This work was presented to the graduate school of Yale University in candidacy for the degree of Doctor of Philosophy

Tentative Compilation

A Design for an APL Compiler

May 1978

78-CS-013

Terrence C. Miller

Institute for Information Systems, C-021
University of California, San Diego
La Jolla, CA 92093
Report 78-CS-001

Dept. of Computer Science
Yale University
New Haven, CT 06520
Research Report No. 133

This work was supported in part by the Mobil Foundation

ABSTRACT

Tentative Compilation
A Design for an APL Compiler

Terrence Clark Miller
Yale University, 1978

The programming language APL has obtained a growing following. Much of its popularity can be ascribed to its terseness (complicated acts can be described briefly) and composability (complete algorithms may be expressed as a single unit - the one-liner). However, the user may pay a high price for these features in terms of inefficiency of execution, particularly in terms of memory space required. This dissertation describes the design of a compiler for APL which significantly lowers the cost of APL execution.

The design includes a notation with which the actions required for the execution of the majority of the APL operators may be expressed. Transformations are applied to the program expressed in this intermediate notation. The transformations re-order independent calculations for a given operation, and intermix calculations for several operations. The intent is to produce an intermediate result only when it is needed (thus avoiding storage) and only if it contributes to the final result (thus eliminating unnecessary calculations). Examples show that significant savings are obtained.

The output of the compiler is expressed in terms of "ladders" - a control structure designed by Alan Perlis to simplify APL execution. The compiler can generate code for the "ladder machine" designed by Charles Minter.

Tentative Compilation:

A Design for an APL Compiler

A Dissertation

Presented to the Faculty of the Graduate School

of

Yale University

in Candidacy for the Degree of

Doctor of Philosophy

by

Terrence Clark Miller

May, 1978

ACKNOWLEDGEMENTS

I am grateful to my advisor, Alan Perlis, for suggesting this problem and for asking the right questions to keep the work going. The other members of my committee, Ned Irons and Larry Snyder, provided valuable insights into how this work could be effectively presented.

The thesis work of Charles Minter provided the hardware environment for this software design. His success made my work possible. He, along with Mike Condry, contributed many valuable suggestions. The text of this thesis was prepared using software much improved by the work of Steve Reiss.

Finally, I would like to acknowledge the support of my wife, Denise Sullivan. She contributed greatly to the author's spelling, grammar, and sanity.

This research was partially supported by the Mobil Foundation.

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 THE PROBLEM

The programming language APL has achieved a growing following (mostly outside the computer science community). In this thesis we present a design of a system for executing the language which attempts to alleviate some of the difficulties that have been cited as limiting the continued growth of the language. In particular we address the following problems:

1. The interpretative execution of APL programs can be slow compared to that attained with programs compiled from languages such as FORTRAN or Algol 60 [12].

2. APL functions will often generate large arrays as intermediate results on the way to a small array (or even a scalar) as the answer. An example of such a function is given in chapter 5.

3. The style of APL programming most efficient for an experienced user (full use of the power of the array operations to eliminate explicit control structure and improve brevity [20]) tends to worsen the problems listed above.

4. Current implementations which process functions on a line by line basis encourage the use of long lines to achieve efficiency. Readability suffers.

The environment for which the compiler is designed is that of a single processor whose instructions act on individual data items only. It has also been designed to be most useful in either of two circumstances:

1. When APL is used in a production situation, a given function will be executed repeatedly. Thus the cost of compiling the function (even if high) will be offset by the savings in execution time. Also the input arrays will tend to be large in such a situation. As the execution time rises, the significance of compiler overhead diminishes. When the outer-product operator is used in place of explicit looping, the size of the intermediate results will often be a power of the size of the input. Given large inputs, actual storage of such intermediate values is not feasible.

2. At the other end of the scale is the small personal APL system for which storage space is the critical resource [~~51~~ 25]. Many functions can not be executed if intermediate results must be stored in memory. The storage required for the more complicated APL system is not of comparable importance. It can be in the form of read-only memory which is of much lower cost. The user of such a system is also in a better position to tolerate longer execution time. (When all else fails, you take the system home, and key in the function just before going to bed).

1.2 PREVIOUS WORK

Currently available implementations of APL have attempted to solve some of the problems mentioned above. In the sections below we describe remaining weaknesses which motivated the design of this thesis.

1.2.1 Simple Interpreter

The original implementation of APL [7] and many that followed were interpreters which execute each operator separately as encountered. All intermediate results are stored in memory. Great speed improvement has been obtained by the fine tuning of the routines for various operators and by the recognition of short special patterns of operations [22]. Our design also recognizes a small number of special patterns (which we call "idioms"). However, as is shown in chapters 3 and 5, the reduction of temporary storage may require interleaving the individual calculations of a sequence of operations. From the examples presented in chapter 5 it is clear that the sequences of operations which can be profitably interleaved are too long (and thus too numerous) to be recognized as special cases.

1.2.2 Translation Into Algol

Jenkins [12] implemented a translator from a sub-set of APL into Algol. He was forced to restrict the language so that compilation could take place before any data was available, and so a compiled module would always remain valid. The features of APL which present difficulty in

that regard are discussed in chapter 2. While the compiled code is significantly faster than interpreted APL for scalar calculations the advantage almost disappears for large arrays. Jenkins did not investigate the reasons for the inefficiency of array calculation. However, experience gained in this implementation suggest that it resulted from sequential execution of operators which require large amounts of temporary storage, and from the cost of array element address generation.

1.2.3 Beating And Dragging (Interpreter)

In his 1970 thesis "AN APL Machine" [1] Philip Abrams investigated the semantics of APL and developed two techniques for improving execution efficiency. They are:

1.2.3.1 Beating - Abrams recognized that a set of operations he called selection operations (Take, Drop, Reverse, Transpose, and Subscription by vectors of the form $A+B\times\iota C$) could be implemented by changes to the parameters used to generate array item addresses, and did not require actual creation of the result array. He also showed that an expression in which a selection operation was applied to the result of certain operators could be transformed so that selection (which may decrease but never increase the number of elements) was applied before the operation, possibly reducing the number of calculations.

1.2.3.2 Drag-Along - Abrams' interpreter deferred execution of operators as long as possible. Possible meant:

1. The value was not required for assignment to a variable.

2. The function mapping position in the result to position in the input was simple (ie. Grade-up was not deferred since the function would be "sort").

3. The calculations for each resultant array position were independent.

This resulted in savings of storage of intermediate results and improved opportunities for beating.

To a large extent the work of this thesis is an extension of the work of Abrams. Three weaknesses in particular are addressed:

1. An Abrams interpreter will always store the operands and the result of certain operators, even thought it is possible to defer them in many cases. The operators include Compression, Expansion, Catenation, Rotation, general Subscription, Scan, Encode, and Decode.

2. Function lines are processed independently. We will see in chapter 5 that important storage savings can result from the elimination of variables that are used only to carry a value between two lines.

3. Even if assignment occurs in the interior of a line, it is never deferred. No consideration is given to eliminating storage specified by the user when he recognizes a common sub-expression.

The elimination of user specified storage requires analysis of the

entire function to verify that the data is not used elsewhere. That analysis would not be feasible for an interpreter.

1.2.4 Beating And Dragging (Compiler)

The APL for the Hewlett Packard HP-3000 II computer is a compiler [13]. In contrast to the approach of Jenkins, no restrictions are placed on the APL to be compiled, and no declarations are required. Compilation is deferred until the execution of a line is required. At that time properties of the input data are available to guide compilation. When a line is executed a second time, the properties of the new input must be inspected to verify that the previous compilation remains valid. This technique, which has also been described by Perlis [19], is an extension of the concept of incremental compilation as described by Mitchell [18].

The existence of the HP-3000 compiler is important to the work of this thesis in that it disproves the contention [11] that APL can not be compiled. However, it shares the limitations of the interpretative implementation of Abrams work. The compiler does not consider more than one line at a time, nor does it eliminate user specified storage.

1.2.5 APL Emulator

Significant speed improvements over an interpreter may be obtained by writing micro-programs to directly execute some of the APL operators (for example [11]). However, this approach prevents the interleaving of operations required for beating and dragging. The emulator will still perform all the unnecessary operations done by the interpreter, only faster. Also no reduction in temporary storage is possible.

1.3 A MULTI-LINE COMPILER

The design presented here is an extension to the HP-3000 APL compiler which differs from that system in the following ways:

1. When a function is executed, the compiler will determine if several lines can be compiled together as a unit.

2. The execution of a larger class of operators (including assignment) can be deferred.

3. If the definition of a variable is active only within a single compiled unit, the elimination of that storage will be attempted.

In order to lower the frequency of recompilation and increase the size of compiled units (both become more important as compiler overhead increases), restrictions are placed on the input language. They are detailed in Chapter 2, and are less severe than those proposed by Jenkins. Chapter 2 describes the compilation procedure.

In common with the HP-3000 compiler, this compiler makes no attempt to do mathematical analysis of the users algorithm.

The object code of the compiler is the description of a network of ladders - a control structure consisting of nested loops connected by co-routines designed by Perlis [19]. The ladder structure was designed

to facilitate the access to array elements and the implementation of the selection operations. Chapter 3 describes ladders in detail and discusses the extensions needed to make possible the deferment (interleaving) of the additional operators listed above. The process of handling each APL operator and minimizing overall temporary storage is described in chapter 4.

We consider the major contributions and accomplishments of this work to be:

1. The development of a procedure (described in Chapter 2) which determines the requirements for the legal execution of an APL expression, precisely locates the small sub-set of those requirements that may not be verified at compile-time, and identifies the point at which information required for compilation and execution will first become available.

2. The development of a representation in which the actions required to execute a majority of the APL operators can be expressed. Taking advantage of that representation, this compiler can defer the execution of catenation, compression, expansion, general subscription, rotation, encode, decode, and scan as well as the simpler operations handled by earlier systems.

3. The development of a translation procedure which can handle those operators which make it impossible to move all selection operators to the operands of an expression. In particular we handle those cases in which selection operations may be efficiently handled by being moved to the root of the parse tree for the expression.

4. The development of an implementation for compression which does not require the entire left operand to be calculated in advance. Massive savings in storage can result in those cases when the same array appears in both the left and right operands (a common APL technique is to compress an array using a function of itself).

5. The development of a storage minimization algorithm which will identify when only part of an array must be in memory and which can make improvements even in those cases where the more complex operations trap a selection operator in the middle of an expression.

We show in our examples large gains in performance which resulted from the new capabilities listed above. In contrast, the very recent work of Guibas and Wyatt [10] makes no attempt to handle items 2-4 and deals with 5 in very weak way.

The purpose of this work was to study the design of an APL compiler and not to build one. No actual software exists. The production of a useful, complete APL system is not a one-person task. In addition, much of the work that would be involved (programming workspace control, function editing, ...) has little connection to the design issues considered in this thesis.

CHAPTER 2

COMPILING A DYNAMIC LANGUAGE

The execution of a program is a process of binding (the last step is the binding of a particular value to the output variable). A compiler for a language affects bindings without reference to the input data. The bindings thus made are permanent. An interpreter binds only after inspection of the input and only for the duration of that one execution. Most actual systems use a combination of the two techniques and are labeled according to which predominates.

It has been believed (for example see [11]) that the definition of APL makes it impossible to eliminate variability before each actual execution. That conjecture has, however been disproved by the creation of the HP-3000 APL compiler. Compilation of APL is possible if we include in the definition of compilation given above bindings made with reference to the actual data for the first execution which then may become permanent. Since the continued validity of the bindings is not guaranteed, it must be re-verified whenever the compiled code is executed with new data. Efficient execution will only occur when the bindings remain valid and compiled code may be reused. Fortunately the

dynamic features of APL which could cause the most recompilation are not often used. Their use is also an example of a style of programming which I am quite happy to discourage.

Since compilation references input values and we need to verify the continued validity of compiled code, the program compiled must be divided into "units" which are compiled separately. The division must be done so that the verification of the choice of several possible compilations for the unit depends solely on properties of input variables before the unit is executed. The compiler will generate a preamble for each unit which specifies the required operand characteristics. The testing for data dependent situations for which no compilation is correct may be done by code compiled into the unit.

The Hp-3000 compiler will never compile more than one line in a single unit, nor will it ever include a called function into a line it is compiling. Since efficiency increases with the size of the compiled units, my compiler will, when possible, do both. It also tries to locate those cases in which potential binding variability resulting from a given operation can not be legally realized, and thus to eliminate the need for division into separate units.

In the course of this discussion, we will introduce modifications to the language APL which are assumed by this design. They are presented individually in the section of this chapter which first presents the design decision which motivated them. They are summarized in Figure 2-1.

Language Changes

Functions	- Function definitions are global and may not be masked. - Valence of a definition may not change. - Local variables are not available to a called function unless explicitly indicated in function header (new syntax).
One-Element Array	- A one-element array is not equivalent to a scalar. Monadic ι (new operator) creats a scalar.
Operation Modifiers	- The dimension to be affected by an operator must be implicit or specified as a constant.
Index Origin	- Origin may be changed only while in calculator mode.
Take	- Take may not return more elements than in the right operand.
Goto	- The GOTO operator may not branch to a line which is not labeled.
Partial Assignmet	- When assignment changes only part of an existing array, the right operand must have the same shape as the sub-array assigned to. - The selection operators (↑, ↓, ⌽, ⍉, and [ι]) may be used to specify the sub-array to be changed.
Execution Order	- Right-to-Left order of execution is not garanteed except for the operands of the new line separator operator ◊.

Figure 2-1

2.1 BINDINGS

The bindings which must be made in order to execute APL are discussed below. For each the sources of variability are listed and the binding time is given.

2.1.1 Valence

An APL identifier may have valence 2 (dyadic function), 1 (monadic function), or 0 (niladic function or variable). The valence of an identifier can not always be determined from the syntax of an APL expression. For example, in the expression *A B-C* the identifier B could have any valence. As a result an APL expression may only be fully parsed in the context in which it will be executed, and valence bindings made at first execution may fail in three cases:

2.1.1.1 Function Entry - A global symbol may be redefined between calls on a function. Unless this is ruled out, any function which references a global symbol might require recompilation each time it is executed. The entry to the function must begin a new unit.

2.1.1.2 Function Call - A global or local variable may be redefined as a side effect of a function call (the execute operator is considered a function call). Unless this is ruled out, the function must return to a different unit than itself to permit checking for such side effects (done by interpreter as part of the process of beginning the execution of a unit).

2.1.1.3 Multiple Definitions - If a line of an APL function can be reached from more than one predecessor (target of a GOTO), then there may exist multiple definition points for a symbol used in that line. The possibility of the two definitions having different valence makes it necessary to have all such statements begin a compilation unit.

Strawn [23] showed that for APL without local function definitions (assumed here), only 2% of the identifiers in a sample of programs had ambiguous valence. Once a valence is resolved (first execution or user query) the use of the identifier remains fixed in almost all cases (1 change in 1 million possibilities) [22].

We therefore restrict the function definition mechanism in APL slightly. All function definitions must be global. The masking of a global function name by a local variable or formal parameter will be an error. Any operation which changes the valence of an existing symbol is an error. Violation of this restriction is a run-time error from ⎕FX. These restrictions eliminate the necessity of re-parsing due to valence change.

2.1.2 Rank

Both the control structure of the object code and conformability checks depend on the rank of input operands and intermediate results. Therefore compilation requires knowledge of the ranks of the result of all nodes of the parse tree. In the majority of cases a given unit will be syntactically correct with only one set of operand ranks. If this

occurs, then compilation can take place without reference to information about the operands. The following circumstances may introduce rank variability:

2.1.2.1 Multiple Assignments - If a line of an APL function can be reached from more than one predecessor (target of a GOTO), then there may exist multiple assignment statements defining the operands of the line. To permit the interpreter to check whether an alternate path has resulted in ranks different than at first execution, any such statement must begin a compiled unit.

2.1.2.2 Transposition - A transposition operator with a variable left operand has a result of unknown rank (diagonalization may or may not be specified). If other constraints do not eliminate this variability, the transposition may not be compiled until information fixing the rank is available. It can be provided by the user, or the transposition may be placed in a separate compilation unit from the calculation of the left operand. The value of the left operand will be used to guide the compilation of the unit containing the transposition or verify its reusability.

2.1.2.3 Reshape - A reshape operator with a variable left operand has a result of unknown rank. If other constraints or information from the user do not eliminate this variability, this implementation will not compile the operation. The generation of operands and the use of the result will be placed in separate compiled units.

2.1.2.4 Function Entry - The rank of global variables and arguments may change between calls on a function. In many cases only one possibility will be legal, but certainly a function whose syntax allows variable rank arguments may be written. In two special circumstances the rank variability may be hidden from the function by the caller [3].

1. If no global variables are referenced, the function processes each argument item (or pair of items for a dyadic function) independently, and result is a scalar item for each input item, then the function may be compiled so as not to care about the structure of input.

2. If no global variables are referenced, if the function processes its argument(s) by row (or plane or ....), and if the result for each group is either a scalar or the same shape as the input, then the function can be compiled to be called repeatedly once for each group.

Otherwise the function must be recompiled when global or argument ranks vary, and thus must begin the compiled unit containing it. The determination that a function falls into one of the special cases listed above requires only a simple examination of the code produced when the function is compiled independently (ie - if all computation is in the inner-most loop, then each operand item is handled independently).

2.1.2.5 Function Return - A Function may return results of variable rank or change global variables. Unless the result rank is a function of argument rank only and no global variables are changed, the location to which the function returns must begin a compiled unit.

Fortunately the rank of a variable usually has a connection to the semantics of the program which results in its being fixed. The work of Bauer and Saal [4] suggests that 80% of ranks may be determined statically (without access to actual operands). Our experience is that except for the case of universal functions, which can be compiled, use of the same expression to generate results of different rank on successive executions is rare. Most array ranks are derived from a fundamental characteristic of the problem being solved.

This design identifies at compile time those scalars which must be used repeatedly in a single operation in order to have conformability. Since array sizes may not be known at this time, this compiler will not allow one element arrays to be used as scalars unless they have been converted into a scalar using a new operation Monadic ɩ. The process which checks for rank conformability will insert the conversion operator where needed if the size of the array is known to be 1 at compile time (ex. 1↑A), and if the operation using the value requires a scalar (ex. monadic ɩ). The ravel operator "," must be used to convert a scalar into a one element vector.

2.1.3 Operation Dimension

Several array operations apply to one (implicitly or explicitly specified) dimension of their operand(s). The code compiled for these operations is heavily dependent on which dimension is affected.

Therefore, we restrict APL to use only constants when the dimension to be operated on is explicitly specified.

2.1.4 Index Origin

This compiler produces code which may (depending on operators in the expression) be invalid if the index origin changes. In order to avoid constant testing, we allow index origin to be changed only via the ")ORIGIN" command executed in calculator mode.

2.1.5 Length

The compiler will attempt to use syntactic constraints to fix the size of array operands, but if it fails, the compiler will not bind the compiled code based on the sizes at first execution. This approach contrasts with the HP-3000 APL system which does bind on size, resulting in frequent recompilations. The object code has been designed so that operand size is reflected in a small number of parameters which must be given values by the interpreter before a compiled unit is executed. All length conformability tests actually required will be done by the interpreter. The compiler will generate a preamble for each compiled unit which instructs the interpreter what calculations and tests to perform.

If a compiled unit contains an operation whose result size can not be calculated before the unit executes, the unit will interrupt its execution when the size is first available. The interpreter can then

perform any necessary conformability checks and calculate any parameters which depend on that size. There are 7 APL events which can cause length variability.

2.1.5.1 Compression - The length of the compressed dimension is equal to the number of 1's in the left operand. That length will be available the first time the left operand has been completely used (compression of other than first dimension will result in repeated access to left operand). Parameter adjustments may be required even if conformability checking is not.

2.1.5.2 Take And Drop - The length of the result of take or drop depends on the value of the left operand. Since in the object code these operations are implemented by changes to addressing parameters which must be calculated by the interpreter, take or drop with a variable left argument is compiled so that the operation does not begin until the left operand has been fully evaluated. At that point the interpreter will calculate the parameters which control access to the selected elements of the right operand.

2.1.5.3 Over-take - If the Take operation is allowed to return more elements than exist in its right operand, a Take operator with a variable left operand has a result of unknown size. Also the performance improvement algorithm used by this implementation tries to move the Take operation so that it is performed as early as possible. That is correct only if the Take operator will not return more elements than exist in the right operand. The restriction is imposed dynamically when the take is executed. If the over-take option is desired, it could be included as a separate operator (which would be interpreted).

2.1.5.4 Reshape - A reshape operator with a variable left operand has a result of unknown size. If other constraints do not eliminate this variability, the operation will be performed by the interpreter.

2.1.5.5 Function Entry - If a function references global variables, or if internal conformability is not implied by conformability of arguments, its execution will require interpreter processing on entry.

2.1.5.6 Function Return - If a function sets global variables, or if the result shape is not that of some scalar operator applied to the argument(s) (possibly reduced), then conformability checking will be required after a call on the function.

2.1.5.7 Multiple Assignments - If a line of an APL function can be reached from more than one predecessor (target of a GOTO), then there may exist multiple assignment statements defining the operands of the line. Thus conformability checking will be required.

The output of the compiler is a co-routine with the interpreter. The interpreter will do parameter computation and conformability checks and the compiled code will evaluate the APL. The two will interleave as needed. The work of Bauer and Saal [4] suggested that only 38% of the potential length conformability checking is actually required and that length checking was required in an average of two places in each of a collection of functions. Thus the amount of interleaving will not be

excessive.

2.1.6 Value

The compiler attempts to perform calculations at compile time so as to increase efficiency and tighten syntactic constraints. The interpreter will perform the size operation (monadic ρ), since that requires access to symbol table information. Code to detect value dependent errors will be compiled into the object code (index, domain (ex. divide by zero), and right operand of expansion with wrong number of 1's). We have imposed an additional constraint on the dyadic scalar operations used with Scan. All items of the operand must be in the range as well as the domain of the operator. This is done to permit the use of a technique developed by McDonald [16] for executing the scan operator without repeated access to elements of the operand.

2.1.7 Type

Operand type must be known at compile time. If syntactic constraints do not eliminate potential variability, compilation before first execution will require user interrogation. There exists no APL operation whose result type is not given by operand types but the following situations may require type checking and recompilation:

2.1.7.1 Multiple Assignment - If a line of an APL function may be reached from more than one predecessor (target of a GOTO), the type of variables referenced might be derived from different operands.

Thus that line must begin a compiled unit.

2.1.7.2 Function Entry - A function which has alternate legal compilations depending on the type of arguments or global variables may not be included in a larger compiled unit.

2.1.7.3 Function Return - A function may not be part of a larger compiled unit if its result could be of variable type or if it changes global variables.

Bauer and Saal [4] found that in a sample of programs only 12% of the domain checking (which includes type checking) could not be performed statically. This suggests that type variability is rare.

This compiler will not process integer and floating point numbers as two separate types. It assumes the arithmetic instructions of the target machine are type sensitive and convert automatically as needed. It also allows a numeric value to be used as a boolean operand (which is standard for APL). When this conversion is required, the compiler inserts the new operation monadic ⊃ into the expression. This operator will signal a domain error at run time if its operand has values other than 0 and 1.

2.1.8 Position

The actual storage location for an array is not known until each execution takes place. The compiler code will access array elements using pointers which are initialized from parameters at entry. The interpreter will perform storage allocation and set the parameters. A

reshape operation with variable left operand is performed by the interpreter since no upper limit can be placed on the storage required until the operand is calculated. Storage allocation may be interleaved with the execution of the compiled code in the case of compression.

2.1.9 Summary Of Binding Problems

The preceding sections have listed those places which may require interpreter intervention. Some, in particular length conformability checking, are handled by interleaving the execution of compiled code and the interpreter. However, in other cases the validity of the compiled code about to be executed is in question. These circumstances require division into separate compiled units so that execution of a unit is only started if the entire unit is valid (bindings still hold). The locations of possible unit divisions are:

1. The beginning of a statement which may be reached from more than one place in the function (Goto target) is a unit boundary. To make the location of such lines feasible, we restrict the Goto operation so that its right operand must be either the empty vector (no branch), or 0 (function exit), or the line number of a line which is labeled. This restriction is imposed at run-time (Goto is interpreted). Every labeled line then becomes a unit boundary.

2. The entry to a function will be a unit boundary except in special circumstances (see Section 2.2.2).

3. The return from a function will be a unit boundary except in special circumstances (see Section 2.2.2).

4. The point at which the left operand of a Transpose is first available will be a unit boundary.

5. Unit boundaries are required before and after the Reshape operation except in special cases (see Section 2.2.6.5).

6. All system functions and those APL operators which represent a complex algorithm are processed by the interpreter. Figure 2-2 lists the parts of APL which are not compiled. Unit boundaries appear before and after each such operation.

In Chapter 6 we discuss briefly the problem of eliminating these restrictions. The inability of my design to handle control structure (GOTO) can be costly. As an example we consider the APL function:

```
       Z←M COMPOSE P;X;T;U
[1]    U←X←(⍴P)⍴1-Z←,T←1
[2]L:  →L×⍳M>⍴Z←Z,T←⌊/U←P×Z[X←X+T=U]
```

which finds the first M numbers which have the form ×/P*I where P is a vector of distinct prime numbers and I is a vector of non-negative integers. My compiler would separate the GOTO operator from the body of line 2. As a result, the stream generator code would exit to the interpreter after each iteration. However, all information needed to compile this function as a single unit is available (including size of Z which is M). The single unit would execute with much lower interpreter overhead.

Operations Not Compiled

Roll and Deal - ?

I/O Operators - ⎕, ⍞, [illegible], [illegible], etc.

Laminate

Goto - →

Matrix Division - ⌹

Execute - monadic ∊ or ⍎

I-beam - ⌶

Sort Operators - ⍒ and ⍋

All system functions - example ⎕FX

Figure 2-2

2.2 OVERVIEW OF COMPILATION

The compiler will be evoked by the interpreter as a result of two different circumstances:

1. When an expression is executed in calculator mode or an un-compiled function is executed. At this time the entire line or function will be parsed and divided into compilation units. The first of these is then translated and executed. Each unit will be translated the first time it must be executed. Since compilation takes place after the definition of all operands and called functions, all information necessary for compilation is available.

2. The user may also request the compilation of an entire function. This would be done to build a library of functions or to cause a function to be compiled before the function that called it (so that information about the called function is available when the caller is compiled). If the function has not previously been executed in the automatic compilation mode, or if arguments are not defined, then valence, rank or type information not given by the function syntax would have to be supplied by the user.

   If the user recompiles a function that has been executed, he can request that bindings be made based on the properties of its arguments and intermediate results from the prior execution. This facility would be used as follows:

   1. The user would execute a function (possibly as part of testing it). As part of this execution all functions called would be

   compiled.

   2. He would then request re-compilation of the original function. The called functions would be in compiled form and thus the information needed to identify functions which could be linked into the compiled unit of the caller would be available. At the same time the binding information developed during the first execution would be used to guide compilation.

   However, a user will never have to request compilation in order to get correct execution of a function.

In both cases the code resulting from the translation of a function is saved and will be re-used by the interpreter if possible. The steps of compilation are described below. They are motivated by the binding requirements given above. Figure 2-3 lists the steps of an example execution. Routine names shown in all capital letters give the major modules of the design.

SAMPLE EXECUTION

1. COMMAND/SCANNER gets input line and reduces *D←(C≠0)/C ◇ C←A+B* into tokens.
2. CONSTRAINT/PROPAGATION procedure determines that A and B and thus C must be numeric, that C and thus A and B must be vectors, and that A and B must be conformable.
3. The IDIOM/RECOGNIZER makes no changes in this example.
4. The OPERATOR/CONVERSION procedure explicitly indicates that / affects the second dimension of C.
5. The DATA/DEPENDENCY procedure recognizes C as local.
6. The TRANSLATOR generates two outputs:
   a. a stream generator which will execute this expression.
   b. instructions for the set-up and management of the stream generator (see 7 below).
7. The INTERPRETER executes the set-up instructions which include storage allocation for D and transferring the location and size of A, B, and D (C has been eliminated) into the local storage of the stream generator. The last instruction of the set-up program is a co-routine jump to the stream generator.
8. When the INTERPRETER regains control, it executes an instruction to fetch the actual size of D from stream generator local storage, and then exits.

Figure 2-3

2.2.1 Parsing

When all symbol valences are known, APL is a very simple language to parse. Indeed it has been shown that a 3 state finite-state machine augmented by a stack to handle nested expressions is sufficient [24]. The function is parsed into one tree with lines joined by a successor operator. The nodes of the parse tree are labeled to permit other stages of compilation to reference individual nodes. This document uses strings (most often of length 1) of lower-case letters as parse tree node labels. They are assigned in lexagraphic order during a right-left-root order traversal of the parse tree.

2.2.2 Function Calls

Function calls are handled in two different ways. The most general form (always correct) is to create unit boundaries before and after the function call and handle the function call in the interpreter. (The called function may be a compiled user function but the transition is handled by the interpreter.) This type of function call requires that the function arguments and result be held in storage across unit boundaries. The parse tree is altered so that

<left argument> *FUNCTION* <right argument>

becomes

T1←<right argument>
T1←<left argument>
T3←*T2 FUNCTION T1*

System functions and the APL operators listed in table 2-1 are always handled in this way.

User functions which have been compiled previously and which meet requirements listed below may be linked into the code of the calling compiled unit. A description of the linking mechanism is given in Chapter 4 after the control structure of the compiled code has been described. For a function call to be included inside a compiled unit, the following conditions must be satisfied:

1. The function itself compiled into a single unit.

2. The single compiled unit which is the called function does not contain the calling function. (Conditions 1 and 2 rule out direct or indirect recursion.)

3. The compiled unit which is the function does not access any global variable which is accessed by the calling unit.

4. There is only one possible legal compilation for the function (ie no rank or type variability as described in Section 2.1).

In order to simplify this analysis we require that the local variables of a function be accessible to a called function only if explicitly designated in a function header entry of the form:

(*FUN1*;*VAR1*;*VAR2*....)

Similar changes have been proposed by others for the purpose of decreasing opportunities for errors. This modification to APL does not result in a static name scoping system as used by Algol. Its effect is to hide un-named local variables from a called function which is looking back along its call chain to satisfy a global reference. More detailed

analysis would permit some relaxation of these restrictions.

2.2.3 Control Structure

Every Goto target begins a compiled unit which is started by the interpreter. Thus the Goto operator is not compiled and is preceded by a unit boundary. As a result of this design decision, the compiler is heavily biased in favor of the style of APL programming which avoids use of Goto. The whole design is oriented towards the execution of array operations.

The parse tree is now converted to an ordered forest by eliminating all arcs which cross unit boundaries. Each unit is a tree.

2.2.4 Constraint Propagation

The compiler will then attempt to determine the properties of each node of the parse trees. This will be done by propagating information derived from constants and syntax restrictions. The procedure is concerned with 4 characteristics of the value produced at each node of the parse tree:

1. Rank (number of dimensions - a non-negative integer)

2. Type (numeric, boolean, numeric-or-boolean, or character)

3. Length (of each dimension - a non-negative integer)

4. Value (scalars and vectors only)

The constraint propagation procedure attempts to derive this information based on (in order of use and decreasing desirability):

1. The rank (0 or 1), type, length, and value of all constants.

2. Operator semantics (ex. monadic ⍳ always produces a numeric vector and requires a numeric right argument).

3. The properties of previously compiled, called functions.

4. The rank, type, and length of operands represented as compile-time variables (initially with no value) which may be propagated as if they were fixed values.

5. The actual rank and type of each operand (from existing definition or user specification). This information gives values to the compile-time variables defined above.

6. The actual length of each operand.

7. The values of scalar operands.

As each item of information is applied, an attempt is made to propagate that information to other positions in the parse tree (ex. 4↑A is numeric if A is numeric). Appendix B gives the propagation procedure and lists all operator characteristics used. Constraint propagation is done independently for each compiled unit. The handling of and requirements for information about each of the result properties is

described below:

2.2.4.1 Rank - Ranks must be known for compilation to take place. Therefore rank information must exist for each node, and if given as a compile-time variable (operand property), the variable must be defined (operand defined). If, after propagation of the initial information listed above, there exists a node with no rank prediction, a new compile-time variable is created to represent the rank of the highest such node and that information is than propagated and the process repeated until all such nodes have an (undefined) compile-time variable representing rank associated with them. Since the propagation process never removes information from a node, the above will terminate.

The lowest occurrence in the parse tree of an undefined compile-time variable representing the rank of a node indicates when, in the computation, the information needed to fix the rank will be available. The most common situation is for the variable to represent the rank of an operand. Otherwise, the variable will represent a value or length of a position at or below the left argument of a transpose or reshape operation which has caused rank variability. If the compilation of the entire function has been requested by the user, a declaration will be requested for each rank variable which does not have a value. When the compilation is taking place at first execution, the point at which the variable will receive a value must be at a leaf (unit boundary). If this is not initially true, the unit must be subdivided. Execution of the first sub-division of the the unit thus generated will produce the information needed for compilation of the dependent units.

The lowest appearance of a defined rank variable indicates locations where rank checking must be performed. If they apply to intermediate results, stream generator interruption will be required.

2.2.4.2 Type - Types must be known for compilation to take place. All nodes will have a type prediction after the propagation of initial predictions. All compile-time variables appearing in these predictions must have values before compilation can take place. If the compilation of the entire function has been requested by the user, a declaration will be requested for each undefined compile-time variable representing a node type. When the compilation takes place at first execution, there never will be any remaining type variability.

The appearance of defined type prediction variables indicates locations where type checking is required.

2.2.4.3 Length - Length values take three forms - actual length, minimum length, and maximum length. Every node must have a maximum length defined to permit storage allocation. The only operation which will not always propagate a maximum length prediction upwards is Reshape. A reshape which does not have a maximum length prediction will be interpreted.

Every node must also have an actual length prediction to permit conformability checking. If after propagation of the initial

information there exists a node with no entry representing its actual length, a new compile-time variable is created to represent the length of the highest such node. This information is then propagated and the process repeated until all nodes have a length prediction.

Undefined compile-time variables representing lengths do not prohibit compilation but give the location for conformability checking or parameter calculation, and if not at a leaf, force an interruption of stream generator execution. Since all AFL operators generate rectangular structures, only one unit of the dimension of unknown length must be tested for length conformability. The stream generator can then run uninterrupted and the test is executed only once. Defined variables representing length locate requirements for length checking.

The length tests imposed for constraint verification also permit the interpreter to detect null arrays. Since the loop control of the stream generators tests after execution of the body, the loops will always execute once. Therefore when the interpreter detects a null array it aborts the execution of that section of the stream generator and performs the calculation directly.

2.2.4.4 Value - Value information comes from constants, scalar operands, and the operations which convert a predicted length (ρ) or rank ($\rho\rho$) into a value. The information is needed for compilation or parameter generation when:

1. A variable representing rank or maximum length gets its value (lowest occurrence) from the value of a node. These are handled as described in the sections for those properties (2.2.4.1 and 2.2.4.3).

2. Dyadic take, drop, transposition, or reshape appear (left operand).

Operator domain, index, or conformability requirements are checked by code compiled into the stream generator.

Compile-time variables which represent the value of a scalar operand or the actual length of an operand are never assigned fixed values based on the those characteristics of the operands, unless required to define a rank prediction. However, the actual values may be used to test relations between expressions involving compile-time variables. An example is the expression $(N,M)\rho A$ where N and M are scalars and A is a matrix with predicted lengths X and Y. If at first execution $N\times M$ equals $X\times Y$, then the reshape may be compiled as requiring no duplication. The equality must be tested before each execution.

As the blocks are further subdivided into units, temporary storage arrays will be created to hold values which are calculated in one unit and used in another. New nodes will be added to the parse tree at the point of division to represent the assignment and reference. The new variables are operands to the units referencing them and may be assigned predictions. If requirements imposed by the same node cause subdivision at two different places in the tree, only the highest is actually done

(the other requirement is assumed not to propagate past that point).

Susan Gerhart [9] has designed a system which determines the properties the operands of an APL function must have for the function to execute. Syntactic constraints are generated and propagated in a manner similar to that described above. However, she makes no attempt to develop information needed to select between alternate legal interpretations of the function. Nor does she locate those places at which such information will later be available (undefined compile-time variable).

Our attempt to advance binding times is similar philosophically to the work of Jones and Muchnick [14]. However, their technique and that proposed by Kaplan and Ullman [15] are oriented to determining properties which hold at entry to simple statements. APL requires intra-statement analysis. They also do not handle information to be available in the future or the inter-dependence of different properties (such as a rank depending on a value). A detailed description of the constraint propagation algorithm and the characteristics of the APL operators appears in Appendix B.

2.2.5 Idioms

One goal of this compiler design is to process the language APL using one consistent procedure. However, it has become apparent that there exist a small set of combinations of operators and operands which have a much more efficient implementation then that produced by translating each operator separately. A common characteristic of these patterns is the occurrence of the same operand on both the left and right of an operator or group of operators. These patterns will be recognized and replaced in the parse tree by a unique new internal operator. An example is *V⍳⌊/V* which will require two passes over V if translated directly, but can be easily implemented using just one. A list of all such "idioms" currently recognized is in Appendix A

Idioms are recognized by applying a pattern matching procedure to the parse tree. Each node of the tree is visited. If it could be the root of a sub-tree headed by one of the idioms, its immediate descendants (maximum number 4) are examined to determine if they match the idiom. The nodes which are operands in the idiom description will match any node which is predicted to have the rank or constant value required by the idiom. The pattern matcher will never have to look lower in the parse tree.

Some of the idioms require that the same value be used in two places in the expression. These will only be recognized if the corresponding nodes in the parse tree are leaf nodes referencing the same variable. No common sub-expression recognition will be done by the idiom recognizer.

When an idiom is recognized, the pattern is locally contracted into a single internal operator. In the case of multiple references to the same variable, only one will be retained. Since both the search for idioms and the transformation of them requires access to a small (<4) number of nodes for each possible idiom, the entire process requires a time which is linear in the size of the parse tree.

2.2.6 Operator Conversion

The parse tree of the APL function has now been split into a forest of parse trees for units each of which will be compiled separately. Based on information obtained by constraint propagation, operations in the parse tree will be modified to distinguish special cases.

2.2.6.1 Take Or Drop - If the left argument of take or drop is known to be a constant, the operation becomes monadic with the former left operand as a modifier.

2.2.6.2 Subscription - Subscription will be expanded into a node for each dimension of the subscripted array. The left operand of each node will be the subscript for that dimension. The right operand for the lowest (last dimension) will be the subscripted array and the remaining nodes will use successive results as their right operand. If a subscript is null, the node is removed from the parse tree. If a subscript is known to have a constant value of the form *A+B×⍳C*, the node is converted to a monadic operator with the value as modifier. If the subscript is of that form, but not all of the scalars A, B, and C are known to be constant, the + and ⍳ operations are replaced by the successor operator in forming the left operand.

2.2.6.3 Transposition - If the left operand is known to be a constant, the operation becomes monadic with the left operand value as a modifier.

2.2.6.4 Ravel - The ravel operation (monadic ,) will be modified with the dimensions that it affects. The special case of a scalar right operand will be converted to a unique internal operator if the scalar is an intermediate result, otherwise the operation becomes a simple reference to a vector (which will be created by the interpreter).

2.2.6.5 Reshape - The information known about the length of the right operand will be compared to the information known about the value of the left operand to recognize 3 special cases.

1. If the reshape is the duplication of the right operand in a new first dimension, and if the duplication factor is known to be a constant, the node becomes the monadic duplicate operator with the duplication factor as a modifier.

2. If the reshape is the duplication of the right operand in a new first dimension, and if the duplication factor is variable, the node becomes the dyadic duplicate operator with the scalar duplication factor as left operand.

3. If the reshape is a partial ravel of the right operand, the node becomes a ravel operation appropriately modified.

4. If none of the above special cases can be recognized, if it is known that result has the same number of elements as the right operand, and if the shape of the result is known, the node becomes a monadic operator with the result shape as a modifier. An example is $(((\rho C)\div 2),2)\rho C$. the number of elements in the result which is $((\rho C)\div 2)\times 2$ equals ρC which is the number of elements in C (when ρC is even).

A combination of 1, 2, and 3 will be split into separate nodes. If one of these special cases can not be identified, the reshape is split out into a separate unit (storage added as needed) which will be interpreted. Except for cases 1 and 3, a reshape with both operands constants will be done at compile time.

2.2.6.6 Single Dimension Operators - All operations which apply to a single dimension are modified with the dimension. The parser will have absorbed explicit dimension indicators such as in +/[2] into the node for the operation. Implicit dimension designations which depend on operand rank (i.e. last dimension) are now filled in.

2.2.6.7 Lamination - If the dimension modifier for dyadic "," has a non-integer value indicating lamination, the operation is split out into a separate unit which will be interpreted.

2.2.6.8 Functionals - (Scan, Reduction, Inner Product, and Outer Product) The scalar operations associated with these operations are modifiers to the node.

2.2.6.9 Scalar Conversion - (the new operation - monadic ι) If a vector operand is not an intermediate result, this operation becomes a simple reference to a scalar variable.

2.2.7 Data Dependency

The compiler considers user specified array variables which are active within only one compiled unit to be temporary storage. This allows the elimination of storage added to improve readability of the code or because of the users recognition of a common sub-expression. If an array variable is referenced in any unit without an assignment having appeared earlier in that unit, is a global variable, or is potentially accessible to called functions, that variable will be considered as global to the compiled unit and will always actually exist in memory.

Conventional control-flow analysis [2] could be done to determine when variables are active, but that is not required. Other standard program transformations such as common sub-expression elimination and moving invariants out of loops could be done at this time, but are not included in this design.

2.2.8 Stream Generator Creation

All of the above serve as preliminaries to the real work of the compiler - the making explicit of the control structure implied by the array operations of APL. This is accomplished by translating APL into what I call stream generators. The actions required to execute the majority of the APL operators can be expressed in the stream generator notation. Chapters 3 and 4 contain a complete description of stream generators and the translation process outlined below.

Each tree of the parse forest which is to be compiled is translated separately at the first time that all requirements can be evaluated. The stream generators are comprised of sets of nested loops connected as co-routines. They are generated by traversing the parse tree in right-left-root order. For an array leaf the code to access the array

is generated. At an operator node the stream generators for its operand(s) are combined, and code to calculate the result is inserted into the control structure. The way the generators are combined depends on the operation. The monadic selection operators can often be absorbed into the array access parameters unless some previous calculation can not be reordered. At any point where temporary storage may be required for the correct or efficient execution of the APL the required assignments are generated.

2.2.9 Stream Generator Refinement

The stream generators created initially may compute values which are never used. These calculations must be eliminated. The next step is to eliminate all unnecessary storage. This will be done by re-ordering independent calculations so that as soon as an intermediate result is available, it is consumed. When this is possible both the consumer and producer share the same control structure and only a single scalar item of the intermediate result will exist at any one time.

The use of the same address generation mechanism to process several arrays in parallel requires that in addition to having the same number of elements, they have the same shape. In order for this restriction to apply to the right and left operands of all assignment operations, shape conformability will be imposed on assignment to a sub-array. The result of such an assignment will be the right operand. The assignments of values to each element of a sub-array are considered to be independent and may be re-ordered. As a result, if the same position in the array is selected more than once, the result is not fully defined (it will be one of the values assigned to that position - no error). However, the use of the same control structure for array access on both sides of the assignment operator means that all the selection operations may be used to select the target sub-array (ex. (1 1⍉*A*)←1 sets the diagonal of A to 1).

If a variable, whose assignment and reference are thus synchronized, is a temporary created by the compiler or a user specified variable which has been identified (see Section 2.2.7) as being temporary storage for the unit being compiled, the assignment (and the variable) may be eliminated. It is this elimination which provides the major benefit of using this compiler.

The re-ordering of operations described above takes advantage of the fact that the definition of APL specifies right association but does not fix order of execution. This compiler will not guarantee right to left execution order. In particular, absurdities such as *X*[*X*←1 2] are undefined. Since the compiler can combine lines and eliminate storage used only to hold values between lines, more legitimate uses of execution order such as (*A*≠0)/*A*÷*B*÷*C* will execute efficiently when written on two lines. To permit the above and similar expressions to be handled efficiently in calculator mode, we will use a successor operator ◊ to combine logical lines into one input line. The two lines are executed in right to left order. This is the only operator which imposes right to left order of evaluation on its operands.

2.2.10 Interpreter Instructions

Associated with each stream generator is a set of instructions to the interpreter specifying the actions necessary to verify constraints, calculate array access parameters, and transfer information to and from the interpreter symbol table. These instructions combine with the parts of the APL function which are not compiled to produce a new function which is executed by the interpreter. The stream generators are co-routines whose execution is interleaved with the interpreted part of the function. Re-compilation of a unit, because of binding failure, will change part of the code about to be interpreted in addition to the stream generator (indeed the test which just failed will be changed to succeed with the new binding).

CHAPTER 3

STREAM GENERATORS - A MODEL FOR THE EXECUTION OF APL

Chapter 2 considered the question of how and when to bind information so as to permit compilation of an APL function. Now we look at the problem of efficiently executing the array expressions of APL. This chapter describes the syntax and semantics of the intermediate representation into which the APL expressions are translated. The compiler algorithms described in Chapter 4 are stated in terms of how they manipulate this representation.

3.1 ARRAY OPERATION EFFICIENCY

The efficient execution of a scalar oriented language such as FORTRAN or Algol 60 requires the elimination of unnecessary calculations and control overhead. Some common transformations are:

1. elimination of repeated calculation of the same value (common sub-expression and loop invariants).

2. elimination of calculation of unused values (dead variables).

3. reduction of control overhead (loop jamming).

As a result of these transformations, the value of expressions not assigned to variables must be retained for varying periods. The same situation arises with intermediate results of expression evaluation and control parameters. This requires careful allocation of machine registers in order to minimize memory access. In doing the above, the compiler will take advantage of the close correspondence between the operators and operands of the language and those of the machine. The operations compute only scalars, and intermediate values may thus be held in a machine register.

However when APL is executed on a machine having only scalar operations (the environment to which the work of this thesis is applicable), the problem is more complicated because:

1. Intermediate results may be arrays which can not be kept in machine registers. If temporary storage in memory is to be avoided, calculations must be re-ordered so that each scalar item is consumed as soon as it is produced. The re-ordering depends on the fact that for many APL operators the computations using array components are independent of each other. Operations for which that is not true (Ex. ⌽\ ) may prevent the necessary re-ordering and force the use of temporary storage.

2. The expressions generate complex control patterns when array operations are mapped into machine instructions acting on single

items. Overhead may be reduced by combining two calculations which run in parallel. Analysis of control patterns implied by array operations is simplified by the connection between the control and operand shape and operator semantics. Less information is available when trying to transform user specified control patterns.

3. Because of selection operators only part of an intermediate result may be needed. Thus the partial execution of operators must be possible.

The more common transformations are applicable (some before and some after translation into scalar code) but will not be discussed further here since known algorithms apply. The examples described below show some of the transformations unique to APL.

3.1.1 Dragging And Beating

An example of the need for operator interleaving and partial execution of operators is the simple APL expression *A←5 5↑B+C+D* where B, C, and D are matrices of size 10 by 10. An Algol program which performs this calculation is shown below:

```
FOR I:=1 STEP 1 UNTIL 10 DO FOR J:=1 STEP 1 UNTIL 10 DO
                            T1[I;J]:=C[I;J]+D[I;J];
FOR I:=1 STEP 1 UNTIL 10 DO FOR J:=1 STEP 1 UNTIL 10 DO
                            T2[I;J]:=B[I;J]+T1[I;J];.
FOR I:=1 STEP 1 UNTIL 5 DO  FOR J:=1 STEP 1 UNTIL 5 DO
                            A[I;J]:=T2[I;J];
```

This program represents the standard way of executing APL, which is to do each operation separately, storing all intermediate results. The

program consists of three sets of nested loops, does 200 additions, 425 loads, 225 stores, and uses at least 100 words of temporary storage (200 if T2 is not the same storage as T1). The temporary storage is clearly unnecessary as can be seen in an improved Algol version of the same expression:

```
FOR I:=1 STEP 1 UNTIL 5 DO FOR J:=1 STEP 1 UNTIL 5 DO
                                   A[I;J]:=B[I;J]+(C[I;J]+D[I;J]);
```

which reflects the transformations Abrams [1] called "beating and dragging", and has only one set of nested loops, does 25 additions, 75 loads, and 25 stores, and uses no temporary storage. The 2 operations have been interleaved but the items of each operand are accessed in the same order. Also the actual additions for each element are not re-ordered. The result is the same even if the additions are non-associative floating point operations.

3.1.2 Operator Transposition

In other cases the items must be processed in different order. For example, the standard execution of *S←×/+/[1]A÷B* with A and B being 10 by 10 matrices is represented by:

```
FOR I:=1 STEP 1 UNTIL 10 DO FOR J:=1 STEP 1 UNTIL 10 DO
                                T1[I;J]:=A[I;J]/B[I;J];
FOR J:=1 STEP 1 UNTIL 10 DO T2[J]:=0;
FOR I:=10 STEP -1 UNTIL 1 DO FOR J:=1 STEP 1 UNTIL 10 DO
                                T2[J]:=T1[I;J]+T2[J];
S:=1
FOR J:=10 STEP -1 UNTIL 1 DO S:=T2[J]*S
```

This code has 6 loops (2 nested), does 210 arithmetic operations, 410 loads: and 211 stores (S is kept in a register), and uses 110 words of temporary storage. Changing the order of calculation so that intermediate values may be used as soon as produced yields:

```
S:=1;
FOR J:=10 STEP -1 UNTIL 1 DO
   BEGIN T:=0;
         FOR I:=10 STEP -1 UNTIL 1 DO T:=(A[I;J]/B[I;J])+T;
         S:=T*S
   END
```

which has 2 loops, does 210 arithmetic operation, 200 loads, and 1 store, and uses no temporary storage (T will be a register). It takes advantage of the fact that the calculations for each element of *A÷B* are independent. The correctness of this transformation which may be applied at the APL level to yield *S←×/+/⍉A÷B* was proved by Abrams.

3.1.3 Filtering

For a number of APL operators the above simple transformations are not sufficient to produce reasonable execution. An example is the expression *B←(∨/A)/[1]E∨A ◇ A←C∧D* where C, D, and E are 10 by 10 boolean matrices. This expression removes a row of *E∨A* if that row of A is all zero. An Algol program for the standard execution of this expression is:


```
FOR I:=1 STEP 1 UNTIL 10 DO FOR J:=1 STEP 1 UNTIL 10 DO
                                       A[I;J]:=C[I;J] AND D[I;J];
FOR I:=1 STEP 1 UNTIL 10 DO FOR J:=1 STEP 1 UNTIL 10 DO
                                       T1[I;J]:=E[I;J] OR A[I;J];
FOR I:=1 STEP 1 UNTIL 10 DO
   BEGIN T2[I]:=FALSE;
         J:=10 STEP -1 UNTIL 1 DO T2[I]:=A[I;J] OR T2[I]
   END;
K:=0
FOR I:=1 STEP 1 UNTIL 10 DO
   BEGIN IF T2[I] THEN BEGIN K:=K+1;
                                      FOR J:=1 STEP 1 UNTIL 10 DO
                                         B[K;J]:=T1[I;J]
                              END
   END
```

which consists of 4 sets of nested loops, does 300 logical operations, between 510 and 610 loads (depending on number of rows preserved), and between 210 and 310 stores, and uses 110 words of temporary storage, and makes two complete passes over A. A more efficient execution of the APL can be obtained using the following program:

```
K:=0
FOR I:=1 STEP 1 UNTIL 10 DO
   BEGIN T:=FALSE;
         FOR J:=10 STEP -1 UNTIL 1 DO
            BEGIN A[I;J]:=C[I;J] AND D[I;J];
                  T:=A[I;J] OR T
            END;
         IF T THEN BEGIN K:=K+1;
                               FOR J:=1 STEP 1 UNTIL 10 DO
                                  B[K;J]:=E[I;J] OR A[I;J]
                         END
   END
```

which has only one loop at the outer level, does between 200 and 300 logical operations, between 200 and 400 loads and between 100 and 200 stores, uses no temporary storage (T is a register), and makes two passes over each row of A in succession. The change in order of access to A is a significant transformation. A common occurrence in APL functions is the generation of a large array, followed by an expression such as the example which filters out selected components of the original array based on their values. Further processing then uses only the surviving components. Thus the variable A will be referenced only in the given filter expression. The first implementation which does two complete passes over A would require all of A to be in storage. The second which uses A a row at a time would require only a row of A to exist at any one time saving 90 words of storage in this example. (A[I;J] would become T2[J].)

3.1.4 Merging

APL also has operations which select between two data sources instead of filtering one. An example is the expression $S\leftarrow+/+/B,C,[1]D$ where B is a 10 by 5 matrix and C and D are 5 by 5 matrices. The conventional execution is given by:

```
FOR I:=1 STEP 1 UNTIL 5 DO FOR J:=1 STEP 1 UNTIL 5 DO
                              T1[I;J]:=C[I;J];
FOR I:=1 STEP 1 UNTIL 5 DO FOR J:=1 STEP 1 UNTIL 5 DO
                              T1[I+5;J]:=D[I;J];
FOR I:=1 STEP 1 UNTIL 10 DO
   BEGIN FOR J:=1 STEP 1 UNTIL 5 DO T2[I;J]:=B[I;J];
         FOR J:=1 STEP 1 UNTIL 5 DO T2[I;J+5]:=T1[I;J]
   END
FOR I:=1 STEP 1 UNTIL 10 DO
   BEGIN T3[I]:=0;
         FOR J:=10 STEP -1 UNTIL 1 DO
           T3[I]:=T2[I;J]+T3[I]
   END
S:=0;
FOR I:=10 STEP -1 UNTIL 1 DO S:=T3[I]+S
```

which uses 10 loops, does 260 loads and 161 stores, and uses 160 words of temporary storage. T1 (50 words of storage, 50 loads, and 50 stores) may be eliminated by using half of T2 as T1. However to eliminate T2

and T3 the program must be transformed to:

```
S:=0;
FOR I:=5 STEP -1 UNTIL 1 DO
   BEGIN T:=0;
         FOR J:=5 STEP -1 UNTIL 1 DO T:=D[I;J]+T;
         FOR J:=5 STEP -1 UNTIL 1 DO T:=B[I+5;J]+T;
         S:=T+S
   END;
FOR I:=5 STEP -1 UNTIL 1 DO
   BEGIN T:=0;
         FOR J:=5 STEP -1 UNTIL 1 DO T:=C[I;J]+T;
         FOR J:=5 STEP -1 UNTIL 1 DO T:=B[I;J]+T;
         S:=T+S
   END;
```

which has 6 loops, does 100 loads and 1 store, and uses no temporary storage. The loops calculate a function between position in the result of catenation and position in the input.

3.2 ARRAY ACCESS AND LADDERS

From the number of occurrences of subscripted variables in the examples above it is clear that an important part of the execution of an APL expression is the generation of the addresses of elements of an array. In developing an address generation algorithm we take advantage of the fact that the sequence of array positions for which addresses are needed is often independent of calculated values (as true in the examples). Index origin 0 is assumed for all the equations of this section.

3.2.1 Array Storage

Following the suggestion of Minter [17] we store array elements so that the function mapping subscript positions into addresses uses only arithmetic operations, and in particular we want certain sequences of addresses to require only the fast arithmetic operation addition. The expression for the address of an array element (PI) given the subscripts is:

```
PI←BETA++/I×G                                  (3-1)
```

where BETA is the address of the element with all subscripts equal to zero, I is the vector of subscripts, and G is a vector of constants which depend on the size of the array. Given an array there are several possible storage orders for which it is possible to assign a G satisfying the above expression. However, APL defines a linear order on the elements of an array. This is ravel order or row-major order (right-most subscript changing most rapidly). The position in ravel order of an element with subscript vector I is given by:

```
RHO⊥I                                          (3-2)
```

where RHO is the vector of dimensions of the array. (This is known as the odometer function.) The ravel operation will not require copying, and sequencing through an array in ravel order will be simplified if there exists a scalar GR such that:

```
(BETA++/I×G)=BETA+(RHO⊥I)×GR                   (3-3)
```

is true (equal 1) for all I which satisfy:


```
∧/(I≥0),I<RHO                                  (3-4)
```

(all legal subscripts). When this is true the same address generation mechanism can be used to access the elements as an array or as a vector. We show in Appendix C that if:

```
G←GR××\1,⌽1↓RHO                                (3-5)
```

then equation (3-3) will be satisfied. If GR is the number of addressable units per data word this will cause the array to be stored in ravel order in consecutive locations.

3.2.2 Ladders

The "ladder" is an algorithm developed by Perlis [PER] which generates addresses of successive elements of an array in ravel order. The control structure developed to represent this algorithm provides a framework for the execution of APL. In this thesis we use the term "ladder" to refer to those components of the intermediate or final representation of the compiled program which have that structure. We will define ladders by giving rules for writing an Algol program which represents the address generation algorithm. The ladder will consist of n+1 purely nested loops where n is the number of dimensions of the array. The program will be built up from program fragments of the following form:

fragment id	text
A	L[0]: PI := BETA
B[i] i in (1....n)	I[i] := 0; L[i]:
C[i] i in (1....n)	I[i] := I[i] + 1; IF I[i] < RHO[i] THEN BEGIN PI := PI + DELTA[i]; GOTO L[i] END
D	GOTO L[0]

where PI, BETA, I[1:n], and RHO[1:n] are the quantities defined in the previous section and DELTA[1:n] holds the values used to increment PI. The Algol program representing a ladder of depth n is given by:

```
A;
*;
B[1] *;
.....
B[n-1] *;
B[n] $;
C[n]; *;
.....
C[1]; *;
D
```

where '*' represents a location in the program at which additional computational statements may be inserted, and '$' is a '*' at which PI contains the address of the array element whose subscript position is I.

For n=3 the program is:


```
L[0]: PI := BETA; *;
    I[1] := 0;
   L[1]: *;
        I[2] := 0;
       L[2]: *;
            I[3] := 0;
           L[3]: $;
                I[3] := I[3] + 1;
                IF I[3] < RHO[3] THEN
                  BEGIN PI := PI + DELTA[3]; GOTO L[3] END;
            *;
            I[2] := I[2] + 1;
            IF I[2] < RHO[2] THEN
              BEGIN PI := PI + DELTA[2]; GOTO L[2] END;
        *;
        I[1] := I[1] + 1;
        IF I[1] < RHO[1] THEN
          BEGIN PI := PI + DELTA[1]; GOTO L[1] END;
    *;
    GOTO L[0]
```

Figure 3-1 is a flowchart of the minimum required actions of this program (called the "fixed part" of the ladder).

The boxes of the flowchart have been labeled with the identifiers of the program fragments from which they were derived. That flowchart clearly shows the origin of the name ladder for this structure. We consider the ladder to consist of n+1 "rungs" consisting of the 0^{th} rung A and n rungs formed by B[i] and C[i] for i in (1....n). When this program is executed, I will take on all legal subscript values in odometer order. At each transition a single element from DELTA is added to PI. We show in Appendix C that if the array is stored in ravel order, then:

$$\wedge/GR=DELTA \tag{3-6}$$

holds (adding GR (1 if word addressing) will always produce the address of one data element from the address of the previous data element). The address sequence is not generated by a single loop, since computation of certain operators such as reduction depends on the array

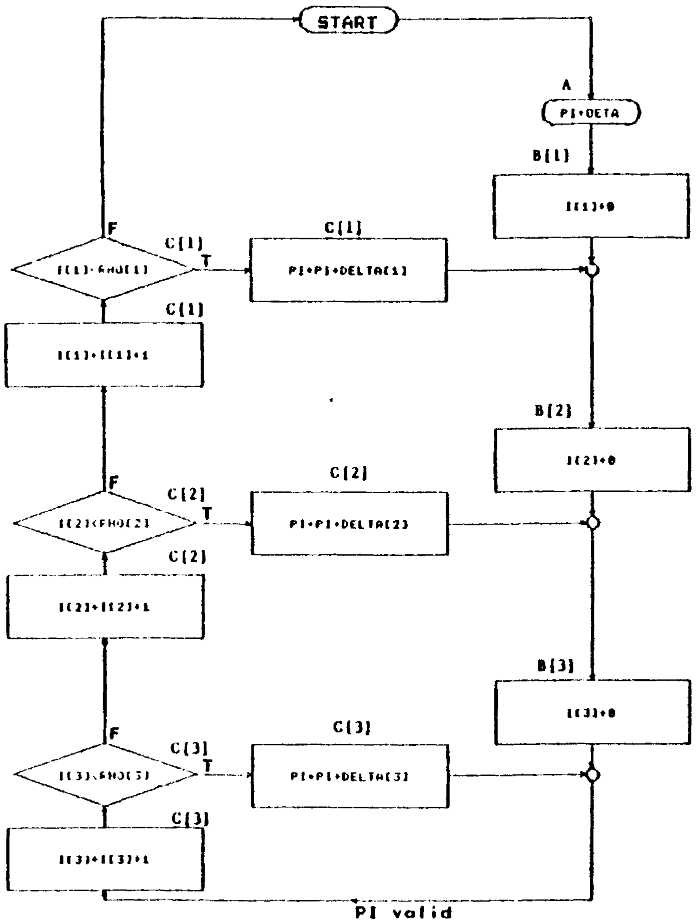


Figure 3-1

structure.

This is a special case of the result derived in Appendix C that for any array whose storage is defined by equation (3-1) there exists a DELTA which will allow a ladder to access that array in ravel order. In Appendix C it is shown that the application of certain common APL operators to an array which is stored in ravel order yields an array which can also be accessed in ravel order by a ladder, but with different BETA, RHO, and DELTA. Since no data is moved in storage when these operators are applied, and since these operators sometimes change the ordering of the data and even the number of data items in the array to which they are applied, the resulting array is not stored in ravel order. The fact that ladders can be used to access these resultant arrays means that the storage orderings which differ from the ravel ordering and which the ladders can handle are commonly occurring ones. The operators, which Abrams called selection operators, are reverse, transpose, take, drop, and certain types of subscription. Other access orders can be generated by directly calculating PI using equation (3-1).

The ladder fixed part defined above generates the sequence of addresses needed to access a single array. However, that definition is not complete as it makes no provision for calculations with the array elements when they become available. Figure 3-2 shows the same ladder as before except that in addition to the fixed part of the ladder shown in Figure 3-1 seven numbered boxes called "splices" have been added. Code to perform scalar calculations may be placed in each box. A splice may be inserted into any edge of the flowchart defining the fixed part of a ladder corresponding to the

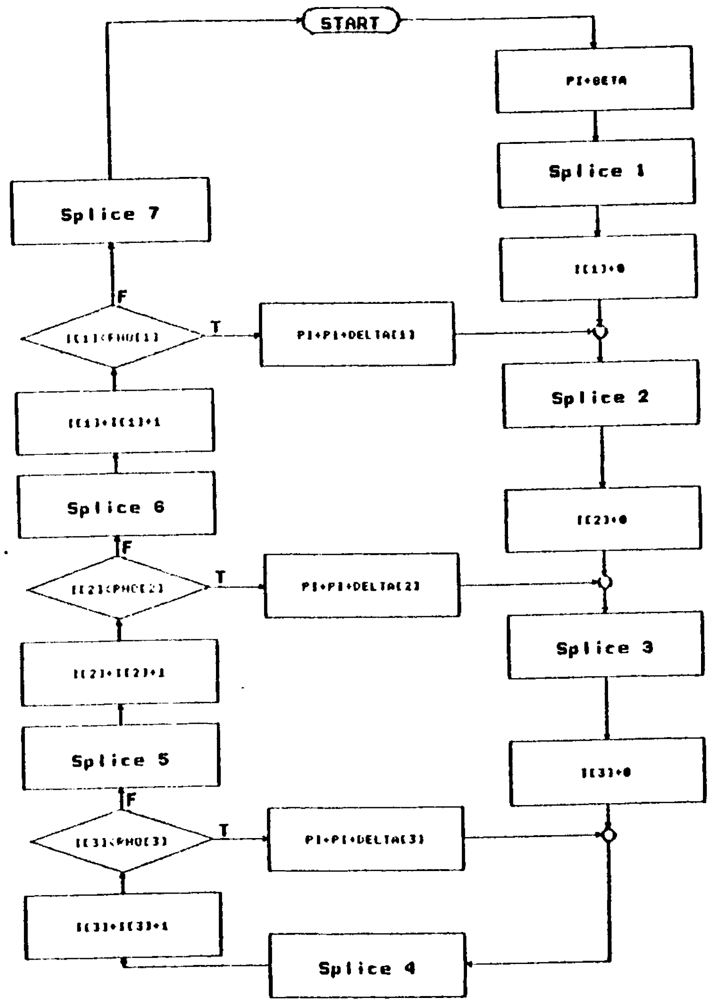


Figure 3-2

location of a "*" or "$" in the Algol program. Figure 3-2 shows all possible splice locations for a ladder of depth 3 (splices numbered by order of first execution). The splice which is in the inner-most loop (Splice 4 or "$") may fetch from or store into the array element pointed to by PI. (PI will have successive values of the address sequence at each execution of splice 4.) All the splices may contain code using the control variables (BETA, PI I, RHO, G, and DELTA) and items from a local memory T.

We have now specified a structure which allows access to and calculation with the elements of a single array. Since almost all APL expressions involve more than one array, the ladder definition also includes a facility for combining several ladders into a larger structure. Each ladder is a co-routine. The control variables (PI, BETA, I, RHO, DELTA, and G) are local to each ladder, and they share a global vector T. Control will pass between ladders as specified by co-routine jumps placed in the splices. The collection of ladders is a "ladder network" which is a co-routine with the interpreter.

The requirements of fixed rank and type present in the constraint propagation phase of the compiler were imposed because the ladder structure depends on the rank of the associated array, and the splice code instructions are dependent on the type of the data. However, the length of the data is reflected only in the values of the control variables. Thus we only require length to be known at execution time when control variables (and T) are initialized by the interpreter.

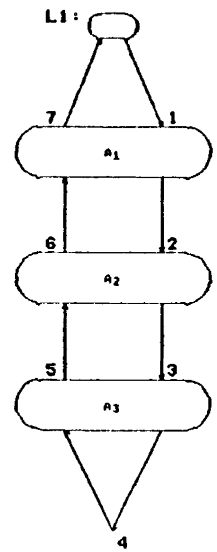
3.3 STREAM GENERATORS

A "stream generator" is a ladder network. However, we have modified the definition of the ladder given by Perlis [PER] (presented in previous section) so as to provide a closer match between the capabilities of the ladder and the requirements for efficient execution of APL seen in the preceding examples. We have borrowed the term stream used by Burge [BUR] since it aptly describes the flow of data right to left through an APL expression. However, the finite, array-based sequences of data items described are very different from those described in [6], and the notation used to describe them is unrelated.

In this section we will progressively modify the original definition for the ladder to arrive at the definition a stream generator. Each change will be motivated by reference to an example. In parallel with the modification of the structure we will introduce new notation for specifying stream generators. The reader should keep in mind that the notation presented here is designed for human processing. It shows the information needed by the compiler algorithms, but not the form that information would take internal to the compiler.

The examples presented above will now be re-done in terms of ladders. In these examples a simpler picture will be used to describe a ladder. The actions of the fixed part which can not be separated (a single rung) are collected into one box in the flowchart. The j^{th} rung will contain the exit test for the loop at level j. Except for the 0^{th} rung the new boxes have in and out-degree 2. These edges are the ladder "rails".

The fixed code is omitted and the address sequence being generated is indicated with the name and dimension of the array upon which RHO and DELTA are based (eg. A_1 stands for the first dimension of A). The labeling will distinguish between assignment and reference (A_1 - reference, A_1← - assignment) and will incorporate the selection operators which affect only address sequencing (eg. a label may be $5\uparrow A_1$ not just A_1). The ladder of Figure 3-2 would be shown as:



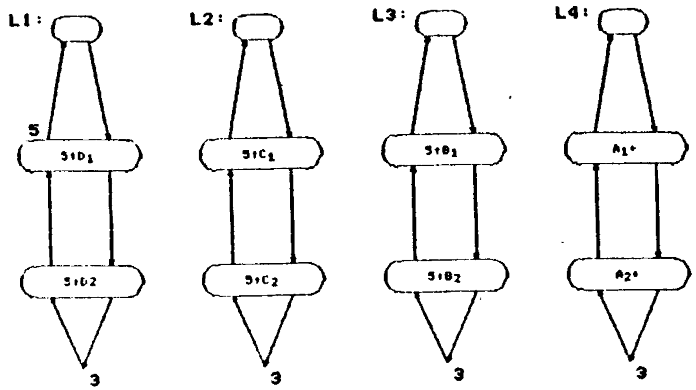
where L1 is the ladder label. The splice code is listed, labeled by splice number and is scalar APL augmented by the function EVOKE L (where L is a ladder label) which does a co-routine jump, and by [PI] which refers to the array element pointed at by PI. Execution starts with the interpreter doing EVOKE L1.

3.3.1 Beating And Dragging

The APL expression *A←5 5↑B+C+D* was translated into the Algol program:

```
FOR I:=1 STEP 1 UNTIL 5 DO FOR J:=1 STEP 1 UNTIL 5 DO
                                   A[I;J]:=B[I;J]+(C[I;J]+D[I;J]);
```

However, the ladder network for this expression is:



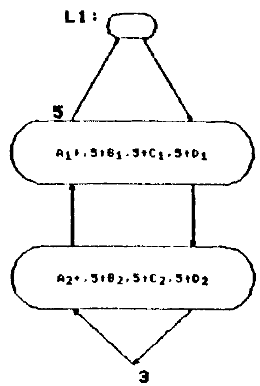
3:π[1]←[*PI*] EVOKE L2 | 3:π[1]←π[1]+[*PI*] EVOKE L3 | 3:π[1]←π[1]×[*PI*] EVOKE L4 | 3:[*PI*]←π[1] EVOKE L1

5:EVOKE Interpreter

which consists of 4 ladders instead of the single loop of the Algol version. The overhead generated by 4 sets of loop control and the co-routining is undesirable. In addition, the perfect synchronization of the access to the 4 arrays is obscured. We thus wish to modify the ladder concept to permit more than one array to be accessed by a single ladder. To accomplish this we make PI and BETA into a vector of pointers and initial values. Each position is associated with an array

(not a ladder) and these variables are global to the entire network. Similarly DELTA and G which were vectors whose length was given by the rank of the array, now become matrices with a row for each item in PI. Only I and RHO (the loop control parameters) remain local to the ladders. In Figure 3-3 we see the fixed part of a ladder structure accessing two arrays.

Using this new facility the ladder network shown above becomes:



3:[*PI*[1]]←[*PI*[2]]+[*PI*[3]]+[*PI*[4]]

5:EVOKE Interpreter

which generates addresses efficiently with the same low control overhead as the Algol version.

3.3.2 Operator Transposition

Given this new feature we can also translate the expression *S←×/+/[1]A↑B* into a single ladder. The change in access order shown in the Algol version:

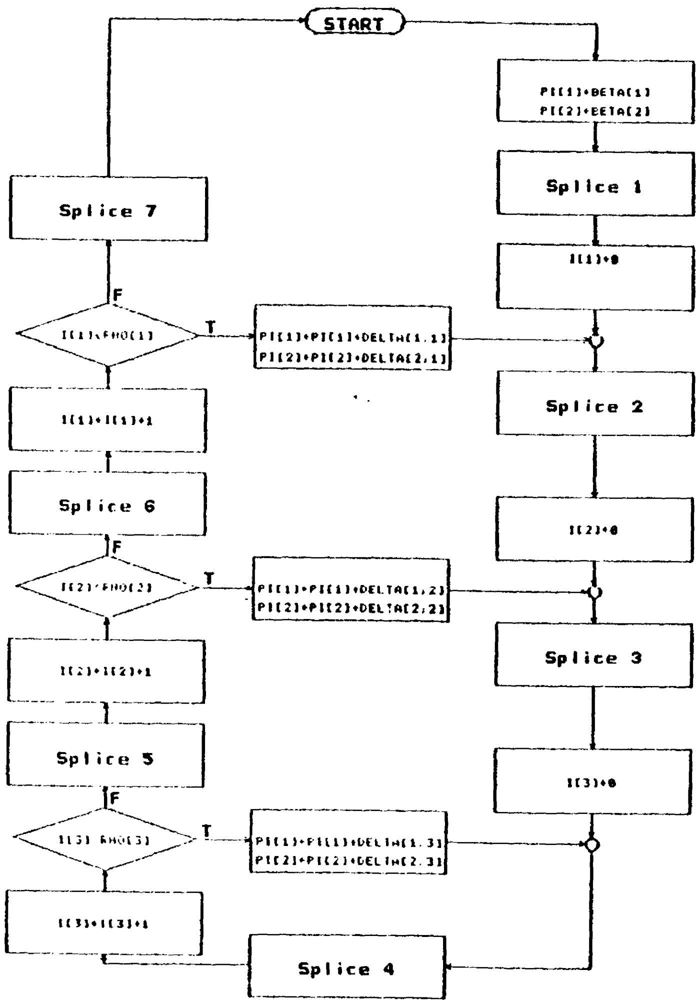
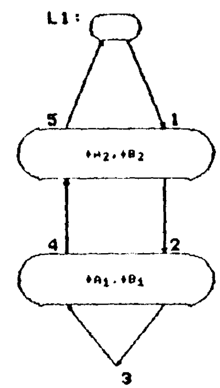


Figure 3-3


```
S:=1;
FOR J:=10 STEP -1 UNTIL 1 DO
   BEGIN T:=0;
         FOR I:=10 STEP -1 UNTIL 1 DO T:=(A[I;J]/B[I;J])+T;
         S:=T*S
   END
```

as reversal of nesting order for the loops controlling the subscripts I and J is reflected in the ladder:



1:T[1]←1

2:T[2]←0

3:T[2]←([PI[1]]÷[PI[2]])+T[2]

4:T[1]←T[2]×T[1]

5:EVOKE Interpreter
  (which stores into S from T[1])

by the inversion of array dimensions shown by the fixed part labels.

3.3.3 Filtering

The Algol code for B←(∨/A)/[1]E∨A ◇ A←C∧D:

```
K:=0
FOR I:=1 STEP 1 UNTIL 10 DO
   BEGIN T:=FALSE;
         FOR J:=10 STEP -1 UNTIL 1 DO
            BEGIN A[I;J]:=C[I;J] AND D[I;J];
                  T:=A[I;J] OR T
            END;
         IF T THEN BEGIN K:=K+1;
                         FOR J:=1 STEP 1 UNTIL 10 DO
                            B[K;J]:=E[I;J] OR A[I;J]
                   END
   END
```

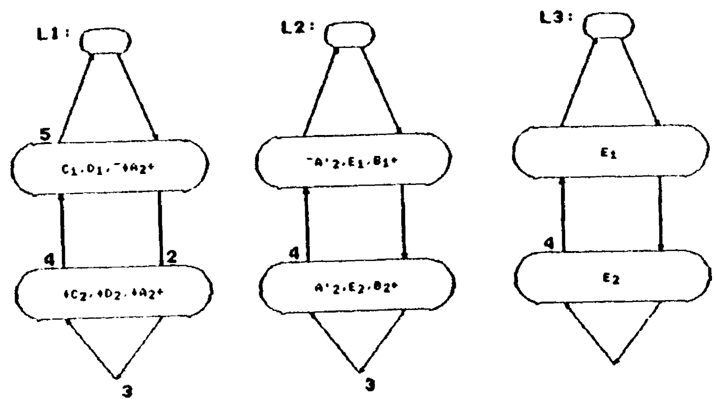
uses one subscript (K) which is not a loop index. Also, the use of variable E occurs only for some values of its index I, the selection being dependent on values of A. In order to avoid having to calculate an address sequence in splice code the co-routine facility will be used to select one of two ladders to execute in each step. The splice code is extended to include an if-then-else construct. The value of the if clause must be in a register, and the alternatives may only contain an EVOKE. Both ladders will sequence the pointer (element of PI) associated with E but only one will actually access the array and sequence the pointer to B. Additional processing necessary to get correct sequencing will be described in the section on stream generators

The variable A occurs 3 times in this expression. Since the assignment is the first access, each references the same values in the same storage, but there is no guarantee that the 3 uses of A will proceeed in synchronization. Therefore we may need 3 different pointers to A. These "aliases" for *A* will be written as *A′* and *A′′*. In this example two pointers do move together and may be combined.

We also eliminate the storage of all but a single row of A which is accessed repeatedly. Thus the ladder rung which would have moved the

pointer to the start of the next row must reset it to the beginning of the single row. This is done in the ladder fixed part by using a different DELTA (calculated from a G of zero), and indicated by a label formed prefixing the label for the dimension reset by "¯" (ex. $^{-}A_1$).

Using these new features the ladder network is:




```
2:T[1]←0

3:T[2]←[PI[1]]∧[PI[2]]   3:[PI[6]]←[PI[5]]∨[PI[4]]
  [PI[3]]←T[2]
  T[1]←T[2]∨T[1]

4:if T[1] then EVOKE L2 4:EVOKE L1                4:EVOKE L1
          else EVOKE L3

5:EVOKE Interpreter
```

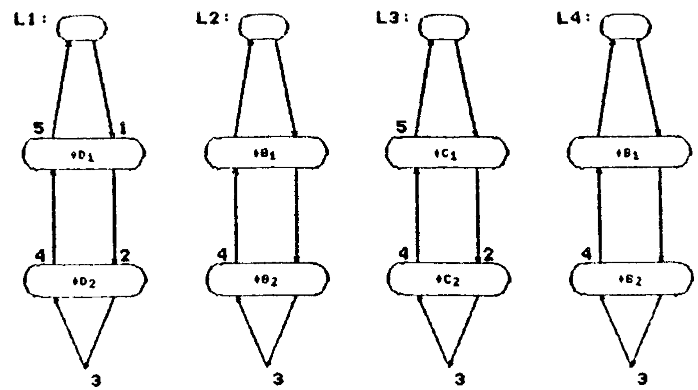
which has the same pattern of access to A as the Algol program. Because this structure moves the same PI in two different ladders at the same level, adjustments must be made to the ladder fixed part. They are described in section 3.4.16.

3.3.4 Merging

The expression $S\leftarrow +/+/B,C,[1]D$ translated into an Algol program:

```
S:=0;
FOR I:=5 STEP -1 UNTIL 1 DO
   BEGIN T:=0;
         FOR J:=5 STEP -1 UNTIL 1 DO T:=D[I;J]+T;
         FOR J:=5 STEP -1 UNTIL 1 DO T:=B[I+5;J]+T;
         S:=T+S
   END;
FOR I:=5 STEP -1 UNTIL 1 DO
   BEGIN T:=0;
         FOR J:=5 STEP -1 UNTIL 1 DO T:=C[I;J]+T;
         FOR J:=5 STEP -1 UNTIL 1 DO T:=B[I;J]+T;
         S:=T+S
   END;
```

which does not have simply nested loops. Thus the ladder network for this expression is one:



L1	L2	L3	L4
1: $T[1]\leftarrow 0$			
2: $T[2]\leftarrow 0$		2: $T[2]\leftarrow 0$	
3: $T[2]\leftarrow(PI[1]) + T[2]$	3: $T[2]\leftarrow(PI[2]) + T[2]$	3: $T[2]\leftarrow(PI[3]) + T[2]$	3: $T[2]\leftarrow(PI[2]) \times T[2]$
4: EVOKE L2	4: $T[1]\leftarrow T[2]+T[1]$ EVOKE L1	4: EVOKE L4	4: $T[1]\leftarrow T[2]+T[1]$ EVOKE L3
5: EVOKE L3		5: EVOKE Interpreter	

which, as in the first example, uses co-routines to synchronize parts of the network driven by the same loops in the Algol program. Since only the upper level is synchronized we can not merge the two ladders as before. We must allow more then one loop to be nested at one level. If the ladder has multiple nesting then the remaining local control variable vectors (I and RHO) must become global matrices. A ladder will use the same number of rows as the maximum number of nodes at a given level. Figure 3-4 shows the fixed part of a ladder using the new I and RHO.

The modified network is:

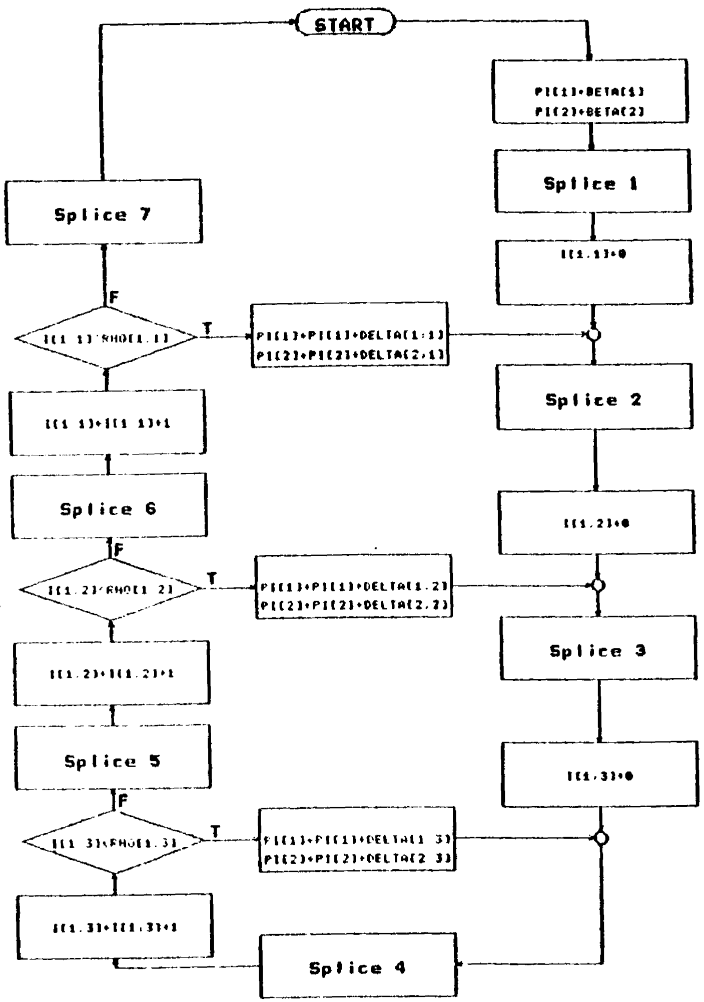
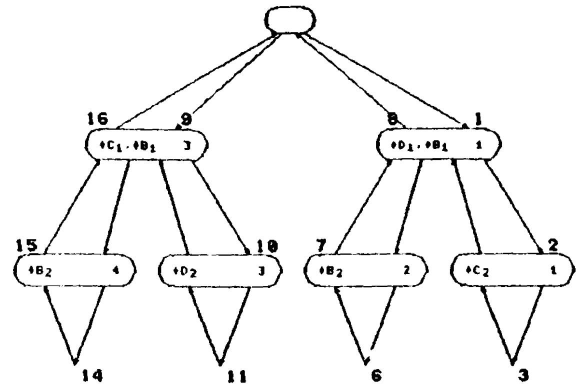


Figure 3-4



1:T[1]←1

2:T[2]←0

3:T[2]←[PI[3]]+T[2]

6:T[2]←[PI[1]]+T[2]

7:T[1]←T[2]+T[1]

10:T[2]←0

11:T[2]←[PI[2]]+T[2]

14:T[2]←[PI[1]]+T[2]

15:T[1]←T[2]+T[1]

16:EVOKE Interpreter

in which each rung is labeled with the row of I and RHO it references. Because this structure moves the same PI in two loops nested at the same level, it will require adjustments to the ladder fixed parts in order to get correct addressing. These will be described in section 3.4.16.

3.4 STREAM GENERATOR GRAPHS

As the power of the ladder has grown, the flow charts needed to describe them have gotten increasingly complex, and the condensed notation is not adequate for completely specifying the ladder fixed part. We will now modify the notation to eliminate redundant information, and at the same time increase the information contained in the diagram of the stream generator so that significant processing can be done without reference to the splice code. As each new feature is described, we will specify how that information can be used to determine the actual ladder fixed part needed.

3.4.1 Loop Nesting

The flow chart draws each loop of the control structure, and in the condensed notation we retained both arcs connecting the body of a loop to the rung containing the exit test for that loop. However, no information is lost if the flow chart is converted into a directed graph by removing all arcs which carry the flow of control out of a loop body (upwards pointing in examples shown). Figure 3-5 shows an example of this conversion.

The nodes of the graph represent ladder rungs. If an edge goes from node a to node b, then the loop controlled by the ladder fixed part defined by node b is nested inside the loop defined by the ladder fixed part defined by node a. Node b is defined to have a nesting level one higher than that of node a. Since the ladder control structure allows only pure nesting, the graph is a tree.

3.4.2 Header Node

The graph node derived from the 0^{th} rung of the flowchart is labeled to distinguish it (drawn smaller) and called the "header node". It will be a root node in the nesting graph (forest if several ladders). These

header nodes will contain a label which gives the nesting level (≥ 0) for the header. Every node now has a fixed level. This level is used as the second subscript for all references to DELTA, G, RHO, and I in the fixed part code defined by the node. Node (1) in Figure 3-5 is the header node.

3.4.3 Raveled Nesting

The ravel operation of APL may reduce two dimensions to one. This is shown in a stream generator graph by labeling the edge connecting the two nodes (drawn ═════ instead of as a single line). (The control structure represented does not change.) Both nodes are considered to be at the same level. The loop limit for a raveled structure is the product of the limits for the nodes. In Figure 3-6 nodes (2) and (3) form a raveled structure.

3.4.4 Splice Order

The stream generator graph does not have a distinct edge associated with each splice and splice numbers are not shown. They may be calculated as follows:

1. Start at the header of ladder L1 and with a current splice number of 0.

2. Traverse the tree in root-right-left-root-order. (All nodes except leaves are visited twice, and all edges are traversed twice - once backwards.)

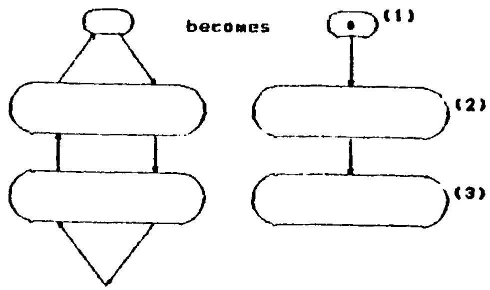


Figure 3-5 - Nesting Graph

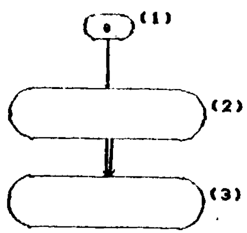


Figure 3-6 - Raveled Nesting

3. When an edge is traversed, the current splice number is incremented and the new value is assigned to that position and direction (of the traversal, not of the edge) of the ladder structure.

4. When a leaf node is reached, the current ladder number is incremented and the new value is assigned to the body of the (innermost) loop which is controled by the fixed part defined by that node.

5. The process is then repeated for each ladder in order.

Figure 3-7 shows the splice order for a ladder of depth 2.

When the graph is not a straight line, the sub-trees are executed in right to left order for each step of the loop defined by the node with multiple sons. For each step (a single execution of the body of the loop) of any node, the list of splices executed inside, with duplicates removed, will be sorted in increasing order.

The ladder structure defines an infinite loop. However the instruction "EVOKE Interpreter" will always occur in the highest numbered splice of some ladder in the network, and the co-routine connections will guarantee that no header node at level 0 executes more than one step.

3.4.5 Co-routine Graph

The ladders in a stream generator are linked together using the EVOKE function. The ladders are co-routines. When a ladder is evoked, its execution is resumed from the point after the last EVOKE it executed (ladders start at the beginning of Splice 1). However, the control dependency relationship is that of non-recursive subroutines. A ladder will always execute an EVOKE of the ladder that evoked it. This instruction will be the final instruction of a splice which is traversed in an upwards direction. (A ladder evoked will execute one step at some level and return.) The compiler will replace a string of successive EVOKEs by one equivalent EVOKE.

The linkage structure may be shown graphically by adding evocation edges to the stream generator graph. They are directed edges leading from the node defining the loop containing the EVOKE to the node defining the loop ended by the return. They will be labeled (drawn as - - - - - - -) to distinguish them from the edges indicating nesting which are drawn as solid lines. We call the source of an evocation edge a "choice" node if more than one such edge leaves it. The nodes entered are called "target" nodes. In Figure 3-8 we see a ladder network in which the splice code for the loop defined by node (1) contains an EVOKE of the ladder containing node (2), and the last instruction of the splice code of the loop defined by node (2) is an EVOKE of the ladder containing node (1).

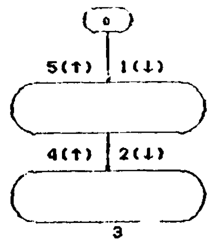


Figure 3-7 - Splice Order

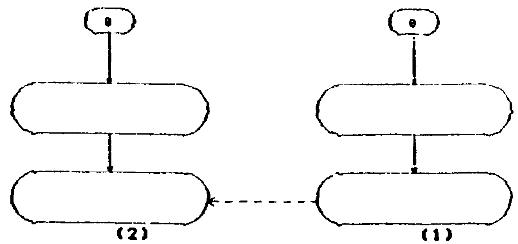


Figure 3-8 - Evocation Graph

3.4.6 Control Structure Sanity

The stream generator description mechanism described above is too powerful in that it will allow the specification of patterns of control which can not actually be executed. To eliminate some of the danger and to give a clearer picture of the way the stream generators will be used, a set of restrictions on the graph structure is given:

1. The header node of ladder L1 which is the starting point of the ladder network must be at level 0. The "EVOKE interpreter" instruction which ends the execution of the network must be the last instruction of the last (highest number) splice. The header node of ladder L1 is called the "entry point" of the stream generator.

2. If the trees defined by header and loop nodes and nesting edges (the ladders of the network) are considered to be single nodes of a super graph, the evocation edges define a directed graph. Since each ladder returns to the ladder that started it by evoking explicitly a particular ladder, a ladder may be evoked from only one place. As a result the graph defined by the evocation edges is a tree. Figure 3-9 shows the graph and super tree for *A*,[1](*B*,[2]*C*).

3. The co-routine facility is used only to synchronize two ladders (only one evocation edge leaves the super node) or to select one of two possible calculations to do in conjunction with that step of the evoking ladder. In that case two evocation edges leave the super node. However the splice code must guarantee that only one is executed for each step of the loop containing the EVOKEs. This will be accomplished by putting them in opposite branches of an

A,[1]*B*,[2]*C*

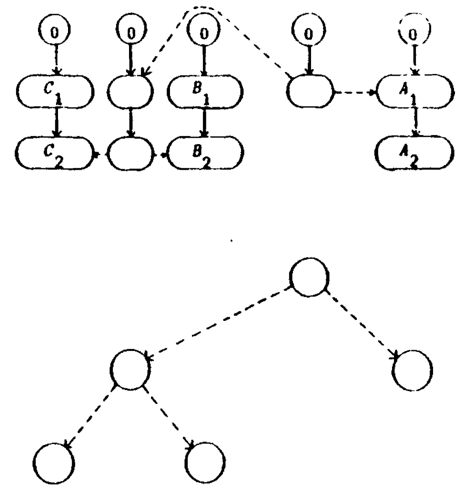


Figure 3-9 - The Super Tree

If-Then-Else statement. (The above also means that the two evocation edges will leave the same loop node in the actual graph.)

4. Except as needed to implement reshape, evocation edges will not connect (synchronize) nodes (loops) having different levels.

5. A "step" of a raveled structure is a step of the inner-most loop. Therefore, since evocation edges synchronize steps, they may not connect to a node from which a raveled nesting edge leaves.

These restrictions are not checked and enforced by any part of the compiler. Rather they served as guidelines for the writing of the actual procedures used to build stream generators (described in Chapter 4). Figure 3-10 shows violations of the last two rules.

In a ladder network meeting these restrictions, it is possible to determine for any given loop X what other loops in the network have been entered but not completed when X is executed. That sequence of nodes called the "control path" leading to X is identified by the following procedure:

1. The sequence is initialized to contain X.

2. If the first node in the sequence is not a header (root) node, then put its father at the beginning of the sequence, and repeat this step.

3. If the first node in the sequence is a header node but not the entry point, put the source of the evocation entering that ladder at the beginning of the sequence, and return to step 2.

This pattern is not allowed

but reshaping *V* into *M* requires

This pattern is not allowed

but ravel will be represented by

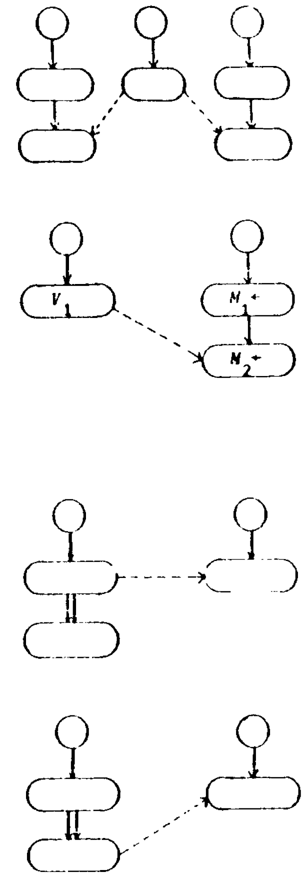


Figure 3-10 - Control Structure Errors

All control paths start at the entry point. We define the "separation point" of two control paths to be the last node common to both sequences. Figure 3-11 shows an example control path.

3.4.7 Loop Indices

The assignment of the row of I to be used in a given ladder fixed part is made by the compiler based on the graph structure. The rules are:

1. The same row of I will not be used in two different ladders (nodes of the super tree).

2. One son of each non-leaf node in a nesting tree will use the same row of I as the father. (This is simply to reduce the number of rows used.)

3. Other nesting sons will use different rows of I than the father.

Since two nodes are in the same ladder if connected by a nesting edge, no conflicts can arise.

3.4.8 Loop Limit

The value used to control the number of times a loop executes is defined by a label on the graph node for that loop and has one of the following forms:

1. ρA_1 - The loop limit is $(\rho A)[I]$.

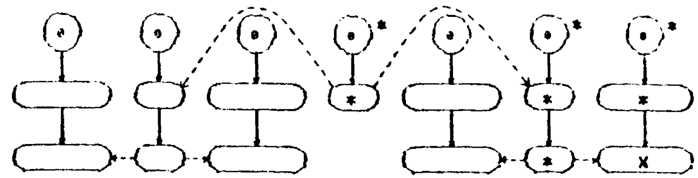


Figure 3-11 - Control Path (*) to Node X

2. eρA_i - The loop limit is the current value of the index of the loop associated with the i^{th} dimension of A in the code for expression e.

3. ριs - the loop limit is given by the scalar value of node s of the parse tree of the APL expression.

4. ρe_i - the loop limit is set from the current value of the index (+1) whenever the node labeled e_i reaches its limit. (This is used for compression. The compressed dimension of the result ends when the compressor does.)

or an expression combining these values using +, − ,⌊ ,⌈ , and scalar constants. The value specified will be stored in the row of RHO matching the row of I selected for that loop.

3.4.9 Array Storage Pointers

A unique element of PI and row of G is associated with each array accessed by a ladder network. The following labels indicate that the ladder fixed part they define increments the PI associated with the variable named, and specify the dimension of the array which defines that entry in G.

1. A_i - The loop advances the PI pointing to A the amount corresponding to incrementing the i^{th} subscript. A is referenced.

2. A_i← - The loop advances the PI pointing to A the amount corresponding to incrementing the i^{th} subscript. A is assigned to.

3. $\bar{A}_i$ - The PI pointing to A is backed up to the first position of the i^{th} dimension (to permit repeated access).

4. ιρA_i - The pointer takes on successive values of ι(ρA)[i]. (If possible the loop index will be used.)

5. ιs - The pointer takes on successive values of ι <result of parse tree node s>.

Since there is a one-to-one association between array names and pointers, the array name (un-subscripted) will replace the notation [PI] in splice code to indicate access to array elements.

When an assignment occurs inside an expression, all references to that variable appearing to the left of the assignment operator are considered to refer to a different variable (may be a different area in memory). A distinct name will be used in stream generator labels. If the same array name appears more than once in an expression, local aliases will be assigned to permit the use of different pointers. An in-line assignment will be assumed to be two occurrences of the array name (one for the assignment, and one for the use of the value). The use of an alias will be indicated in our examples by following the original variable name by "'". The compiler will merge aliases if it detects synchronized access to the same storage.

3.4.10 Storage Spacing

There will be a row of G corresponding to each item of PI. The ordering of the values in a row of G depends on the positions of labels in the stream generator graph.

3.4.11 Address Increments

There will be a row of DELTA corresponding to each item of PI. The value of DELTA is calculated from G, RHO, and the nesting relations given by the graph. The calculation is described in detail in Appendix C.

3.4.12 Address Calculation

In addition to the simple selection operators which may be affected by changes to the BETA, RHO, and G of the ladder fixed part, there are two APL selection operators which must be implemented using splice code to calculate addresses. These are rotation and general subscription. Since we want the address generation process to be completely described by the labels on the nodes of the stream generator graphs, new labels are defined below which indicate that the appropriate address computation occurs at the beginning of the loop defined by the graph node. They are:

1. A_i[e] - the pointer to A is adjusted to point to the element whose i^{th} coordinate is equal to the current value of the expression defined at node e of the parse tree of the APL expression. A is referenced.

2. eϕA_i - the pointer to A is adjusted to account for the rotation of the i^{th} coordinate of A specified by the expression e. A is referenced.

3. A_i[e]← - the pointer to A is adjusted to point to the element whose i^{th} coordinate is equal to the current value of the expression defined at node e of the parse tree of the APL expression. A is assigned to .

4. eϕA_i← - the pointer to A is adjusted to account for the rotation of the i^{th} coordinate of A specified by the expression e. A is assigned to.

Both rotation and subscription employ similar splice code. The previous value of the subscript is kept as an item of T and the multiple of the item of G associated with A_i necessary to move to the new subscript position will be added to PI at each step. The ladder fixed part will have removed the effect of incrementation in lower level loops in a similar fashion to the action indicated by the label $^{-}A_i$.

3.4.13 Special Labels

In order that the graph representation alone, without splice code, be sufficient for analysis needed to eliminate unnecessary temporary storage, labels for two special arrays and a set of functions which modify the meaning of labels are defined:

1. ZERO - represents an array of unspecified size containing the numeric value zero.

2. BLANK - represents an array of unspecified size containing the character value blank. The compiler will not actually generate code to access stored values when ZERO or BLANK are used.

3. e_i - a function which marks its left operand as the source of the i^{th} dimension of the result (subscript is omitted for a scalar) of the expression defined by node e of the parse tree. A node so labeled is a "result node" for e. The nodes on a control path to a result node are called "active".

4. $n\backslash e_i$ - a function with the same meaning as above which also indicates that the calculation depends on n previous values of result. (fixes order)

5. e/ - a function which signals that the labels modified contribute to the expression for e. The node does not produce a dimension of the result, and its loop must be completed before values are available.

6. SKIP - a function that indicates that all computation is omitted from its left operand and only pointer movement is done. Since no values are produced, any result labels are removed from the operand of SKIP

The set of labels of a node will be given as a text string which should be parsed as an APL expression, with the exception that the symbol ',' is used as a list element separator and has a precedence lower than all other operators, including function application.

3.4.14 Address Generation Sanity

The stream generator description method described above is too powerful in that it will allow the specifications of ladders which do not generate valid address sequences. In particular an item of PI associated with a given array is only valid if a loop associated with each dimension of the array has been entered and not exited. To eliminate some of the danger and to give a clearer picture of the way the stream generators will be used, a set of restrictions on the labeling is given: (examples of violations of these restrictions appear in Figure 3-12)

1. Pointers will only be used at the highest level at which they are valid. This means that they will only appear in a splice leading down from, or up to, a fixed part whose graph node is labeled to indicate that that loop changes the pointer. The pointer is valid deeper in the nesting structure but the same value would be fetched repeatedly, so it should be moved into a register at the higher level.

2. Any label indicating pointer movement for reference (A_i,A_i[e], or e⍉A_i) must appear on a control path to a leaf on which must be found one and only one such for each dimension of A. (Aliases generated because of multiple references to the same array in the APL are considered distinct.)

3. Any label indicating pointer movement for assignment ($A_i\leftarrow$, $A_i[e]\leftarrow$, or $e\iota A_i\leftarrow$) must appear on a control path to a leaf on which must be found one and only one such for each dimension of A.

4. Two labels for the same dimension of a given array reference or assignment must be at the same level. Labels for two dimensions may be at the same level only if they are in the same node. The ordering along the control path and the ordering by level must agree.

5. If the same control path to a leaf holds both an assignment and a reference to the same area of storage, the assignment and reference for each dimension must be at the same level. The assignment label for a dimension must not be later along the control path than the reference to that dimension.

6. If the separation of two different control paths which define an assignment and a reference to the same area of storage occur at a node with multiple nesting, the assignment must be in the right branch (executes first). If it occurs at a choice node, the assignment must be in the ladder containing the choice.

7. The address calculation algorithm assumes that there are no repeats in the address sequence. If the control structure specified by the stream generator graph make repeated passes over an array or part of an array, the address pointer must be reset. The reset operation is indicated in the graph by labels of the form $^{-}A_i$. The graph specifies a repetition if the control path to the lowest node associated with the given reference or assignment contains any loop

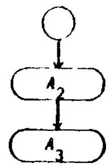
nodes not labeled to be part of that operation. The reset operation must be placed on all the nodes in the gap.

8. If a node contains an address computation label referencing the result of parse tree node e, then a value of the expression must be available at that point. This will be true if the control path to the node has on it labels for each dimension of e.

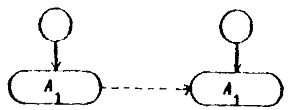
9. The control path leading to a node labeled with $e\rho A_i$ must contain a node labeled with $e_j A_i$.

10. The restrictions of uniqueness and distinct levels which apply to address generation labels also are imposed on the labels for e.

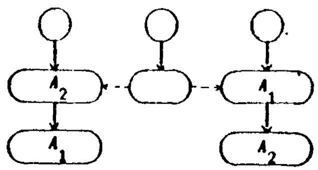
As was the case for the control structure rules, no part of the compiler checks and enforces all of these restrictions. Rather they served as guidelines for the writing of the actual procedures used used to build stream generators (see Chapter 4).



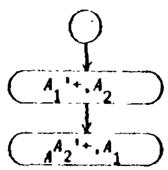
(missing level)



(duplicated motion)

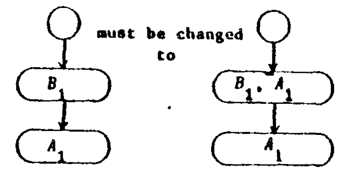


(order conflict)



(order conflict)

Figure 3-12 - Address Generation Errors



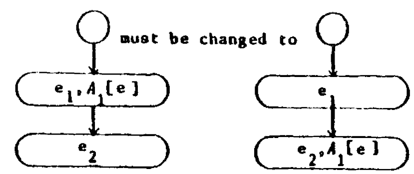


Figure 3-12 - Address Generation Errors
(cont.)

3.4.15 Loop Limit Validity

When a choice node selects between alternative targets, the number of alternatives must equal the number of choices. The alternatives must also match in size at the other levels. In addition, since the increment DELTA at one level is calculated on the assumption that all lower levels exhaust their corresponding array dimensions, the effective loop limits must match the size of the array. Thus we require: (see examples in Figure 3-13)

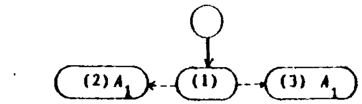
1. If two loop limit labels appear on the same node, they must agree. (If not based on the same array, conformability requirements from constraint propagation may be used to establish equality.)

2. If evocation edges leave a node, the loop limit of the choice node must equal the sum of the limits of the target nodes.

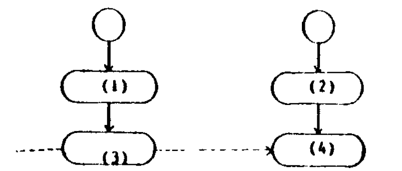
3. If the control path connecting two active nodes at the same level does not pass over an evocation edge at that level, the loop limits of the two nodes must be equal.

4. The sum of the loop limits of all nodes containing pointer increment labels for a given array dimension must equal the length of that dimension. If there exists more than one node incrementing a single pointer, the control paths leading to those nodes must separate at either a node with multiple nesting which is their father or at a choice node which is at their common level.

These requirements are checked explicitly (described in Chapter 4).



ρA = limit (1) = limit (2) + limit (3)



limit (1) must equal limit (2)
but limit (3) need not equal limit (4)

Figure 3-13 - Loop Limit Eerrors

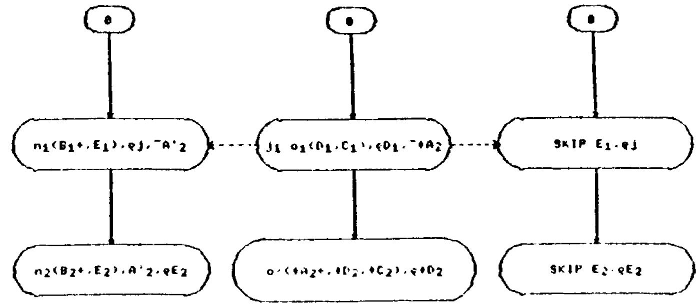
3.4.16 Sequencing Correction

When the pointer movement for one dimension of an array is accomplished in more than one node, the standard ladder fixed parts will not produce the correct address sequence because at both the first and the last (test fails) steps of a loop the pointers are not incremented. Therefore at the transition between two nodes moving the same pointer one increment will be omitted. It must be done in the separation node before the first execution of the second option to execute. Pointer reset applied to pointer motion in a different ladder will require the same correction.

3.4.17 Examples

The stream generator graphs for two of the examples presented earlier are shown below. In these drawings we have included only those result labels which are needed to show special properties of nodes (/ \) and relationships between them. This was done to save space and avoid clutter. In Appendices F and G graphs with all labels shown are presented. An inspection of those drawings will quickly reveal why labels are omitted here. The procedure used to generate these graphs is described in Chapter 4.

3.4.17.1 Filtering - The stream generator for the filtering example - $B←(∨/A)/[1]E∨A$ ◇ $A←C∧D$ is described by:



2: $T[1]←0$

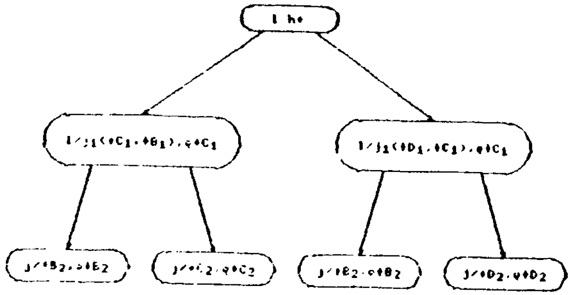
3: $B←E∨A$

3: $A←T[2]←C∧D$
$T[1]←T[2]∨T[1]$

4: if $T[1]$ then EVOKE L2
else EVOKE L3

in the new notation.

3.4.17.2 Merging - The merging example $S←+/+/B,C,[1]D$ now appears as:



```
1:T[1]←1                10:T[2]←0

2:T[2]←0                11:T[2]←C+T[2]

3:T[2]←D+T[2]           14:T[2]←B+T[2]

6:T[2]←B+T[2]           15:T[1]←T[2]+T[1]

7:T[1]←T[2]+T[1]
```

which is a considerable simplification.

3.5 COMPILER OBJECT CODE

The object language of the compiler has been designed to simply and efficiently implement the control structures given by the graph representation and to be easily translatable into machine code for the ladder machine designed by Charles Minter (see Appendix E for a description of the ladder machine and an example program). A BNF definition of the syntax follows:


```
<network>           ::=<ladder> | <network>";"<ladder>;
<ladder>            ::=LADDER <ladder #>":"<statement>;

<statement>         ::=<cond statement> |
                       <assign statement> |
                       <init statement> |
                       <loop statement> |
                       <switch statement> |
                       <empty>|
                       (<statement list>);
<statement list>    ::=<statement> | <statement list>";"<statement>;

<cond statement>    ::=<temporary>"=""">"<statement> |
                       <temporary>"=""">"<statement>ELSE<statement>;

<assign statement>::=<var>_<expr>;
<expr>              ::=<var><dop><expr> |
                       <constant><dop><expr> |
                       <mop><expr> |
                       (<expr>)|
                       <var> |
                       <constant>;
<var>               ::=<memeory> | <temporary> |
                       <pointer> | <base> |
                       <index> | <limit> |
                       <step> | <spacing>;

<init statement>    ::=INIT <pointer # list>;
<loop statement>    ::=REPEAT <statement> AT <level #>
                       USING <index #>
                       <stepping list>;
<stepping list>     ::=MOVING <pointer #> |
                       <stepping list>,<pointer #>
                       <empty>;

<switch statement>::=EVOKE<ladder #>;

<memory>            ::=[<pointer>];
<temporary>         ::=T[<register #>];
<pointer>           ::=PI[<pointer #>];
<base>              ::=BETA[<pointer #>];
<index>             ::=I[<index #>,<level #>];
<limit>             ::=RHO[<index #>,<level #>];
<step>              ::=DELTA[<pointer #>,<level #>];
<spacing>           ::=G[<pointer #>,<level #>];

<mop>               ::=+|-|×|÷|⌈|⌊|*|⍟|"|"|○|!|~;
<dop>               ::=+|-|×|÷|⌈|⌊|*|⍟|"|"|○|!|
                       ∧|∨|⍲|⍱|<|≤|=|≥|>|≠;

<ladder #>::=<level #>::=<pointer #>::=<register #>::=<limit #>
                    ::=<constant>::=<unsigned integer>
```

The semantics of the language are defined informally below with reference to the existing languages IMP-10 [5] and APL. With the exception of the APL operators the language is an extension of IMP-10 and programs which use only the standard arithmetic operations will compile into PDP-10 machine code. Appendix D shows the IMP extensions necessary and a sample of IMP-10 compiler output for stream generator code.

1. the operators (mop and dop) are equivalent to the corresponding scalar operators of APL.

2. _ is scalar assignment. (Algol :=)

3. => is the IMP conditional. (Algol IF .... NE 0 THEN ......)

4. The init statement is expanded to:

   ```
   PI[<pointer #>]_BETA[<pointer #>]
   ```

   for each element of its pointer # list. An init statement must be executed for a pointer before that pointer is referenced.

5. The loop statement is expanded to:

   ```
   I[<index #>,<level #>]_0;
   <unique label>:<statement>
   I[<index #>,<level #>]_I[<index #>,<level #>]+1;
   I[<index #>,<level #>] LT RHO[<index #>,<level #>] "=">"
     (PI[<pointer #>]_PI[<pointer #>]+DELTA[<pointer #>,<level #>];
         ... { repeat for each item in stepping list }
         GOTO <unique label>);
   ```

   which can not be written directly. The index #'s and level #'s have values given in the USING and AT clauses of the loop statement.

6. Each ladder stores a separate program counter. An EVOKE statement makes the designated ladder active (that ladder # must appear as a label). All program counters start at the beginning of each ladder and execution starts with an implied EVOKE 1.

7. Control, upon reaching the end of a ladder, returns to the beginning.

8. The construct [...] is an indirect reference to the memory which holds array elements.

All features not described should be considered as IMP-10 extended to handle matrices. We assume that the operands and the values for BETA, RHO,DELTA, and G have been pre-loaded.

CHAPTER 4

BUILDING STREAM GENERATORS FOR APL EXPRESSION

The translation of an APL expression into a stream generator takes place in three steps. First, stream generators, which have the maximum control and pointer flexibility and temporary storage which may be required, are generated using the graph representation. Second, transformations are applied to the graph in order to eliminate control overhead and unnecessary storage. Finally, the graphical representation is translated into the matching program. Figure 4-1 is a flow chart for the translator. For the purposes of describing the translator in this thesis, we have presented several components of it as interpreters of a command language. We then give the "program" to be executed in different situations. Such an approach could, but need not, be used in a machine implementation.

4.1 GRAPH TRANSFORMATION

The first two steps involve transformations of stream generator graphs. To simplify the description we define below a set of standard operations

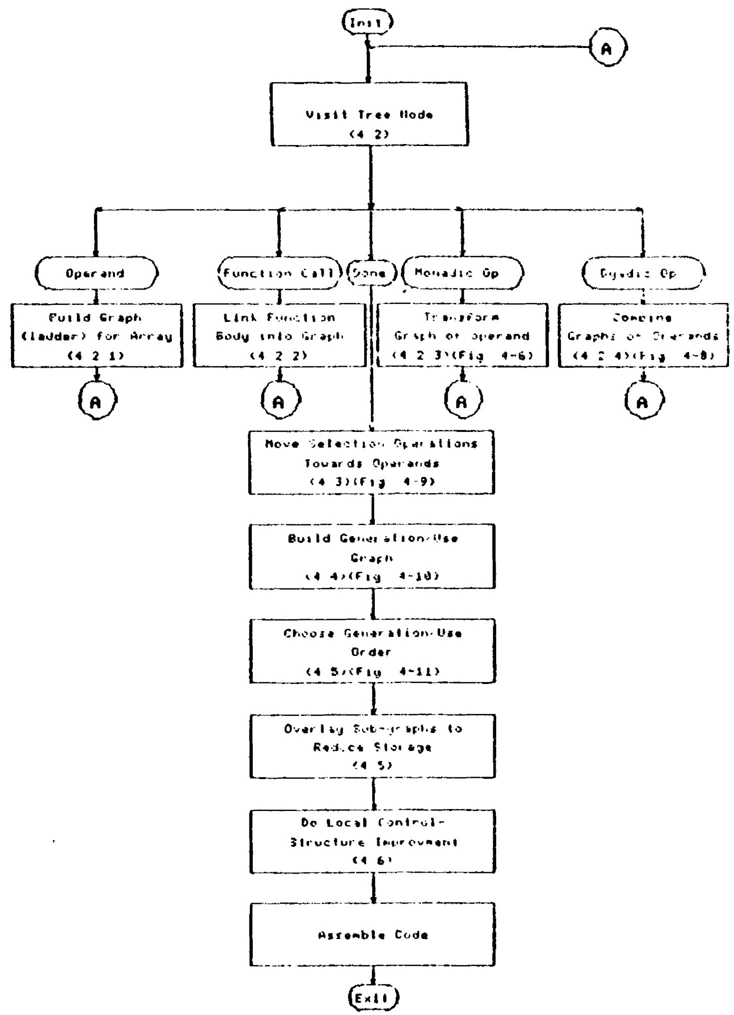


Figure 4-1 - The Translator

(commands). The definitions of the operations are given in terms of the structure of the graphs. When a node moves, its labels move with it. These operations (except Overlay) assume that the graphs which they transform have the following properties:

1. Only the entry point of the graph may have more than one nesting son, and the control path to any result node must include the left-most son of the entry point. We are building a control structure which does a succession of complete operations at the highest level. The last one produces the result.

2. If a node has evocation edges leaving it, it has no nesting sons.

3. All header nodes are at level 0.

4. All result nodes are in leaves of the super (evocation) tree.

The operations (except Overlay) preserve these properties if they hold for their input. Since they hold for ladders created to access an array, they will be preserved until the final stages of the improvement of the stream generator, at which time Overlay may be applied as the last use of these operations.

A stream generator graph having these properties may be partitioned into the entry point and one or more "sub-graphs". Two nodes are in the same sub-graph if and only if the control path to both of them includes the same nesting son of the entry point. A sub-graph is active if it contains result nodes.

4.1.1 Commands

The compiler will process the graph using the operations given below. The steps carried out for each command are given and many are illustrated by examples. In the examples, the graphs have been edited to remove labels not needed to identify result nodes.

Certain of these operations are not applicable in all cases. If an operation can not be correctly applied, it is said to "fail". When failure occurs, the graph is restored to its state before the operation was invoked. Failure of a command is reported to whatever process invoked it.

4.1.1.1 Adjust - A stream generator graph A is "adjusted to fit" a ladder B by changing loop limits in A if necessary so that:

1. For each node in B, the node in the entry ladder of A at the same nesting level has the same limit.

2. Graph A meets the requirements for the relation between the limits of choice nodes and their targets given in Section 3.4.15.2.

Graphs which have been adjusted may be in violation of the restrictions given in Sections 3.4.15.3 and 3.4.15.4 which specify the correct relation between loop limits of nodes at the same level and between loop limits and array sizes. Subsequent commands may correct the situation, but when Adjust is used (in Merge below), the command Check must be executed at the end of processing to verify that the final graph is correct.

4.1.1.2 Check - Check does not alter a graph. It verifies that loop limits meet the restrictions given in the three sections mentioned above. If the restrictions are violated, then the command using Check fails. Figure 4-2 shows examples for Adjust and how it can cause Check to fail.

4.1.1.3 Overlaying - When two stream generators with the same structure are operating (or can operate) in synchronization, it is desirable to use the same control structure for both. This will be accomplished by overlaying the two generators. Two graphs (or sub-graphs) may be completely overlayed if:

1. The two graphs are isomorphic.

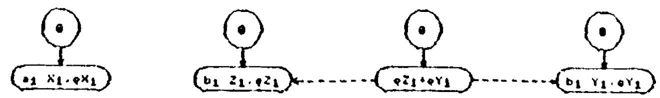
2. The equating of nodes of the isomorphism preserves not only adjacency but also the type of connection (nesting, raveled nesting, or evocation), the level, and the right-to-left order of multiple edges leaving a node.

3. The loop limits for each equated pair of nodes agree.

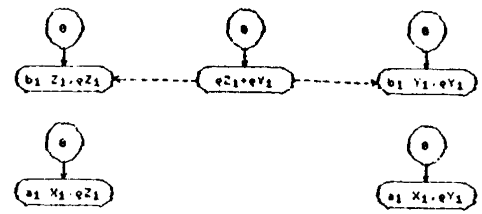
4. If two choice nodes are equated, they must both make the same selection at each step.

Because of the requirement of preserving right-to-left order of edges, the overlay process involves only a single simultaneous traversal of each graph (tree). A simple interpretation of the first two requirements is that the pictures of the two graphs must

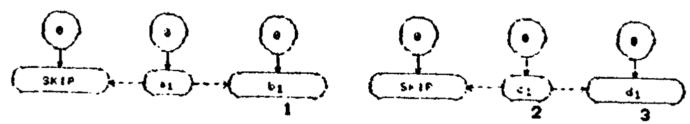
The command: Merge a and b (as in $X+Y,Z$)



requests that a be copied and adjusted to fit both active ladders for b.



Since $(\rho X)=(\rho Y)+\rho Z$ is true by conformability, Check will succeed after Overlay. However the command: Merge b and d (as in $(V/A)+(W/B)$)



requests that d be adjusted to fit the active ladder for b (1). This means that the limits for node 1 and node 2 must agree, but conformability requires nodes 1 and 3 to have the same limit. Thus Check will fail.

Figure 4-2 - Adjust and Check

look the same. Obviously this operation will not change the nesting pattern or change header level.

It is also possible to do a partial overlaying in two cases which do not permit complete overlaying. They are:

1. If complete overlaying would combine two choice nodes with different selection patterns, the nodes may be combined, but overlaying stops with that node. Both sets of evocation edges leave the new node.

2. If two nodes can not be overlayed because of count or addressing restrictions or because one is a raveled structure and they are nesting sons of nodes that can, they may become separate nesting sons of the combined father. Overlaying stops at this division.

The last form of incomplete overlaying can create multiple nesting at any point. However it is performed only during the final stage of the improvement process. The merging operation (below) which uses Overlay requests complete overlay only.

4.1.1.4 Transpose - Several APL operators (ex. Transpose) require the permuting of the order of nesting of the nodes labeled to be part of an operand. When this is done, it is necessary to also move the other active nodes. Transposing a stream generator graph is accomplished using the procedure given below which is illustrated in Figure 4-3.

1. The active sub-graph is detached from the entry point and transformed as specified in steps 2 thru 5.

2. The nesting edges connecting active nodes are broken.

3. All active nodes at each level are moved to the specified new level as a unit.

4. Each set of nodes formerly connected by nesting edges are reconnected in their new order. If gaps exist in the nesting, new (unlabeled) nodes are created. (Gaps will be created when a node which has no nesting sons is moved below a level formerly beneath it.)

5. If nodes become inactive due to transposition they are discarded. (They will have been generated earlier to fill a gap.)

6. The transformed sub-graph is then reconnected to the entry point. Right to left order will be preserved.

The above procedure is only correct if there is not multiple active nesting. This transformation will preserve that status. It will also leave all level 0 header nodes at that level since level 0 is not subject to transposition.

4.1.1.5 Reversal - Several APL operators (ex. reverse) require that the processing of a given dimension of a stream generator be reversed. This is done by reversing all pointer increment labels in active nodes at the specified level. There are 2 special cases.

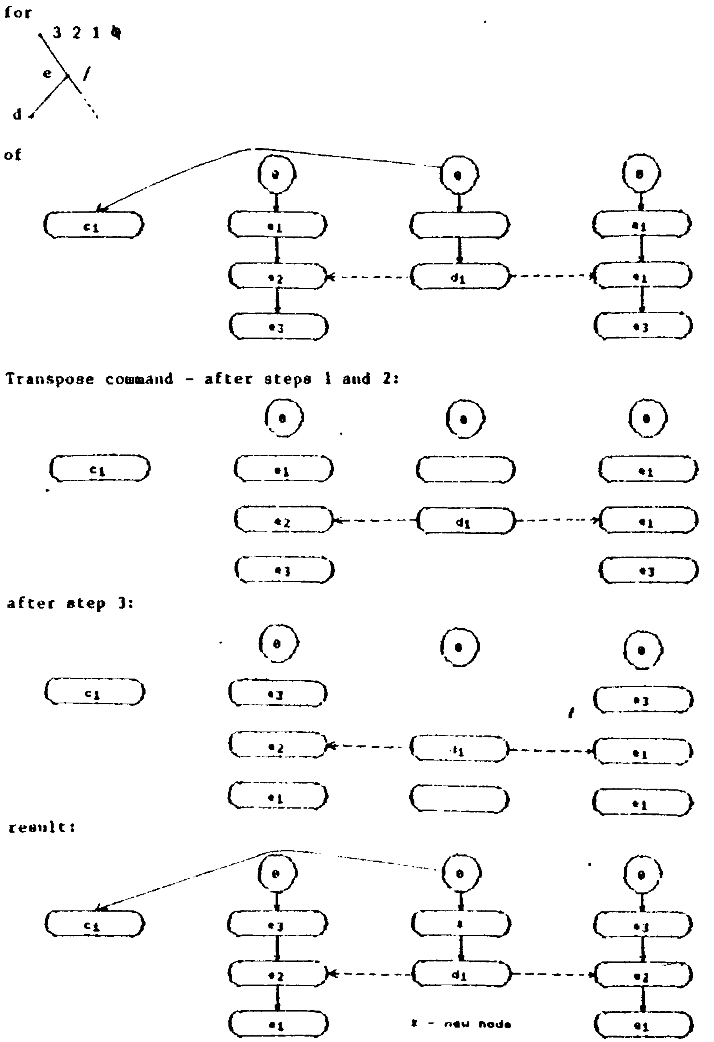


Figure 4-3 - Transpose

1. The operation may not be applied to a node containing the label functions / or \.

2. If reversal is applied to a raveled structure, all nodes comprising the structure must be reversed.

4.1.1.6 Merging - The generation of the stream generator graph for an operation which requires synchronized access to its two conformable operands (ex. dyadic scalar operations) requires merging the entry points and the active sub-graphs of the two operands (inactive sub-graphs are first disconnected). There are 3 cases, each described below and illustrated in Figure 4-4.

1. If in each graph the result nodes are in the ladder containing the entry point (ex. *A+B*), the two graphs are completely Overlayed if possible.

2. If one graph has result nodes in a different ladder from the entry (control path uses evocation edges) (ex. *A+(C,D)*), each ladder containing result nodes is detached and completely Overlayed (if possible) with a copy of the graph for the other operand which has been Adjusted to fit. The resulting ladders are re-connected and the whole Checked.

3. If both operands have evocation edges as part of the control path to a result node (ex. *(A,[1]B)+C,[2]D*) then we select the operand in which the first such occurs at the highest level (selection can be arbitrary if levels are equal). Each ladder

containing result nodes is replaced by a copy of the the graph for the other operand which has been Adjusted to fit the ladder it replaces.

For each graph B which replaced a ladder A, we detach each ladder in B containing result nodes and completely Overlay it (if possible) with a copy of A which has been Adjusted to fit. The results are reattached, and the whole is Checked.

The inactive sub-graphs of each operand are then attached to the header of the merged active structure (to the right). If either Check or Overlay fails then Merge fails.

This procedure will not create multiple active nesting since complete overlaying can not cause new nesting. Since ladder structures are moved intact, headers will stay at level 0 if at that level in both operands.

4.1.1.7 Nesting - Certain APL operators (ex. outer product) require a stream generator which accesses one operator inside the inner-most loop accessing the other. The generation procedure which is shown in Figure 4-5 is:

1. The active sub-graph for the operand to be nested under the other is detached.

2. A copy of it is attached to each result node of the other stream generator which is at the end of an active control path.

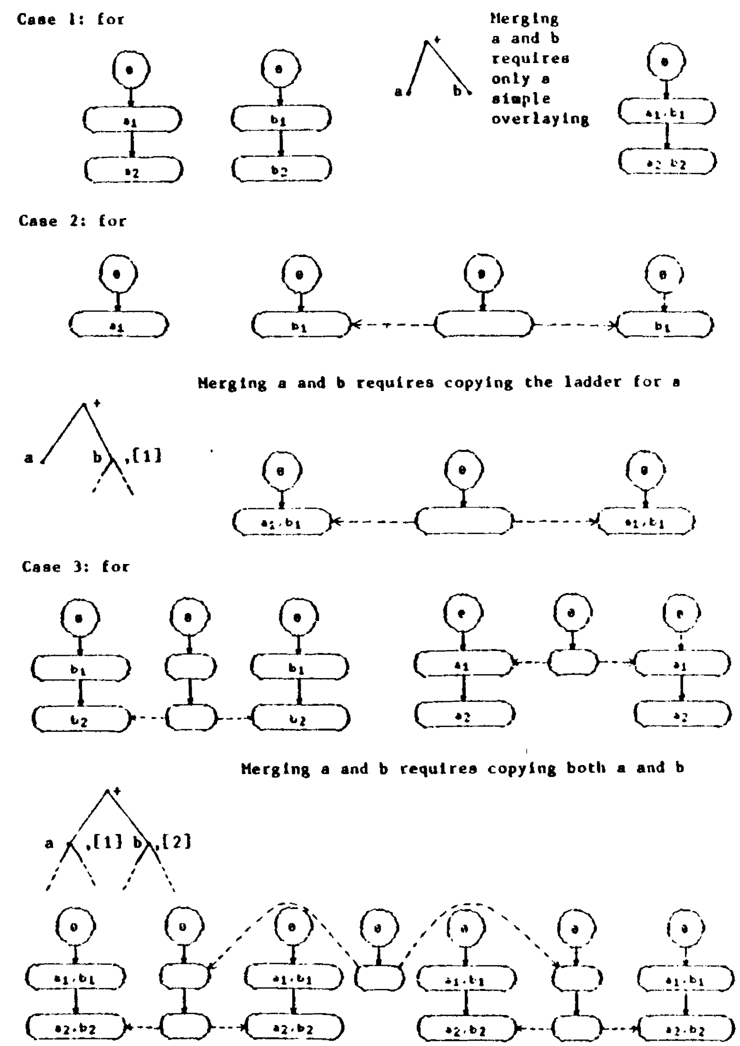


Figure 4-4 - Merging

3. If the newly positioned sub-graph contains any header nodes, new empty nodes are inserted between each header and its nesting son until nodes which had equal nesting level in the original sub-graph are again at the same level. If a new node is a nesting ancestor of a node modified by SKIP, then that modifier is placed on the new node.

4. The header of the lower operand is combined with the header of the upper. Any sub-graphs nested below that header become sons of the combined header (on the right).

The result of this procedure for nesting will have all its header nodes at level 0 if so positioned in both operands. Since the new nesting connections are made to nodes with no active sons, and only one connection is made, no multiple active nesting can be created.

4.1.1.8 Alternatives - The APL operations which require the selection of one of two possible sources for the result (ex. expansion) are implemented using co-routine evocation. New evocation edges are added, running from the lowest result node(s) of the graph which makes the selection, to the active node in the entry ladder of each alternative which is at the level of the operation. The procedure is illustrated in Figure 4-6. There are 2 special cases:

1. If the node from which the evocation edges leave is above the level of the operation (as in V\[n]A where n is not highest dimension of A), it will be nested under new empty nodes until at a matching level.

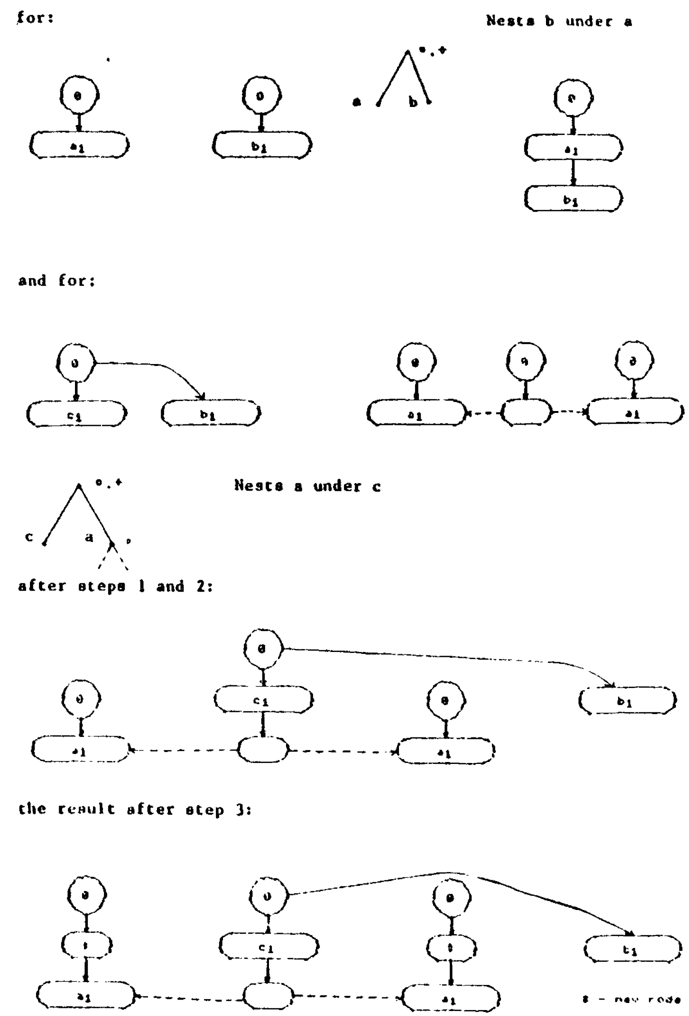


Figure 4-5 - Nesting

2. If one of the alternatives does not have an active node in the entry ladder at the level of the operation (as in *A*,[2]*B*,[1]*C* - see Figure 4-7), no legal connection can be made. The Evocation Order demon described in 4.1.2 must first be applied. It will guarantee that the new destination for the evocation edge has a proper target.

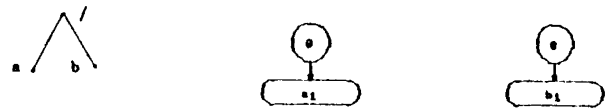
This operation will fail if an alternative contains inactive target nodes at the level of the operation (ie, the result of a compression may not be compressed over the same dimension without being stored) unless the operation is being used to represent catenation (see Appendix G - Example 4).

The creation of alternatives has no effect on nesting structure or header levels. There will be nodes nested below the choice node only if it is possible to have an inactive node nested below a result node (which has been ruled out).

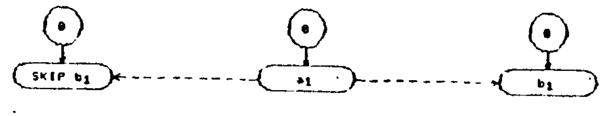
These operations will be used both to create and improve stream generators. We show in Appendix C that reversing or transposing a sub-graph of the entry point will not change the contents of storage.

None of these operations will create multiple active nesting or pull header nodes down from level 0. New nesting connections are made only to the entry point and at the lowest result node of a ladder. Multiple nesting will only be created at the entry point unless an inactive sub-graph hangs below a lowest result node.

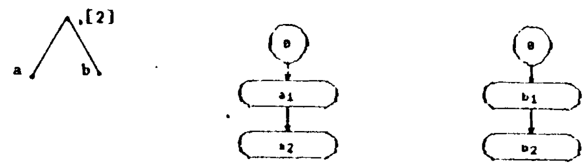
for:



the Alternative command produces:



and for:



the result is:

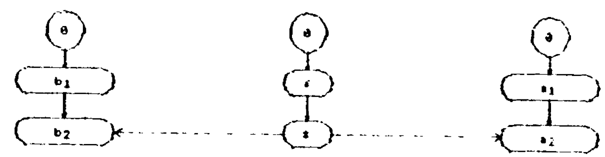


Figure 4-6 - Alternatives

4.1.2 Demons

Faithful application of the above procedures can yield graphs not suited for later phases in the compilation. To avoid this the following transformations will be carried out whenever applicable.

4.1.2.1 Address Calculation - If nodes containing address calculations are moved by a transposition, the address calculation will be moved, if necessary to the location where the subscript is available (see Section 3.4.14.8).

4.1.2.2 Empty Nodes - If empty nodes are created, they must be assigned loop limits meeting the requirements for loop limit validity given in Section 3.4.15.

4.1.2.3 Pointer Reset - When graphs are transposed or nested, pointer reset labels must be added as required by Section 3.4.14.9, and removed when they are not required.

4.1.2.4 Redundant Choices - If a control path passes through two choice nodes, and if the branch selected by the first determines the branch which will be selected by the second, the branchs which may not be selected can be deleted. If that results in a choice having only one branch, the remaining alternative can be overlayed with the choice. An example is $(A,B)\downarrow A,C$. We saw in Figure 4-4 how Merging the results of two Alternative operations generates a graph with two choice nodes on each control path to a result node. In this case they are at the same level and identical.

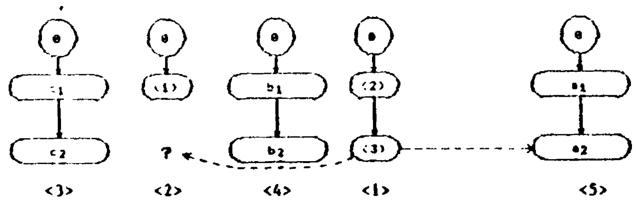
4.1.2.5 Evocation Order - The co-routine pattern described in Chapter 3 requires that if an evocation edge enters a ladder, it connects to a node at the same level as the source of the edge. The above transformations may produce a graph which does not have a suitable target for a subsequent operation. If so, the stream generator must be transformed to create one. The procedure which operates at the level of the super tree defined by the evocation edges is: (see Figure 4-7 which shows the example $A,[2]B,[1]C$)

1. Locate a node with a son which does not contain a legal target.
2. Break the connections into the father and into and out of the son.
3. Put the son in place of the father, and in the place of each grandson attach a copy of the father (including any descendants not detached).
4. Attach each grandson to the remaining broken link on the appropriate copy of the original father.
5. Repeat until the super graph is ordered.

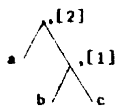
When the evocation edges are broken and reconnected the choice nodes remain the same and the targets become the nodes at the matching level. This procedure has no effect on nesting structure or header level.

4.1.2.6 Scalar Operands - If an operand of a dyadic operation is a scalar, its header is merged with the header of the other operand.

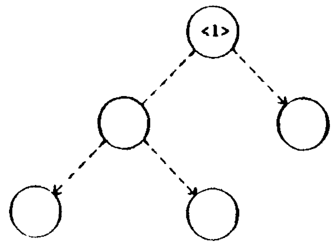
the graph:



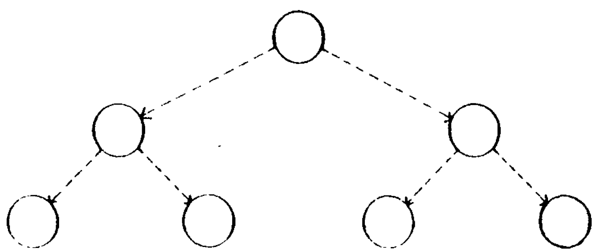
which results from



has an evocation tree with <1> as a "father" whose son <2> does not have a proper target node (<3> and <4> are grandsons)



this is transformed to:



which is:

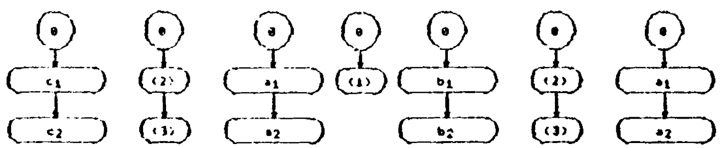


Figure 4-7 - Evocation Order Demon

Labels to reference that scalar are placed on all the result nodes of the other operand.

4.1.2.7 Repeated Calculations - If an operation will build a graph which has, or could have, after transposition, nodes of one operand nested below nodes not of that operand, and if the labels modified to be the result of that operand are not simple pointer movement reference only, then labels specifying assignment to a temporary array are added to each modified node and they become the result.

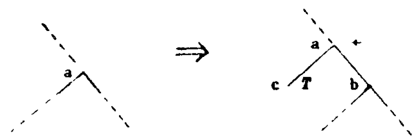
4.1.2.8 In-Line Assignment - If, in the graph for an operand, the modifiers indicating which labels represent the result of that node of the parse tree are applied only to simple pointer increment assignment labels, and if the operation is not assignment, the stored quantity will be used as the operand. The changes made to the graph are detailed in the section of Appendix F describing the assignment operation. (Note: This interpretation of assignment implies that the value of an assignment to part of an array is the right operand of the assignment. It also forces shape conformability.)

This operation nests an active structure under the entry point, but it also removes the result labels from the old sub-graph. Thus the result will not have multiple active nesting. The new active sub-graph has no multiple nesting and contains no inactive nodes. Header level is not affected.

The last two demons reflect that one intent of the generation process is to leave maximum flexibility for the improver by including all temporary storage that could possibly be needed. The improver will never need to generate temporaries, only eliminate them.

4.2 CREATION OF STREAM GENERATORS

The input to the translator is the parse tree with its associated predictions and requirements. The parse tree will be traversed in right-left-root order. If, during the processing of a node for an operator, temporary storage is created to hold the result, or if an operand must be placed in a temporary before the operation can be applied, the parse tree will be modified to reflect the change:



where b and c are new parse tree labels. The action to be taken for each operand or operator is given below. Examples of the graphs produced for each operator are shown in Appendix F. Figure 4-8 shows a flowchart of the processing of an operator.

4.2.1 Operands

When a leaf node (storage reference) is encountered, it is handled as follows:

4.2.1.1 Arrays - If the variable (or constant) is an array, the graph is a header node over loop nodes nested to a depth equal to the rank

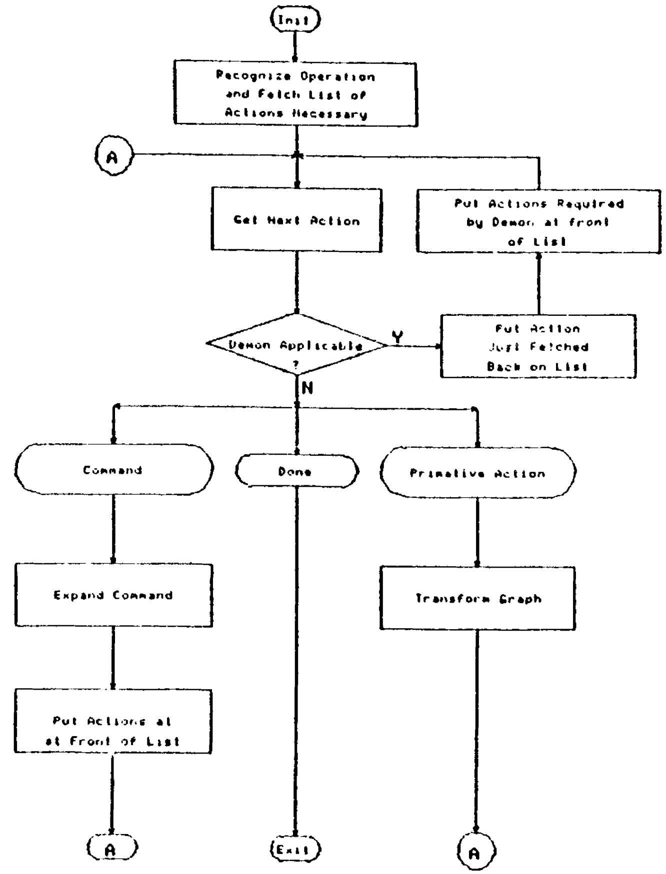


Figure 4-8 - Translation of an Operator

of the array. Each loop node is labeled with $e_i A_i, \rho A_i$. This is a single ladder with no multiple nesting or inactive nodes.

4.2.1.2 Scalars - If the variable is a scalar, the graph is only a header node.

Each ladder created is based on an array. New ladders will be generated for operations only when a new temporary array is created.

4.2.2 Functions

User written functions may be handled in one of two ways:

4.2.2.1 Separate Unit - If earlier phases of the compiler have separated the function call into a separate compiled unit, that call is not compiled, but serves to instruct the interpreter to compile (if necessary) and execute the function body.

4.2.2.2 Stream Generator Subroutine - If earlier analysis has determined that a previously compiled function may be included in the calling compiled unit, it will appear in the parse tree as an operator of appropriate valence. The command program for processing this operator is:

```
for each operand of the function do
   (build a simple ladder with shape of operand;
    label the new ladder to assign to a temporary;
    execute Merge command between new ladder and operand);
build a simple ladder with shape of result;
label it to assign to a temporary
                  and as result of the function call;
Overlay entry points of operand and result ladders
                  with result as left-most sub-graph
```


The result sub-graph represents the action of the function which is actually connected to the calling stream generator by evoking it from the entry point. Since that ladder does not actually exist, it can never be overlayed.

When a function is compiled a preamble is generated containing instructions to be followed by the interpreter in performing validity checks and initialization. The preamble also lists all global variables appearing in the function together with the constraints that must be imposed upon or between the global variables and operands of the function.

4.2.3 Monadic Operators

When a monadic branch node (monadic operator) is traversed, the graph of the result is built by transforming that of the operand. The transformations applied for each operator are given in Appendix F (with examples).

Reduction is the only operation which can create an inactive node in a ladder containing result nodes. Since the result of a reduction is always placed in storage, when this stream generator is used for an operand, the In-line Assignment demon will create a new active sub-graph which contains only result nodes. Thus when the transformation operations are applied no result node for a monadic operation may have an inactive nesting son. Therefore no multiple nesting can be created by a monadic operation except at the entry point. Since the graphs produced for operands have headers at level 0 and no multiple nesting, and the transformations preserve these properties, they hold for the results of monadic operations.

4.2.4 Dyadic Operators

The graph for the result of a dyadic operation is built up from those of the operands. The procedure used for each APL operator is given in Appendix F (with examples).

The dyadic operations create inactive nodes only through use of the monadic operation reduction which, as seen above, does not cause violation of the nesting restrictions. Therefore, like the monadic operations, they create multiple nesting only at the entry point. Also the ladder containing the result nodes used for the choice in the alternative operation will have only result nodes. Since the alternative operation uses the lowest result node as the choice node, no nodes will be nested under it. Thus the dyadic operation will also preserve the desired properties.

4.3 ELIMINATION OF UNNECESSARY CALCULATIONS

Compression and the selection operators (except transposition without diagonalization) discard part of their right operand. If the value is used nowhere else it should not be calculated. During the course of stream generator creation the selection operations were applied to the stream generators for their right operands, but the In-line Assignment prevents any operation from affecting the calculation of a value which

has been stored.

Now, after the completion of the generation phase, if all references to any given dimension of an array which is local to the stream generator are affected by the same selection operator, then the equivalent operation will be applied to the sub-graph of the entry node which is active for the generation of that array. When the operation is not directly applicable (would generate temporary storage), it is abandoned. If the application is successful, then the selection operation is removed from all the labels for the array. One application of this transformation may create more opportunities to apply it. However, since operations are moved towards the leaves of the parse tree, the process will terminate. Figure 4-9 is a flowchart for this phase of the compiler.

The "reject" side of the alternative construction used to implement compression does only pointer movement for the source(s) of the right operand. If the right operand is an expression which has not been put in temporary storage, calculation of unneeded elements will be skipped. If the right operand had been stored, only the final reference to the stored value is skipped. However, the compiler attempts to eliminate temporary storage by moving the calculation to point of use. In the case of compression, a side effect will be the elimination of unneeded calculations.

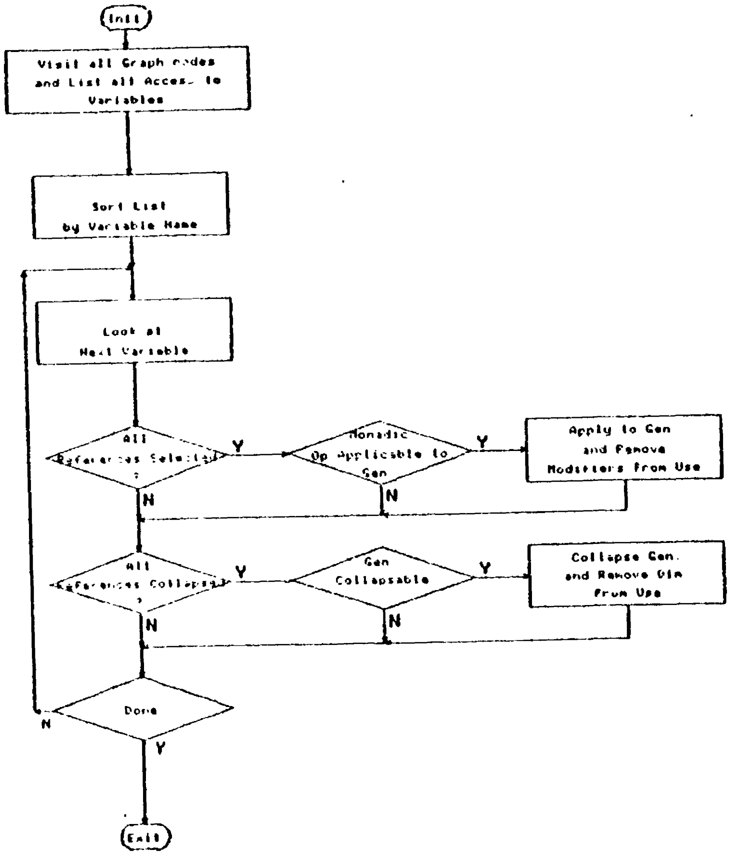


Figure 4-9 - Un-necessary Calculations

4.4 REDUCTION OF TEMPORARY STORAGE

The primary reason for translating APL into stream generators is to facilitate the elimination of unnecessary temporary storage. The stream generator graph reveals the order in which intermediate results are generated and used. It also contains the information needed to determine alternate orders. The ideal situation is a stream generator in which the order of generation and use match for all intermediate values. When this occurs no temporary storage is required. The compiler will attempt to find the orde which comes closest to this ideal. During this process a constraint is imposed that no calculations will be repeated in order to reduce storage. The task of determining the best ordering is complex because the constraint of no repeated calculations requires that certain operations be done in one particular order (ex. Scan), and because certain operations require internal temporary storage unless done in a specific order (ex. Reduction).

If the level 0 header which is the entry point of the stream generator has more than one nesting son, each sub-tree executes independently in succession, communicating array data only via memory. The creation rules have produced a graph in which the entry point is the only node with more than one nesting son. They have also produced a configuration in which a successful application of the commands transpose or reverse to the generation of an array (result labels for that parse tree node used to determine active sub-graph) will not change the result in storage (see Appendix C). These two operations which we call "reordering" may thus be applied to the stream generator as needed.

The elimination of temporary storage requires that the stream generator be transformed so that the calculations are done in synchronization instead of in sequence. The desired transformation is to overlay all the sub-graphs of the entry point. When overlaying is blocked only by the presence of raveled nesting, two sub-graphs may be synchronized as co-routines using evocation.

The compiler attempts to select the order for each sub-graph which permits the maximum overlay. Storage may be eliminated in the following cases:

1. Both assignment and all references to an array dimension are in the same node.

2. The assignment to an array dimension is in a choice node and the only references are in all the targets.

3. The assignment and reference of the entire array are at matching levels of two ladders synchronized by evocation.

If an array is used in a node where none of the above apply, or if there exist a node where the storage is modified by / or \, then storage of that array may not be eliminated lower in the graph.

The most straightforward approach to picking an order for each sub-graph would be to try all possible combinations of ordering and select the one requiring the minimum storage. Unfortunately that approach requires the examination of a number of cases that grows exponentially with the number of sub-graphs. The sections below

describe a procedure which is not guaranteed to generate an optimal ordering, but it is computationally feasible (time) and has proved to perform well.

4.4.1 Generation And Use

The determination of best order for each sub-graph is simplified by the use of an alternate representation for the generation and use of arrays. We draw a graph in which each node represents one of the sub-graphs of the entry point of the stream generator graph. The nodes are labeled with the names of the arrays (or scalars) assigned to by that sub-graph. If one sub-graph uses a value generated by another, a directed edge is drawn from generation to use in the new graph. The edge is labeled with the name of the array (or scalar). If a node has out-degree 0, the values it assigns are not referenced in this stream generator. If the arrays (and scalars) generated are local to the stream generator, the node (and the associated sub-graph) are deleted (only functions without side-effects have been included in larger stream generators).

Each edge indicates a possible storage savings to be obtained by overlaying the sub-graphs represented by the two nodes connected by that edge. Thus we first eliminate edges connecting sub-graphs which we can determine can not be overlayed (the complexity of the recognition process is discused below). This occurs in the following situations: (considered in order listed)

1. Required Storage - Storage of a generated array is required if:

1. A use of that array is subscripted or rotated.

2. A dimension of the use is modified by a selection operator.

3. Two dimensions of the use are in the same node (diagonalized)

4. Two uses of the same array in one sub-graph (aliases) are not in the same order at the same levels.

When an array is used in that way, edges for that array entering that node connect two nodes which may not be overlayed. Those edges are deleted.

2. Scalars - A sub-graph generating a scalar intermediate result in a register must complete before the value may be used. Thus edges representing a scalar are deleted from the graph.

3. Alternatives - If both the generation and use of an array are in ladders which are alternatives selected by a choice node (reached via evocation edges), and if the alternative is not implementing the compression operator, then overlaying is possible (see definition) only if the choices in the two sub-graphs are identical. Otherwise overlaying is impossible, and the connecting edge is removed from the graph.

   In the case of compression it may be possible (see Section 4.5.1) to move an array use from the alternatives into the choice ladder. If this occurs, overlaying is possible. Thus edges representing that situation are retained.

4. Order Conflict - Any path through the graph defines a function which maps each ordering of the node at the beginning of the path to that ordering(s) of the node at the end of the path which causes generation and use to be synchronized. If there exists a node with out-degree greater than 1, and if paths which leave that node along different edges rejoin, the ordering functions defined must be consistent. If this is not true, either the fork or join must be eliminated.

5. Repeated Use - If one use is nested under another, no order exists which permits both to be completely overlayed with its generation.

6. Required Sequencing - If two nodes may not be overlayed, this may block the overlaying of other pairs of nodes in three circumstances:

   1. A node which depends (path exists) on both may be overlayed with neither.

   2. A node depended on by both may be overlayed with neither.

   3. Two nodes which may not be overlayed may not be overlayed with the same node.

When a choice exists as to which edge in the graph to eliminate, the one representing the smallest quantity of temporary storage is selected (ie - best is a global array - next the smallest temporary). If any edge for a given array has been eliminated, that array must be stored, and it is considered global in any further processing. As a final step, all edges for arrays considered to be global are eliminated. Thus the procedure that selects sub-graph orderings will only attempt to match generation/use order when storage savings are possible. Figure 4-10 shows an example of this process.

Detection of order conflict or required sequencing may require tracing all pathes from each node, and in worst case could could require time proportional to the square of the number of nodes. The other cases depend on properties of single nodes or edges (number of edges bounded by square of number of nodes or by size of original function).

4.4.2 Graph Order For Maximum Overlay

The removal of edges may have divided the generation/use graph into several unconnected components. Each will be processed separately in what follows. Each edge remaining indicates the possibility of elimination of temporary storage via one of four patterns of overlaying.

1. The generation and use are both in the entry ladder, and are overlayed.

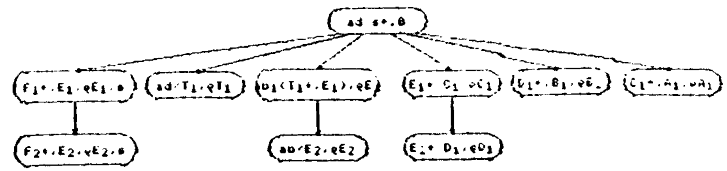
2. Copies of the entry ladder containing the generation (adjusted to fit) are overlayed with the ladders of use in their position as alternatives. (Because of the in-line assignment demon, an entry ladder which is a generation will not have alternatives.)

3. The entry ladders of copies of the sub-graph containing a use are overlayed with the alternatives containing the generation.

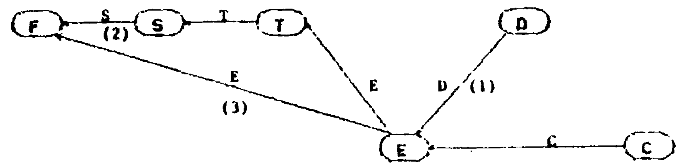
The APL expressions:

```
C←⍳A
D←-B
E←C∘.×D
S←+/+/E
F←S+E
```

are translated into



The Generation/Use Graph is



edge (1) is deleted due to nesting conflict (repeated use)
(2) is deleted because S is a scalar
(3) is deleted because (2) was deleted (required sequencing)

leaving

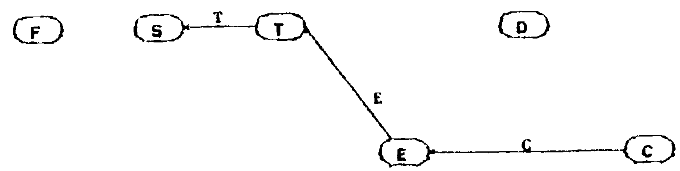


Figure 4-10 - Generation/Use

4. When the choices are identical, two sub-graphs with the generation and use in alternatives are overlayed.

An edge may represent one of two situations:

1. The storage may be eliminated by overlaying the stream generator as currently ordered.

2. The storage may not be eliminated because use and generation are in different orders. Reordering the sub-graph corresponding to either node will correct this situation.

The end result of this algorithm is a graph with only edges of type 1. However, that is not sufficient to eliminate unnecessary storage. There are 2 cases which require a sub-graph to have a particular order before storage may be eliminated. These are:

1. The storage may not be eliminated when generation occurs below a node labeled with / or \ (/[n] or \[n] where n is not lowest dimension will require storage unless transposed).

2. The storage may not be eliminated if use is repeated (nested under another array - *B* in *A*∘.+*B* or nested under a dummy node which was created for an alternative operation - *V* in *V*/[n]*A* where n is not highest dimension).

When the same array is used in both the choice and target ladders of a compression, storage of that array can only be eliminated at or above the level of the choice. However, we have not included compressing the

lowest dimension as one of the requirements since it conflicts with the elimination of repeated use of the left operand. Since we can use the storage allocated for the final result to hold candidate components, this is not a serious problem.

In addition there will be sub-graphs (produced to handle reshape) which can not be reordered.

The algorithm presented below gives priority to satisfying the requirements indicated by edges and will, if necessary, leave the requirements indicated by nodes unsatisfied. Node requirements are relaxed in a fixed order depending on the operation that was the source of the requirement. They are never reinstated even if the higher priority requirement with which they conflicted is later eliminated. Requirements are relaxed in the order given below:

1. Reshape Input - The monadic reshape operation generates a sub-graph which may not be reordered. If that causes an order mismatch along an edge entering the reshape node when all node requirements are honored, the edge is deleted. (The node requirement is absolute).

2. Repeated Use - The preference to avoid repeated use will be the next one abandoned. If saving storage is critical, values used repeatedly may be calculated repeatedly.

3. Reduction and Scan - The preference that the dimension reduced or scanned be the lowest one is the last abandoned since the resulting storage may not be avoided. However, no examples have been seen in which the storage of internal intermediate results required when the preference is not honored exceed the storage resulting from mismatch

of generation and use.

4. Reshape Output - If, after all node requirements are relaxed, an edge conflict involves an edge leaving a reshape node, the edge is deleted.

The compiler proceeds as follows:

1. Visit each node of Generation/Use graph and establish preferences.

2. Visit each each node which has a preference in order of decreasing priority, and trace each path leaving that node. If a node with a conflicting preference is reached, the preference with the lower priority is abandoned. If a node with no preference is reached, a preference is established.

3. If a node now exists with no preference, select one with in-degree 0. Select an ordering and trace all paths leaving the node. If conflicts with a preference are discovered, select a new ordering and repeat. If no conflicts are found, establish this order as a preference for all nodes on the path. This step is repeated until all nodes have an order preference.

Figure 4-11 is a flow chart of the ordering procedure.
This procedure which establishes preferences explores the whole graph starting from each node. Thus it may require time proportional to the square of the number of nodes. The final order selection traces the graph for each alternative tried. The number of alternatives is given by n! where n is the depth of the graph (highest array rank which

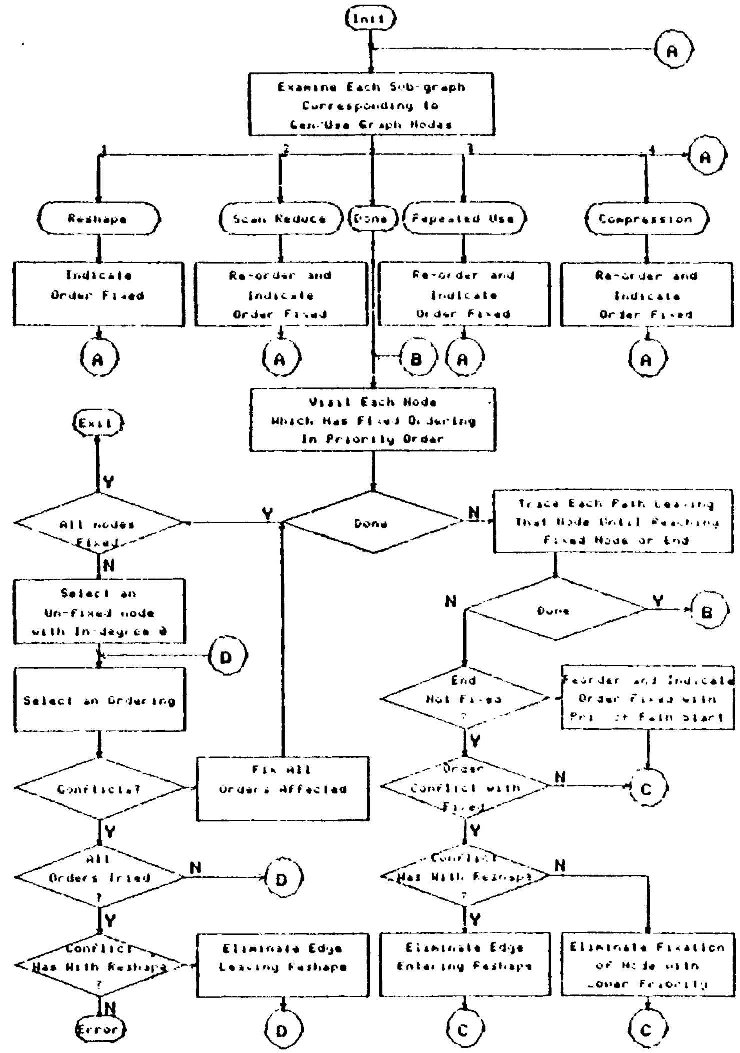


Figure 4-11 - Generation/Use Order

rarely exceeds 4).

In Chapter 5 the performance of the compiler for a set of examples is discussed. The action of the above procedure for one example is shown in Appendix G.

4.5 ELIMINATION OF EXTRA CONTROL STRUCTURE

Once unnecessary temporary storage has been eliminated, the compiler will attempt to reduce overhead and code size by simplifying the control structure.

4.5.1 Syncronization Within Sub-graphs

The alternative construction produces a stream generator in which the nodes above the level of the choice in the choice ladder and the alternatives are synchronized. Unless a level consists of a raveled structure, all pointer movement can be done in the choice ladder. Since having three loops doing the work of one is undesirable, the pointer movement labels are moved. All nodes now without pointer movement labels and not nested under nodes with them are removed. This operation will create header nodes at a level greater than 0. The pointer initialization operations are moved into the choice ladder.

If the choice node was created to execute a catenation, it exhausts first one alternative than the other. The same access pattern can be obtained with less overhead by making the two target nodes nesting sons of the father of the choice. The choice node is deleted. The merging

example of chapter 3 shows this transformation.

If the choice node was created to execute a compression, the pointer movement labels for the right operand may be moved from the target into the choice if either:

1. The target nodes are at the lowest level of the alternatives.

2. The choice node contains a pointer movement label for the same array (different alias). The labels must be exactly identical. When the movement label is removed from an active target, it must be replaced with a pointer reset label.

If the target node modified by SKIP now has no pointer movement labels, that alternative may be eliminated.

Figure 4-12 shows examples of the above transformations.

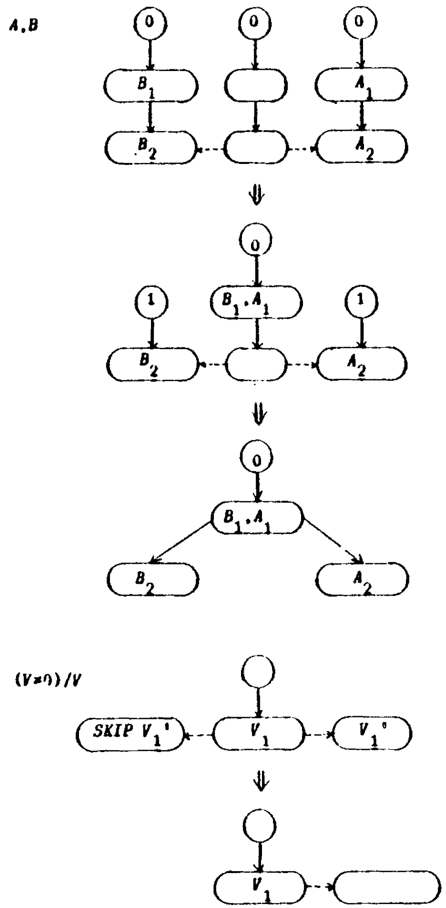


Figure 4-12 - Synchronization Within Sub-graphs

4.5.2 Loop Jamming

The loop control required by the stream generators may be reduced in the following situations: (see Figure 4-13)

1. Two sub-graphs match in shape and can be fully or partially overlayed without resulting in reference to an array before assignment (the order of reference and assignment along the control paths of the resulting stream generator must be checked).

2. If in all ladders comprising a sub-graph of the entry point two adjacent levels use the same DELTA value for all pointers, and if there is no splice code in the ladder rails connecting the two levels, the two levels can be collapsed into one.

The translation into stream generators has considerably simplified the problem of recognizing the opportunity to apply these two transformations.

4.5.3 Alias Elimination

Given two pointers *A* and *A'* which refer to the same array, if for each occurrence of A'_j A_j is present in the same form on the same node, *A'* may be eliminated. Only one pointer is needed.

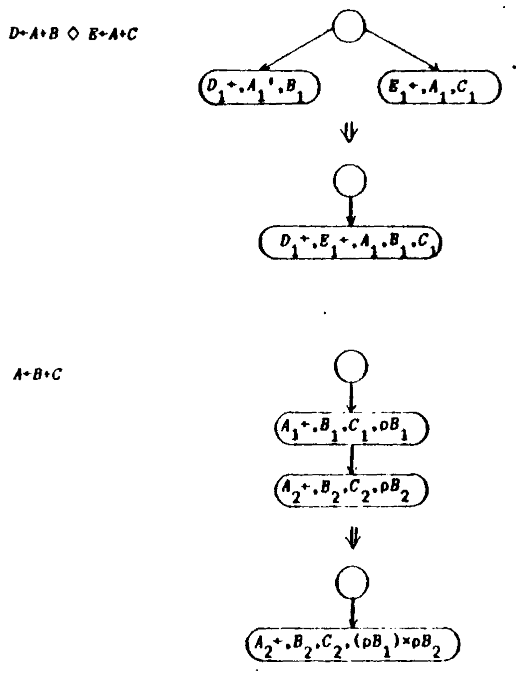


Figure 4-13 - Loop Jamming

4.5.4 Tight Linkage Of Called Functions

If a function which has been linked into a compiled unit accesses its arguments (results) in the same order that it is generated (used), and if there are no repeats, then the two access points may be linked with an evocation edge. A different version of the compiled routine, which was created to expect that quantity to be in a register, will be used.

4.5.5 Subroutines

If the stream generator contains identical sub-graphs (created by copying), one copy may be created (with combined limits). It will be evoked from each node formerly adjacent to one of the original copies. The evocation edges will no longer define a tree.

CHAPTER 5

THE EXECUTION OF STREAM GENERATORS

The previous sections have described a procedure for translating APL into stream generators. Now we will examine the actual execution of stream generators and evaluate them based on the following criteria:

1. the amount of storage used,

2. the size of the code generated,

3. the speed of execution.

Our primary concern is with the amount of storage required. The other factors are considered to demonstrate the feasibility of this method for executing APL.

5.1 EXECUTION ENVIRONMENTS

Examination of code size and execution time require the specification of exactly how the stream generators will be executed.

5.1.1 Translation Into Machine Language

The object code of the compiler can be translated into machine code for the Digital Equipment Corporation PDP-10 using an extended version of the compiler for the IMP-10 language. The examples below will show that this approach is feasible. However, three problems suggest that this approach is not the ideal one:

1. Most of the APL scalar operations would have to be interpreted. If the arithmetic operations are to be independent of number representation, then all must be interpreted.

2. The control structure will make very heavy use of short patterns of instructions. The lack of single instructions to perform these functions increases code size and slows execution.

3. The control parameters I, RHO, DELTA, and G and the pointers PI and BETA are used frequently, but are too numerous to keep in registers. The high rate of memory access will slow execution.

Because of these factors and the overhead of compilation, no speed advantage will be claimed for a compiler translating into machine code. Only the size of the object module will be considered in order to demonstrate that the storage economy obtained by using the compiler does not require special hardware.

An example of the machine language representation for a stream generator is given in Appendix D. That example assumes that all numbers are represented as integers. It uses machine instructions for the scalar operations.

5.1.2 The Ladder Machine

A better environment for the execution of a stream generator is provided by an auxiliary processor - the ladder machine. Charles Minter [17] has designed two versions of a processing unit designed to execute networks of ladders as conceived by Perlis [19]. With minor modifications they are also well suited for the execution of stream generators. The ladder machine keeps all control parameters in fast local memory. Access to main memory is required only to fetch or store array values (Splice code [PI]).

In Minter's design the ladder machine executes independently of the main CPU. When the ladder machine is executing a stream generator, the main CPU may do other unrelated processing. An alternate approach more suited to a smaller machine would be to provide the ladder machine capabilities as an extension (micro-programmed) to the instruction set of the single CPU.

The modified ladder machine upon which our estimates of code size and speed are based is described in Appendix E . An example of the ladder code representation is given. The stream generator is the same as the one whose PDP-10 machine code representation is shown in Appendix D.

Minter wrote a translator which produced code for his ladder machine. Since the ladder is basically a hardware representation of the array accessing method employed by Abrams, Minter's translator produces code which reflects the transformations Abrams called "beating and dragging". Using this translator and a simulator for his ladder

machine, Minter obtained the following comparisons:

1. For the simple expression $A+B$:

    1. The time to compile the expression and load the ladder machine would be twice that required to set up for interpretative execution.

    2. The time required to load the ladder machine with the results of a previous compilation is half that required to set up for interpretative execution.

    3. If the expression must be compiled, the ladder machine will be slower than the interpreter for A and B with less than 50 elements.

2. Minter's moderate-performance ladder processor executing with a DEC PDP-11 host CPU will execute +/(⍳200)∊⍳200 approximately 4 times faster than a IBM 370/158 running APL.SV.

In the examples given later in this chapter, we will compare the speed of the ladder machine code produced by Minter's translator with the speed of the code produced by this compiler. In making that comparison, we will assume that the same hardware is used to execute the output of both translators.

5.2 TRANSLATION EXAMPLES

In the sections below we present the output of the compiler for each of a set of examples. The output is analyzed to determine:

1. The size of the object module if translated into PDP-10 machine code (see Appendix D).
2. The size of the object module if translated into ladder machine code (see Appendix E).
3. The number of array element loads and stores executed (given the sizes of input).
4. The amount of temporary storage used (given the sizes of input).

The last two measures will be compared against the same values obtained based on the following methods of executing APL:

1. a naive interpreter which performs each operation separately, putting each intermediate result into temporary storage.
2. A compiler which compiles on a line by line basis and incorporates the work of Abrams. The HP-3000 APL compiler [13] and the APL translator developed by Minter are examples of such a compiler.

The stream generators and the final object code for each example are given in Appendix G.

5.2.1 Example 1 - Prime Numbers

The expression *S*←+/2=+/[1]0=(⍳*N*)∘.|⍳*N* will calculate the number of primes less than or equal to N. It is an excellent example of an expression which generates a large intermediate result on the way to a small answer, and it has been analyzed by several authors [1] [Weg] [8] [19]. It compiles into 37 PDP-10 instructions or 16 ladder instructions.

For N = 10, the expression performs as follows:

	Array Element References	Temporary Storage
Naive Interpreter	570	220
HP-3000 Compiler	0	0
Stream Generator	0	0

Results similar to that obtained using stream generators were obtained by Wegbright [Weg]. Daniels [8] proposed to go further by considering mathematical properties of the operators. The space requirements when this expression is interpreted grow as *N*⋆2 and will limit N sooner than excessive execution time.

5.2.2 Example 2 - Roman Numbers

The expression *R*←(,((7⍴5 2)⊤*N*)∘.≥⍳4)/,⍉4 7⍴'*MDCLXVI*' converts an integer (N) into its representation in Roman numerals. It compiles into 68 PDP-10 instructions or 31 ladder instructions.

When the result contains 7 characters, the execution of this expression requires:

	Array Element References	Temporary Storage
Naive Interpreter	273	74
HP-3000 Compiler	165	70
Stream Generator	28	7

The stream generators performance is similar to that described for this example in Daniels [8].

5.2.3 Example 3 - J Choose N

The function

```
      D← J CHOOSE A;B;C;N;V
[1]   N←ι1↑ρ A
[2]   B←(N,N×J!N)ρ2 1 3⍉1 2 2 3⍉((ιN)∘.=ιN)∘.∨A
[3]   V←2⊥B
[4]   C←((ιρV)=VιV)/B
[5]   D←((J+1)=+/[1]C)/C
```

takes as its argument A a boolean matrix each column of which has J elements equal to 1. Each column is unique and together they give all the ways of choosing J elements from N. The output of the function D is the same information for J+1. The function makes N copies of each column of the input and forces "on" (1) a different position in each copy. It then discards all duplicate columns and those which still have only J 1's.

The stream generator (see Appendix G) reveals the characteristics of the function. It will run without conformability checking (no fixed loop limit label appears on the same node as a variable label). Each column of A is used once (the label A_2 appears only once and at level 1). That column is used repeatedly (the label A_1 appear below a

pointer reset label for A) to generate candidates for columns of the output. This means that each column must be in memory, but the function could be entered repeatedly (as a co-routine since V accumulates) once for each input column. If we wished to go directly from J to J+2, two copies of this stream generator could be strung together eliminating the storage of all but a single column of the matrix for J+1.

It compiles into 206 PDP-10 instructions or 96 ladder instructions. For N = 10 and J = 5 this function requires:

	Array Element References	Temporary Storage
Naive Interpreter	7,318,578	277,200
HP-3000 Compiler	6,533,604	50,400
Stream Generator	6,414,660	2,530

The execution time is dominated by that required to execute $V\iota V$ which is of order $0.5\times(N\times J!N)\star 2$ (6,350,400). A compiler, which could recognize that the process of locating duplicate entries in a vector (V) could be speeded up considerably by maintaining V in sorted order and inserting each new value if unique, could produce much more efficient code. However, that level of sophistication was not considered in the design of this compiler.

The function generates a large number of candidates for inclusion in its output and then tests each one to determine if they really belong in the output. As seen above, the execution of such a function may require a compiler which can execute the function without ever storing all the candidates at one time. This is especially true if execution is to take place on a small machine with a limited workspace and no virtual storage. The value of a stream generator compiler depends considerably on the prevalence of this style of programming in APL. My impression, based on the programs writen by the introductory programming classes tought by Professor Perlis at Yale, is that this compiler is needed. However, the detailed investigation of a large sample of programs neccessary to confirm that impression has not been done.

It is also not clear to what extent gains made by the compiler depend on a failure by the user to properly analyze the problem. However, the answer to that question for this particular problem is suggested by the function below:

```
      X←N CHOOSE Y;D;T;M1;M2
[1]   T←2*⌽0,⍳N-1
[2]   M2←M1∧~<\[1]M1 ◇ M1←0=T∘.|Y
[3]   D←(,M2)/,1 2 1⍉Y∘.+1 1 2⍉T∘.×M2
[4]   X←(0=2|D)/D
[5]   ⎕←(Nρ2)⊤D
```

which is also a solution. This function makes a copy of each input column for each 0 preceding the first 1 in that column. It then turns "on" (1) a different one of those 0's in each copy. No duplicates are created and all columns have J+1 1's. The old result is stored with each column converted to a single number. Since a column starting with 1 does not contribute to the new value, odd entries are discarded. The execution time of the new version is of order *N×J!N* instead of *(N×J!N)*2* for the original. However, the naive interpreter and the HP-3000 compiler would store at least an extra 635,040 and 5,440 elements respectively when N=10 and J=5. My conjecture, supported by this example, is that user cleverness has more effect on speed than space (provided he adheres to the loop-free style of programming).

5.2.4 Example 4 - Symbol Table Update

The function

```
      SYM X;Y
[1]   Y←,~X∊A
[2]   A←A,Y/,X
[3]   B←(B,Y/,0)+X=A
```

uses global variables A and B which are respectively a vector of single character symbols and a count of the number of times each symbol has been encountered. The function argument X is a character. It will be appended to A if required and the matching item of B will be updated or created.

It compiles into 85 PDP-10 instructions or 48 ladder instructions. When A has 10 elements and X is a new element, this function requires:

	Array Element References	Temporary Storage
Naive Interpreter	137	22
HP-3000 Compiler	107	11
Stream Generator	42	0

5.2.5 Example 5 - String Search

The two line APL expression:

```
D←(B[C∘.+¯1+⍳ρA]∧.=A)/C ◇ C←(B=1↑A)/⍳ρB
```

searches a string B for occurrences of string A and puts all starting positions into D.

It compiles into 110 PDP-10 instructions or 54 ladder instructions. When A is 10 characters long, its first character occurs 10 times in B which is 100 characters long, and A occurs once in B, the expression requires:

	Array Element References	Temporary Storage
Naive Interpreter	1181	210
HP-3000 Compiler	951	210
Stream Generator	301	210

5.2.6 Example 6 - Selection

The APL expression *A←5 5↑B+C+D* which was used in chapter 3 to introduce multiple array ladders will compile into 45 PDP-10 instructions or 25 ladder instructions. When the inputs are 10 by 10 matrices, this expression requires:

	Array Element References	Temporary Storage
Naive Interpreter	650	200
HP-3000 Compiler	100	0
Stream Generator	100	0

5.2.7 Example 7 - Transposition

The expression *S←×/+/[1]A⍉B* which was used in chapter 3 to show the importance of re-ordering calculations compiles into 38 PDP-10 instructions or 20 ladder instructions. When the inputs are 10 by 10 matrices this expression requires:

	Array Element References	Temporary Storage
Naive Interpreter	420	110
HP-3000 Compiler	200	0
Stream Generator	200	0

5.2.8 Example 8 - Filtering

The two line expression

B←(∨/A)/[1]E∨A ⋄ A←C∧D

which was used in chapter 3 to introduce the use of co-routines compiles into 133 PDP-10 instructions or 74 ladder instructions. When the inputs are 10 by 10 matrices and 5 rows survive, this expression requires:

	Array Element References	Temporary Storage
Naive Interpreter	870	210
HP-3000 Compiler	7:0	210
Stream Generator	450	10

5.2.9 Example 9 - Merging

The expression *S←+/+/B,C,[1]D* which was used in chapter 3 to demonstrate the need for multiple nesting compiles into 86 PDP-10 instructions or 47 ladder instructions. When B is a 10 by 5 matrix and C and D are 5 by 5 matrices this expression requires:

	Array Element References	Temporary Storage
Naive Interpreter	320	160
HP-3000 Compiler	300	150
Stream Generator	100	0

5.2.10 Summary

5.2.10.1 Code Size - The table shows the size (in bytes) of the 3 representations for an APL expression and gives the blow-up caused by translating the original APL:

Example	APL	Ladder Code	Blow-up	PDP-10 Code	Blow-up
1	21	64	3.0	185	8.8
2	37	124	3.4	340	9.2
3	82	384	4.7	1030	12.6
4	30	192	6.4	425	14.2
5	38	216	5.7	550	14.5
6	11	100	9.1	225	20.5
7	12	80	6.7	190	15.8
8	20	296	14.8	665	33.3
9	13	188	14.5	430	33.1
			7.6		18.0

Considering the conciseness of the APL notation, the blow-up factor is reasonable. For comparison Algol 60 programs for the algorithm (both before and after the transformations made in producing the stream generator) of example 3 compiled into 230 and 158 words, respectively, of PDP-10 code (vs. 206 PDP-10 words for stream generator). (I found

the Algol versions harder to write than the APL.) An object module of 206 PDP-10 words for the function of example 3 (the largest) is certainly of tolerable size. A workspace containing 20 such functions would require approximately 4K(octal) words of PDP-10 memory for storage of ladder code. That size seems a reasonable cost for the 20 such powerful functions.

The last two examples, which experienced the largest blow-up, were those in which multiple copies of a ladder referencing one array were created as the stream generators were built, and then not eliminated as they were improved.

5.2.10.2 Array References (Time) - The table below show the improvement in number of references to array elements obtained by using the stream generators: (100% improvement means all references were eliminated)

Example	vs. Interpreter	vs. HP-3000 Compiler
1	100%	0%
2	90%	84%
3	13%	2%
4	70%	61%
5	75%	69%
6	85%	0%
7	53%	0%
8	95%	94%
9	69%	67%
	72%	41%

The comparison with the HP-3000 compiler gives an approximate indication of the speed difference between a ladder machine running code produced by this compiler and a ladder machine running code produced by Minter's (Abrams based) translator.

5.2.10.3 Temporary Storage - The table below shows the improvement in the amount of temporary storage obtained when the stream generators are used: (100% improvement means no temporary storage)

Example	vs. Interpreter	vs. HP-3000 Compiler
1	100%	0%
2	91%	90%
3	99%	95%
4	100%	100%
5	100%	100%
6	100%	0%
7	100%	0%
8	96%	96%
9	100%	100%
	--------	--------
	99%	65%

The streaming technique is effective, and including additional operators over those handled in the HP-3000 compiler does make a significant difference.

5.3 COMPILER OVERHEAD

Minter included in his performance estimates an allowance for the time taken to generate ladder code. In order to compare the compilation method he implemented with the one described here, we must estimate the cost of the extra processing required in the creation of stream generators. The extra work appears in three places.

5.3.1 Data Dependency

The data dependency analysis needed to identify temporary storage within a compiled unit is, in the simple form described in chapter 2, a trivial book-keeping operation during a traversal of the parse forest. In the examples shown it is not required, since each example consists of only one compiled unit.

5.3.2 Constraint Propagation

Minter's compiler implicitly propagates predictions upwards in the parse tree during the generation of ladder code. In all of the examples given, with compilation taking place at first execution, the upwards propagation produced all the information necessary for compilation and the propagation phase ended after performing an equivalent amount of work.

5.3.3 Stream Generator Refinement

Most of the compiler algorithms have been described in terms of actions involving the visiting of nodes of various graph structures. Using number of nodes visited as rough estimate of execution time, the stream generator refinement phase of translation of Example 3 requires approximately 80% as much time as the remainder of the compilation common to both compilers. The number of nodes visited is reduced considerably by the fact that (as in all other examples) the first ordering considered (preferences satisfied) yielded the maximum possible overlay.

Accurate performance estimates await the implementation in production form of the translators from both Minter's and this thesis. However the examples suggest that a reasonable estimate is that the stream generator compiler would require twice the time to generate a block of code for the ladder machine. This raises the estimated array size needed, before compilation is faster than interpretation, to 125 elements.

CHAPTER 6

CONCLUSIONS

6.1 THE COMPILER

The results presented in chapter 5 suggest that the compiler presented in this thesis can and should be implemented. However, the cost of the final processing needed to reduce temporary storage is sufficiently high to preclude its application in all cases. The stream generator output by the creation phase requires very little further processing to be executable. If the compiled code will not be used again, and if the storage required is available, immediate (but slower) execution may be more efficient. But in those situations (production and/or space limited software) which need the full compiler, the difference in execution time performance will often be critical.

APL expressions may be divided into three classes:

1. Expressions which are executed efficiently even by a naive system (FORTRAN written in APL). For such expressions, a naive system is preferable, since the compiler achieves no gain in performance to offset the increased overhead (except when compiled code is used

repeatedly).

2. Expressions which require the processing of this compiler for efficient execution (such as example 3 of chapter 5).

3. Expressions which are beyond help from this system. (The expression *S*←+/×/(⍉*A*)+*A* ⋄ *A*←*V1*∘.↑*V2* will require storage of A unless either each element of A is calculated twice or the calculation of A is done in a order which is not ravel order for *A* or ⍉*A*. This compiler would store A).

This work will be truly valuable only if a significant percentage of APL expressions being written fall into class 2. We do not have any experimental evidence concerning that question. However, the style of APL programming advocated by Perlis in [20] and demonstrated in [21] certainly decreases the size of class 1, and seems to result in consistency of orderings of access to arrays. That consistency permits optimization.

Exact information on the performance and applicability of this design (including data on the trade-off described above) will not be available until the design is implemented as part of a complete APL system. The Hewlett Packard HP-3000 APL system would provide an ideal base for the implementation. The controller which manages the interaction of the incremental expression compiler and the interpreter exists. Also the current output of the compiler is code for a virtual machine. That machine could be changed to be a ladder machine without requiring a major restructuring of the compiler.

I estimate that a man-year of effort would be required to add the features of this design to the HP-3000 system. Once in operation, the compiler could collect data about its own effectiveness. A complicating factor in any such investigation would be the tendency of users to adjust their programming style to match what executes efficiently. An analysis of current APL usage might not show the style of programming (heavy use of outer-product and compression) for which the compiler makes the most difference simply because the naive execution is intolerably inefficient (or impossible).

6.2 FUTURE WORK

The current design limits the size of compiled units by assuming that

1. Every labeled statement (potential Goto target) must begin a new compiled unit.

2. No two functions which use the same global variables may be part of a single compiled unit.

The work of Jones and Muchnick [14] and Kaplan and Ullman [15] suggests an approach which may make it possible to exactly identify those places in the APL function at which bindings might become invalid. This would be a valuable addition to the compiler.

A second open question is the relation between this work and the execution of APL on parallel or pipeline machines. The stream generator graph does identify the level of loop nesting below which all steps are

independent and thus could be performed in parallel. The stream generator graph and the ladder structure also define a repeated sequence of operations which could be pipe-lined. However, as mentioned in the discussion of APL emulators, there seems to be a conflict between storage economy and parallelism. This time/space trade-off should be investigated.

APPENDIX A

IDIOMS

A.1 IDIOMS

The constructs which this compiler recognizes as idioms are listed below. Many of them are the result of unusual properties of the the APL operators ⍳, ⌈, and ⌊. The operator ⍳ is the only APL operator which selects as its result the first quantity meeting some criteria. The operators ⌈ and ⌊ differ from all the other dyadic scalar operators in that they select one of their operands instead of combining them.

In the idiom descriptions that follow the variable V is a vector and the variable S is a scalar.

A.1.1 Niladic

A.1.1.1 Rank - ⍴⍴*A* - Rank is a scalar constant computed at compile time.

A.1.1.2 Indices Of Array - ⍳1(⍴*A*)[1] or ⍳1⍴*V* - This idiom generates a vector whose length (given by shape of A) is precalculated by the interpreter.

A.1.2 Monadic

A.1.2.1 Self Indexing - *V*⍳*V* - The result of this expression is the position of the first occurrence in V of each element of V. Normally execution of ⍳ requires a search of the left operand for each item in the right operand. However, when the operands are the same, only the part of the left operand at and before the current position in the right operand need be examined. This permits the idiom to be executed as the items ot V are calculated.

A.1.2.2 Extremum Position - *V*⍳⌈/*V* or *V*⍳⌊/*V* - The results of these expressions are the positions of the maximum and minimum elements of V respectively. Normal processing would result in two passes over V. Since the pass that finds the maximum (minimum) can also remember its location, only one is needed.

A.1.2.3 Span - ⌈/*V*∘.-*V* - This expression calculates the difference between the largest and smallest elements of V by performing all possible subtractions and taking the maximum. The same result is obtained from (⌈/*V*)-⌊/*V*. We also recognize that both the maximum and minimum can be found in a single pass over V. The graph is the same as for the reduction of V.

A.1.3 Dyadic

A.1.3.1 End Around - ¯1⌽*V*,*S* or 1⌽*S*,*V* - These expressions are equivalent to simple catenation in the reverse order.

A.1.3.2 First-found - $\lfloor/V1\iota V2$ - This expression gives the first position in V1 of whichever element of V2 first occurs. Normal translation would result in a pass over V1 for each element of V2. A better way is to test each element of V1 against all of V2, stopping at the first match.

A.1.3.3 Bounded Extremum - $\lfloor/S,V$ or $\lceil/S,V$ - The scaler S is used instead of the normal identity value for the reduction of V.

A.1.3.4 Take-till - $(V\iota S)\uparrow V$- The result of this expression is V up to and including the first occurrence of S. It can be executed as a single pass over V which is equivalent to compression of V. Thus the expression can be executed as V is calculated.

A.1.3.5 Delay - $S,\bar{}1\downarrow V$ or $\bar{}1\downarrow S,V$ - These expressions can be executed with a pass over V which saves the current value in a register for delayed access.

A.1.3.6 Select Index - $A[V/\iota\rho A]$ - This construct is compiled as V/A.

APPENDIX B

CONSTRAINT PROPAGATION

B.1 CONSTRAINT PROPAGATION PROCEDURE

The constraint propagation procedure operates on the parse tree of the APL function. The function has been sub-divided into compiled units which have no internal control structure, and each block is analyzed independently. Thus no flow analysis is involved.

B.1.1 Node Properties

The procedure is concerned with 4 characteristics of the value produced at each node of the parse tree:

1. Rank (number of dimensions - a non-negative integer)

2. Type (numeric, boolean, numeric-or-boolean, or character)

3. Length (of each dimension - a non-negative integer)

4. Value (scalars and vectors only)

The information regarding the result of a node is defined to have a "position" in the parse tree. It is located on the edge entering (leading downwards into) the node. We assume an imaginary edge entering the root of the tree so as to provide a location for the information about the final value of the expression. For any node we can refer to information located:

1. Above it. (properties of the result of that node)

2. To the right. (properties of right operand)

3. To the left. (properties of left operand)

In the case of monadic operators or operands, some of the positions (ie. left for a monadic operator, and left and right for a leaf) will not exist.

Knowledge of a property (or a requirement for that knowledge) is represented by the appearance of an expression in the slot for that property at the appropriate position. In the case of length and value, a vector of separate expressions may be required. The expressions have the form given by:

```
<high limit>        ::= "≤"<constant>|
                        "<"<constant>;
<low limit>         ::= "≥"<constant>|
                        ">"<constant>;
<range>             ::= <high limit> ∩ <low limit>;
<property term>     ::= <constant>|
                        <high limit>|
                        <low limit>|
                        <range>;
<property value>    ::= <property term>|
                        <variable>|
                        <property term> ∩ <variable>;
```

The limit operators may only appear in expressions for rank and length.

The property values define sets of possible values for the property (only one element for type and value properties). The constants define one element sets containing themselves. The limit operators (≤,< ,≥ , >) define sets which contain all the possible rank or length values which have the indicated arithmetic relationship with some element of the operand set. Since the legal values of rank and length form a finite set (non-negative integers bounded by an arbitrary maximum such as machine size), all property value sets are finite. The "∩" operator is set intersection. The compile-time variables used in these expressions represent information which has been (or will be) derived from the current values of the APL variables appearing at a parse tree leaf. They will not appear in the expression for the value of a property which has been completely determined by syntax restrictions and/or values of constants.

B.1.2 Property List

The constraint propagation algorithm keeps a list of information known about the expression being processed, but not yet represented in the parse tree. At each step an item from the head of the list is transfered to the proper position. The entries in the list have the form given by:

```
<propagation value> ::= <property value>|
                        (<property value>) "+" <constant>|
                        (<property value>) "-" <constant>|
                        (<property value>) "+" (<property value>)|
                        (<property value>) "+" (<property value>)
                              "+" <constant>|
                        (<property value>) "+" (<property value>)
                              "-" <constant>|
                        "≤" (<property value>)|
                        "<" (<property value>)|
                        ">" (<property value>)|
                        "≥" (<property value>);
```

The operators "+" and "-" produce the set consisting of those elements generated by taking the outer product of the two sets using the indicated arithmetic operator and eliminating all values which are not possible elements. These operators are used in expressions for rank and length information only.

These expressions give the largest property value set which may exist for a given position either absolutely (a property term) or in terms of the property values of other positions. The propagation values contain constructs which are not allowed in a property value. When a propagation value is taken off the list, evaluated and placed in the given position as a property value, a new variable is created to represent that part of the propagation expression. The value of a variable will be given as an expression of the form defined by:

```
<variable sum>       ::= <variable> + <variable>;
<variable atom>      ::= (<variable sum>)|
                         (<variable sum> + <constant>)|
                         (<variable sum> - <constant>)|
                         (<variable atom>)|
                         <variable>;
<variable limit>     ::= "≤" <variable atom>|
                         "≥" <variable atom>|
                         "<" <variable atom>|
                         ">" <variable atom>;
<variable term>      ::= (<variable limit>)|(<variable term>)|
                         <variable atom>;
<variable expr>      ::= <variable term>|(<variable expr>)
                         <variable expr> ∩ <variable term>;
<variable value>     ::= <variable expr>|
                         <variable expr> ∩ <property value>;
```

where the sum and limit forms are used for rank and length information only.

Items on the list come from two sources.

B.1.2.1 Generated Information - List items are generated by the algorithm based on properties of the operators or operands at individual nodes of the parse tree. These items are:

1. Property terms derived from constants.

2. Property terms derived from the properties of operators.

3. Variable names generated to be the property values of leaf nodes which are APL variables, not constants. The variables are assigned to represent rank, type, and length of all leaf nodes and the value of a leaf known to be a scalar. If two leaves refer to the same APL variables, the same compile-time variables will be assigned

4. Property terms derived from the current values of the APL variables. They are the current values of the compile-time variables defined above.

5. Variable names generated to represent a property value which must be known by the compiler.

Chapter 2 has discussed the order of generation and the significance of these items.

B.1.2.2 Propagated Information - We call two positions in the parse tree "adjacent" if the two edges directly connect to the same parse tree node. In a binary tree a set of properties (an edge) may be adjacent to a maximum of 4 others. The operator at the node which provides the connection defines relations between adjacent sets of properties. These relations are given in section B.2.

For each operator there is a set of propagation expressions which give (as propagation values) the maximum property value set for a given slot in terms of adjacent property values. An expression may be used only if all its components are defined. Some require the property term of a given component to have a particular value.

If the transfer of a list item into its position in the parse tree results in new information being added to the property value at that position, both ends of the edge for the position affected are examined to determine if information about adjacent positions is implied. If it is, the propagation expression is added to the tail of the list. We define new information to mean:

1. The appearance of a variable in a property value which was formerly empty or contained only a property term. This can happen only once for each property and position.

2. The number of items in the set defined by the property term has decreased. When the property value also contains a variable, we immediately place on the list propagation values consisting of the new property term for every position where the property value also contains that variable. Normal propagation is only considered towards adjacent positions not using that variable.

An item on the propagation list is tagged with the name of the position generating it as well as the position to which it is to be applied. When an item is applied it will never generate a new list item propagating that change back to its source.

When propagation values are placed in the list, the property values are represented by name (property and position). When these expressions reach the head of the list, the actual expressions are substituted for the names.

B.1.3 Property Insertion

When a list item is processed for insertion of its information into the parse tree, the following steps are taken:

1. The propagation value is joined to the property value for the slot using the ∩ operator.

2. All variables appearing in the new expression are replaced by the expression given by "(<the variable> ∩ <variable value for the variable>)". This process is repeated for any new variable which appears. (Note: this requires marking of expanded variables so as to avoid infinite expansion. The system remembers which terms of the result were present before expansion and the variable from which others were derived.

3. The resulting expression is simplified as described in section B.1.5, and will have the form:

```
<expansion> ::= <property term> | <variable expr>|
                <property term> ∩ <variable expr>|
                <expansion> ∩ <variable>;
```

4. If two variables appear in the expansion, the two variables are equated by replacing all occurrences of one (choice arbitrary) with the other.

5. If the expansion contains a variable expression but no variable a new variable is created and joine' to the expansion.

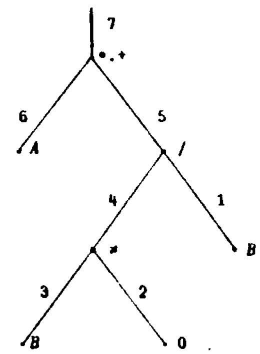
6. The new property value is the union of the property term and variable from the expansion.

7. The new value of the variable consists of the union of all components of the variable expression marked as coming from the original expression or from the expansion of that variable

(including any name equating).

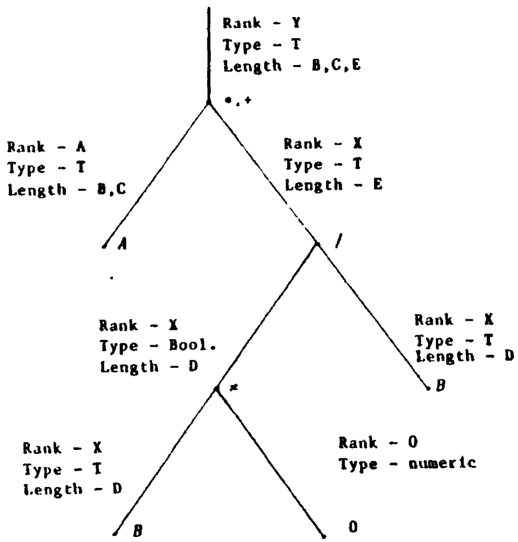
If at any point in the process a contradiction appears (ex. (>5) ∩ (<5) or X ∩ (<(X+Y))), it indicates an error in the APL expression. Constraint propagation is abandoned.

An example is shown in Figure B-1. The first page shows the list items in the order they were applied to the parse tree. Indented entries indicate an item which was generated by a constraint propagation rule and added to the end of the list when the item immediately above it was applied. The application of a generated item is marked with a '*'. The other list items reflect information known about individual nodes. The second page shows the final property values and their significance.




```
5: Type is numeric                    3: Length is D
    1: Type is numeric                    4: Length is D
7: Type is numeric                     7: Length is B, C
6: Type is Numeric                    *1: Type is numeric
4: Rank is 1                               T←numeric
4: Type is boolean                    *3: Type is numeric
4: Type is boolean                    *3: Rank is 1
2: Type is numeric                         X←1
    3: Type is numeric                *5: Type is T
2: Rank is 0                               7: Type is T
    3: Rank is 1                      *2: Type is T
1: Type is T                          *5: Rank is X
    5: Type is T                           7:Rank is A+X
3: Type is T                          *4: Rank is X
    2: Type is T                      *5: Length is E←≤D
1: Rank is X                               7: Length is B, C, E
    5: Rank is X                      *4: Length is D
3: Rank is X                          *7: Type is T
    4: Rank is X                      *7: Rank is Y←A+X
6: Rank is A                          *7: Length is B, C, E
1: Length is D                        *7: Type is R
    5: Length is  ≤D                       R is replaced by T
6: Type is R
    7: Type is R
```

Figure B-1 - Constraint Propagation - List Items



T = numeric - type check for B and A

X = 1 - rank check for B

E = ≤D - length of result of compression is not fixed so interpreter parameter calculation will be interleaved with stream generator calculations

Y = A + 1 - rank of A is not fixed by syntax, and must be known for compilation

Figure B-1 - Constraint Propagation - Final Results

B.1.4 Algebra Of Properties

While the notation used is non-standard, the objects described are finite sets of the non-negative integers. We take advantage of the well known properties of such objects to establish that:

1. <limit op 1> (<limit op 2> <set>) equals <limit op 3> <set> for all possible combinations of op 1 and op 2.

2. (<limit op 1> <set 1>)+(<limit op 2> <set 2>) equals either <limit op 3> (<set 1>+<set 2>) or <limit op 3> <set 1 or 2>.

3. (<limit op 1> <set 1>)+<set 2> equals <limit op 2>(<set 1>+<set 2>).

4. (<set 1> ∩ <set 2>)+<set 3> equals (<set 1>+<set 3>) ∩ (<set 2>+<set 3>)

Figure B-2 shows all actual combinations. These transformations will convert the formula produced in step 2 of the procedure given above to a legal expansion. The compiler builds the expressions internally as a string of tokens in Polish postfix form. Thus the transformations are done as simple string pattern matching and replacement.

$<(<X) \Rightarrow <(X-1)$	$<(\geq X) \Rightarrow$ no restriction *
$<(\leq X) \Rightarrow <X$	$<(>X) \Rightarrow$ no restriction *
$\leq(<X) \Rightarrow <X$	$\leq(\geq X) \Rightarrow$ no restriction
$\leq(\leq X) \Rightarrow \leq X$	$\leq(>X) \Rightarrow$ no restriction
$\geq(\geq X) \Rightarrow \geq X$	$\geq(<X) \Rightarrow$ no restriction
$\geq(>X) \Rightarrow >X$	$\geq(\leq X) \Rightarrow$ no restriction
$>(\geq X) \Rightarrow >X$	$>(<X) \Rightarrow >0$
$>(>X) \Rightarrow >(X+1)$	$>(\leq X) \Rightarrow >0$

(* - we ignore the fact that this set may not contain the largest element of the set of possible values)

$(<X) + (<Y) \Rightarrow <(X+Y-1)$	$(\geq X) + (<Y) \Rightarrow \geq X$
$(<X) + (\leq Y) \Rightarrow <(X+Y)$	$(\geq X) + (\leq Y) \Rightarrow \geq X$
$(<X) + (\geq Y) \Rightarrow \geq Y$	$(\geq X) + (\geq Y) \Rightarrow \geq(X+Y)$
$(<X) + (>Y) \Rightarrow >Y$	$(\geq X) + (>X) \Rightarrow >(X+Y)$
$(\leq X) + (<Y) \Rightarrow <(X+Y)$	$(>X) + (<Y) \Rightarrow >X$
$(\leq X) + (\leq Y) \Rightarrow \leq(X+Y)$	$(>X) + (\leq Y) \Rightarrow >X$
$(\leq X) + (\geq Y) \Rightarrow \geq Y$	$(>X) + (\geq Y) \Rightarrow >(X+Y)$
$(\leq X) + (>Y) \Rightarrow >Y$	$(\geq X) + (>Y) \Rightarrow >(X+Y+1)$

$(>X) + Y \Rightarrow >(X+Y)$

$(\geq X) + Y \Rightarrow \geq(X+Y)$

$(\leq X) + Y \Rightarrow (\leq(X+Y)) \cap (\geq Y)$

$(<X) + Y \Rightarrow (<(X+Y)) \cap (\geq Y)$

Figure B-2 - Set Algebra

B.1.5 Termination For Constraint Propagation

The constraint propagation procedure terminates when the list of items to be put into parse tree positions is empty. It is clear that the process will terminate. A new list item is generated only when the set for some property value decreases in size. Since all sets are finite, this process can not continue indefinitely. What must be evaluated more carefully is the possibility of decreasing the size of a very large set in tiny steps resulting in execution time not given as a function of parse tree size. However it can be shown that this is not possible. A change to the property value at one position in the parse tree may result in changes at four other adjacent positions. We can imagine markers moving on the parse tree outwards from an initial point of distubance. A marker may split into 3 at every property position. However, since we do not allow a change to propagate back to the position that caused it, no marker may retrace a path it or its immediate ancestor has followed. Since the graph (tree) has no cycles, no marker can get back to the initial position.

Thus each position in the parse tree is affected at most once for every new item of information introduced. These items of information come from the nodes of the parse tree in the form of syntax restrictions and operand properties. Each node generates a maximum of one item for each property of each of the three positions surrounding it. No slow refinement occurs. Thus the number of steps of constraint propagation is of order $N*2$ where N is the size of the parse tree.

B.2 SYNTAX CONSTRAINTS

This section lists the syntax constraints and propagation rules used by the constraint propagation. In these rules we use two letter variable names to refer to properties. The first letter gives the property (T(ype), R(ank), L(ength), and V(alue)). The second gives the position (R(ight), L(eft), and A(bove)). The positions are in relation to the operator which establishes each set of rules.

B.2.1 Monadic Operators

B.2.1.1 Monadic Arithmetic Operations – + - × ÷ ⌈ ⌊ * ● | ! ○

```
TA := numeric
TR := numeric-or-boolean
RA := RR
LA := LR
if VR is constant then VA := op VR
RR := RA
LR := LA
```

B.2.1.2 Not – ~

```
TA := boolean
TR := boolean
RA := RR
LA := LR
if VR is constant then VA :=~ VR
RR := RA
LR := LA
```

B.2.1.3 Size – Monadic ρ

```
RA := 1
TA := numeric
LA := RR
VA := LR
RR := LA
```


B.2.1.4 Index Generator – monadic ι

```
RA := 1
TA := numeric
RR := 0
TR := numeric-or-boolean
LA := VR
```

B.2.1.5 Ravel

```
RA := 1
TA := TR
LA := ×/ LR
if RR < 2 then VA := VR
```

B.2.1.6 Reduction

```
TA := as required by operation
RR := >0
TR := as required by operation
RA := RR - 1
LA := LR with reduced dimension removed
RR := RA + 1
RL := RA for un-reduced dimensions
```

B.2.1.7 Scan

```
TA := as required by operator
RA := >0
RR := >0
RA := RR
TR := as required by operator
RR := RA
LA := LR
LR := LA
TA := TR
TR := TA
```

B.2.1.8 Reverse

```
RA := >0
RR := >0
RA := RR
TA := TR
LA := LR
```

```
RR := RA
TR := TA
LR := LA
```

B.2.2 Dyadic Operators

B.2.2.1 Dyadic Arithmetic Operators - + - × ÷ ⌈ ⌊ * ⍟ | ! ○

```
TA := numeric
TR := numeric-orboolean
TL := numeric-or-boolean
if RR > 0 then RA := RR else RA := RL
if RR > 0 then LA := LR
if RL > 0 then RA := RL
if RL > 0 then LA := LL
if RL = 0 and RR = 0 then LA := 1
if VR is constant and VL is constant then VA := VL op VR
if RR > 0 then RR := RA
if RR > 0 then LR := LA
if RL > 0 then RL := RA
if RL > 0 then LL := LA
```

B.2.2.2 Dyadic Logical Operators - ∧ ∨ ⍲ ⍱

```
TA := boolean
TR := boolean
TL := boolean
if RR > 0 then RA := RR else RA := RL
if RR > 0 then LA := LR
if RL > 0 then RA := RL
if RL > 0 then LA := LL
if RL = 0 and RR = 0 then LA := 1
if VR is constant and VL is constant then VA := VL op VR
if RR > 0 then RR := RA
if RR > 0 then LR := LA
if RL > 0 then RL := RA
if RL > 0 then LL := LA
```

B.2.2.3 Dyadic Equality Operators - = ≠

```
TA := boolean
TL := TR
TR := TL
if RR > 0 then RA := RR else RA := RL
if RR > 0 then LA := LR
if RL > 0 then RA := RL
if RL > 0 then LA := LL
```



```
if RL = 0 and RR = 0 then LA := 1
if VR is constant and VL is constant then VA := VL op VR
if RR > 0 then RR := RA
if RR > 0 then LR := LA
if RL > 0 then RL := RA
if RL > 0 then LL := LA
```

B.2.2.4 Dyadic Relational Operators - < ≤ ≥ >

```
TA := boolean
TR := numeric-or-boolean
TL := numeric-or-boolean
if RR > 0 then RA := RR else RA := RL
if RR > 0 then LA := LR
if RL > 0 then RA := RL
if RL > 0 then LA := LL
if RL = 0 and RR = 0 then LA := 1
if VR is constant and VL is constant then VA := VL op VR
if RR > 0 then RR := RA
if RR > 0 then LR := LA
if RL > 0 then RL := RA
if RL > 0 then LL := LA
```

B.2.2.5 Reshape - dyadic ρ

```
TA := TR
RA := LL
LA := LV
TR := TA
LL := RA
```

B.2.2.6 Catenation - dyadic ,

```
RA := >0
if RR > 0 then RA := RR
TA := TR
LA := LR except for dimension catenated
if RL > 0 then RA := RL
TA := TL
LA := LL except for dimension catenated
if RR = 0 and RL = 0 then RA := 1
LA := LL + LR for catenated dimension
if RL > 0 then RL := RA
TL := TA
if RR > 0 then RR := RA
TR := TA
```

B.2.2.7 Indexing – [ ]

```
RA := >0
RR := >0
TL := numeric-or-boolean
TA := TR
RA := +/ RL
LA := LL
RR := ≤ RA
TR := TA
LL := LA
RR := number of subscripts
```

B.2.2.8 Inner Product

```
TA := as required by left operator
TR := as required by right operator
TL := as required by right operator
if RR = 0 xor RL = 0 then RA := RR + RL - 1
if RR > 0 and RL > 0 then RA := RR + RL - 2
if RR = 0 and RL = 0 then RA := 0
LA := ¯1↓LL,1↓LR
if RR > 0 and RL > 0 then
      last position of LL := first position of LR
if RR > 0 and RL > 0 then
      first position of LR := last position of LL
```

B.2.2.9 Outer Product

```
RA := >0
TA := as required by operator
TR := as required by operator
TL := as required by operator
RA := RL + RR
LA := LL,LR
RR := <RA
RL := <RA
```

B.2.2.10 Take

```
RA := >0
RR := >0
RL := <2
TL := numeric-or-boolean
RA := RR
RA := LL
RR := RA
```



```
LA := | VL
LA := ≤LR
LL := RA
```

B.2.2.11 Drop

```
RA := >0
RR := >0
RL := <2
TL := numeric-or-boolean
RA := RR
RA := LL
RR := RA
LA := ≤LR
LL := RA
```

B.2.2.12 Transpose

```
RA := >1
RR := >1
RL := 1
TL := numeric
RA := ≤ RR
TA := TR
RA := maximum of VL
RR := ≥ RA
LL := RR
RR := LL
```

B.2.2.13 Rotate

```
RA := >0
RR := >0
TL := numeric-or-boolean
RA := RR
TA := TR
LA := LR
RR := RA
TR := TA
LR := LA
RL := RA - 1
RA := RL + 1
LL := LA without rotated dimension
LA except rotated dimension := LL
```

B.2.2.14 Compress

```
RA := >0
RR := >0
RL := 1
TL := boolean
RA := RR
RR := RA
TA := TR
TR := TA
LA := LR except for compressed dimension
LA := ≤ LR for compressed dimension
LR except for compressed dimension := LA
LR := ≥ LA for compressed dimension
LL := LR for compressed dimension
compressed dimension of LR := LL
```

B.2.2.15 Expand

```
RA := >0
RR := >0
TL := boolean
RL := 1
TA := TR
TR := TA
LA := LR for un-expanded dimensions
LA := LL for expanded dimension
unexpanded dimensions of LR := LA
LL := LA for expanded dimension
```

B.2.2.16 Index

```
TA := numeric
RL := 1
RA := RR
LA := LR
RR := RA
LR := LA
TL := TR
TR := TL
```

B.2.2.17 Membership

```
TA := boolean
RA := RL
LA := LL
RL := RA
```



```
LL := LA
TR := TL
TL := TR
```

B.2.2.18 Decode

```
TA := as required by left operator
TR := as required by right operator
TL := as required by right operator
if RR = 0 xor RL = 0 then RA := RR + RL - 1
if RR > 0 and RL > 0 then RA := RR + RL - 2
if RR = 0 and RL = 0 then RA := 0
LA := ¯1↓LL,1↓LR
if RR > 0 and RL > 0 then
      last position of LL := first position of LR
if RR > 0 and RL > 0 then
      first position of LR := last position of LL
```

B.2.2.19 Encode

```
TA := numeric
TR := numeric or boolean
TL := numeric or boolean
RA := RL + RR
LA := LL,LR
```

APPENDIX C

ARRAY ADDRESSING WITH LADDERS

This appendix describes the procedures used to calculate the array accessing parameters used by the ladders of the stream generators. It is largely a reformulation of similar presentations by Perlis [19] and particularly Minter [17].

C.1 ADDRESS SEQUENCING

The storage locations for the elements of an array are given by equation 3-1:

PI←BETA++/I×G (C-1)

The semantics of APL impose an ordering - ravel order - which orders array elements with right-most subscript varying most rapidly. This is the same order obtained by considering the subscripts as "digits" in a number and ordering the subscripts according to their value as single numbers. If we use equation 3-2 to calculate the numbers J, then:

J←RHO⊥I (C-2)

Since each subscript I[k] may legally range from 0 to RHO[k] independently, J may have values ranging from 0 to ¯1+×/*RHO*.

C.1.1 Storage Spacing - G

So that the APL operators which must access the array as a vector in ravel order do not require recopying of the data, we want to use values of BETA and G such that the equation:

PI←BETAR+J×GR (C-3)

will generate the same value of PI as equation C-1. If we set BETA to be equal to BETAR, then the equality required is:

(+/I×G)=J×GR (C-4)

Replacing J by the equation which calculates it value gives:

(+/I×G)=(RHO⊥I)×GR (C-5)

in which the Decode operation can be expanded according to its definition to give:

(+/I×G)=(+/I×⌽×\1,⌽1↓RHO)×GR (C-6)

If we apply the distributive law we obtain:

(+/I×G)=+/I×(GR×⌽×\1,⌽1↓RHO) (C-7)

which will obviously be satisfied if we set G as follows:

$$G \leftarrow GR \times \phi \times \backslash 1,\phi 1 \downarrow RHO \tag{C-8}$$

If GR is the number of addressable memory units used to store an item of the array (may not be smaller), then the array is stored in ravel order in consecutive locations.

C.1.2 Pointer Increment - DELTA

Figure 3-1 is a flowchart of the fixed part of a ladder. Each time control reaches the arc labeled "PI VALID", I will have a legal subscript value. All items of the initial I will be zero (the downwards path from the start box includes statements which clear I), and successive values will be in ravel order (the right-most subscript is advanced in the bottom loop and thus most rapidly). It is obvious from the flowchart that after execution of the arc labeled "PI VALID" only one horizontal rung will be executed before the bottom arc is re-entered. Therefore one item from the vector DELTA is added to PI at each step. Equation C-1 defines the relation we want to hold between I and PI each time control reaches the bottom of the ladder. A value for DELTA must be found which generates the required sequence.

If the horizontal rung last executed was at level k, then I[l] = 0 for l > k (I[k+1] is cleared on the downwards path from level k). If I[k] does not have its maximum value, then the next time the increment and test at level k is executed the horizontal branch at level k is executed. On return to the bottom of the ladder, I[l] = 0 for l > k, and I[j], (j < k) are unchanged. Only I[k], which has been increased by 1, is different. By equation C-1 the change in PI must equal G[k].

If we assume control has reached the bottom of the ladder after executing rung j, then I[j+1] is zero. Control will reach rung j again only when the test at level j+1 fails. This will occur on the RHO[j+1]st execution of the increment and test at level j+1. The first RHO[j+1]-1 times the test executes, it succeeds and the rung at level j+1 is executed. As we saw above, PI will have been incremented by G[j+1]. That increment consists of DELTA[j+1] plus the effect of lower levels which we will call LOW[k+1]. By this definition the equation

$$G = DELTA + LOW \tag{C-9}$$

holds. At the RHO[j+1]st execution of the increment and test at level j+1 the test fails and rung j executes resulting in a change to PI of DELTA[j] + LOW[j+1]. Since the total change to PI after the second execution of rung is shown above to equal G[j], the relation:

$$\begin{aligned} G[j] = {} & DELTA[j] + LOW[j+1] \\ & + G[j+1] \times {}^{-}1 + RHO[j+1] \end{aligned} \tag{C-10}$$

must hold. Using equation C-9 to eliminate LOW and simplifying we obtain a recurrence relation for DELTA:

$$\begin{aligned} DELTA[j] \leftarrow {} & G[j] + DELTA[j+1] \\ & - RHO[j+1] \times G[j+1] \end{aligned} \tag{C-11}$$

Since LOW[n] is obviously zero when n is the depth of the ladder, then

$$DELTA[n] \leftarrow G[n] \tag{C-12}$$

provides an initial value and we can calculate DELTA. A more rigorous derivation of this result is presented by Minter [17].

C.2 THE SELECTION OPERATORS

The operations Take, Drop, and Subscription by a vector of the form $A+B\times\iota C$ require only changes to the address generation parameters. The formulae for calculating the new values are given below.

C.2.1 Take

When the Take operation with T as the vector left operand is applied to an array, the position in the old array corresponding to subscript IT of the new array is given by:

$$I \leftarrow IT+(RHO-T)\times T<0 \tag{C-13}$$

The new addressing parameters must generate the same value of PI for each subscript IT as are generated using the I calculated above and the old values of BETA and G. This requires that:

$$(BETAT++/GT\times IT)=BETA++/G\times IT+(RHO-T)\times T<0 \tag{C-14}$$

be true for all legal IT. It will be true if:

$$\begin{array}{l} BETAT \leftarrow BETA++/G\times(RHO-T)\times T<0 \\ GT \leftarrow G \end{array} \tag{C-15}$$

are used to calculate the new values. The size of the result has also changed requiring:

$$RHOT \leftarrow |T \tag{C-16}$$

New values for DELTA are then calculated using the formulae derived in C.1.2.

C.2.2 Drop

When the Drop operation with D as the vector left operand is applied to an array, the position in the old array corresponding to subscript ID of the new array is given by:

$$I \leftarrow ID+0\lceil D \tag{C-17}$$

The new addressing parameters must generate the same value of PI for each subscript ID as are generated using the I calculated above and the old values of BETA and G. This requires that:

$$(BETAD++/GD\times ID)=BETA++/G\times ID\times 0\lceil D \tag{C-18}$$

be true for all legal ID. It will be true if:

$$\begin{array}{l} BETAD \leftarrow BETA++/G\times 0\lceil D \\ GD \leftarrow G \end{array} \tag{C-19}$$

are used to calculate the new values. The size of the result has also changed requiring:

$$RHOD \leftarrow RHO-|D \tag{C-20}$$

New values for DELTA are then calculated using the formulae derived in C.1.2.

C.2.3 Subscription

When the array is subscripted and the subscript in position k is given by A[k]+B[k]×ιC[k] (set A[k]←0, B[k]←1, and C[k]←RHO[k] for those dimensions not subscripted), the position in the old array corresponding to subscript IS of the new array is given by:

$$I \leftarrow A + B \times IS \qquad \text{(C-21)}$$

The new addressing parameters must generate the same value of PI for each subscript IS as are generated using the I calculated above and the old values of BETA and G. This requires that:

$$(BETAS + +/GS \times IS) = BETA + +/G \times A + B \times IS \qquad \text{(C-22)}$$

be true for all legal IS. It will be true if:

$$BETAS \leftarrow BETA + +/G \times A \qquad \text{(C-23)}$$
$$GS \leftarrow G \times B$$

are used to calculate the new values. The size of the result has also changed requiring:

$$RHOS \leftarrow C \qquad \text{(C-24)}$$

New values for DELTA are then calculated using the formulae derived in C.1.2.

C.3 RE-ORDERING

The other two operations which require only changes to the address generation parameters are transpose and reverse. The same transformations are used during the generation and improvement of stream generators to change the order in which array items are accessed.

C.3.1 Transpose

When the Transpose operation with TR as the vector left operand is applied to an array, the position in the new array corresponding to subscript I of the old array is given by:

$$ITR \leftarrow I[TR] \qquad \text{(C-25)}$$

The new addressing parameters must generate the same value of PI for each subscript ITR as are generated using I and the old values of BETA and G. This requires that:

$$(BETATR + +/GTR \times I[TR]) = BETA + +/G \times I \qquad \text{(C-26)}$$

be true for all legal I. It will be true if:

$$BETATR \leftarrow BETA \qquad \text{(C-27)}$$
$$GTR \leftarrow G[TR]$$

are used to calculate the new values. The size of the result has also changed requiring:

$$RHOTR \leftarrow RHO[TR] \qquad \text{(C-28)}$$

New values for DELTA are then calculated using the formulae derived in C.1.2.

C.3.2 Reverse

If the reversal operation is applied to an array, and if the boolean vector R is true for each position corresponding to a dimension which is reversed, the position in the old array corresponding to subscript IR of the new array is given by:

$$I \leftarrow IR + R \times RHO - 1 + 2 \times IR \qquad \text{(C-29)}$$

The new addressing parameters must generate the same value of PI for each subscript IR as are generated using I and the old values of BETA and G. This requires that:

$$(BETAR + +/GR \times IR) = BETA + +/G \times IR + R \times RHO - 1 + 2 \times IR \qquad \text{(C-30)}$$

be true for all legal I. It will be true if:

$$\begin{aligned} &BETAR \leftarrow BETA + +/G \times R \times RHO - 1 \\ &GR \leftarrow G - 2 \times R \times G \end{aligned} \qquad \text{(C-31)}$$

are used to calculate the new values. The size of the result has not changed so that:

$$RHOR \leftarrow RHO \qquad \text{(C-32)}$$

New values for DELTA are then calculated using the formulae derived in C.1.2.

C.4 STREAM GENERATORS

The procedure described in chapter 4 which creates stream generators from APL expressions was designed so that only the entry point of the graph could have more than one nesting son. It also produces a graph with all the header nodes at level 0. Thus each sub-graph hanging below the entry point contains only one or more (if connected by evocation edges) simple ladders with the address generation mechanism described

above.

C.4.1 Address Calculation

The labels on the stream generator graph nodes specify the addressing desired as follows:

1. If a node is labeled with a pointer movement label (A_i or $A_i\cdot$) the value of G assigned to that node will be that given for the i^{th} dimension of A.

2. If a node is labeled with a pointer reset or address calculation label, the corresponding item of G is zero.

3. If the array name in a label is modified by ϕ the corresponding item of G is negated.

4. The values of RHO to be used in the calculation of BETA and DELTA are taken from the length of the array dimension specified by the label (not from loop limit label).

The calculation of BETA and DELTA proceed as specified earlier.

C.4.2 Re-ordering

The sub-graphs of the entry node of a stream generator communicate only via registers (scalars) and memory (arrays). Each sub-graph executes to completion before the next is started. Thus the order that calculated values are placed into storage is not important. Only the final position in memory of each item matters.

The transposition and reversal operations described in chapter 4 are applied to all arrays being accessed by that ladder. It is obvious from the preceeding sections that elements of two arrays with the same I will have the same ITR or IR after re-ordering. Thus for those operator which calculate each element independently, the current operand values contributing to a given position in the result will be the same after re-ordering.

Of those APL operations which are compiled into stream generators the ones for which each position is not independent are reduction and scan (and others defined in terms of those which include inner-product, encode, decode, and membership). Both require sequential access along one dimension in a fixed direction, and accumulate a value. When these operators are translated into stream generators, temporary storage for the running values is provided. Thus the dimension being reduced or scanned may be at any level (ladder may be transposed), but it may not be reversed. The definition of reversal does not permit that operation to be applied to a graph node labeled (/ or \) as being reduced or scanned. Therefore legal re-orderings will not affect final array content.

APPENDIX D

STREAM GENERATORS AS IMP-10

The listing below is the (edited) listing output of the IMP-10 compiler produced during the final compilation into machine code of the stream generator description representing the APL expression F←A+-/(-/B+C)+.--/D+E . A complete description of the language and the extension mechanism may be found in [5].

```
  IMP 1.6     7-SEP-74      STREAM.110[50,130]      15-JUL-77   10:30

0 #
0 ********** SYNTAX EXTENSIONS ********************************
0
0  LADDER LABEL, EVOKE STATEMENT, AND CO-ROUTINE INITIALIZATION
0 #
0 !$CURLD! IS REGISTER, RESERVED;
0 !$LNKRG! IS 33 LONG;  # 32 LADDERS POSSIBLE#
0 <ST>::=LADDER <EXP,A>':'<ST,B>
0     ::=LOCAL SL,EL IN "!$LNKRG![A]_LOC(SL);
0                         GO TO EL;SL:B;
0                         GO TO SL;
0                         EL:0";
0 <ST>   ::= EXIT  ::= "DATA(047000000012B)";
0 <ST>::=EVOKE <EXP,A>
0     ::=LOCAL L IN "!$LNKRG![!$CURLD!]_LOC(L);
0                    !$CURLD!_A;
0                    GO TO [!$LNKRG![!$CURLD!]];
0                    L:0";
0 <ST>::=BEGIN::=LOCAL EL IN
0                 "!$CURLD!_0;GO TO EL;
0                  !$INTER!:EVOKE 1;EXIT;
```



```
0                       EL:0";
0 <ST>::=END::="GO TO !$INTER!";
0 #
0   INIT STATEMENT
0 #
0 <PTL>::=<EXP,B>::="PI[B]_BETA[B]";
0 <PTL>::=<PTL,A>,<EXP,B>::="A;PI[B]_BETA[B]";
0 <ST>::=INIT <PTL,A>::="A";
0 #
0   MATRICES  WITH 5 COLUMNS
0 #
0  <ST>    ::= <VBL,A> IS <EXP,I> BY 5
0          ::= "A IS I*5 LONG";
0  <VBL>   ::= <VBL,A>[<EXP,I>,<EXP,J>]
0          ::= "A[IOCON,JOREG]" => "A[5*I][J]" ELSE
0                "A[I,JOCON]" => "A[J][I*5]"
0                 ELSE "A[J+I*5]";
0 #
0   REPEAT STATEMENT
0 #
0 !$LEV! IS REGISTER, RESERVED;
0 <INCL>::=MOVING <EXP,B>
0       ::="PI[B]_PI[B]+DELTA[B,!$LEV!]";
0 <INCL>::=<INCL,A>,<EXP,B>
0       ::="A;PI[B]_PI[B]+DELTA[B,!$LEV!]";
0 <ST>::=REPEAT <ST,A> AT <EXP,B>
0        USING <EXP,C>
0        <INCL,E>
0     ::= LOCAL RL IN
0           "I[C,B]_0;
0            RL:A;!$LEV!_B;
0            (I[C,!$LEV!]_I[C,!$LEV!]+1) LT RHO[C,!$LEV!]->
0                                                  (E;GO TO RL)";
0 <ST>::=REPEAT <ST,A> AT <EXP,B>
0        USING <EXP,C>
0     ::= LOCAL RL IN
0           "I[C,B]_0;
0            RL:A;!$LEV!_B;
0            (I[C,!$LEV!]_I[C,!$LEV!]+1) LT RHO[C,!$LEV!]->
0                                                     GO TO RL";
0 #
0 *********** LOCAL STORAGE ***************
0 #
0 T IS 17 LONG;  #16 REGISTERS#
0 PI IS 17 LONG;#16 POINTERS#
0 I IS 17 BY 5; #16 INDICES - 4 LEVELS#
0 #  (THOSE BELOW ACTUALLY DEFINED BY DATA STATEMENTS)
0 BETA IS 17 LONG;  16 POINTER INITIAL VALUES
0 DELTA IS 17 BY 5;   16 INCREMENTS - 4 LEVELS
0 RHO IS 17 BY 5;     16 LIMITS - 4 LEVELS
0
0 ********* ARRAY STORAGE *****************
0 #
```

```
 0 A,F ARE  5 LONG;
 0 B,C,D,E ARE 125 LONG;
 0 /
 0 ********* CONTROL PARAMETERS ************
 0 /
 0 REMOTE(
 1  BETA: DATA(0,A,B[104],C[104],D[124],E[124],F);
 1  RHO:  DATA(0,5,5,5,5,
 2             0,0,0,0,5);
 1  DELTA:DATA(0,1,0,0,0,
 2             0,34,29,-21,-1,
 2             0,34,29,-21,-1,
 2             0,124,-1,-1,-1,
 2             0,124,-1,-1,-1,
 2             0,1,0,0,0));
 0 /
 0 ********* CO-ROUTINE INITIALIZATION *****
 0 /
 0 BEGIN;
 0 /
 0 *****   STREAM GENERATOR CODE ********************
 0 /
 0 /
 0  APL - F_A+-/(-/B+C)+.--/D+E
 0 /
 0 LADDER 1:(INIT 1,2,3,4,5,6;
 1           REPEAT(T[1]_0;
 2                  REPEAT(T[2]_0;
 3                         REPEAT(T[3]_0;
 4                                REPEAT(T[3]_([PI[4]]+[PI[5]])-T[3])
 4                                AT 4 USING 1
 4                                MOVING 4, 5;
 4                                T[4]_0;
 4                                REPEAT(T[4]_([PI[2]]+[PI[3]])-T[4])
 4                                AT 4 USING 2
 4                                MOVING 2, 3;
 4                                T[2]_(T[4]-T[3])+T[2])
 3                         AT 3 USING 1
 3                         MOVING 2, 3, 4, 5;
 3                         T[1]_T[2]-T[1])
 2                  AT 2 USING 1
 2                  MOVING 2, 3, 4, 5:
 2                  [PI[6]]_[PI[1]]+T[1])
 1           AT 1 USING 1
 1           MOVING 1, 2, 3, 4, 5, 6;
 1           EVOKE 0);
 0 /
 0 *****************************************************
 0 /
 0 END%%%%%%%%%%%%%%

** CODE PRODUCED BY PROGRAM STREAM
```



```
000002  $INTER
000006  %L3
000007  %EL2
000012  %SL15
000027  %RL12
000031  %RL10
000033  %RL8
000035  %RL4
000051  %IF5
000053  %RL6
000067  %IF7
000107  %IF9
000126  %IF11
000152  %IF13
000155  %L14
000157  %EL16


000000        MOVEI   1,0
000001        JRST    %EL2
        $INTER:
000002        MOVEI   3,%L3
000003        MOVEM   3,$LNKRG(1)
000004        MOVEI   1,1
000005        JRST    @$LNKRG(1)
        %L3:
000006        CALLI   12
        %EL2:
000007        MOVEI   3,%SL15
000010        MOVEM   3,$LNKRG+1
000011        JRST    %EL16
        %SL15:
000012        MOVE    3,BETA+1
000013        MOVEM   3,PI+1
000014        MOVE    4,BETA+2
000015        MOVEM   4,PI+2
000016        MOVE    5,BETA+3
000017        MOVEM   5,PI+3
000020        MOVE    6,BETA+4
000021        MOVEM   6,PI+4
000022        MOVE    7,BETA+5
000023        MOVEM   7,PI+5
000024        MOVE    10,BETA+6
000025        MOVEM   10,PI+6
000026        SETZM   I+6
        %RL12:
000027        SETZM   T+1
000030        SETZM   I+7
        %RL10:
000031        SETZM   T+2
000032        SETZM   I+10
        %RL8:
000033        SETZM   T+3
```

```
000034         SETZM   I+11
        %RL4:
000035         MOVE    3,@PI+5
000036         ADD     3,@PI+4
000037         SUBB    3,T+3
000040         MOVEI   2,4
000041         AOS     4,I+5(2)
000042         CAML    4,RHO+5(2)
000043         JRST    %IF5
000044         MOVE    4,DELTA+24(2)
000045         ADDB    4,PI+4
000046         MOVE    5,DELTA+31(2)
000047         ADDB    5,PI+5
000050         JRST    %RL4
        %IF5:
000051         SETZM   T+4
000052         SETZM   I+16
        %RL6:
000053         MOVE    3,@PI+3
000054         ADD     3,@PI+2
000055         SUBB    3,T+4
000056         MOVEI   2,4
000057         AOS     4,I+12(2)
000060         CAML    4,RHO+12(2)
000061         JRST    %IF7
000062         MOVE    4,DELTA+12(2)
000063         ADDB    4,PI+2
000064         MOVE    5,DELTA+17(2)
000065         ADDB    5,PI+3
000066         JRST    %RL6
        %IF7:
000067         MOVE    3,T+4
000070         SUB     3,T+3
000071         ADDB    3,T+2
000072         MOVEI   2,3
000073         AOS     4,I+5(2)
000074         CAML    4,RHO+5(2)
000075         JRST    %IF9
000076         MOVE    4,DELTA+12(2)
000077         ADDB    4,PI+2
000100         MOVE    5,DELTA+17(2)
000101         ADDB    5,PI+3
000102         MOVE    6,DELTA+24(2)
000103         ADDB    6,PI+4
000104         MOVE    7,DELTA+31(2)
000105         ADDB    7,PI+5
000106         JRST    %RL8
        %IF9:
000107         MOVE    3,T+2
000110         SUBB    3,T+1
000111         MOVEI   2,2
000112         AOS     4,I+5(2)
000113         CAML    4,RHO+5(2)
```



```
000114         JRST    %IF11
000115         MOVE    4,DELTA+12(2)
000116         ADDB    4,PI+2
000117         MOVE    5,DELTA+17(2)
000120         ADDB    5,PI+3
000121         MOVE    6,DELTA+24(2)
000122         ADDB    6,PI+4
000123         MOVE    7,DELTA+31(2)
000124         ADDB    7,PI+5
000125         JRST    %RL10
        %IF11:
000126         MOVE    3,T+1
000127         ADD     3,@PI+1
000130         MOVEM   3,@PI+6
000131         MOVEI   2,1
000132         AOS     3,I+5(2)
000133         CAML    3,RHO+5(2)
000134         JRST    %IF13
000135         MOVE    3,DELTA+5(2)
000136         ADDB    3,PI+1
000137         MOVE    4,DELTA+12(2)
000140         ADDB    4,PI+2
000141         MOVE    5,DELTA+17(2)
000142         ADDB    5,PI+3
000143         MOVE    6,DELTA+24(2)
000144         ADDB    6,PI+4
000145         MOVE    7,DELTA+31(2)
000146         ADDB    7,PI+5
000147         MOVE    10,DELTA+36(2)
000150         ADDB    10,PI+6
000151         JRST    %RL12
        %IF13:
000152         MOVEI   3,%L14
000153         MOVEM   3,$LNKRG(1)
000154         MOVEI   1,0
000155         JRST    @$LNKRG(1)
        %L14:
000156         JRST    %SL15
        %EL16:
000157         JRST    $INTER
               END
```

112 WORDS OBJECT CODE

APPENDIX E

STREAM GENERATORS AS LADDER MACHINE CODE

This appendix describes the version of the ladder machine upon which were based the code size and speed estimates of Chapter 5. The original design by Minter [17] was for a machine to execute the ladder control structure defined in Section 3.2. The design sketched below incorporates changes reflecting the additional features added to the ladder structure (Section 3.3).

The local memory of the ladder machine is organized as 32 bit words. The first 1144 words are reserved for the control variables used by the compiler object code: (Section 3.5)

```
   0 -  255  is  T[register #]
 256 -  511  is  DELTA[pointer #,level #]
 512 -  767  is  G[pointer #,level #]
 768 - 1023  is  I[index #,level #]
1024 - 1279  is  RHO[index #,level #]
1080 - 1191  is  PI[pointer #]
1112 - 1143  is  BETA[pointer #]
```

with $0 \le$ register # ≤ 255
$0 \le$ pointer # ≤ 31
$0 \le$ index # ≤ 31
$0 \le$ level # ≤ 7

The remainder holds instructions.

The instructions executed by the ladder machine are each single words (32 bits) and consist of:

1. Two address instructions for each of the APL monadic scalar operators.

   MOP register $\#_1$,register $\#_2$

2. Three address instructions for each of the APL dyadic scalar operators.

   DOP register $\#_1$,register $\#_2$,register $\#_3$

3. Main memory load and store (indirect using pointers PI).

   ```
   GET  register #,pointer #
   PUT  register #,pointer #
   ```

4. Local memory "load" and "store" (transfer to or from T).

   ```
   LOAD register #,address
   STO  register #,address
   ```

5. Register clear.

   CLR register #

6. Unconditional branch

```
BR    address
```

7. Conditional branch

```
(T[register #] => GOTO address)

BNE  register #,address
```

8. Pointer initialization

```
(PI[pointer #]BETA[pointer #])

INIT pointer #
```

9. Index increment and test

```
(I[index #,level #]I[index #,level #]+1;
 I[index #,level #] GE RHO[index #,level #]
               => GOTO address).

INC  index #,level #,address
```

10. Pointer increment

```
(PI[pointer #]PI[pointer #]+DELTA[pointer #,level #]).

STEP pointer #,level #
```

11. Co-routine evocation.

```
EVOKE ladder #
```


The op-code is is 8 bits wide and addresses occupy 16 bits.

The stream generator for $F{\leftarrow}A+-/(-/B+C)+.--/D+E$ is given by:

```
LADDER 1:(INIT 1,2,3,4,5,6;
          REPEAT(T[1]_0;
                 REPEAT(T[2]_0;
                        REPEAT(T[3]_0;
                               REPEAT(T[3]_((PI[4])+(PI[5]))-T[3])
                               AT 4 USING 1
                               MOVING 4,5;
                               T[4]_0;
                               REPEAT(T[4]_((PI[2])+(PI[3]))-T[4])
                               AT 4 USING 2
                               MOVING 2,3;
                               T[2]_(T[4]-T[3])+T[2])
                        AT 3 USING 1
                        MOVING 2,3,4,5;
                        T[1]_T[2]-T[1])
                 AT 2 USING 1
                 MOVING 2,3,4,5;
                 (PI[6])_(PI[1])+T[1])
          AT 1 USING 1
          MOVING 1,2,3,4,5;
      EVOKE 0);
```

This translates into the ladder code given below. We have added labels to this listing to permit symbolic addresses.

```
LOOP0:  INIT  1
        INIT  2
        INIT  3
        INIT  4
        INIT  5
        INIT  6
        CLRI  1,1
LOOP1:  CLR   1
        CLRI  1,2
LOOP2:  CLR   2
        CLRI  1,3
LOOP3:  CLR   3
        CLRI  1,4
LOOP4:  GET   5,4
        GET   6,5
        +     5,5,6
        -     3,5,3
        INC   1,4,SKIP1
        STEP  4
```

```
        STEP  5
        BR    LOOP4
SKIP1:  CLR   4
        CLRI  2,4
LOOP5:  GET   5,2
        GET   6,3
        +     5,5,6
        -     4,5,4
        INC   2,4,SKIP2
        STEP  2
        STEP  3
        BR    LOOP5
SKIP2:  -     3,4,5
        +     2,3,2
        INC   1,3,SKIP3
        STEP  2
        STEP  3
        STEP  4
        STEP  5
        BR    LOOP3
SKIP3:  -     1,2,1
        INC   1,2,SKIP4
        STEP  2
        STEP  3
        STEP  4
        STEP  5
        BR    LOOP2
SKIP4:  GET   7,1
        +     7,7,1
        PUT   7,6
        INC   1,1,SKIP5
        STEP  1
        STEP  2
        STEP  3
        STEP  4
        STEP  5
        STEP  6
SKIP5:  EVOKE 0
        BR    LOOP0

        *** 59 instructions ***
```

The 32 bit instructions would permit a STEP instruction to specify up to 4 pointer #'s. However, all ladder code instruction counts in this thesis assume the form of step used above.

APPENDIX F

STREAM GENERATORS FOR THE APL OPERATORS

F.1 DEFINITIONS

The procedure the compiler uses to handle each of the APL operators which is compiled is given together with an example for each. The short APL fragments which identify the idioms use the convention that S is a scalar and V a vector. In these procedures we will use the term "problem node" to refer to a node with any of the following properties:

1. One or more of the labels has been modified by \.

2. The node is a choice node.

3. The node is part of a raveled structure.

The name was chosen because such nodes prevent the direct application of several operations.

F.2 THE OPERATORS

F.2.1 Monadic Operators

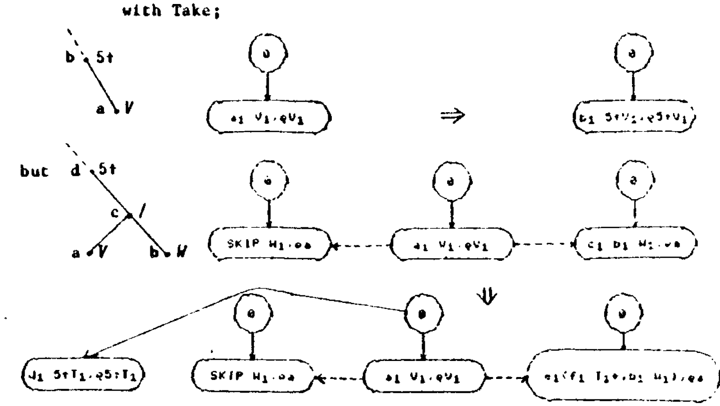
When a monadic branch node (monadic operator) is traversed, the graph of the result is built from that for the operand as follows:

F.2.1.1 Scalar Operation

No change is made to the graph for the operand. Code for the operation is built into the splice code at the point(s) where values are available.

F.2.1.2 Take

```
IF problem nodes are active at dimension affected
  THEN (Assign operand into temporary array;
        /* In-line Assignment demon acts here */
        apply take to new operand)
  ELSE modify labels of active nodes of operand at level affected
       with Take;
```

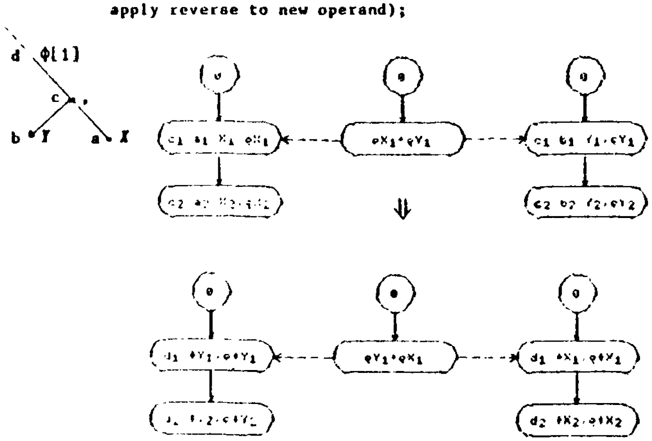


F.2.1.3 Drop

```
IF problem nodes are active at dimension affected
  THEN (Assign operand into temporary array;
        /* In-line Assignment demon acts here */
        apply drop to new operand)
  ELSE modify labels of active nodes of operand at level affected
       with Drop;
```

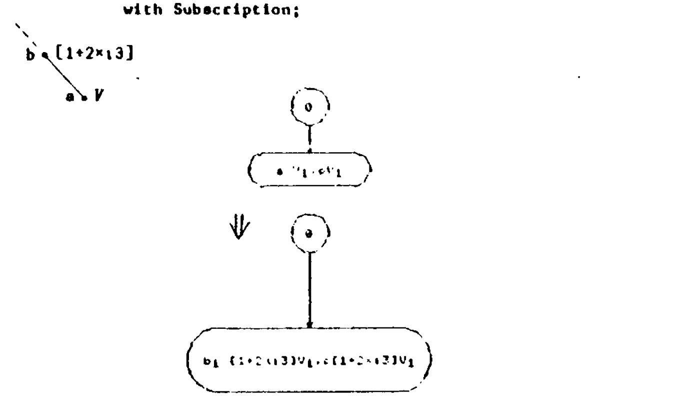
F.2.1.4 Reverse

```
execute Reverse command;
IF FAILURE
  THEN (Assign operand into temporary array;
        /* In-line Assignment demon acts here */
        apply reverse to new operand);
```



F.2.1.5 Subscription

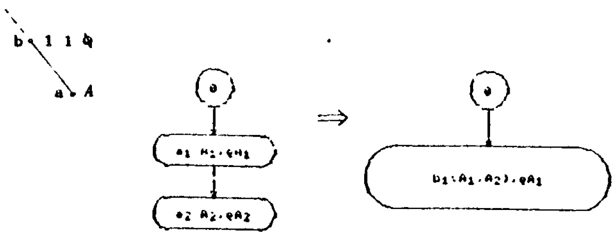
```
IF problem nodes are active at dimension affected
  THEN (Assign operand into temporary array;
        /* In-line Assignment demon acts here */
        apply subscription to new operand)
  ELSE modify labels of active nodes of operand at level affected
       with Subscription;
```



F.2.1.6 Transposition

```
execute Transpose command;
IF diagonalization is specified /* now adjacent */
  THEN (IF problem nodes are active at levels affected
          THEN (Assign operand into temporary array
                /* In-line Assignment demon acts here */);
        collapse affected levels of each ladder);
```

In a similar fashion to the requirement of the re-order operation, the collapse procedure assumes the absence of multiple nesting. It will preserve that property. Since no active node has an inactive son, the collapse operation will reduce the depth of each ladder in the active sub-graph of the entry point uniformly by one.

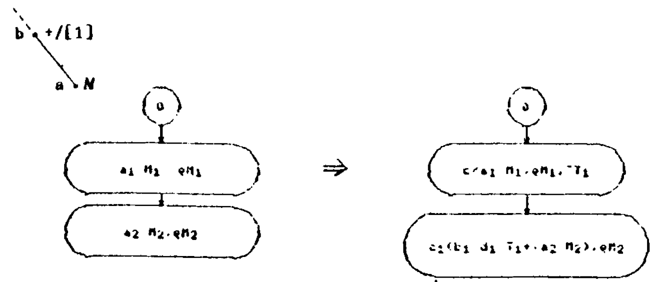


F.2.1.7 Reduction

```
build a simple ladder with shape of operand;
label dimensions not being reduced to assign to a temporary;
label reduced dimension with /;
execute Merge command between new ladder and operand;
```

For a non-associative operation, the reduced dimension must be reversed or alternate operations performed. (For associative operation this dimension my be altered to match delivery.)

The reduction operation will be most efficient if the the reduction is applied to the last dimension. The temporary storage is not accessed repeatedly, and thus its elimination may be possible.

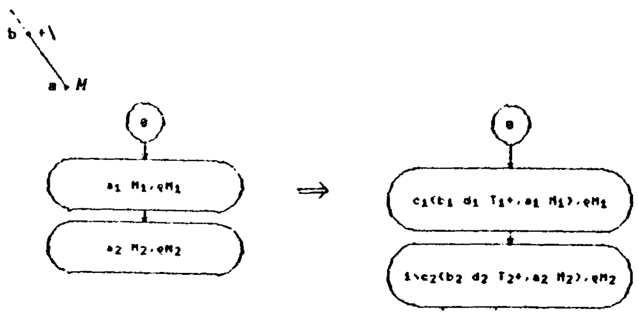


F.2.1.8 Scan

build a simple ladder with shape of operand;
label new ladder to assign to a temporary;
label all levels with 1\;
execute Merge command between new ladder and operand;

We require that all data scanned be in the range of the operator as well as in its domain. Non-associative operations require special handling [16].

The scan operation will be most efficient if the scan is applied to the last dimension. The temporary storage is not accessed, and thus its elimination may be possible.

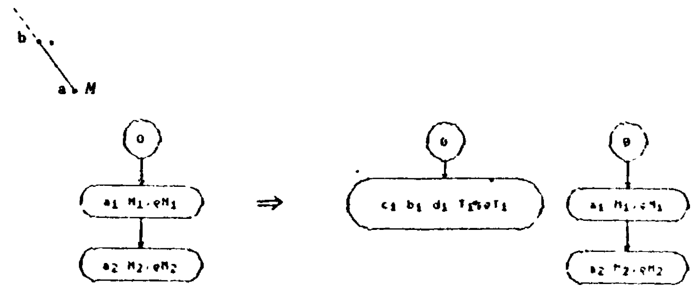


F.2.1.9 Iota

add a new (left-most) nesting son to the header
 of the graph for the scalar operand;
label the new node as ριs,ιs and as the new result;

F.2.1.10 Ravel

change nesting edges connecting active nodes at
  levels affected to raveled nesting edges;
build a new ladder with the shape of the result and
  label it to assign to a temporary and as the result;
create a copy of the new ladder for each result ladder
  in the operand and Adjust to fit;
connect an evocation edge from
the lowest active node in each operand result ladder
  to the lowest active node in the matching copy of the new ladder;

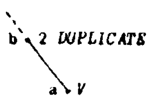


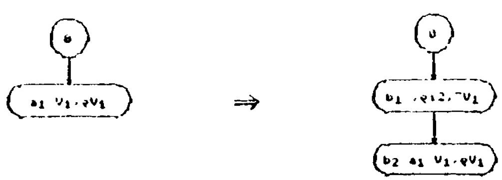
F.2.1.11 Shape

Most uses of shape become modifiers of monadic operations. If this has not occurred, the set-up code will store the vector, and the stream generator is a simple reference to that new vector. The constraint propagation phase will have subdivided the function so that the information is available when the generator is started. If the operand is not a variable used elsewhere, then only those values which determine size need be calculated (may be none).

F.2.1.12 Duplication

```
create a new one level ladder with limit
 equal to duplication factor;
Nest operand under new ladder;
```

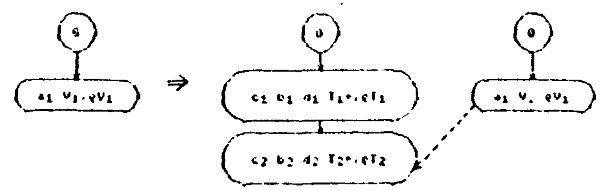




F.2.1.13 Reshape

```
build a new ladder with the shape of the result and
  label it to assign to a temporary and as the result;
create a copy of the new ladder for each result ladder
  in the operand and Adjust to fit;
connect an evocation edge from
the lowest active node in each operand result ladder
  to the lowest active node in the matching copy of the new ladder;
Neither sub-graph may be altered during later processing.
```

(reshaping *V* to have shape of *T*)



F.2.1.14 Scalar Creation

Splice code in the entry point will fetch the operand into a register. If the operand of this operation is a parse tree leaf (or 1↑*V*), then the fetch will be done by the interpreter. All scalar operands and intermediate results are kept in registers.

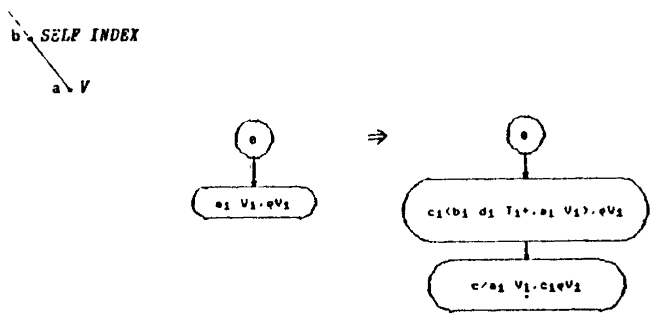
F.2.1.15 Boolean Creation

The graph for the operand is unchanged. Splice code to check value range will be added to the point where values are available.

F.2.1.16 Self Indexing - $V \iota V$

Nest the graph for the operand under a copy of itself;
change limit label of lower copy to $e\rho V_1$;

Reduce the last dimension;



F.2.1.17 Extremum Position - $V\iota\lceil/V$ or $V\iota\lfloor/V$

The graph is the same as for the reduction of V.

F.2.1.18 Span - $\lceil/V\circ.-V$

The graph is the same as for the reduction of V.

F.2.1.19 Rank - $\rho\rho A$

Rank is a scalar constant computed at compile time.

F.2.1.20 Indices Of Array - $\iota(\rho A)[I]$

The graph is a single active node labeled $\iota\rho A_1 \cdot \rho A_1$.

F.2.1.21 Scalar To Vector

create a one level graph to reference the vector;
attach it as the left-most nesting son of the entry point of
of the graph for the right operand;

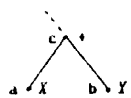
The remainder of the monadic operators are treated as function calls.

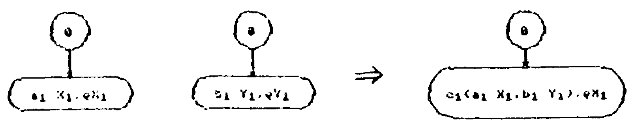
F.2.2 Dyadic Operators

The graph for the result of a dyadic operation is built up from those of the operands as follows:

F.2.2.1 Scalar Operation

```
execute Merge command;
IF FAILURE
  THEN (Assign operand causing failure to temporary;
        /* In-line Assignment demon acts here *\
        apply scalar operation)
  ELSE build actual calculations into splice code;
```



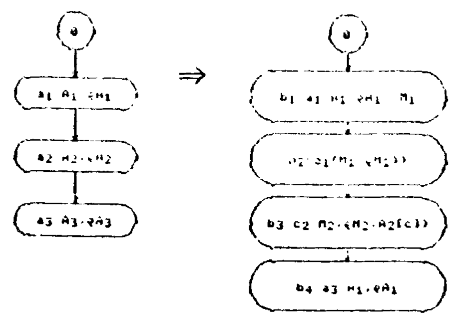


F.2.2.2 Subscription

```
IF subscript is an ι vector
  THEN (attach right operand to the header of
          the left operand as a new left-most nesting son;
        apply monadic subscription)
  ELSE (IF active nodes of right operand are problem nodes
         THEN (Assign right operand to a temporary;
               /* In-line Assignment demon acts here *\
               subscript new right operand)
         ELSE (Nest left operand under right operand;
               Transpose to bring subscript adjacent to
                 level subscripted;
               remove nodes being subscripted;
               FOR each array reference removed DO
                    add a subscription label to
                    the lowest node of the subscript);
```

If the result of a subscription is re-ordered, the subscription labels will have to be moved to the lowest node labeled as generating the subscript.

(A[;M;])



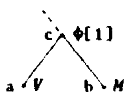
F.2.2.3 Rotation

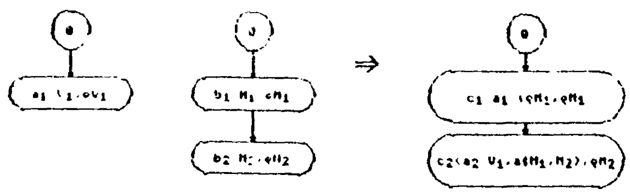
```
create a one level ladder for ιρR_r;

     /* R_r is rotated dimension of right operand *\

Nest it under left operand;
Transpose to put new node at level of rotation;
IF active nodes at the rotated level of the right operand
      are problem nodes
   THEN (Assign right operand to temporary
         /* In-line Assignment demon acts here *\);
execute Merge command;
IF FAILURE
  THEN (Assign operand causing failure to temporary;
        /* In-line Assignment demon acts here *\
        execute Merge command);
remove pointer movement labels for rotated dimension
  of right operand;
FOR each label removed DO add a rotation label to the
                            lowest node of left operand;
```

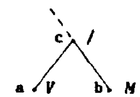
If the result of a rotation is re-ordered, the rotation labels must be moved to the lowest node of the right operand.

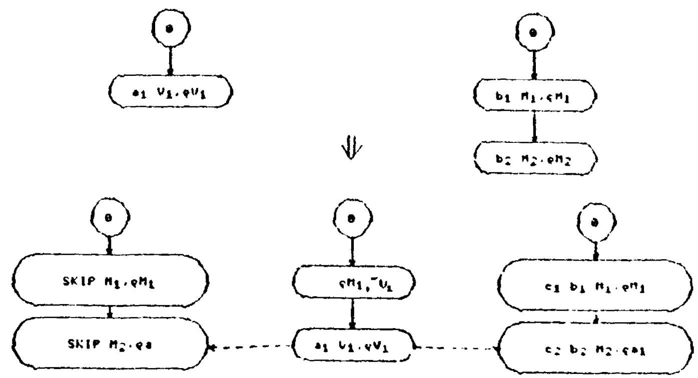




F.2.2.4 Compress

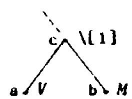
```
IF right operand has choice node at level
   of compression and one alternative is inactive
  THEN (Assign right operand to a temporary
        /* In-line Assignment demons acts here *\);
copy the right operand;
label one copy as the result of the compression;
add the modifier SKIP to all
  (except header nodes) of the other;
use the left operand to select between
  two Alternatives which are the two copies;
change the limit labels of the target nodes
  to be ρl_1; /* l labels left operand *\
```

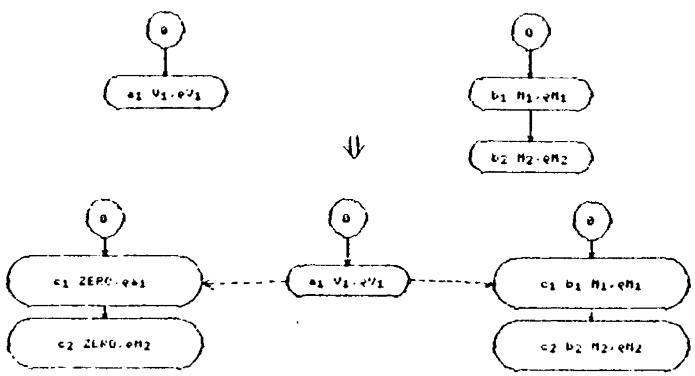




F.2.2.5 Expand

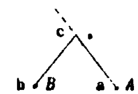
```
IF right operand has choice node at level
   of expansion and one alternative is inactive
  THEN (Assign right operand to a temporary
        /* In-line Assignment demons acts here *\);
build a new ladder with shape of right operand;
IF right operand is numeric
  THEN label new ladder with ZERO
  ELSE label new ladder with BLANK;
label right operand and new ladder as result;
use the left operand to select between
  two Alternatives which are the right operand and
  the new ladder;
change the limit label of the target nodes in the new
  ladder to be ρl₁; /* l labels left operand *\
```

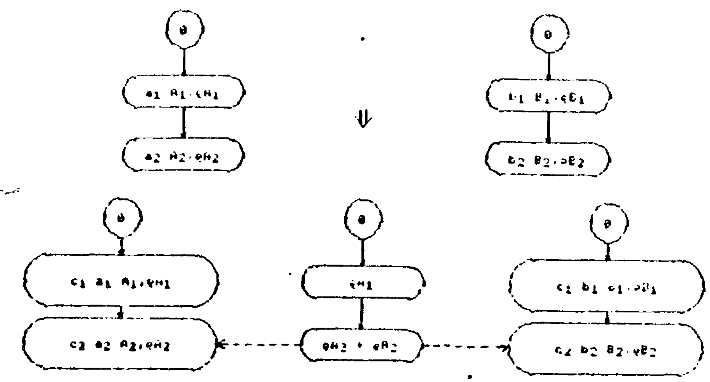




F.2.2.6 Catenation

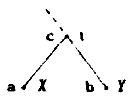
```
IF right operand has choice node at level
   of compression and one alternative is inactive
  THEN (Assign right operand to a temporary
        /* In-line Assignment demons acts here *\);
IF left operand has choice node at level
   of compression and one alternative is inactive
  THEN (Assign left operand to a temporary
        /* In-line Assignment demons acts here *\);
build a new one level ladder with limit equal to the
  sum of the lengths of the catenated dimension;
the new ladder selects between the right and left
  operands as Alternatives at level of catenate;
```

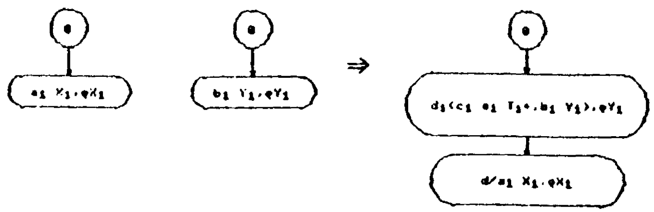




F.2.2.7 Index

Nest left argument under right operand;
Reduce each dimension derived from left operand;



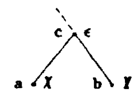


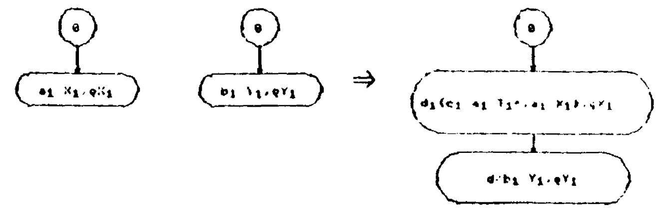
F.2.2.8 Membership

Nest right operand under left operand;
Reduce each dimension derived from right operand;

(Inactive nodes may be re-ordered here and in other multiple

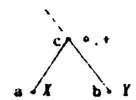
reductions using the same associative, commutative operation.)

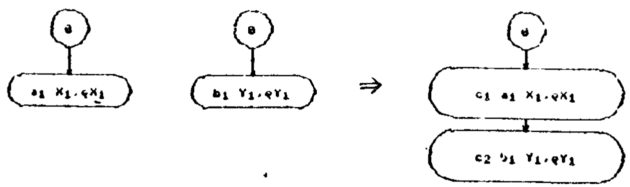




F.2.2.9 Outer Product

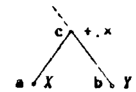
Nest the right operand under the left operand;
label both results as result of operation;

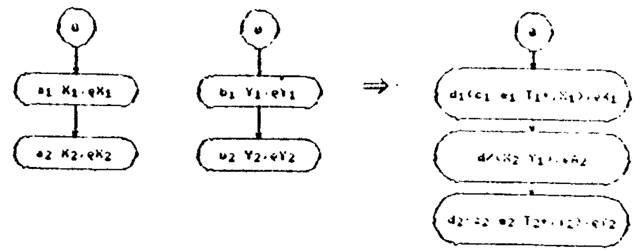




F.2.2.10 Inner Product

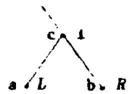
perform Outer Product;
Diagonalize the last dimension of left operand
with first dimension of right operand;
Reduce resulting dimension;

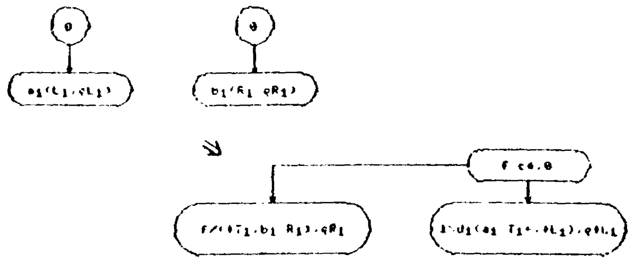




F.2.2.11 Decode

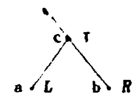
Reverse the last dimension of the left operand;
Scan the last dimension of the left operand;
Reverse the last dimension of the left operand;
perform Inner Product;

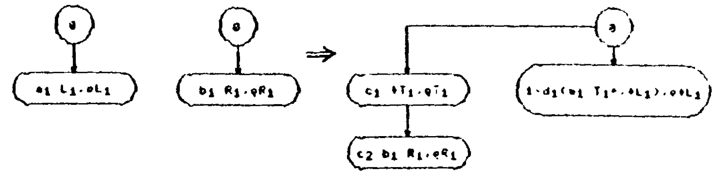




F.2.2.12 Encode

Reverse the last dimension of the left operand;
Scan the last dimension of the left operand;
Reverse the last dimension of the left operand;
perform Outer Product;



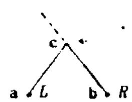


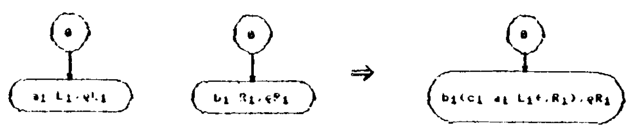
F.2.2.13 Assignment

```
relabel the result nodes of the single ladder
  for the left operand to assign to the single
  array it references;
label left operand as the result;
Merge the right and left operands;
```

The In-line Assignment demon will not be applied before assignment. If the assignment is to a whole array (replacement), the loop limit labels of the left operand will be ignored. If the assignment is to a sub-array (left operand includes transposition, subscription, or simple selection), then limits must match.

If a graph is created which assigns the same value to more than one array, temporary storage arrays will be eliminated until only one copy of the data is stored. Result labels will be transferred.

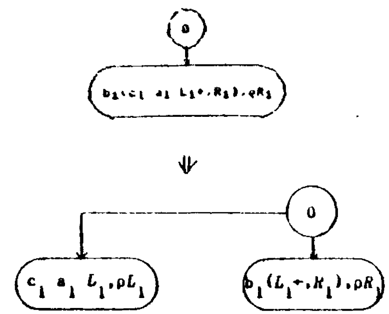




Note that the labels from the left operand are placed so as to be modified by the result labels of the right operand. If L is a temporary which is subsequently assigned to a different array, the labels c_1 a_1 L_1← would be replaced by those for the new assignment.

When the In-line Assignment demon is applied after an assignment, the actions are:

```
build a simple ladder with shape of stored result;
label to reference that array;
transfer modifiers originally from left operand
                    of assignment to new ladder;
            /* c and a in above *\
Overlay entry points (new ladder on the left)
```



F.2.2.14 Reshape

Dyadic reshape is not compiled, so no graph is ever created. The interpreter will remove extra elements or create duplicates from the stored operand (and if necessary copy the stored operand into actual ravel order).

F.2.2.15 Transposition

The right operand becomes the successor of the left operand. If the right operand is a single ladder, and if the transposition does not specify diagonalization (this will be known due to rank constraints), then the interpreter can apply the equivalent of monadic transpose to that ladder by changing address sequence parameters. Otherwise, the right operand must be assigned to a temporary, and monadic transposition applied.

F.2.2.16 Take And Drop

The right operand becomes the successor of the left operand. The monadic operation is then applied to the right operand.

F.2.2.17 Duplication

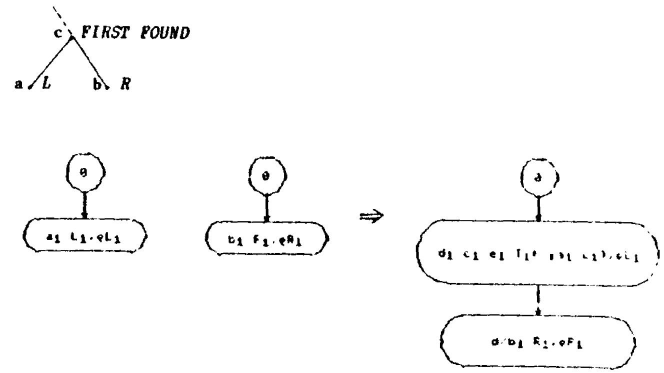
The right operand becomes the successor of the left operand. The monadic operation is then applied to the right operand.

F.2.2.18 End-around - ¯1⌽V,S or 1⌽S,V

The graph is equivalent to catenation in reverse order.

F.2.2.19 First-found - ⌊/V1⍳V2

Nest V2 under V1;
Reduce lower level;



F.2.2.20 Bounded Extremum - ⌊/S,V or ⌈/S,V

The graph is the same as for reduction of V.

F.2.2.21 Take-till - (V⍳S)↑V

The graph is that produced for V/V with the choice node modified by \. The header is labeled with the parse tree lablel for the leaf S.

F.2.2.22 Delay - *S*,¯1↓*V* or ¯1↓*S*,*V*

This will be graphed as simple access to entire vector with scaler reference in header.

F.2.2.23 Select-index - *A*[*V*/⍳⍴*A*]

This construct is compiled as *V*/*A*.

F.2.2.24 Successor - ◇

The two entry points are Overlayed. The right to left order of sub-graphs is preserved within and between the sub-graphs of the two operands.

All other dyadic operations are considered to be function calls.

APPENDIX G

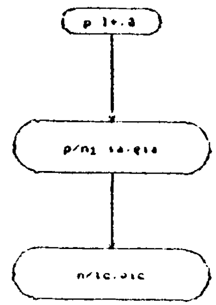
EXAMPLE STREAM GENERATORS

The stream generators and final object code for the examples of Chapter 5 are given below. For all except example 3 only the final stream generator graphs are included below and they have been edited to remove all parse tree node labels not essential in showing the structure of the generator. Each stage of stream generator development is presented for example 3.

G.1 EXAMPLE 1 - PRIME NUMBERS

The expression S←+/2=+/[1]0=(⍳N)∘.|⍳N will calculate the number of primes less than or equal to N.

The stream generator for this expression is:



The object code produced is:

```
LADDER 1:(T[1]_0;
          REPEAT(T[2]_0;
                 T[3]_I[1,1];
                 REPEAT(T[4]_0=(I[1,2]|T[3]);
                        T[2]_T[4]+T[2])
                 AT 2 USING 1;
                 T[1]_T[1]+(T[2]=2))
          AT 1 USING 1;
          EVOKE 0)
```

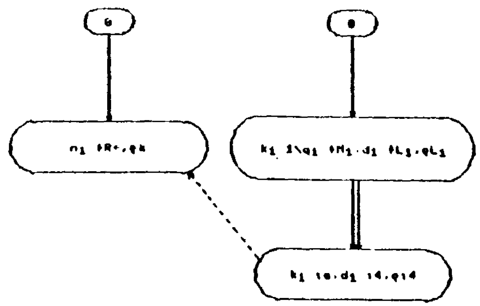
which compiles into 37 PDP-10 instructions or 16 ladder instructions.

For N = 10, the expression performs as follows:

	Array Element References	Temporary Storage
Naive Interpreter	570	220
HP-3000 Compiler	0	0
Stream Generator	0	0

G.2 EXAMPLE 2 - ROMAN NUMBERS

The expression *R*←(,((7ρ5 2)⊤*N*)∘.≥ι4)/,⍉4 7ρ'*MDCLXVI*' converts an integer (N) into its representation in Roman numerals. The stream generator is:



The object code produced is:

```
LADDER 1:(#N IS IN T[1] #
          INIT 1,2;
          T[2]_T[1];
          REPEAT(T[3]_[PI[1]]|T[2];
                 T[2]_⌊(T[2]÷[PI[1]]);
                 T[4]_[PI[2]];
                 REPEAT(T[5]_T[3]≥I[1,2];
                        T[5] => EVOKE 2)
                 AT 2 USING 1)
          AT 1 USING 1
          MOVING 1, 2;
          RHO[2,1]_I[2,1]+1;
          EVOKE 0);
LADDER 2:(INIT 3;
          REPEAT([PI[3]]_T[4];
                 EVOKE 1)
          AT 1 USING 2
          MOVING 3)
```

which compiles into 68 PDP-10 instructions or 31 ladder instructions. When the result contains 7 characters, the execution of this expression requires:

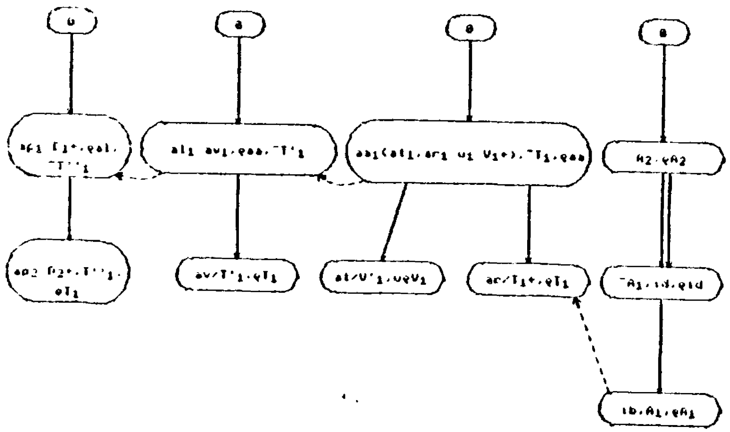
	Array Element References	Temporary Storage
Naive Interpreter	273	74
HP-3000 Compiler	165	70
Stream Generator	28	7

G.3 EXAMPLE 3 - J CHOOSE N

The function

```
      D← J CHOOSE A;B;C;N
[1]   N←ι1↑ρ A
[2]   B←(N,N×J!N)ρ2 1 3⍉1 2 2 3⍉((ιN)SO.=ιN)∘.∨A
[3]   V←2⊥B
[4]   C←((ιρV)=VιV)/B
[5]   D←((J+1)=+/[1]C)/C
```

takes as its argument A a boolean matrix each column of which has J elements equal to 1. Each column is unique and together they give all the ways of choosing J elements from N. The output of the function D is the same information for J+1. The stream generator is:



The object code is:


```
LADDER 1:(INIT 1,2,7;
        REPEAT(T[1]_0;
              REPEAT(EVOKE 2;
                    T[1]_(2×T[1])+T[2]; #2⊥B#
                    [PI[4]]_T[2]) #T_#
              AT 2 USING 1
              MOVING 1;
              [PI[2]]_T[1];   #V_#
              T[3]_1;
              RHO[2,2]_I[1,1]+1;  # ιρV#
              REPEAT (T[4]_([PI[7]]=T[1])∧T[3];
                     T[4] ->(T[3]_0;T[5]_I[2,2])) #ι#
              AT 2 USING 2
              MOVING 7;
              T[4]_T[5]=I[1,1];   #(ιρV)=#
              T[4] => EVOKE 3)
        AT 1 USING 1
        MOVING 1, 2, 7;
        RHO[4,1]_I[4,1]+1; #ρaa#
        EVOKE 0);
LADDER 2:(INIT 3;
        REPEAT(T[6]_I[3,1]; #ιN#
              REPEAT(REPEAT(T[7]_I[3,3]; #ιN#
                          T[2]_(T[6]=T[7])∨[PI[3]];
                          EVOKE 1)
                    AT 3 USING 3
                    MOVING 3 )
              AT 2 USING 3
              MOVING 3)
        AT 1 USING 3
        MOVING 3);
LADDER 3:(# (J+1) IS IN T[9]#
        INIT 5;
        REPEAT(T[8]_0;
              REPEAT(T[8]_[PI[5]]+T[8]) #+/C#
              AT 1 USING 4
              MOVING 1;
              T[8]_T[8]=T[9];
              T[8] => EVOKE 4 ELSE EVOKE 1)
        AT 1 USING 4
        MOVING 5;
        RHO[5,1]_I[5,1]+1);   #ρa1#
LADDER 4:(INIT 4,6;
        REPEAT(REPEAT([PI[4]]_[PI[6]])
              AT 2 USING 5
              MOVING 6,4;
              EVOKE 1)
        AT 1 USING 5
        MOVING 6, 4)
```

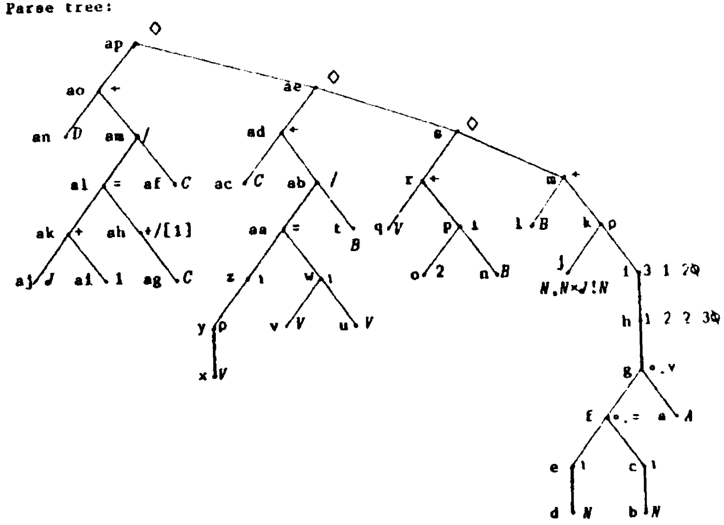
which compiles into 206 PDP-10 instructions or 96 ladder instructions.

For N = 10 and J = 5 this function requires:

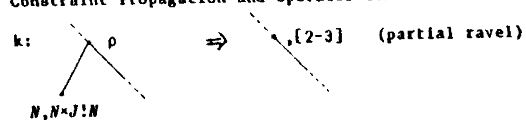
	Array Element References	Temporary Storage
Naive Interpreter	7,318,578	277,200
HP-3000 Compiler	6,533,604	50,400
Stream Generator	6,414,660	2,530

Figure G-1 which follows shows each stage of the development of this result.

Parse tree:



Constraint Propagation and Operator Conversion:



Idiom Recognition:

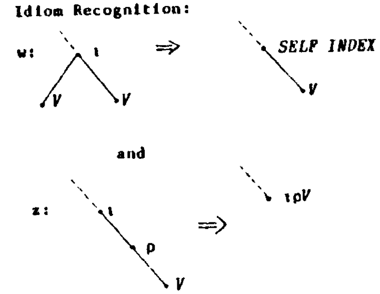
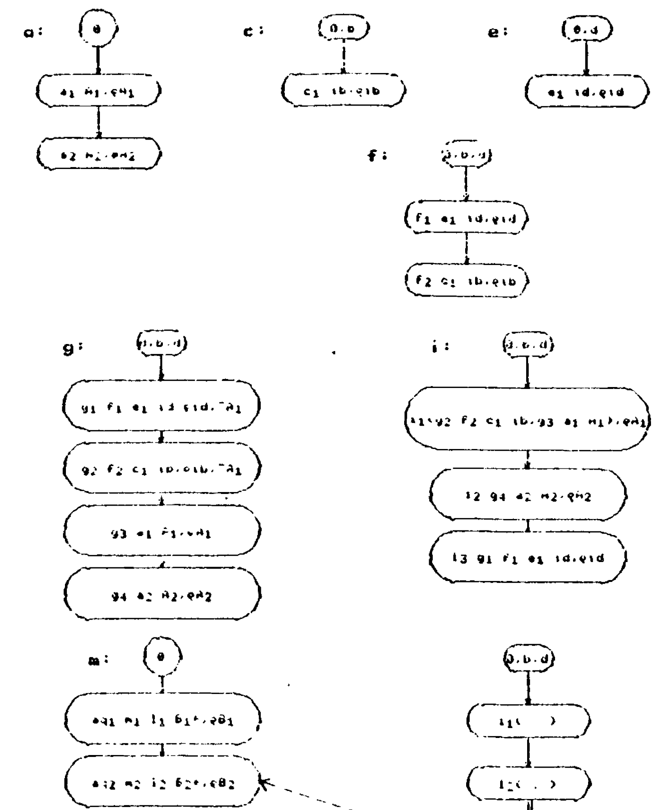
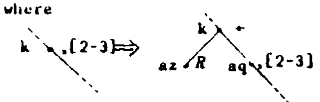


Figure G-1 - Example 3

Stages of Stream Generator Creation:
(active sub-graph at each step only)

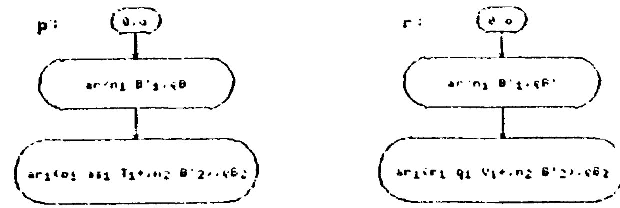


where

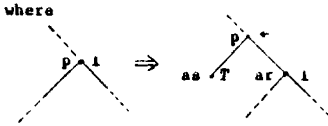


to reflect creation of temporary storage for ravel

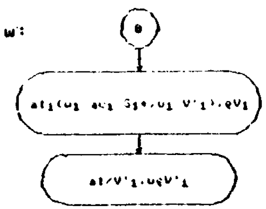
Figure G-1 - Example 3 (cont.)



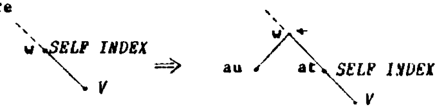
where



to reflect creation of T as storage for Decode (reduction)



where



to reflect creation of S as storage for Self Index (reduction)

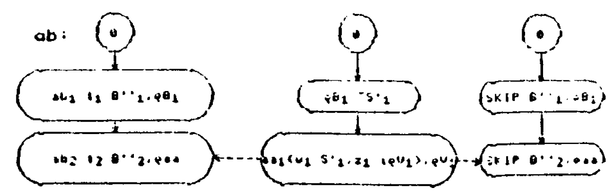
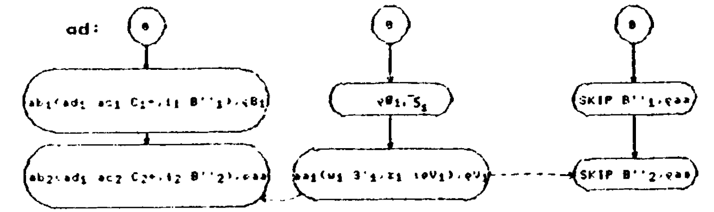
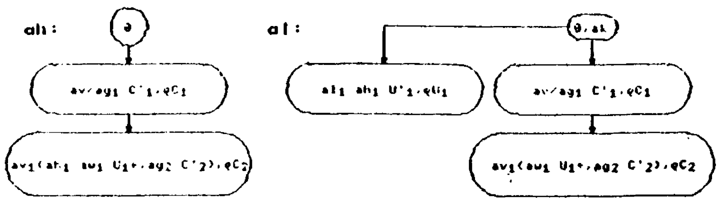
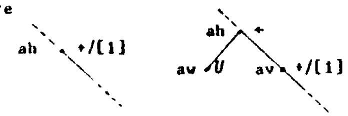


Figure G-1 - Example 3 (cont.)

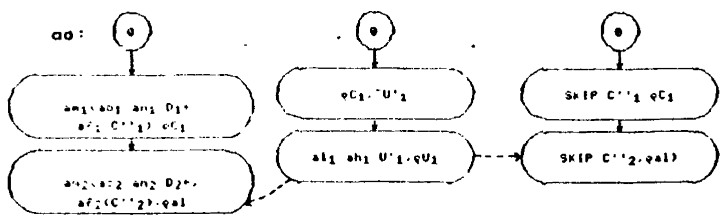




where



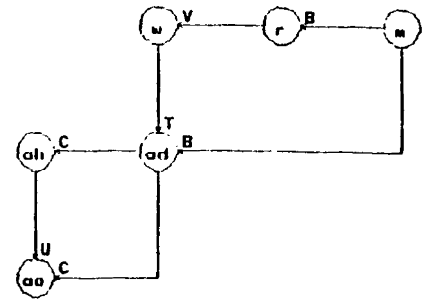
to reflect creation of U as storage for Reduction



the final graph for ap has the graphs for ao, ah, ad, w, and r as sub-graphs. (Some result labels will have been removed from generation sub-graph by in-line assignment demon.)

Figure G-1 - Example 3 (cont.)

Generation/Use Graph



ad: preference conflict - elimination of repeated use of T conflicts with having compression affect lowest dimension.
- elimination of repeated use of T has higher priority and ad is re-ordered

ao: preference conflict - elimination of repeated use of U conflicts with having compression affect lowest dimension
- elimination of repeated use of U has higher priority and ao is re-ordered

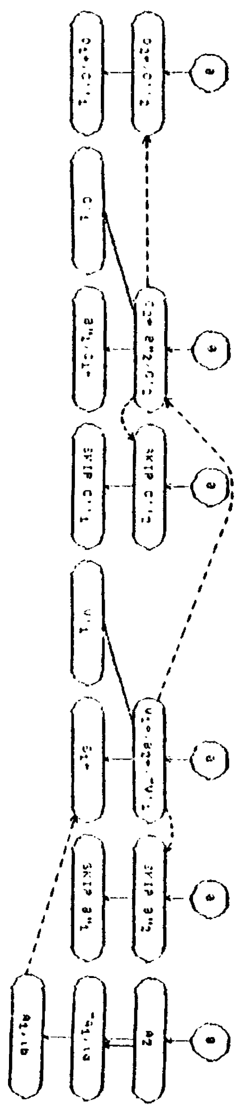
ah: preference - reduction should affect lowest dimension

r: preference - reduction should affect lowest dimension
- r is re-ordered

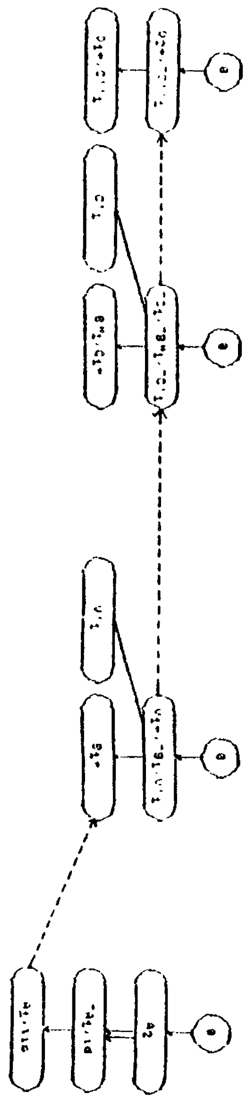
w: preference - reduction should affect lowest dimension

There is no conflict between nodes with an order preference. Testing possible orderings for m reveals that, if m is re-ordered the graph may be overlayed to have the following form:

Figure G-1 - Example 3 (cont.)



which collapses to:



Replacing B and C with the same temporary T gives the final form which was included in the text (section G.3) describing this example. (Note: labels not needed to show source of nodes in earlier drawing have been omitted to save space.)

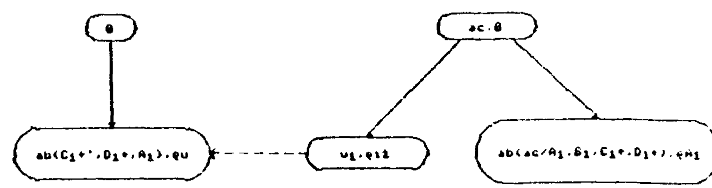
Figure G-1 - Example 3 (cont.)

G.4 EXAMPLE 4 - SYMBOL TABLE UPDATE

The function

```
      SYM X;Y
[1]   Y←,~X∈A
[2]   A←A,Y/,X
[3]   B←(B,Y/,0)+X=A
```

uses global variables A and B which are respectively a vector of single character symbols and a count of the number of times each symbol has been encountered. The function argument X is a character. It will be appended to A if required and the matching item of B will be updated or created. The stream generator is:



The object code is:

```
LADDER 1:(#X IN T[1]#
          INIT 1,2,3,4;
          T[2]_0;
          REPEAT([PI[3]]_[PI[1]];
                 T[3]_[PI[1]]=T[1];
                 [PI[4]]v[PI[2]]+T[3];
                 T[2]_T[2] T[3])
          AT 1 USING 1
          MOVING 1, 2, 3, 4;
          PI[3]_PI[3]+DELTA[3,1];
          PI[4]_PI[4]+DELTA[4,1];
          REPEAT(T[2] => EVOKE 2)
          AT 1 USING 2;
          RHO[3,1]_I[3,1]+1;
          EVOKE 0);
LADDER 2:(#O IN T[3]#
          REPEAT([PI[3]]_T[1];
                 [PI[4]]_T[3]+(T[1]=[PI[3]]);
                 EVOKE 1)
          AT 1 USING 3
          MOVING 3, 4)
```

which compiles into 85 PDP-10 instructions or 48 ladder instructions.

When A has 10 elements and X is a new element, this function requires:

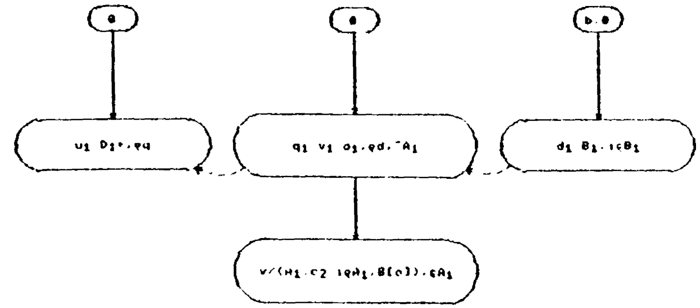
	Array Element References	Temporary Storage
Naive Interpreter	137	22
HP-3000 Compiler	107	11
Stream Generator	42	0

G.5 EXAMPLE 5 - STRING SEARCH

The two line APL expression:

$$D\leftarrow(B[C\circ.+^{-}1+\iota\rho A]\wedge.=A)/C \diamond C\leftarrow(B=\iota 1\uparrow A)/\iota\rho B$$

searches a string B for occurrences of string A and puts all starting positions into D. The stream generator is:



The object code is:

```
LADDER 1:(#1+A IS IN T[1]#
          INIT 1;
          REPEAT(T[2]_I[1,1]; #ιρB#
                 T[3]_T[1]=[PI[1]];
                 T[3] => EVOKE 2)
          AT 1 USING 1
          MOVING 1;
          RHO[2,1]_I[2,1]+1;
          EVOKE 0);
LADDER 2:(INIT 2,3;
          REPEAT(T[3]_1;T[4]_0;
                 REPEAT(T[5]_T[2]+I[2,2]-1; #+¯1+ιρA#
                        PI[2]_PI[2]+G[2,1]×(T[5]-T[4]);
                        T[3]_T[3] ∧ ([PI[2]]=[PI[3]]))
                 AT 2 USING 2
                 MOVING 3;
                 T[3] => EVOKE 3 ELSE EVOKE 1)
          AT 1 USING 2
          MOVING 2, 3;
          RHO[3,1]_I[3,1]+1);
LADDER 3:(INIT 4;
          REPEAT([PI[4]]_T[2];
                 EVOKE 1)
          AT 1 USING 3
          MOVING 4)
```

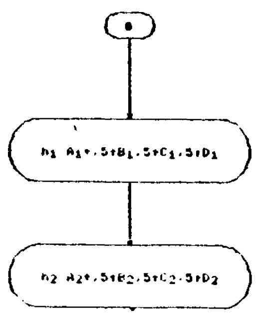
which compiles into 110 PDP-10 instructions or 54 ladder instructions.

When A is 10 characters long, its first character occurs 10 times in B which is 100 characters long, and A occurs once in B, the expression requires:

	Array Element References	Temporary Storage
Naive Interpreter	1181	210
HP-3000 Compiler	951	210
Stream Generator	301	210

G.6 EXAMPLE 6 - SELECTION

The APL expression *A*←5 5↑*B*↑*C*↓*D* which was used in chapter 3 to introduce multiple array ladders is translated into:



The object code is:

```
LADDER 1:(INIT 1,2,3,4;
          REPEAT(REPEAT([PI[1]]_[PI[2]]+[PI[3]]+[PI[4]])
                 AT 2 USING 1
                 MOVING 1, 2, 3, 4)
          AT 1 USING 1
          MOVING 1, 2, 3, 4;
          EVOKE 0)
```

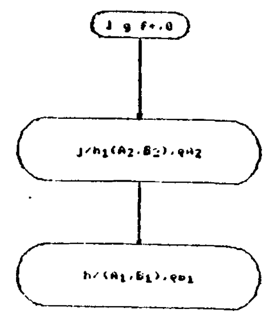
which will compile into 45 PDP-10 instructions or 25 ladder

instructions. When the inputs are 10 by 10 matrices, this expression requires:

	Array Element References	Temporary Storage
Naive Interpreter	650	200
HP-3000 Compiler	100	0
Stream Generator	100	0

G.7 EXAMPLE 7 - TRANSPOSITION

The expression *S*←×/+/[1]*A*:*B* which was used in chapter 3 to show the importance of re-ordering calculations is translated into:



The object code is:

```
LADDER 1:(INIT 1,2;
          T[1]_1;
          REPEAT(T[2]_0;
                 REPEAT(T[2]_([PI[1]]+[PI[2]])+T[2])
                 AT 2 USING 1 MOVING 1, 2;
                 T[1]_T[2]×T[1])
          AT 1 USING 1 MOVING 1, 2;
          #S IS IN T[1]#
          EVOKE 0)
```

which compiles into 38 PDP-10 instructions or 20 ladder instructions.

When the inputs are 10 by 10 matrices this expression requires:

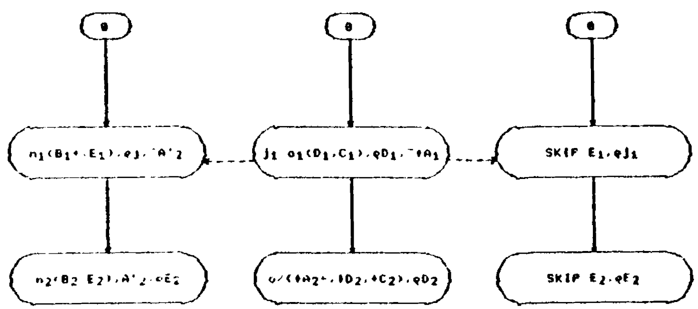
	Array Element References	Temporary Storage
Naive Interpreter	420	110
HP-3000 Compiler	200	0
Stream Generator	200	0

G.8 EXAMPLE 8 - FILTERING

The two line expression

$$B\leftarrow(\vee/A)/[1]E\vee A \diamond A\leftarrow C\wedge D$$

which was used in chapter 3 to introduce the use of co-routines is translated into:



The object code is:


```
LADDER 1:(INIT 1,2,3,4,5,6
         T[2]_0;
         REPEAT(T[1]_0;
                REPEAT(T[2]_(PI[3])∧(PI[4]);
                       (PI[1])_T[2];
                       T[1]_T[2]∨T[1])
                AT 2 USING 1
                MOVING 1, 3, 4;
                T[1] => EVOKE 2 ELSE EVOKE 3)
         AT 1 USING 1
         MOVING 1, 2, 3;
         RHO[3,1]_I[3,1]+1;RHO[2,1]_I[2,1]+1;
         EVOKE 0);
LADDER 2:(T[2] => PI[5]_PI[5]+DELTA[5,1];
         T[2]_1;
         REPEAT(REPEAT((PI[2])_(PI[5])∨(PI[6]))
                AT 2 USING 2
                MOVING 6, 2, 5;
                EVOKE 1)
         AT 1 USING 2
         MOVING 6, 2, 5);
LADDER 3:(T[2] => PI[5]_PI[5]+DELTA[5,1];
         T[2]_1;
         REPEAT(REPEAT(0)
                AT 2 USING 3
                MOVING 5;
                EVOKE 1)
         AT 1 USING 3
         MOVING 5)
```

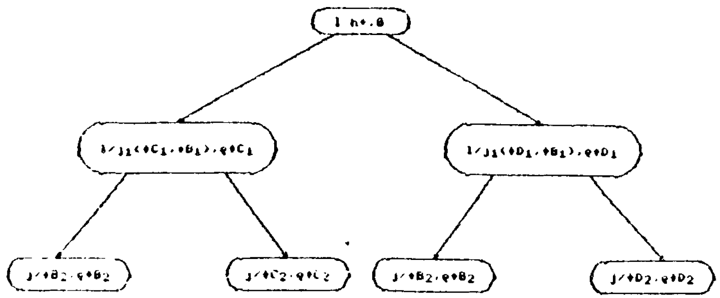
which compiles into 133 PDP-10 instructions or 74 ladder instructions.

When the inputs are 10 by 10 matrices and 5 rows survive, this expression requires:

	Array Element References	Temporary Storage
Naive Interpreter	870	210
HP-3000 Compiler	710	210
Stream Generator	450	10

G.9 EXAMPLE 9 - MERGING

The expression $S \leftarrow +/+/B,C,[1]D$ which was used in chapter 3 to demonstrate the need for multiple nesting is translated into:



The object code is:

```
LADDER 1:(INIT 1,2,3;
         T[1]_0;
         REPEAT(T[2]_0;
                REPEAT(T[2]_[PI[2]]+T[2])
                AT 2 USING 1
                MOVING 2;
                REPEAT(T[2]_[PI[1]]+T[2])
                AT 2 USING 2
                MOVING 1;
                T[1]_T[2]+T[1])
         AT 1 USING 1
         MOVING 1, 2;
         PI[1]_PI[1]+DELTA[1,1];
         REPEAT(T[2]_0;
                REPEAT(T[2]_[PI[3]]+T[2])
                AT 2 USING 1
                MOVING 3;
                REPEAT(T[2]_[PI[1]]+T[2])
                AT 2 USING 2
                MOVING 1;
                T[1]_T[2]+T[1])
         AT 1 USING 1
         MOVING 1, 3;
         #T[1] NOW HOLDS S#
         EVOKE 0)
```


which compiles into 86 PDP-10 instructions or 47 ladder instructions. When B is a 10 by 5 matrix and C and D are 5 by 5 matrices this expression requires:

	Array Element References	Temporary Storage
Naive Interpreter	320	160
HP-3000 Compiler	300	150
Stream Generator	100	0

BIBLIOGRAPHY

1] Abrams, P. S., "An APL Machine", Stanford University Computer Science Department, Report STAN-CS-70-158 (1970).

2] Aho, A. V., Ullman, J. D., Theory of Parsing, Translation and Compiling, Prentice-Hall (1972).

3] Ashcroft, E. A., "Towards an APL Compiler", Dept. of Applied Analysis and Computer Science, University of Waterloo, Report CS-70-01 (1970).

4] Bauer, A. M., Saal, H. J., "Does APL Really Need Run-time Checking", Software-Practice and Experience, 4()129-138 (1974).

5] Bilofsky, W., "PDP-10 IMP72 Reference Manual", Dept. of Computer Science, Yale University (1973).

6] Burge, W. H., Recursive Programming Techniques, Addison-Wesley (1975).

7] Breed, L. M., Lathwell, R.H., "The Implementation of APL360" in Interactive Systems for Experimental Applied Mathematics, M. Klerer and J. Reinfelds editors, Academic Press (1968).

8] Daniels, B., "Very High Level Optimization: Design of an Optimizer for APL", Laboratory for Computer Science, Massachusetts Institute of Technology, thesis proposal (1976).

9] Gerhart, S. L., "Verification of APL Programs", Dept. of Computer Science, Carnegie-Mellon University, Ph.D. thesis (1972).

10] Guibas, L. J., Wyatt, D.K., "Compilation and Delayed Evaluation in APL", Conference Record of the Fifth ACM Symposium on Principles of Programming Languages, Association for Computing Machinery (1978).

11] Hassitt, A., Lyon, L. E., "An APL emulator on System370", IBM Systems Journal, 15(4)358-378 (1976).

12] Jenkins, M. A., "Translating APL - An Empirical Study", Proc. of the APL 75 Conference, Pisa Italy, June 1975, SIGPLAN Technical Committee on APL (1975).

13] Johnston, R. L., Hewlett-Packard Co., Palo Alto, California, personal communication (1977).

14] Jones, N. D., Muchnick, S. S., "Binding Time Optimization in Programming Languages", Conference Record of the Third ACM Symposium on Principles of Programming Languages, Association for Computing Machinery (1976).

15] Kaplan, M. A., Ullman, J. A., "A General Scheme for the Automatic Inference of Variable Types", Conference Record of the Fifth ACM Symposium on Principles of Programming Languages, Association for Computing Machinery (1978).

16] McDonnell, E. E., "The Caret Functions: Efficient Algorithms for the Scans and Reductions of Eight Boolean Functions", Proceedings of the Sixth International APL Users Conference, Anaheim CA (1974).

17] Minter, C. R., "A Machine Design for Efficient Implementation of APL", Dept. of Computer Science, Yale University, Research Report 81 (1976).

18] Mitchel, J. G., "The Design and Construction of Flexible and Efficient Interactive Programming Systems", Dept. of Computer Science, Carnegie-Mellon University, Ph.D. thesis (1970).

19] Perlis, A. J., "Steps Toward an APL Compiler - Updated", Dept. of Computer Science, Yale University, Research Report 24 (1975).

20] Perlis, A. J., "In Praise of APL - A Language for Lyrical Programming", SIAM News, 10(3) (1977).

21] Perlis, A. J., Rugaber, S., "The APL Idiom List", Dept. of Computer Science, Yale University, Research Report 87 (1977).

22] Ryon, J., Burroughs Corp., personal communication (1977).

23] Strawn, G. O., "Does APL Really Need Run-time Parsing", Software - Practice and Experience, 7 (1977).

24] Zaks, R., "A Microprogrammed APL Implementation", Dept. of Computer Science, Univ. of Calif., Berkeley, Ph.D. thesis (1971).

25] Irons, E. T., Yale University, personal communication (1977)