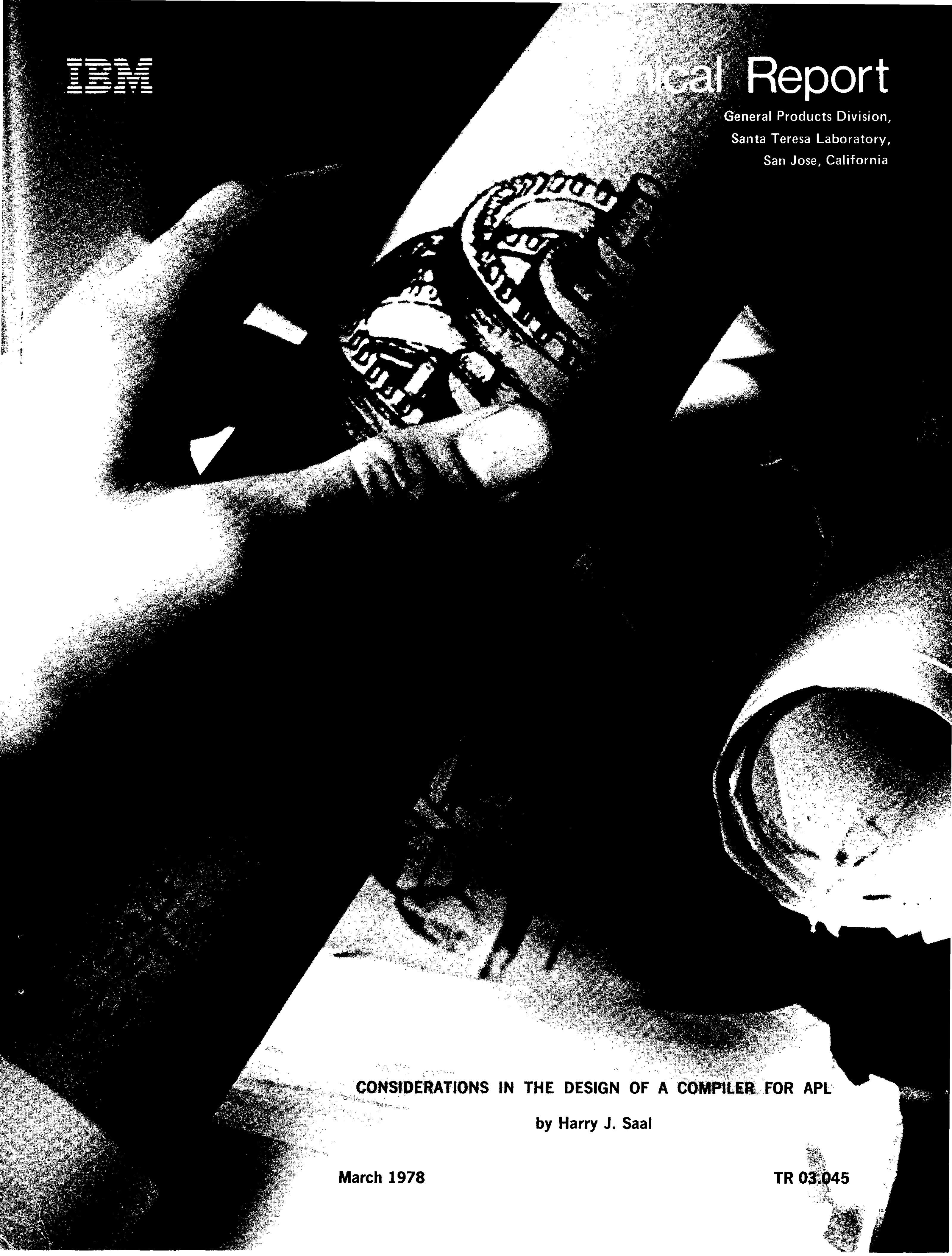

IBM

nical Report

General Products Division,
Santa Teresa Laboratory,
San Jose, California

# CONSIDERATIONS IN THE DESIGN OF A COMPILER FOR APL

by Harry J. Saal

March 1978

TR 03.045

March 1978

TR 03.045

# CONSIDERATIONS IN THE DESIGN OF A COMPILER FOR APL

by

Harry J. Saal

International Business Machines Corporation

General Products Division

Santa Teresa Laboratory

San Jose, California

## ABSTRACT

Existing implementations of the APL language are generally referred to as "interpretive." APL users rely on system behavior of an interpretive nature, such as "desk calculator mode" debugging, and modern interpreters retain that external view even though internal interpreter organization is highly optimized. Increased use of APL has led to many requests for an APL compiler.

This paper first presents a selected annotated bibliography of eight papers which address the question of the efficient execution of APL programs. Based on the results of these studies, the problem of efficiently executing APL expressions seems to be well understood.

The second half of the paper presents a number of environment related questions which must be resolved if one were to design a compiler for APL. Almost all of these issues involve potential tradeoffs of performance for other desirable attributes of an APL system.

# CONSIDERATIONS IN THE DESIGN OF A COMPILER FOR APL

by

Harry J. Saal

## OVERVIEW

Although APL systems have been available for over ten years, almost all implementations share a fairly uniform approach to the execution of the APL language, that of interpreting. If one studies the internal details of existing systems, one sees that this does not necessarily imply that APL systems interpret APL programs in their source form, nor that there are not some fairly high powered techniques used to speed up execution within the framework of interpreting.

From the very first APL implementations within IBM, source statements have been scanned and partially analyzed during function creation or editing, producing a compact internal form which is converted to external form for display purposes. Studies of compiler performance for other languages have shown that this stage, lexical analysis, often dominates the total time required for compilation (assuming no elaborate global optimization or fancy code generation is later performed). In VSAPL for example, the statement *A*←12345 in a function executes about ten times faster than the statement ⍎'*A*←12345'.

Some of the more interesting techniques beyond straightforward interpreting of tokenized source found in various interpreters from IBM or other manufacturers include: parsing and retaining of the parsed internal form as long as it is valid; on-the-fly generation of machine code; copy optimization and usage of arithmetic progression vectors; and incremental compilation with loop merging (discussed later).

Techniques such as those described above provide for quite efficient implementation of APL. These implementations compare very favorably to existing small, fast compilers for conventional languages such as FORTRAN. Notwithstanding the above, the growth in usage of APL, particularly for large production applications, results in continued concern with the performance of APL systems, of which CPU time is one factor. This has led to considerable interest in the production of a compiler for APL.

This paper is organized in three sections. The second part is an annotated bibliography of what I consider to be seminal papers relating to the techniques likely to be useful in any compiler for APL. I will mention only a few of many articles devoted to the subject, and suggest that the interested reader locate the others by following the references in the papers listed here. The papers discussed in the following section show a steadily increasing understanding of techniques suitable for providing higher performance execution of very high level languages, such as APL, without necessarily imposing costs that are prohibitive.

The third section is devoted to enumerating some of the tradeoffs and considerations involved in defining what is meant by an "APL compiler." It appears difficult to finesse many of the issues raised therein so that no tradeoffs are involved, and "everyone is happy."

APL systems are expected to provide much more than just the APL Language itself. A processor for a conventional language like FORTRAN or COBOL relies on host system components for all services. In contrast, APL implementations provide not only the execution vehicle for the APL notation, but a consistent editor, a debugging system, library storage scheme and access to external files. This attitude towards the APL environment probably accounts in large measure for the resemblance among implementation technologies from many different computer manufacturers.

## ANNOTATED BIBLIOGRAPHY

The papers discussed are presented in approximate historical sequence. Differences between the date of development, publication as reports versus publication in a journal, etc., make exact sequencing somewhat difficult. There has been a significant amount of re-invention as well. This may be due in part to the lack of suitable bibliographic sources for this material.

Let me reiterate that these papers are not the only relevant ones. This is due in part to my own lack of appreciation of the significance of some, and in part to the lack of space in this paper. Consider this a set of pointers or milestones along the path towards a compiler for APL.

1. An APL Machine
   Philip Abrams
   SLAC Report No 114
   Stanford Linear Accelerator Center
   Stanford University, Stanford, California
   February 1970

Although his interest is in machine organization, Abrams introduces two techniques useful in efficient evaluation of APL expressions. He shows that a class of APL primitives, "select operations", can be viewed as transformations on addressing descriptors accessing materialized APL objects, without explicit evaluation. Thus one can "beat" operations by manipulating descriptors instead of generating temporary results. Abrams uses the term "dragging" in his scheme to mean deferring application of a class of non-select functions, so as to provide loop merging when actual evaluation is ultimately required.

Abrams demonstrates the mathematical correctness of his transformations, and evaluates the performance of his machine organization on several examples, compared with the so-called "naive" interpretation methods. Abrams was the first to raise the issue of the suppression of side effects due to optimizations, e.g., must one receive a *DOMAIN ERROR* during the evaluation of 1↑2 2÷1 0?

2. Efficient Evaluation of Array Subscripts of Arrays
   A. Hassitt and L. E. Lyon
   IBM Journal of Research and Development
   January 1972, pp. 45-57

Hassitt and Lyon treat the problem of optimizing the evaluation of array subscripting in APL where the subscripts take a variety of forms. They present algorithms to simplify the addressing patterns involved in order to minimize the number of loops and loop variables required. They show how the select operations of Abrams can be transformed into cases of subscripting, and their techniques then are applied. They also extend the class of select operations to include compression and expansion, omitted by Abrams.

The techniques described were applied in the implementation of APL/CMS and the later VSAPL system, both in software and microcoded assist versions.

Hassitt and Lyon remark that "ravel and some cases of reshape can be considered as selection operations but it is usually not profitable to do this." We will return to this point in the discussion of the paper by Guibas and Wyatt.

3. APL in PL/I
   M. T. Compton
   RC 4481
   IBM Yorktown Heights Research Center
   Yorktown Heights, New York
   August 1973

Compton describes a preprocessor for APL-like extensions to PL/I. Objects are restricted in shape and type through standard declaration statements, and error checking is essentially ignored. Compton treats transpose, reverse, compress, expand, take and drop as forms of select operations, in that they do not produce explicit code, but control the order and number of loops over element generators. The output of the translator is executable PL/I code, but no timing comparisons are presented with interpretive APL execution.

Compton's interpretation of the rules of APL is rather imprecise, leading to all kinds of interesting extensions, e.g., *A∧.∊B*, *A+./B*, *A+.⍳B*, vector - matrix conformability, etc.

4. Does APL Really Need Run-time Checking?
   Alan M. Bauer and Harry J. Saal
   Software-Practice and Experience
   Vol. 4, 129-138 (1974)

The authors show that eighty percent of the checking performed by "naive" APL interpreters could be validated statically. The sample was 39 APL programs, containing about 1500 references to APL primitive functions.

The APL statements were parsed manually, and pseudo-interpreted to analyze attribute information. The lack of precise information about flow of control in programs containing branches limits the accuracy of analysis possible. Static validation can eliminate any need to check for possible *DOMAIN* and *VALUE* errors in almost ninety percent of the cases, whereas *INDEX* checking can almost never be removed (although present in very low frequency).

The authors conclude from the success of static analysis that the addition of declarations to APL is not a prerequisite for compilation.

5. Steps toward an APL Compiler
   Alan J. Perlis
   Yale University
   Computer Science Research Report 24
   January 1974

Perlis addresses the question of efficient execution of APL expressions, but like Abrams questions the need for translation to conventional machine organizations. Perlis' scheme treats APL objects as "streams" of values produced by a "generator." A sequence of generators is called a "ladder", a coroutine that is controlled by accessing descriptors similar in flavor to Abrams'. Rather than maintaining an access polynomial for addressing succesive elements of an array in ravel order, Perlis uses a scheme by which the "next" element index is generated. This introduces additional generality into the accessing scheme. Perlis presents rules by which select operations (extended as in Hassitt and Lyon) can be transformed into ladder manipulations.

Perlis describes the overall flow of a compiler based on ladders. Only APL expressions which are rank invariant are compiled to ladder code. They are translated on their first occurrence during execution, and code is generated to check rank changes on all later executions. If the ranks change, the compiled code is invalidated, and will be interpreted on later executions.

The forthcoming thesis by Terrence Miller (Yale University, 1978) gives more detail on the translation process, considering both conventional high level languages and a "ladder machine" as targets. The process is essentially a bottom-up method similar to the "dragging" described by Abrams.

6. Type Determination for Very High Level Languages
   Aaron M. Tenenbaum
   Courant Computer Science Report 3
   Courant Institute of Mathematical Sciences
   New York University, N. Y.
   October 1974

Tenenbaum's work is based on problems associated with the translation of programs written in SETL, a very high level language based on set theory. Like APL, SETL is declaration free, and identifiers may refer to objects of varying data type and size. This work can be viewed as an extension of the investigation of Bauer and Saal beyond shape analysis.

SETL datatypes are treated as a finite lattice rather than completely nested (since SETL objects can contain other objects to any depth). This leads to little loss of information in practical cases. Global flow analysis is performed on the programs, and datatype information is then propagated starting with constants and using the properties of the SETL primitives.

Six programs were analyzed, containing 403 primitive functions. The analyzer correctly determined the datatype resulting from the primitive in 77 percent of the cases.

7. A Dynamic Incremental Compiler for an Interpretive Language
   Eric J. Van Dyke
   Hewlett-Packard Journal
   Volume 28, No. 11, 17-23 (July 1977)

This translator for APL is very heavily based on the work of Abrams. It compiles incrementally, generating code for a statement when it is first encountered during execution. A class of further optimizations of subscript expressions similar to those described in Hassitt and Lyon is incorporated.

The code produced is always preceded by "signature" code, which validates the assumptions made during the translation phase. Initial translation of an expression produces "hard" code, in which the actual sizes of objects (e.g. the length of a vector) are compiled into the code. On later executions, if the code "breaks", i.e. does not pass the signature tests, "soft" code is generated which is somewhat slower and less dense, but more likely to survive further executions. The compiler accumulates information in a bottom-up fashion similar to Abrams' dragging.

The actual code produced is not HP/3000 machine code due to the addressing limitations that would be imposed on the size of an APL workspace. Instead, the code is for a pseudo-machine which is supported by special microcode that does virtual storage management. Once the appropriate data is present, conventional HP/3000 code executes any data manipulations or calculations required. Source APL statements are maintained in an internal representation which can be translated back to external form for editing or recompilation, as well as in their compiled form.

8. Compilation and Delayed Evaluation in APL
   Leo J. Guibas and Douglas K. Wyatt
   Fifth Annual ACM Symposium on
   Principles of Programming Languages
   Tucson, Ariz.
   January 1978

Guibas and Wyatt extend the work begun by Abrams and Perlis and describe several extensions and improvements in translation efficiency. In particular, they show how to compute "steppers" (similar to Perlis' ladders) by parsing an expression, performing initial rank and shape analysis, and then propagating information in a top-down fashion.

The concept of a "conforming reshape" is introduced, where a reshape of an array essentially preserves the shape of the "last" axes, and merely adds "leading" axes. This mechanism not only handles scalar extension, but more significantly provides the means to deal with inner and outer products, which must recycle over their arguments several times during evaluation.

Merging of adjacent coordinates is performed using essentially the same techniques described by Hassitt and Lyon. A means of saving partial results, called "slicing," is introduced, whereby intermediate results that must be generated because they cannot be deferred can be saved for reuse in outer loops.

The authors state that the "code generated is well suited for execution by a minicomputer" although no examples are shown. The overhead remaining due to residual signature checking for datatypes and shape analysis is not evaluated in the paper.

## SOME DESIGN ISSUES FOR AN APL COMPILER

The view of APL as a system rather than as a language raises a number of issues for the design of an APL compiler. This section is an overview of some of the questions which arise in the consideration of any compiler for APL. Certain issues seem to require a tradeoff to be made, whereas others relate to the goals which motivate the construction of a compiler.

1. Language compatibility

APL contains a number of features which require either interpretation or dynamic compilation during execution. These features relate to the name space of APL objects, rather than being strictly value oriented. For example, full treatment of ⎕-input, ⎕*FX*, ⍎, etc., cannot be done during a static compilation. One can consider alternatives which restrict or eliminate this type of feature, or still provide it but at the cost of potential performance degradation or additional storage requirements.

Another problem arises from the dynamic interpretation of the syntax of APL statements. It is well known that APL statements cannot be parsed statically due to dynamic localization effects. Even worse,due to the potential effects of features such as ⎕*EX* or ⎕*FX*, one cannot fully parse an APL statement prior to executing it. The syntax of a statement after execution may be different from what it was prior to execution!

Another issue is that mentioned in the review of Abrams' work, namely the suppression of certain error indications due to optimizations. Must all expressions be executed as literally as conventional interpreters have historically done?

2. Localization of errors

APL users currently are given a fairly good indication of where an error occurred during execution. A compiler which performed any rearrangement of code sequencing (moving constant expressions out of "loops" for example) could indicate where an error was encountered, but the rules for determining what had or had not been executed prior to the detection of the error might be very different. In cases where checking of conformability was factored out of complex expressions (as in the work of Guibas and Wyatt described previously), one might receive a *LENGTH ERROR* for a primitive function prior to evaluating the entire right hand subexpression (which might produce a *DOMAIN ERROR* if executed).

Current APL systems assume that once an error has been detected the terminal users will use the features of APL to determine exactly what went wrong. To permit arbitrary expressions to be entered and evaluated one needs some form of interpreter or dynamic compiler. Alternatively, one could provide a source level dump facility, or a facility which converts a workspace to a form which could be interpreted by current systems.

## 3. High level optimizations

Some rather elegant optimizations are feasible in the APL environment which are not very applicable to lower level languages. Among them are the recognition of idiomatic usage of APL. For example, ρρ could be recognized as a single operation which gives the same result as the successive executions of the individual primitives, but didn't actually create an intermediate result. Certain cases of recursive function calls can be transformed into iterative execution, becoming both faster and less space consuming. User defined functions (*IF* is a classic example) could be integrated inline in the calling sequence, rather than actually invoking a function call and return.

These optimizations are conceivable in the context of APL due to its compact nature. However, they would require additonal compilation time, and make the correlation of the translated code with the source code difficult.

## 4. Declarations

As described in the discussion of Bauer and Saal's paper, adding declarations en masse to APL seems unnecessary. However, a compiler might find certain pieces of information very valuable in producing better code. For example, knowing which functions are "external" and which are just called from within as subfunctions make it feasible to do global flow analysis within an APL workspace. (Even so there appear to be numerous problems.) Knowing which variables are eventually shared during executions and which are not also permits optimized treatment of non-shared variables, which are the bulk of the references.

## 5. Scope of compilation

A compiler could "look" at various units of text in gathering information and performing optimizations. These could be at the level of individual statements, functions, or workspaces. The larger the scope of the compilation, the more optimizations are feasible, but at the expense of longer compile time. One possible approach is to permit users to select among a variety of optimization levels, as in FORTRAN H.

A compiler might be able to produce better code in various cases if it could interact with a user when it could not determine certain facts.

Certain of the techniques discussed in the second section will produce especially good code for a certain type of APL program (i.e. novice, scalar code versus advanced production code). This complicates choosing a representative sample of code by which to evaluate a compiler.

A compiler could produce "object" code that can be saved, but with the attendant risk of not having the source and object always agree. Undoubtedly some users would like to be able to "patch" the object code and improve it beyond the abilities of the compiler. Should there be a "link-editor" for constructing modules from various sub-compilations? How important is the ability to link to precompiled subroutines from other languages, such as FORTRAN?

## 6. Space-time tradeoffs

There are a large number of fairly dramatic space-time tradeoffs to consider for an APL compiler. For instance, how important is the time (real or CPU) taken for compilation? Is the absolute maximum performance after compilation the sole criterion? What about the size of the compiled code?

Code compiled directly to machine language level is likely to be huge when compared to current APL internal tokenized form. Alternatively, one could translate to some more compact intermediate form which is interpretable at high speed, possibly with a microcode assist available.

How large a run-time package should there be? Certain cases of APL primitive functions which occur with low frequency (say 0.1%) may have some super optimizations (say 100 times faster than the normal case). How important are including these cases (there are lots of them!)?

Dealing with model dependent tradeoffs is quite complex. Certain constructs are superior on some 370 models, whereas they are inferior on other models. Should the emphasis be on the fastest or slowest models?

One way to resolve some tradeoffs is to rely on frequency information. An APL compiler might collect execution frequency information dynamically, or use information provided by the user, or from automatic static analysis. Collecting this information adds some costs on its own.

## SUMMARY

I have tried to review some of the important papers relating to techniques useful in compiling APL programs. There has been considerable progress in understanding how to deal with many of the data manipulation primitives of APL, without actually performing data movement.

One could produce a compiler for almost the entire APL language, for programs which are "correct" and don't require interactive debugging. On the other hand, it appears that subtle issues relating to the entire APL environment are yet to be understood.

Replacing today's interpretive APL systems by a compiler is unlikely to be a universally beneficial step. We have presented a list of some of the issues which seem to require a tradeoff in the design, and we hope that APL users consider the relative importance and costs of the alternatives.