1987

PROTRAN II: Preliminary Report

H. S. McFaddin

John R. Rice
Purdue University, jrr@cs.purdue.edu

Report Number:
87-698

PROTRAN II - PRELIMINARY REPORT

H. R. McFaddin
John R. Rice

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

CSD TR #698
August 1987
(Revised December 1987)

PROTRAN II
PRELIMINARY REPORT

H.S. McFaddin*
J.R. Rice†
Department of Computer Science
Purdue University
CSD-TR-698
August 1, 1987
Revised December 1, 1987

Abstract

This report presents an analysis of the possibilities of incorporating and merging the new Fortran 8X array facilities into the PROTRAN Language and system.

* Research supported by IMSL fellowship
† Research supported in part by NSF grant CCR-8619817

CONTENTS

1 Introduction and Summary

The **objective** of this study is to analyze the potential for defining a new version of PROTRAN, called PROTRAN II here, which incorporates the array facilities of Fortran 8X. PROTRAN is a software product of IMSL, Inc. which already has considerable matrix-vector facilities and, indeed, much of the incorporation consists of syntax changes. We shall assume the reader is familiar with the existing PROTRAN language. A brief summary is included as Appendix 3. Appendices 1 and 2 describe Fortran 8X.

PROTRAN II is visualized as a general purpose language oriented towards computing with arrays, matrices, vectors, etc. having the following characteristics:

- It incorporates all of the high level **problem solving structures** of the current PROTRAN. Creation of a new version might motivate adding a few more problem solving features to PROTRAN but these additions would be incidental to the present analysis.
- It subsumes the syntax and array manipulation facilities of Fortran 8X.
- PROTRAN II code would be translated into either Fortran 8X or Fortran 77 code. It would then serve as a bridge from Fortran 77 to Fortran 8X in the sense that array-matrix code could be developed in PROTRAN II and executed on current machines and then later moved to a Fortran 8X environment with no or trivial changes. Thus, for PROTRAN II sources which involve only Fortran 8X facilities, the PROTRAN preprocessor would serve as a Fortran 8X to Fortran 77 translator.
- It would be implemented with heavy use of the BLAS[1], BLAS2 and perhaps BLAS3 to allow one to achieve high performance on vector and parallel machines. These high performance BLAS may be written by someone else.

Note that the third characteristic allows one to work in a mixed environment. For example, one has a workstation with PROTRAN II and Fortran 77 and a vector supercomputer with PROTRAN II and Fortran 8X. One can develop, debug and execute code on the workstation and then later polish its performance or make production runs on the supercomputer. Indeed, the supercomputer might not have PROTRAN at all if the code only involves Fortran 8X array features.

PROTRAN operates as a set of high level preprocessor directives embedded in a Fortran host program. PROTRAN facilities are implemented with Fortran primitives, much as Fortran is implemented with assembly language primitives. With the advent of Fortran 8X we find that the host language has itself adopted some of these higher level facilities. Because we compute with both

[1] Basic Linear Algebra Subroutine

host and high level language in the same program[2], this causes interferences, redundancies, and annoying "near miss" similarities. We are lead to reconsider the structure of PROTRAN. There are three principal approaches to be considered:

1. Keep the PROTRAN II and Fortran 8X syntax and typing systems distinct. This is the situation with the current versions of PROTRAN and Fortran. PROTRAN would have the simple style, as it does now, of preprocessor directives embedded in a Fortran 8X program. PROTRAN II would be then upwardly compatible with current PROTRAN and merely add all the array facilities of Fortran 8X. Note that PROTRAN has the data types VECTOR and MATRIX while Fortran 8X has the separate type ARRAY so that this is technically feasible. The principal drawbacks here are a) the language is more complicated and b) there are quite different ways to do essentially the same things. The added difficulty in learning and using such a PROTRAN II seem to outweight the advantage of compatibility with the established PROTRAN.

2. Make Fortran 8X completely dominant, remove the vector-matrix types from PROTRAN II and just use arrays. PROTRAN II would still have the high level problem solving capabilities. This makes the simplest language and the one most convenient for those simply looking for a path to use Fortran 8X. A disadvantage is that many of nice facilities of the PROTRAN and the IMSL library are lost or become clumsy (e.g., band matrices no longer exist).

3. Make the Fortran 8X "style" dominant but retain the matrix-vector structure of PROTRAN. Thus one could be a Fortran 8X purist and ignore the matrix-vector facilities or one could use all the PROTRAN II features.

The analysis of this study divides into two parts. Section 2 presents items where difficulties occur in the implementation or where the choices to be made are not clear. The latter tend to arise where PROTRAN has a more extensive or better designed facility than Fortran 8X and yet the Fortran 8X syntax and style prevent a nice merger.

Section 3 presents those changes that are easy to do. They primarily are syntactic in nature or involve features that do not "interfere" with other language features (e.g., adding the Fortran 8X intrinsic functions for arrays).

Section 4 is an outline of a candidate PROTRAN II. This is not a complete, detailed discription but it is sufficient to assess the nature of the proposed language. Three appendices contain reference material. Appendix 1 reproduces some excerpts from documents defining PROTRAN. Appendix 2

[2]Consider the Fortran-assembler analogy, except imagine that one routinely computes with assembly language and Fortran in the same program. Then imagine that the power of the assembly language is increased to understand arrays, if statements, automatic loops, etc., but with a slightly different syntax and semantics.

is a brief summary of the array facilities in Fortran 8X. The current defining document for Fortran 8X is not easy to read. Appendix 3 lists the functions in Fortran 8X associated with arrays.

We conclude that creating PROTRAN II along the lines presented here is quite feasible. The PROTRAN preprocessor would need substanial modifications and become more complex. For example, we believe it would have to read all declaration statements and identify the occurences of all variables. The latter is to handle aliasing situations for arrays. There are a few, relatively infrequent, situations where the PROTRAN II implementation of arrays is inherently less efficient than a Fortran 8X compiler implementation. These also involve aliasing of arrays.

2 Areas With Difficult or Interesting Choices

2.1 Array-Matrix Interaction and Interference

Let us explore the notion of "matrix" as special data type. To the working scientist or mathematician, matrices are used within a limited scope – they have their own special characteristics (banded, symmetric, hermitian, etc.) and defined operations (inverse, various norms, matrix multiplication, transposition, etc.). Consistent with the notions of strong typing, it seems natural to create and distinguish a special mathematical data type for matrices and vectors, a syntax, and a specially defined set of relevent operations. The non-mathematician would appreciate special type enforcements such as size checks on matrix multiplications, or perhaps being alerted to certain unforeseen optimizations such as attempting to place the result of the multiplication of two banded matrices into full matrix (the result fits nicely into a banded matrix). Furthermore, to the scientist, the scalar-vector-matrix domain is closed under the usual operations : application of these operations does not create objects of different types.

At the same time, a scientist would rarely write a computer program dealing only with formal mathematical data types. Almost certainly, he would use arrays, if only for mundane purposes, such as storing a set of problem parameters. Furthermore, it would often be practical for the scientist or mathematician to forego the use of mathematical data types altogether. Consider the task of modelling a domain for a partial differential equation. Typically, the domain is represented by a square grid of data points. The only relation the data grid has to a matrix is that it is square and thus **looks like** a matrix. It is never inverted, multiplied, etc. The natural representation of this set of numbers is an array, and not a matrix. (Usually the matrix representing a discretization of a PDE over an n by n data grid is of order n^4, not n^2 as is the array. Because of the regularity and locality of such discretizations only the grid need be stored – a vast savings.) Thus, a general purpose scientific language dealing only with mathematical data types would be of limited value.

We are faced with the problem of reconciling two languages, both in the immediate sense of

PROTRAN versus Fortran 8X, and in the conceptual sense – the mathematical language of matrices and vectors versus the computer scientist's language of arrays. The scientific computer language must assist, or at least not hinder, the scientist in distinguishing the two concepts while allowing him to use and intermingle both.

The first question of this section is whether we should create a language with two principle data types – arrays and matrices. Earlier, in Section 1, we presented three basic possibilities for the new language. The second possibility was to banish the scalar-vector-matrix data types altogether and simply use arrays in all calculations. This seems attractive for its simplicity, and would indeed work well for FULL matrices. Fortran 8X facilities could be accessed directly, since the PROTRAN II data types would be the same as those of Fortran 8X. In addition, Fortran 8X implements ranging[3], which was formerly a unique advantage of PROTRAN. However, this idea is immediately crippled, primarily because Fortran 8X does not offer any opportunity for packed storage formats like PROTRAN. Ordinary ranging statements do not work on a banded matrix, for example. Fortran does not offer a "reference interface" for packed matrices. Element references would be mapped directly to positions in the storage array, and not the "virtual" matrix. Thus, the packed storage formats would not be part of the language, resulting in terrible loss of storage efficiency. In addition, if the major data structures are shifted "down" a level to the host language, it becomes the duty of the host language to enforce security, for the preprocessor alone is not sufficient to forecast run time errors in mathematical computations. However, Fortran 8X is not designed to protect a computation from erroneous use of mathematics, such as trapping non-zero assignments to off diagonal elements of a banded "array". Also, Fortran 8X is liberal in areas in which the mathematical programmer would like stricture. For example, arrays may be numbered with an arbitrary lower bound, but the only start index which makes sense for mathematical matrices is 1. We conclude that the language should separate the scalar-vector-matrix data types from the array types of the underlying Fortran 8X.

Having decided to include both data types in the language, the next question of this section is whether array and matrix manipulation should be interleaved, or whether PROTRAN II should retain the old style, which effectively separated matrix manipulation statements of PROTRAN from the array manipulation statements of the background Fortran. In other words, should we allow expressions which contain both array objects and matrices, and should we have the ability to pass matrices to array manipulation routines and/or perform matrix operations on arrays? There are several arguments for allowing the mix:

1. It would be useful to reference matrix objects in non-PROTRAN parts of the program, particularly at control points. Consider the IF statement in the following program fragment

[3]See the next section

written in the style of current PROTRAN:

```
LOOP: CONTINUE
      $PROTRAN
      LINSYS   A*X = RESID
      SOL = SOL + X
      RESID = RESID - A*SOL
      $
      IF( NORM(RESID) > EPS ) GO TO LOOP
```

Here, the matrix RESID and the mathematical operator NORM are PROTRAN objects being referenced in "non-PROTRAN" areas of the program.

2. A limited amount of type mixing is already possible in current PROTRAN, in the sense that matrix **storage areas** may be freely accessed in the Fortran parts of a program. Also there is essentially no distinction made between PROTRAN scalars and ordinary Fortran variables.

3. Fortran 8X will allow arrays to be manipulated in much the same fashion as matrices are now manipulated in the existing PROTRAN. Thus statements such as

```
A1 = A2 + A3
```

   are possible, where A1, A2, and A3 are arrays. Meanwhile, we can make PROTRAN matrix assignments such as

```
M1 = M2 + M3
```

   where M1, M2, and M3 are matrices. Why not allow such a statement where the arguments are both arrays and matrices?

4. There are redundant operations. For example, the PROTRAN statement

```
M1 = M2*M3
```

   has the same effect as the Fortran 8X call

```
A1 = MATMUL(A2,A3) .
```

   The programmer should not have to learn two ways of doing the same thing.

5. If arrays and matrices are unified, then we will be able to apply Fortran 8X operations to matrices and vectors, as well as arrays. A particularly useful construct is the WHERE statement in 8X, which provides a very clear "scalar-like" syntax for altering portions of arrays under a conditional. For example, we could use the statement

```
WHERE( M < 0 ) M = 0
```

   to trim all negative values in a matrix to zero. The importance of such statements is not just syntactic, but also efficiency. In a parallel environment, it is expected that the use of a WHERE statement would be translated into tailor-made code designed to exploit the architecture of the underlying machine.

The primary argument against allowing the mix is the complexity of the preprocessing job. The preprocessor would be required to understand not only the special PROTRAN code segments but also the purely Fortran portions. It is clear, we feel, that a greater degree of interaction should be allowed. However, we shall demonstrate that a completely freewheeling style is not reasonable, so that the resulting language will make interaction possible, but only in a carefully controlled manner.

Once one has decided to make arrays and matrices separate data types but more interchangeable, one must address several issues:

1. Some attempt should be made to unify the syntax of the two data types. There are several syntactic features in Fortran 8X that would be convenient in PROTRAN II, or should be adopted to make the languages more uniform.

   - Subsection notation.
   - Range manipulation (see next section) .
   - Array constructors.

2. Certain facilities, such as the Fortran 8X intrinsic functions, should be duplicated for use on matrix objects. The relieves the programmer of the burden of remembering which built-in operators may be applied to which objects. Whenever reasonable, the same function name should be available to execute analagous operations on arrays and matrices.

3. What should be the type of expressions in which arrays and matrices both appear? For example, let `A` be an array and `M` be a matrix of the same size. Then what should be the type of the expression `Q = A + M` ? It does not suffice to take the type of the target `Q`, for an expression such as `(A + M)` may appear as an intermediate quantity in a calculation, or a

procedure argument, where the assignment is made to a temporary variable generated by the compiler.

4. What should be the meaning of an operation such as * ? In PROTRAN, this indicates the usual matrix multiplication. In Fortran 8X this symbol means "element by element" multiplication and actual matrix multiplication is relegated to the intrinsic function, MATMUL. It is clear that in an expression such as M1*M2, where M1 and M2 are both matrices, the operation should be matrix multiplication, but in mixed cases, which operation should be used ? This issue is further complicated by the fact that in larger expressions, the type of the operands themselves may be unclear to the programmer, in which case the operation to be performed would be uncertain. Consider the expression

```
(A1 + M1) * (A2 + M2)
```

where A1 and A2 are arrays and M1 and M2 are matrices. Depending upon the type of the sum expressions, the multiplication might not be the operation the programmer had in mind. This is a more noxious problem than classical type coercion questions, such as the difference between INTEGER and REAL variables in Fortran, for here the result of the coercion may cause a radically different operation to be performed on a large data object.

5. The semantics of even straightforward assignments may be unclear. Consider the innocent looking assignment

```
A = M
```

where A is of type array and M is a banded matrix. If M is the matrix

$$M = \begin{bmatrix} 1 & 2 & 0 & 0 \\ 3 & 4 & 5 & 0 \\ 0 & 6 & 7 & 8 \\ 0 & 0 & 9 & 10 \end{bmatrix}$$

then PROTRAN stores it, invisible to the user, as the array

$$Mstore = \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \\ 9 & 10 & 0 \end{bmatrix}$$

to save space. What should be the meaning of the assignment statement? Should the array A be set equal to the storage array Mstore of M, or a square array containing the full version of the matrix M, including the off-band zeros?

We also consider the symmetric problem: What would be the meaning of the assignment

```
M = A
```

where M has been declared to be, say, a symmetric matrix, and A is the array

$$A = \begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix}.$$

Then A has the storage sequence (1, 2, 2, 1). Is A to be considered the PROTRAN storage sequence of the symmetric matrix

$$M = \begin{bmatrix} 1 & 2 \\ 2 & 2 \end{bmatrix}$$

(the 1 at the end would be ignored) or is A to be considered a template for the symmetric matrix

$$M = \begin{bmatrix} 1 & 2 \\ 2 & 1 \end{bmatrix}?$$

In the first case, it would not be an error if A appeared, say,

$$A = \begin{bmatrix} 1 & 3 \\ 2 & 1 \end{bmatrix}$$

for A is viewed simply as a storage tool, while in the second case the assignment should be illegal, for A is clearly not a symmetric array.

Solutions to many of these problems may be realized through strong type patrolling and explicit coercion statements. Not only would this serve to remove the ambiguities described above, but strong typing could be used to a practical advantage by the preprocessor in distinguishing "look-alike" Fortran 8X and PROTRAN II commands and procedures. The intent of the procedure would be discerned by checking the type of the argument expression.

In the statement A = M, for example, we could introduce an operator which converts a matrix expression into its actual storage, whatever that might be:

```
A = STORE(M)
```

or, on the other hand, we could assume a default storage based semantics in array assignments and have a coercion operator, say ARRAY, which produces an array containing the full version of its matrix argument. If B is, say, a banded matrix, then the assignment

```
A = ARRAY( B )
```

would fill the array A with a "snapshot" of B, including the off-band zeros.

For the corresponding problem of assigning array expressions to matrices and vectors, similar operators could make the meaning explicit. For example, the array to matrix operator FULL could interpret its array argument as the storage for a FULL matrix (in this case a snapshot). Such conversion operators may have added utility in making bizarre matrix-matrix assignments. Consider the statement

```
M1 = FULL(STORE(M2))
```

where M2 has been declared, say, banded symmetric and M1 is a full matrix. M1 would be a non-symmetric matrix which "looks like" the array storage of M2.

It may be preferrable not to require explicit conversion in all mixed expressions, but instead to specify some default semantics. However, it is unclear what the default semantics should be, for we must first resolve the question of what exact relation an array has to a matrix. Is an array simply a storage area for a matrix? Or is it a visual snapshot of a matrix?

The advantage to the storage based semantics is that the programmer is forced to maintain a clear mental separation between the two data types – matrices are mathematical objects and arrays are storage areas for these objects. An analagous situation exists in languages such as C, where character types (bytes) and, say, integer types coexist. An integer is a structured mathematical object composed of bytes. Both types are used simultaneously, but are not safely interchangeable. If this separation is used as a theme throughout the design of the language, hopefully we will avoid much confusion. Also, taking the minimal view of an array as the default semantics would seem to alleviate the need for explicit conversion in more cases than the alternative semantics.

On the other hand, this semantics clashes with the notion most programmers now have of arrays, in which arrays and matrices are the same thing – a square collection of numbers. Can the programmer be trusted to respect a storage based semantics in light of this?

We conclude that because of the confusion possible when allowing one specific notion about arrays to dominate, we should favor neither, and instead require specific coercion in all instances in which an array expression appears in a vector-matrix assignment statement or in which a vector-matrix expression appears in an array assignment. The cost of such type checking is that the

PROTRAN preprocessor becomes much more complicated, essentially having to parse and understand even the pure Fortran parts of a program – which it could previously ignore. The coercion operators are described in the design section.

2.2 RANGE facilities

2.2.1 The use of range facilities in a high level array-matrix language

Non-trivial data structures must usually be defined by a set of dynamic parameters which dictate the size and exact arrangement of the contained data. Fortran 8X arrays, for example, are defined by a type, upper and lower index bounds in each of seven possible dimensions, and a set of "attributes." PROTRAN matrices are described by two "current" upper index bounds, a storage format, and an absolute upper index bound. All of these parameters must be known by subroutines and problem solving statements which operate on such data structures. The primary function of ranging is to provide such information in an automated fashion.

Consider the traditional scenario when writing a program to do matrix computations. We first statically allocate an array whose size is at least as large as any matrix to be operated upon. The actual matrix, whose size will not be known until run time, will occupy only a portion, the upper left corner, of this array. Because of the compile time allocation requirements of current Fortran, this scenario applies not only to matrix computation, but a broad range of problems in which arrays are used to hold an unpredictable amount of information. Information about this amount, called the "working size" or "range" of the array, is held in separate variables, which must be explicitly passed along to subroutines operating on the array. The lack of automatic ranging would present a conceptual obstacle when defining an language having atomic operations upon arrays (matrices), in which only the array (or matrix) name should suffice as the argument. If we truly view arrays (and matrices) as fundamental objects of the language, then descriptive information, such as "working size", "leading dimension", or "storage format" (in the case of PROTRAN matrices) should be only an implicit argument to atomic operations, and should certainly not require programmer manipulation.

2.2.2 Ranging in PROTRAN.

Automatic ranging is one of the major conveniences the designers of PROTRAN provided which is not included in existing versions of the host language, Fortran. As computation progresses, the working size of arrays and vectors can be dynamically altered. This may be forced either explicitly, by making an assignment to one or more of the range variables of a matrix, or implicitly, by making an assignment in which the target object assumes the shape of an expression, or by using a problem solving statement. The current range, actual size, and storage format of a matrix are all known

to PROTRAN problem solving procedures which operate on the matrix. The mechanics of this is invisible both to the programmer, and to the host language, and is a function of the PROTRAN preprocessor.

Put simply, the preprocessor implements ranging by having the range variables show up everywhere the matrix is operated upon. From the viewpoint of the programmer, then, the range of a matrix is an "invisible" attribute which is somehow known to all of the matrix operators and problem solving statements he makes use of. The same is true of the other matrix attributes, such as static size and storage format. By providing the range variables, static sizes, and format of the matrix storage array, the preprocessor allows the problem solving subroutines to find the correct elements of the working matrix, but relieves the programmer from remembering the lengthy calling sequence. From the viewpoint of the host language processor, it is as though the programmer himself explicitly coded the call.

2.2.3 Ranging in Fortran 8X.

Fortran 8X has added the dynamic ranging capability that previously was unique to PROTRAN. While this makes the practice of ranging accessible to a larger language, and thus a larger community of users, it has important ramifications for the redesign of PROTRAN.

- The syntax and pragmatics of ranging are different in Fortran 8X and PROTRAN. Should PROTRAN adopt the Fortran 8X ranging methodology?
- Ranging is implemented outside of the host language in PROTRAN. Since the two languages are to be merged, we must answer the question of how to **implement** PROTRAN ranging — should we continue with the methodologies outlined above, or take advantage of the ranging facilities in Fortran 8X?

Not all Fortran 8X arrays are given the ranging capability. An array is given this capability by including the keyword `RANGE` in the declaration line:

```
real, RANGE array(-9:50,50) ::  a, b, c
```

Here we have defined arrays a, b, and c to be variable sized arrays initialized to have **range** (-9:50,50) and **shape** (60,50). We could have associated all three arrays with a single **range name**:

```
real, RANGE /raynj/ array(-9:50,50) ::  a, b, c
```

This has the same effect on the arrays, plus declares the name `raynj` to be a **range variable** initialized to the range (-9:50,50) and associated with the three arrays. This is similar to range variables in PROTRAN, except that a single vector range variable describes all dimensions of an

array instead of separate range variables for each dimension. In one sense, then, range variables in PROTRAN offer a finer resolution. However, Fortran 8X ranging offers more utility in the sense that a range variable may describe both upper and lower bounds in each dimension. In both cases the same range variable may be used to describe more than one array. We do not need to declare a specific range variable to associate with a ranged array. We may have the name of the range implicitly associated with the name of the array:

```
real, RANGE array(10:50,50) ::  a
```

Range variables are updated by a special form of assignment statement:

```
SET RANGE (10:50,20) /raynj/
  SET RANGE (21:25,5 ) d
```

This means that all arrays associated with the range name `raynj` will now range from 10 through 50 in the first coordinate and 1 through 20 in the second and that `d` will be as if we had declared `d(21:25,5)` . We could have changed only a portion of the range variable by using

```
SET RANGE (10:50,:)  /raynj/
```

This adjusts the first extent and leaves the second extent as before.

Changing the range of an array changes the element sequence associated with the identifier. Suppose we declare a 3×3 array a as follows:

```
real , range array(3,3) ::  a
```

and then initialize a so that it appears

$$a = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

If we change the range of a as follows,

```
SET RANGE (1:2,2:3) a
```

then we restrict our attention to the upper right corner of a. This now holds for all further references to a. This is not a superficial adjustment, as in the case of PROTRAN. When we pass the array a as an argument to a subroutine, the elements received are automatically that of the sub-array:

$$\begin{bmatrix} 2 & 3 \\ 5 & 6 \end{bmatrix}$$

This eliminates the need for explicitly passing "leading dimension" arguments to Fortran 8X subroutines. Notice that this information is still needed by the subroutine, for the elements of the submatrix are **not** found in contiguous storage locations. Apparently, the language processor must operate in a fashion by which this information is known implicitly by the subroutine. The programmer may assist in this by explicitly describing the INTERFACE with the subroutine in the calling program. The actual and dummy arguments need only agree in rank. The subroutine can project its own numbering scheme on the dummy.

2.2.4 A unified syntax for PROTRAN and Fortran 8X ranging.

There are two issues to be faced in light of these changes in Fortran. The first is the syntax of PROTRAN ranging and the second is the manner of implementation. The primary argument for changing the PROTRAN ranging syntax is the simple reason that otherwise the programmer must learn two ways to do the same thing – change the working size of his data objects. There should be one dominant way of doing ranging in a PROTRAN II / Fortran 8X program. We feel it is evident that the PROTRAN approach should give way to Fortran 8X. Because of upward compatibility concerns, however, the old style of ranging could be retained as a transition feature.

- When an explicit range change is desired, the update should be made through an obvious command such as SET RANGE. Thus, we should discourage the use of independent "range variables" in each dimension. Because the SET RANGE statement allows a single dimension to be updated, without changing the others, the SET RANGE command of Fortran 8X has the same functionality as an assignment to a PROTRAN range variable. For the sake of upward compatibility, however, the old style of ranging should probably be permitted.

- It would not be wise to adopt the Fortran 8X facility of giving "irregular" ranges to matrices such as

  ```
  set range m(k:n,k:n)
  ```

  for this violates the mathematical notion of matrix. Also, it would be more complicated to implement such ranges in the pre-processor and would necessitate new calling protocols for the IMSL library, for these routines must then be prepared to receive lower bounds as well as upper bounds in the calling sequence.

2.2.5 The Fortran 8X implementation of PROTRAN ranging.

However evident the need for unification of the ranging methodologies, it is less evident as to how to implement these changes. In this section, we explore two possible implementations of the redesigned ranging in PROTRAN.

- A seemingly simple solution would be to implement PROTRAN matrices as RANGED Fortran 8X arrays. Thus, the declaration

```
real matrix a(50,50)
```

would be translated into

```
real, range array(50,50) ::  a .
```

Because the declared size and range is built into the array a and is made available to subroutines via inquiry functions, there would be less information for the pre-processor to send across subroutine calls.

The clear advantage to this implementation is that in the case of full matrices a tremendous burden has been lifted from the pre-processor. Since matrices, in real life, would be ranged Fortran 8X arrays, there can be a free interaction between the two types. also, the range of the storage would reveal the range of the matrix being represented, if matrices are stored in a predefined manner. Full matrices would be stored directly as an array. Band matrices would be stored as an array with one band in each column, the first column being the diagonal, the next being the first first band above, then the first band below, then the second band above, then the second band below, etc. If k is the max of the number of above diagonal bands and the number of below diagonal bands, then the storage matrix would have $2k + 1$ columns – slightly wasteful. Band symmetric matrices would be stored with $k + 1$ columns. Symmetric and hermitian matrices would be stored in a vector of length $\frac{n(n+1)}{2}$. In all cases, the array ranges could be used to compute the actual matrix range. And no extraneous information, save for the type, would have to be stored separately by the preprocessor.

For all its apparent elegance, this implementation would have some serious disadvantages. One disadvantage is that the IMSL library subroutine package will have to be changed. Recall that we assumed subroutines expect to be given all relevant storage information in the call, and then to use this information to calculate the location of the data in the matrix. In particular, the subroutines will expect to receive the element sequence of the original array, not just the trimmed element sequence of the active portion. It would be this smaller portion that is sent across the call if the matrix is implemented as a ranged Fortran 8X array. The subroutines would have to be changed in light of this more compact implementation.

This solution creates new problems for the upward compatibility of user programs as well. Suppose, for example, the user has written a special "PROTRAN smart" routine to print matrices in a special manner:

```
subroutine myprint(matrix-store, leading-dim, range1, range2, format)
```

The subroutine accounts for the storage scheme. Then the call

```
call myprint(m,50,Nm,Mm,FULL)
```

would result in the substitution

```
call myprint(m(1:10,1:10),50,Nm,Mm,FULL).
```

The element sequence passed as the first argument would be that of the 10 x 10 sub-array and not that of the original 50 x 50 storage array. The intent of the call would be subverted by the processor.

Additionally, we observe that this scheme only serves to pack ranging information into the Fortran 8X storage, and does not reduce the complexity of the ranging tasks of the preprocessor. SET RANGE calls on matrices must still be intercepted and translated into appropriate SET RANGE statements on the underlying array. For example, if S is a symmetric matrix, then SET RANGE S(*k*) would result in the generated code SET RANGE Sstore(*k*(k+1)/2*).

- The other implementation choice is to implement ranging much as it was before. To have a ranged matrix, we statically allocate an array of the declaration size and have the range variables follow the matrix, transparently to the programmer, at all points where problem solving statements are invoked. This is insufficient, of course, if we are to allow direct interaction between Fortran 8X arrays and PROTRAN matrices. But with the appropriate coercion operators discussed before, the preprocessor could handle the transfer.

  Most importantly, this implementation scheme would make possible the retention of the old style ranging. If range values are stored as separate Fortran 8X variables, as before, then any independent range values specified by in old-fashioned PROTRAN program could be equivalenced with the preprocessor generated variables, and both functionalities could be realized.

2.3 ALIASING

2.3.1 Introduction

The general notion of aliasing is to make a collection of data simulataneously accessible via more than one name. Conceptually, this is similar to entering a make-believe procedure, passing the data object in question as an actual argument. Inside the environment of this make-believe procedure, the data object, or a portion of it, may be referred to by a different name – that of the dummy

argument – and may even be viewed as having a different structure. For example, we could pass an 8 × 8 array A to a procedure SUB, where it could be treated as, say, a 64 element vector B(64), or a 3 dimensional array C(4,4,4). With aliasing, however, we need not enter a distinct scoping unit to acheive this effect. The alias object, then, is a local "short hand" for referring to the object. This technique can have great utility in providing data abstraction – removing irrelevant structure from before the eyes of the programmer – without leaving the current environment. For example, the programmer may wish to treat the diagonal elements of an array A as being the elements of a vector DIAG:

$$\texttt{DIAG(I)} \longrightarrow \texttt{A(I,I)}$$

Then a reference or update of DIAG(2) is equivalent to a reference or update of A(2,2)

As a practical matter, the use of aliasing must be restricted. For example, the language processor must avoid circularities in alias identifications. The following sequence of identifications should be illegal:

$$\texttt{b} \longrightarrow \texttt{a}$$
$$\texttt{c} \longrightarrow \texttt{b}$$
$$\texttt{a} \longrightarrow \texttt{c}$$

Indirect aliasing may be perfectly legal, but the processor must insure that actual data may be found at the end of the chain. Also, the processor may want to restrict the mapping between the alias object and the real data. In Fortran 8X, this mapping must be linear in the dummy variables (such as I above) and is further restricted to be injective. Suppose we make the following non-injective alias identification:

$$\texttt{V(I)} \longrightarrow \texttt{S}$$

where V is a one dimensional array and S is a scalar. Then each array element is another name for the scalar. If we set

$$\texttt{S} = 5$$

then it is as if each array element contains a 5. But the assignment

$$\texttt{V(3)} = 7$$

sets **all** elements of the array to 7, not just the third, as the programmer may believe. In the context of PROTRAN matrices, the processor must be even more restrictive than usual in making alias identifications.

2.3.2 Fortran 8X Aliasing

Fortran aliasing has previously been available, in primitive form, with the Fortran **EQUIVALENCE** declaration. The idea was to physically align the memory chunks assigned to one or more arrays so that they would overlap. Then surely, when a reference or update is made to an entry in one array, the same thing happens to some entry in the other array. This rather simple technique provided much of the utility discussed above, finding its primary use in allowing the programmer to treat arrays as vectors and vice versa. However, this data abstraction is inconvenient, for it requires the programmer to have knowledge of the memory layout for arrays. Also, this aliasing is weak, for the identification is restricted to consecutive array elements, so the diagonal identification above would not be possible. It is also impossible to identify one array with a sub-array of the other, say the upper left corner, unless the two arrays had the same row size. Further, the EQUIVALENCE statement is a declaration, so the identification is done at compile time, and does not offer dynamic identification.

Fortran 8X greatly expands aliasing capabilities. The data identification may be deferred until run-time. Allowable mappings between objects are much more general than simple contiguous storage identification. Indirect aliasing is allowed, although the requirement that real memory (or at least allocatable memory) be found at the end of a chain is enforced. Also, one may make identifications with scalars and even individual component elements of heterogeneous "derived type" structures[4], although we shall be primarily interested in identifications between arrays.

Fortran 8X does not allow free-wheeling aliasing between objects, but instead distinguishes a special data type for objects that are to be identified with others. The object must be declared to have the ALIAS attribute. Since the exact arrangement of the data is not known until an explicit data identification, only the type and rank of the object is indicated in the declaration. In the terminology of Fortran 8X, such an object is said to have *deferred shape*, for example

```
REAL, ALIAS ::  DIAG(:), RED(:,:), CURRENT(:,:)
```

The data identification is made via an IDENTIFY statement:

IDENTIFY(*data mapping*)

where the mapping takes the form

alias array element = parent array element , index bounds

and looks much like a statement function. Thereafter, the alias object is said to be *alias identified* with the parent object. Examples of this are:

[4]These are called record types in Pascal and structures in C

```
IDENTIFY( DIAG(I) = A(I,I) , I = 1:N )
IDENTIFY( RED(I,J) = A(2*I, 2*J), I = 1:N/2 , J = 1:N/2 )
IDENTIFY( CURRENT(I,J) = A(I+K-1,J+K-1), I = 1:N-K+1, J = 1:N-K+1 )
```

The mapping must be linear in each of the dummy variables. The mapping is required to be injective and a given alias object may not be repeatedly un-bound and re-identified with other objects. Recursive identifications, in which the alias object itself is referenced, is illegal. Thus, if we were using CURRENT as a template for the "current" sub-matrix being transformed by Gaussian elimination, we could not use the code

```
K = K + 1
IDENTIFY( CURRENT(I,J) = CURRENT(I+1,J+1) , I=1:N-K+1, J=1:N-K+1 )
```

to advance the template. We must reference the original parent.

There are some features, not included in Fortran 8X, which one may desire from a general purpose array language. The mapping from alias object to parent object, as we observed, must be rather simple (a linear expression in the index variables). For example, one may not use the modulo operator on the indices. This would be handy in computing "shifts". Suppose we wished to view an array as a two dimensional torus. This is done by considering corresponding left and right elements, as well as corresponding top and bottom elements, to be adjacent, just like interior elements. To compute a shift around the torus in the (s,t) direction one would like to use the (illegal) identification

```
IDENTIFY( SHIFT(I,J) = A(mod(I+S,M),mod(J+T,N)) , I=1:M, J=1:N )
```

Fortran 8X requires the ALIAS object be identified with only one parent. Only the mapping into that parent may change over time. It would also be nice, however, to identify with more than one parent, not only over time, but possibly simultaneously. This could be implemented with a form of the Fortran 8X SELECT CASE statement. Consider the following example.

```
IDENTIFY( COMPOSITE(I,J) =

      CASE ( J <= N/2 ) A(I,J)
      CASE ( J > N/2 ) B(I,J-N/2)
      , I = 1:N, J = 1:M )
```

The left half of COMPOSITE would be the array A and the right half would be the array B. The introduction of the SELECT CASE statement would also solve the torus problem. The compiled program would have the job of keeping track of the various parents. This is possible, however, because all possible parents referenced throughout a scoping unit could be determined at compile time – this list itself is not dynamic.

2.3.3 Aliasing for PROTRAN II

The main questions of this section are whether aliasing should be allowed on the PROTRAN II data types, and if so, how this can be implemented. It is clear that this facility would have much utility in a mathematical language. The diagonal problem is one of many examples of this utility. If matrix-array interaction is allowed, even with strict type coercion requirements, the issue of aliasing cannot be avoided. Further, it would be awkward for the programmer to have alias capabilities for the array data type and not the matrix type.

There are certain conceptual problems brought on by introducing aliasing into the matrix environment. For example, should one be allowed to alias two matrices of different format type? Consider the problems brought on by the identification

$$\mathtt{F} \longrightarrow \mathtt{B}$$

where F is a full matrix and B is a banded matrix. By using the name F instead of B one could perform operations that should be impossible on a band matrix. Also, the substantial economy sought in using the banded format is bypassed simply by renaming the object. Further, it is not clear what storage identifications, if any, are possible for the off-band elements of F. Since only the band elements of B are stored, it would be as if these elements are identified with thin air. A similar question is raised by the identification

$$\mathtt{S} \longrightarrow \mathtt{F}$$

where S is a aliasable symmetric matrix and F is the full matrix

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}.$$

Basically, we want the matrix formed from the mirror image of one corner of F. Such an construction would not be so uncommon as to disallow. Here, there is no problem finding elements to identify S with, but we must instead ask which elements will be chosen – those in the lower left or upper right triangles of F. That is, should S appear

$$\begin{bmatrix} 1 & 3 \\ 3 & 4 \end{bmatrix}$$

or

$$\begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix}?$$

Also, if S is later updated, then should the "opposite" elements of F be likewise updated?

A possible solution to this problem, aside from the obvious one of uniformly disallowing identifications between objects of different format (full vs. band) is to consider aliasable matrices to be of bf indeterminate type until an explicit identification is made, and then have the type of the parent object be assumed by the alias object. Above, for example the identification F $\longrightarrow$ B would result in the PROTRAN II visible type of F being indeterminate until the identification, and then set to ***banded***. However, simply assuming the format of the parent does not always make sense. Consider the identification

```
M(I,J) ⟶ S(I+1,J)
```

where S is a symmetric matrix. Then M is not a symmetric matrix, for we have simply shifted elements in S up one entry. Simply comparing the types of the alias and parent object is not sufficient. In general, the problem is difficult, and the pre-processor must be sufficiently "smart" to detect the difference. Furthermore, since identifications may possibly depend upon values in certain variables, this check must be deferred until run-time. Consider the following alteration of the above example:

```
M(I,J) ⟶ S(I+K,J)
```

Then the legality of this depends upon the value of K, which is unknown at preprocessing time.

The solution we have chosen is to distinguish a **special vector-matrix format**, ALIAS. An ALIAS matrix would then be treated in a special manner. The rights and priviliges of such a matrix would be have to be the most limited – essentially those of a FULL matrix, except that certain data values could be ignored. For example, if an element of an ALIAS matrix is mapped to an off-banded element of a BAND matrix, an assignment to this element would be equivalent to an assignment to an off-band element of the parent, and would have no effect. If immediately referenced, this element would have value zero, no matter what the assignment.

2.3.4 Simulation of Aliasing

Aliasing seems to present special problems with the implementation. Within the module in which an alias identification is made, there is no problem in accessing and updating an aliased object, for the compiler has available the transformational formula. However, problems arise due to the possibility of separate compilation of subroutines. For an alias object to be passed to a separately compiled subprogram, the receiving subprogram must have the address of the source array and sufficient information to describe the naming transformation from the alias to source. The same problem occurs with passing arbitrary matrix sections across program boundaries:

```
CALL SUB( MAT(1:M:2,1:N) )
```

The question is how we pass the data, or information about the data , in such a way that all subprograms receive matrix expression arguments in a uniform fashion. For example, the receiving program may have been written and compiled in old PROTRAN, in which aliasing and matrix sections do not exist, and receiving procedures expect only the a contiguous chunk of memory containing the matrix storage. The same problem, of course, is encountered by the Fortran 8X compiler. In that language, however, the programmer aids the compiler by describing calling protocols with an explicit INTERFACE block. This essentially tells the compiler of the calling program what the called subprogram expects in the way of arguments. For the sake of simplicity we do not include such a construct in PROTRAN II, but have adopted a standard and more general protocol for passing matrix arguments (see the design section).

To preserve uniformity of calling protocols, we suggest a **copy-restore scheme** for passing alias matrices, matrix sections, and other "virtual matrices" to subprograms. In this scheme, a temporary storage area is created for the actual argument ALIAS matrix and is filled using the transformational formula given at the point of identification. Essentially, we have taken a snapshot of the portion of the parent object which is shared by the alias object. It is this storage area which is passed to the subprogram. Upon return, the calling program again uses the transformational formula, which it has available, to restore the shared portion of the parent object. Because the mapping is injective, the order of restoration does not matter. The following schematic illustrates this idea.

```
SUBROUTINE SUBA(A,B,C)
REAL, MATRIX ::  A(*,*), B(*,*)
REAL, MATRIX ::  D(20,20), X(20)
REAL, ALIAS, MATRIX E(:,:), F(:)
...
IDENTIFY ( E(I,J) = A(2*I-1,J) )
IDENTIFY ( F(I) = D(I,I) )
...
CALL SUBSOL(E,F,X)
...
END
SUBROUTINE SUBSOL(S,T,U)
REAL, ARRAY S(*,*)
REAL, ARRAY T(*), U(*)
solve S T = U
END
```

The call statement translates into the following psuedo code:

```
Copy E ⟶ E.temp
Copy F ⟶ F.temp
CALL SUBSOL(E.temp, F.temp, X )
Copy E.temp ⟶ E
Copy F.temp ⟶ F
```

The good thing about this scheme is its simplicity. The called program need have no knowledge of the transformational mapping found in the calling program. It simply treats the passed matrix as the actual parent of the argument – after all, this is the conceptual idea behind procedural abstraction[5] in formalisms which view the data as a series of snapshots. A snapshot is taken of the argument at the time of call[6] and then this snapshot, being handed to the called subprogram, is operated upon and a new snapshot is handed back to the caller, which is then used as the "current" value of the data. This has the further advantage of allowing the uniform treatment of all matrix arguments, instead of distinguishing a special ALIAS matrix argument type. Thus, no dummy arguments are treated as ALIAS types, and all translations are done in the calling routine.

The most obvious drawback to this scheme is its cost. We not only consume the extra space for the temporaries, but we spend a lot of time in doing it. However, this scheme has the more subtle, and potentially more malicious, effect of changing the calling semantics of Fortran from "call by reference" to "copy-restore". The two are **not** equivalent.

It is not widely known, but buried in the standards of Fortran 77 and 8X are constraints on argument associations which make the copy-restore method non-restrictive. That is, if these constraints are voluntarily obeyed by the programmer, call by reference is equivalent to copy-restore. This is because the standard does not allow an external object of a subroutine to be namable as both a dummy argument and an external object. Also, no datum may be namable by two dummy arguments (i.e. dummy argument arrays may not overlap). Any Fortran program in which copy-restore does not work as expected is an illegal Fortran program. However, this part of the Fortran standard is not enforced by any compiler known to us, so someone with an illegal (i.e. not standard conforming) program might be surprised.

[5]Calling protocols – call by value, call by reference, etc. are just practical approximations to this ideal

[6]There are some argument passing schemes, referred to as "call by need" or "lazy evaluation" in which the snapshot is not taken until the formal parameter is actually referenced in the called subprogram. Still another calling protocol, often found in languages dealing with concurrency such as Flexible Computer Corporation's Concurrent Fortran, makes a distinction between the time at which an argument is determined and the time at which the called program may access it. It is as if the camera is aimed at one point in the program and the snapshot is taken at another point.

2.4 Upward Compatibility Issues

One of the desired, if not necessary, requirements we should make of PROTRAN II is the upward compatibility with old PROTRAN programs. Upward compatibility can be carried out at two levels:

1. **Source code.** Programs written in the old PROTRAN should pass through the PROTRAN II preprocessor without error and produce the same results. Note that this does not mean those results must be arrived at in the same fashion, but that the **meaning** of the programs are preserved.

2. **Object code.** Existing PROTRAN modules, especially those of the PROTRAN library, may be already compiled or at least pre-processed, and we do not want to worry about their interior workings. However, we must preserve the semantics of their calls. As we observed before, changes in PROTRAN facilities, such as ranging and the availability of aliasing, can alter the meaning of the parameters in calls to the interface routines. This threat not only applies to routines in the PROTRAN library, but to user-defined routines.

Without source code compatibility, PROTRAN users would be put to the considerable expense, in terms of human labor, of re-writing their old PROTRAN programs to work under the new system. In light of the fact that PROTRAN II is their only interface to the new Fortran, and thus to the array-vector processor, this expense would seem unavoidable. Without object code compatibility, the entire PROTRAN system must be restructured from scratch. We shall offer suggestions toward both ends.

Source Code Compatibility

Fortunately, most of the changes to PROTRAN which we have suggested have been in the "upward" direction – additions to the language which would bring it into compatibility with Fortran 8X. In the area of ranging, however, we have suggested a "lateral" change. There, we suggest that single dimension range variables be discouraged in favor of the multi-dimension range variables of Fortran 8X. However, for purposes of upward compatibility, we must allow ranging in the old style. This has influenced our decision **not** to implement PROTRAN-object ranging as equivalent ranging operations on Fortran 8X storage objects, but to instead maintain seperate Fortran 8X variables which hold the range of the PROTRAN II object, as before. If we do this, the preprocessor may `EQUIVALENCE` the independent single-dimension range variables declared by the programmer with the pre-processor generated range values.

There is one area in which source code compatibility is not realized – argument passing. In Section 4 we will describe an improved procedure interface in which matrix names as procedure

actual arguments refer to the complete package of information regarding the matrix. Under the old preprocessor, subroutine interfaces are not translated, so matrix names as actual arguments would refer to the storage area of the matrix only. Thus, the object passed in `CALL SUB(MATRIX)` would be the storage area of the matrix under the old PROTRAN and a complete "package" of information about the matrix under the new pre-processor. The solution would be to process old programs with the old PROTRAN system.

Object Code Compatibility

We observed that matrix arguments are passed differently under the old and new systems. There are two considerations we should make when interfacing PROTRAN programs processed under different preprocessors, thus having different calling protocols:

1. Calling programs may already be compiled. Thus we cannot assume their argument list have been transformed to take account of the new matrix passing method. Lets assume we have just written a subprogram which expects a full PROTRAN II matrix as argument.

   ```
   function mysum(m)
   ```

   Notice that the routine does not expect to be passed range and size information – this is implicit. However, we may be dealing with a caller which uses old style argument passing and which deems it necessary to pass more arguments in addition to the storage array name. To allow the routine `mysum` to deal with array arguments of either type, we should have a mechanism which intercepts arguments, decides exactly what type they are, and translates them into a form the subprogram can understand.

   A solution to these considerations is to create an intermediate program between caller and callee which acts as an interface between the argument lists. Thus we would generate, for the example in 1, two functions:

   ```
   function mysum(a,size1,size2,range1,range2)
   real matrix ::  apass(size1,size2)
   apass = full(a)
   set range apass(range1,range2)
   mysum = callmysum(apass)
   ```

   The result is to interpret the argument as an array, use the `full` function to interpret this as a matrix storage area, set the ranges correctly, and pass the resulting matrix on to the real routine. The other function would have only a as an argument and would simply allow it to fall through.

```
function mysum(a)
mysum = callmysum(a)
```

Then the body of the routine is compiled as before except that it is renamed **callmysum**. The **mysum** routines are distinguished by their argument count.

2. Called subprograms may already be compiled or Fortran translated. In keeping with our practice of factoring out unnecessary information, such as leading dimension, we should be able to make a simple call **call oldsub(m)** even if **oldsub** was written under the old system and expects to explicitly receive range variables and leading dimension arguments. The solution is analagous to the above. There is an example in the interface section of the design.

The cost of upward compatibility is high. What we discuss above is basically to have the PROTRAN II preprocessor handle two similar languages (old and new PROTRAN) and perhaps to insure compatibility at the object code level. An alternative, which is much cheaper to implement, is as follows: First, replace the old PROTRAN preprocessor with one which merely translates old PROTRAN into PROTRAN II. Second, create a processor for PROTRAN II. This alternative does not provide object code compatibility, but its cost/benefits ratio needs to be considered seriously.

2.5 Pros and Cons of Extensions to Fortran 8X

There are three types of enhancements to consider:

1. **Programming features:** Features that the Fortran 8X committee considered favorably but in the end did not include. An example is the `FOR ALL` statement, e.g.,

   FOR ALL(I = 1:N, J = N:2*N-1, I .NE. J) A(I,J) = 1.0/(I-J)

2. **Matrix Data Type:** Should one allow matrices of the various formats now supported by the IMSL library to be explicit data types or should all this be done (as now) using two dimensional arrays and packing?

3. **Problem Solving Statements:** One can create problem solving statements like `LINSYS` or `NONLIN` that do useful things in the vector/array/matrix area.

We feel programming features, as in number 1, should not be added in PROTRAN II, except perhaps where they already exist in PROTRAN (with perhaps some minor modifications.) The idea is for PROTRAN II to be a bridge from Fortran 77 to Fortran 8X array facilities. If we tinker with the programming language, then this bridge is much less satisfactory. So, no matter how big

a mistake one feels the Fortran 8X committee might have made, we should make the programming facilities for arrays in PROTRAN II the same as Fortran 8X.

The addition of the matrix data type has already been discussed. These are excellant arguments for adding the data types and furthermore the enhancement is

- somewhat orthogonal to the existing Fortran 8X features yet the syntax, declarations, etc. extend naturally to it.
- conforms naturally with the IMSL library, the PROTRAN problem solving statements and practices in the mathematical software community.

Yet the argument used to rule out programming language feature also applies here. We have elected to include this enhancement, matrix data types, in our design but feel the decision should be studied again and more thought given to finding a nice alternative that does not change Fortran 8X. In particular, one should consider using a combination of library routines and problem solving statements.

Problem solving statements are an intrinsic part of PROTRAN and thus one should not be concerned about adding more as long as they are worthwhile. For example, we roughly sketch a new call `BUILD` which shows how a valuable capability can be added. This is specified in the design section.

2.6 Use of BLAS1, BLAS2, and BLAS3

The implementation of PROTRAN II should heavily involve the use of the BLAS (Basic Linear Algebra Subroutines) at all three levels (BLAS1, BLAS2, BLAS3). A major motivation for PROTRAN II is existence of vector machines and the arrival of parallel machines. The BLAS take good advantage of these machines. The use of BLAS simplifies the structure of the PROTRAN II system and makes it easier to move the system to new machines. Thus we conclude that it is very worthwhile to make the effort of explicitly using BLAS and similar IMSL routines whenever it is possible in PROTRAN II.

3 The Easy Parts of the Merger

In their quest to design a general purpose array oriented language, the designers of Fortran 8X have introduced several mechanisms which would be useful in PROTRAN. These facilities are projected to be easy to implement because the translations appear straightforward. In this section we survey a few of these ideas.

3.1 Subsection notation for matrices

Fortran 8X is very flexible in the specification of general "array expressions". In particular, a portion of an array may be used as part of such an expression. The same utility would be useful in general matrix expressions. One would specify which portion of the source matrix is to be used by

```
lower bound : upper bound : stride
```

in each dimension. Fortran 8X will allow the upper bound to actually be less than the lower bound if the stride is negative. For reasons given in the design section, these specifications may not be used in matrix **declarations**. All matrix indices begin numbering at 1 and have step 1. As an example, the matrix expression `A(2:10:2,2:10:2)` represents a matrix having all elements of `A` in which both indices are even. The `(1,1)` element of this matrix expression is `A(2,2)`, the `(1,2)` element of this expression is `A(2,4)`, etc. Of course, this matrix expression is just an intermediate body in a calculation and need not necessarily be formed.

If this syntax is used for PROTRAN matrix expressions, then additional checks are necessary at run time to insure type integrity. For example, an arbitrary subsection of a banded matrix may not be banded. Thus, one may wish, as a practical matter, to disallow references to arbitrary matrix subsections for all but FULL matrices.

3.2 Matrix constructors for initialization

Currently the only PROTRAN tool for initialization of matrices is the visual format in which the programmer actually enters a two dimensional picture of the matrix he desires. Fortran 8X offers convenient new facilities for the initialization of arrays. These tools could be incorporated into PROTRAN for construction of vectors and FULL matrices. The basic idea is to first "construct" a vector using a primitive syntax and then to call intrinsic procedures to mold and reshape this vector into a higher dimensional matrix. The following expressions all construct the vector `[1,3,5,1,3,5,1,3,5,1,3,5]`:

```
[1,3,5,1,3,5,1,3,5,1,3,5]
4 [1,3,5]
4 [1:5:2]
```

These could then be used to construct matrices. The intrinsic function call

```
RESHAPE( MOLD = [3,4] , 4 [1:5:2] )
```

returns the FULL matrix

```
1  1  1  1
3  3  3  3
5  5  5  5    .
```

RESHAPE is a Fortran 8X array operator, but the same function name could be used for the analogous vector-matrix version. The distinction is made by checking the types of the arguments or the target object.

This methodology could be extended to diagonal or banded matrices as well. A call such as

```
BAND( NLOW=2, NUP=1, SOURCEMOLD = [3,4], SOURCE = 4[1:5:2])
```

would yield a banded matrix such as

```
. 1
  1  3
  1  3  5
  1  3  5
     3  5  .
        5     .
                 .
```

The columns of the source are used as the diagonals. This is why we must specify the shape [3,4] of the source. In order to allow the band matrix to be of any size and to treat all bands uniformly we "center" the columns of the source on the diagonals. We then pad with zeros if they are too short and ignore the first and last few characters if they are too long. The remaining elements are zero. An alternative to using this "matrix operator" approach is to create a new problem solving statement BUILD ARRAY or BUILD MATRIX. This is presented later.

3.3 Allocatable matrices

Fortran 8X allows the programmer greater control over memory management by including statements to allocate and deallocate memory according to actual need. This utility would be helpful in dealing with matrices and vectors whose actual size is not known until run time and could range

from the very small to the very large. Thus, we would have allocatable matrices and vectors. When the programmer needs a new matrix, he may call a PROTRAN statement to provide the space. Depending upon the nature of the matrix (banded, symmetric, etc.) and the declared size, an appropriate Fortran 8X memory allocation call would be generated. It is important to note that the space calculation code must now be part of the output of the preprocessor, instead of just part of the preprocessor itself, since the exact space allocation is done at Fortran run time. Fortran 8X both allocates and deallocates arrays so the PROTRAN II workspace manager can either a) continue to use a stack which sometimes wastes space or b) be modified to use heap storage.

3.4 SAVE attribute for PROTRAN objects

Fortran 8X offers facilities to extend the lifetime of local objects in procedures beyond the time of procedure exit. Arrays declared with the `SAVE` attribute may be reused, without change of data, in every call of the enclosing procedure. This facility would be nice in PROTRAN, in instances in which the programmer wishes to revisit local matrices of a procedure. The translation into Fortran 8X is straightforward – the underlying array representation of the saved matrices is simply given the `SAVE` attribute. This would be implemented by putting the SAVEed arrays at the bottom of workspace and not freeing the space upon exiting a subprogram.

3.5 INTENT Attribute for PROTRAN II Dummy Arguments

Fortran 8X will allow the programmer to optimize subroutine interfaces by explicitly defining an `INTENT` for dummy arguments. Dummy arguments may be declared with intent `IN`, `OUT`, or `INOUT`. The intent of an `IN` argument is simply as input to the subroutine, and `OUT` arguments are strictly return values. Otherwise, an argument is of intent `INOUT`, and is used both for input and output. This makes unnecessary certain data copying at procedure invokation and return. `IN` arguments do not have to be restored on procedure return and `OUT` arguments do not have to be copied at time of call. When single arguments are large data objects such as arrays or matrices, this is an important optimization. Thus, we suggest that the `INTENT` facility be extended to scalar-vector-matrix arguments as well.

3.6 Use of Intrinsics

Fortran 8X provides array oriented intrinsic functions of Fortran 8X which would have useful counterparts in a vector-matrix language. For example, there is a `TRANSPOSE` function which returns the transpose of its array argument. While the transpose facility already exists in PROTRAN vector-matrix expressions, it would be wise to replicate the Fortran 8X operator as a PROTRAN II operator. To avoid programmer confusion, this replication should be carried out for all of the

array intrinsic functions of Fortran 8X for which application to a matrix makes sense. In addition the implementer will probably find that the Fortran 8X operator will serve as the action statement in the generated code, anyway, and will be surrounded by some minor housekeeping code. For example, the Fortran 8X TRANSPOSE function could directly be used to compute the transpose of a FULL matrix. In general, however, we must construct some "front end" code which would intercept such calls. This is because of the special storage formats. If a transpose operation were invoked on a symmetric matrix, no action **should** happen – if the storage array were actually transposed, chaos would result. Given information about the nature of the matrix or vector argument, it would be the job of the front end code to make decisions about the correctness of the call and the most efficient Fortran 8X code to implement it.

Other functions would be useful as well. For example, the Fortran 8X intrinsic function MAXLOC returns the indices of the maximum value in an array. If a similar function is provided for matrices, a call to MAXLOC would be generated with the representation array as argument followed by back end code which would have to translate the result of the Fortran 8X call into the actual matrix coordinates.

The use of such intrinsics should be encouraged for reasons other than convenience. In a supercomputing environment, the Fortran 8X compiler would be fine tuned to translate the intrinsic function into highly efficient parallel or vector micro code. By using these intrinsics at the highest level, one encourages efficient scientific programming. We would implement all the intrinsics of Fortran 8X as regular library routines.

4 A Design

In this section, we will describe the proposed redesign of PROTRAN. The main points of the redesign are:

- Strengthening the role of types. In particular, a distinction is made between a vector or matrix and its storage array. Also, careful distinctions are made between matrices of different formats and special objects, such as `ALIAS` matrices and dummy arguments.
- Formalisms for allowing interaction between array and vector-matrix objects. These are in the form of coercion and conversion operators.
- Changes in ranging. The explicit `SET RANGE` of Fortran 8X is extended to PROTRAN. Old style ranging is retained and the implementation is basically the same.
- A more general user subprogram interface. Matrices and vectors are passed as atomic units.
- Introduction of scalar-vector-matrix aliasing.
- Dynamic allocation of matrices and vectors.
- New and more powerful notational devices (e.g. vector-matrix constructors, subsections, the `BUILD` statement).
- Extension of the assignment statement. This includes the conditional assignment, and a revision of the element-wise assignment to include ranges and conditions.
- Extensions of certain Fortran 8X optimizations to PROTRAN objects. These are the `INTENT` and `OPTIONAL` attributes for matrix dummy arguments, and the `SAVE` attribute for local objects of procedures.
- Replication of Fortran 8X intrinsics as PROTRAN II procedures.
- A more liberal source form.

The new language is upwardly compatible with the old, except in the area of procedure interfaces. This means, for example, that the old problem solving statements are used exactly as before. Most of the above additions are motivated by similar additions for Fortran 8X objects. Since we distinguish matrices and vectors as special data types, extending these facilities is not simply a matter of applying the Fortran 8X operator on the PROTRAN objects, but involves interpretation and understanding by the preprocessor and the PROTRAN library.

4.1 PROTRAN II Data Objects

PROTRAN II data objects have three characteristics: *structure, type*, and *format*, and a few optional *attributes*. The PROTRAN II data *structures* are SCALAR, VECTOR, and MATRIX. They may be alternatively referred to as the mathematical data structures. These structures are homogeneous aggregates of atomic data which are of *type* INTEGER, REAL, DOUBLE PRECISION, COMPLEX, DOUBLE COMPLEX, or LOGICAL. Other types such as CHARACTER or user defined derived types are not allowed. The attributes are ALLOCATABLE, SAVE, INTENT, and OPTIONAL.

Structure.

A SCALAR is composed of a single datum of one of the above types. A VECTOR is a one-dimensional collection of atomic data and a MATRIX is a two dimensional collection. An element of a vector is referenced by a single integer index and a matrix by an ordered pair of integer indices. The indices of a mathematical data structure or expression always begin at 1 and continue up to some maximum. This maximum is called the **range** and varies throughout the execution of the program. Its value is determined in a variety of manners, both implicit and explicit. This numbering scheme is more restrictive than in Fortran 8X, where numbering may begin at an arbitrary lower bound. This is to avoid conceptual difficulties with the common mathematical usage of matrices. What is meant, for example, by the "diagonal" of a matrix A having numbering A(10:20,30:40)? In one sense it would be all elements whose indices are equal. There are none. In the visual sense, the diagonal would be the elements A(10,30), A(11,31), A(12,32), etc. A programmer wishing such arcane utility should probably place his data in a Fortran 8X ARRAY and manipulate the data with special purpose code.

Format.

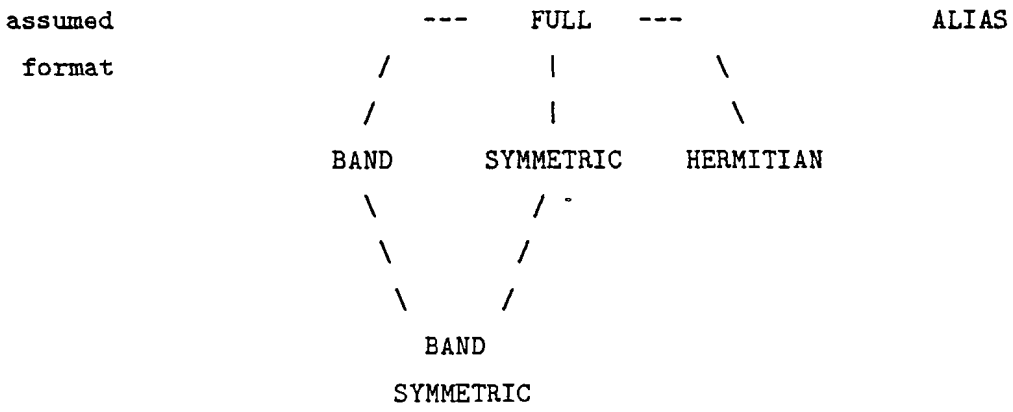
PROTRAN II provides for special *formats* for MATRIX structures. The *format* is one of BAND, SYMMETRIC, BAND SYMMETRIC, HERMITIAN, ALIAS, or *assumed-format*. The introduction of *formats* provide two important features:

- **Efficiency.** Special formats allow for packed storage of matrix elements. For example, it is necessary to store only half of the elements of a symmetric matrix. The preprocessor should generate code which capitalizes on this knowledge. Whenever possible, special routines should exist and be called for standard operations when applied to packed format matrices. Intermediate results should be stored in the smallest area possible.

- **Type checking.** The nature of the different matrix formats are sufficiently different to warrant treatment as different types. While this should not be carried to extremes[7], there are natural points in a program to enforce type checks. One of these is across subroutine interfaces. We have developed a methodology by which a subroutine may be either choosy or relaxed about the type of incoming actual arguments.

The Format Lattice

For the purposes of type checking and efficiency, it is useful to have a scheme by which the formats may be compared. The following format lattice dictates that some formats are "subtypes" of others:



The entries at the top of the lattice represent more general formats and the ones at the bottom are restrictive. When describing the effects of the various operators and assignments, we shall refer to the lowest common ancestor of two formats in this lattice as their "minimally conforming type". Assignments from matrices of restrictive format to matrices of more general format are straightforward. Thus, the result of adding a SYMMETRIC and BAND SYMMETRIC is of minimally conforming type SYMMETRIC and would fit into a SYMMETRIC matrix or a FULL matrix. However, the opposite type of assignment – from a more general format matrix to a more restrictive format – requires explicit coercion. For example, it is not clear how a FULL matrix is to fit into a SYMMETRIC matrix. Should the bottom or the top half be used? Such assignments amount to clippings and/or reflections on the source matrix. We have provided a set of restriction operators which perform such coercions.

[7] A ridiculously stringent type system would not allow many ordinary operations: e.g. F1 = F2 * B1, where F1, F2 are FULL and B1 is BAND type.

Declarations.

PROTRAN II data objects are declared at the top part of a program, along with their fellow Fortran 8X objects. The declaration takes the form

type [, format] [, attributes ...] , structure [: :] name[(specs)] ...

or

type [, format] [, attributes ...] structure(specs) [range-group-name] : : list-of-names

In the second case, all the objects in *list-of-names* are declared at once. If the *range-group-name* is present, they all "share" the same range. and this range is modified either by changing the range of a single member of the list, or by explicitly executing a SET RANGE command with the *range-group-name* as argument. This is implemented by EQUIVALENCE-ing the range variables of the list members. If *structure* is MATRIX, then the default format is FULL. The *specs* give the initial size of the matrix or vector. The declaration is just as before; i.e. *range-var = init-size* or just *init-size* or in the case of dummy arguments *range-var*. Symmetric, Hermitian, and Band Symmetric need only give a single size spec, since they are assumed square. Dummy arguments assume the size of the actual, so the spec must be either null (:) or the name of a local range variable. Assumed format matrices may have no local range variables, for this information is indeterminate, so the specifier (*specs*) is not used (neither is the format specifier).

Vector and Matrix Ranges.

All PROTRAN II vectors and matrices have the range attribute. FULL and ALIAS matrices have two range values. BAND, SYMMETRIC, BAND SYMMETRIC, and HERMITIAN matrices are assumed square and have one range variable. The number of bands is not a variable quantity. A range variable, when defined, has a current ("effective") value which is bounded by an upper bound ("declared" value). At a *point of initialization*, these two values are set equal. There are six categories of objects as far as the point of initialization is concerned:

- **Static** objects are initialized at compile time.
- **Dummy arguments.** The range values are not initialized, but instead assume the values of the actual argument object at time of call.
- **Automatic** objects are local vectors and matrices of procedures which take their size from dummy arguments. This size is initialized at time of call.
- **SAVE** objects are statically defined local objects which retain their "current" size from one invokation of the procedure to the next. The size, then, is initialized at compile time and does not need to be re-initialized.

- **ALLOCATABLE** objects. Range values are initialized at time of allocation.
- **ALIAS** objects have their size initialized at time of identification.

A range variable may be given a name and referred to separately, if desired. This is compatible with old-style PROTRAN. This association is declared by placing the range variable name in the *specs* part of the declaration. For cases in which no name is given to the other range variable, and there is no static size to be declared, the symbol : is used as a place holder. Consider:

```
SUBROUTINE SUB(DUMMY,N)
REAL MATRIX DUMMY(DUMROW,:) MAT(300,MATCOL=8)
REAL BAND MATRIX LOCAL(LOCALSIZE=N,NUP=4,NLOW=3)
```

DUMMY is a dummy argument matrix, which inherits its sizes from the actual argument at time of call. These sizes serve as the upper bound and initial value for the range variables of DUMMY. The first is given an independent name, DUMROW, and may be changed independently. Since there is no upper bound to declare for the second dimension, the colon (:) is used as a place holder. MAT is a static matrix, having one independent range variable. LOCAL is an automatically sized local matrix of the procedure, with the value of N, which should be an argument, determining the upper bound. LOCALSIZE is an independent range variable associated with that size. The diagonal measures are not range-able. NUP and NLOW are simply keywords for the declaration. Range variables are changed explicitly via the SET RANGE statement, described later, or by changing the value of an independent range variable. They also may be changed implicitly by problem solving statements or assignments.

ALIAS objects.

We have described the need for distinguishing PROTRAN II ALIAS objects as a special format type. This is more restrictive than in Fortran 8X, where ALIAS is simply an attribute placed upon an object. An ALIAS format object may not have any attributes. It must not be a dummy argument. It may have individual range variables, but they are not declared with the upper bounds (e.g. NRANGE instead of NRANGE=300). The following statements all declare ALIAS objects:

```
REAL, ALIAS, MATRIX :: TEMPLATE(TROW,TCOL), IMAGE(SIZE,:)
REAL, ALIAS, VECTOR :: DIAGONAL(LEN), TOP(N)
REAL, ALIAS, SCALAR    FRED
```

TEMPLATE is an ALIAS matrix with two independent range variables, TROW and TCOL. IMAGE has only one independent range variable, SIZE, which corresponds to the first dimension. DIAG and TOP are ALIAS vectors with range variables LEN and N. Finally, FRED is an ALIAS SCALAR.

PROTRAN II ALIAS objects may not be referenced until a mapping has been set up with other objects via a PROTRAN II IDENTIFY statement. This will be described later. The range variables, both implicit and explicit, are undefined until an identify is executed, and should not be referenced until then. The IDENTIFY statement initializes the ranges, as well as establishing an upper bound for them. After an identification is made, however, the ranges may be changed, so long as their values remain within the upper bounds established at the last IDENTIFY.

ALIAS format objects are viewed as notational devices allowing a "shorthand" for complicated expressions. They are defined by a specified mapping, which is only known inside a single scoping unit. If they are passed as actual arguments into another scoping unit, it is as if the defining mapping had been repeated by the programmer at the point of call to formulate the argument. For purposes of argument associations, ALIAS actual arguments have the FULL format attribute. The copy-restore scheme mentioned earlier is used to pass such arguments.

Assumed Format Matrices.

Assumed format objects are dummy arguments which take their *format* from the actual argument. If a dummy argument is declared without a format specified, it is assumed to be of assumed format, rather than FULL. This format is discussed at greater length in the procedure interface section.

Dummy Arguments.

A dummy argument cannot be of ALIAS format, or be given the SAVE or ALLOCATABLE attribute. A dummy argument may have the OPTIONAL and/or the INTENT(IN,OUT, *or* INOUT) attribute. The PRESENT inquiry function is extended to matrices, and can be used to check whether or not an OPTIONAL dummy argument has actually been passed. The INTENT attribute was explained in part 3. The association between actual and dummy arguments is explained in the procedure interface section.

Automatic Objects

These are local objects of a procedure. They must not be dummy arguments. They cannnot have SAVE or ALLOCATABLE attributes. At the time the procedure is invoked, these objects take their size from an expression involving dummy arguments. For example, in the interface

```
SUBROUTINE SUB(N)
REAL, BAND SYMMETRIC, MATRIX BS(Nrange = N, N/7)
```

BS is an N × N local band symmetric matrix of the procedure with N/7 bands. The range variable Nrange is initialized at time of call to N. The size of the local object changes from one invokation of the procedure to the next. This space is allocated by the Fortran 8X processor from the stack.

4.2 Mathematical assignment statements.

Assignments of new data must be made into PROTRAN II data objects. Assignments may not be made directly to the underlying storage array. If a programmer wishes to make such an assignment from an array object, he/she must use the conversion operator befitting the structure and format of the object.

Assignments to PROTRAN II objects may be distinguished from surrounding code by checking the type of the target object. Thus, we suggest that the assignment statement be simplified by deprecating the `$ASSIGN ... $` problem solving syntax. Note that this requires the preprocessor to read and understand all variables names.

The General Assignment.

This is the usual PROTRAN assignment, and takes the form

math-target = *math-expression*

The target object takes the form:

math-target	is	*scalar-name*
	or	*vector-name*
	or	*matrix-name*

We note here that this is slightly less general than the asignment available in Fortran 8X, for there the target expression may be an arbitrary array section. The target for a PROTRAN assignment must be the whole matrix, for an arbitrary subsection of a matrix is not necessarily of the same format as the parent matrix or if it is, it is not necessarily storable as such.

The primaries in the *math-expression* are *math-objects*, where

math-object	is	*math-variable*
	or	*math-constructor*
	or	*math-function-reference*
	or	(*math-object*)

and

math-variable	is	*scalar-variable*
	or	*vector-variable*
	or	*matrix-variable*

where

scalar-variable	is	*scalar-name*

and

vector-variable is *vector-name*
or *vector-name*(*section*)

and

matrix-variable is *matrix-name*
or *matrix-name*(*section*,*section*) .

The *math-constructor* is described in Section 3.2. Recall that we require matrix numbering to begin with 1 and have a step of 1. However, when formulating a general expression, a general section may be referred to:

section is *subscript [:subscript [:stride]]*

The format of a general section is allowed to be processor dependent. Sophisticated implementations may make an effort to determine the minimally necessary format of a section. For example, the section `S(1:10:k,1:10:k)` where `S` is symmetric is still symmetric (`k >= 1`). Less sophisticated processors may simply deem the section to be of type `FULL`. At any rate, there should be some effort made toward determining the minimally conformant format type of all expressions and subexpressions. For example, the expression `B1 + B2*BS1` where `B1` and `B2` are `BAND` matrices and `BS1` is `BAND SYMMETRIC` could be determined to be of minimally conforming format type `BAND`. This is important if the expression actually has to be stored, say, to be passed as an actual argument.

Assignment under mask.

We propose to extend PROTRAN to include the `WHERE` assignment statement. This is provided in Fortran 8X. The form is:

`WHERE(MASK)` *math-target* = *math-expression*

`MASK` must be of type `LOGICAL`. `MASK` and *math-target* must be of the same structure and current sizes. If *math-expression* may be evaluated for each element separately, then it is evaluated, and the assignment is made, only for those elements in `MASK` which are true.[8] Otherwise, the entire expression is evaluated and only the assignment is carried out under the mask. Caution is in order when using this expression, for the `WHERE` does not necessarily guard against dangerous values being sent to operators in *math-expression*. For example, if `FLIP` is an operator returning a whole matrix containing the inverses of each element of the argument, then the assignment

[8] In terminology of the Fortran 8X standard, this is called an elemental expression.

```
WHERE(MAT <> 0) MAT = FLIP(MAT)
```

does not insure FLIP will receive only non-zero matrix elements. The programmer should think of the WHERE clause as controlling only the assignment, and not the evaluation of *math-expression*.

Element-wise assignment.

The element-wise assignment is a feature of PROTRAN which closely resembles the defunct FOR ALL assignment of Fortran 8X and is currently explicitly removed from the Fortran 8X standard. We recommend that this facility **not** be removed from PROTRAN II and, in fact, we outline an extension of it.

The extension is to place a mask on the range of I and J values considered in the assignment. There are three possibilities: The first is of the form

FOR(*subscript-triplets*) *matrix*(I,J) = *expression in* I *and* J

FOR(*subscript-triplet*) *vector*(I) = *expression in* I

Here, the affected index values are determined by the intersection of the *subscript-triplets* with the current range(s) of the matrix or vector.

The second possibility is:

FOR(*scalar-logical-expression*) *matrix*(I,J) = *expression in* I *and* J

FOR(*scalar-logical-expression*) *vector*(I) = *expression in* I

Here, the values of I and J used are determined by the intersection of the values of I and J for which the logical expression is true with the current range(s) of the matrix or vector.

The third possibility is the old PROTRAN form:

vector(I) = *expression in* I

and

matrix(I,J) = *expression in* I,J

The valid (I,J) values are determined by the current range(s) of the matrix or vector.

In none of these forms is the range of the target changed, no matter how much the *subscript-triplets* look like a valid subrange. While this is an extension of Fortran 8X, which we promised not to do, we feel it is justified since this is simply a modification on existing PROTRAN.

4.3 Matrix-Matrix Restriction Operators.

For the purposes of the assignment statement, general matrix sections, ALIAS matrices, and FULL matrices are of the same format, for there is no optimization available – all elements must be treated upon reference and/or update. The semantics of an assignment from a restricted-format source to a more general target is clear (e.g. F = B where F is a FULL matrix and B is a BAND matrix), and if the programmer is cautious, the converse is true as well. For example, an assignment of form S = F, where S is of SYMMETRIC format and F is FULL is unambiguous given that the elements in F are truly arranged symmetrically about the diagonal. However, this is not the case in general, and special restriction operators are provided.

VECTOR(MATRIX)

Matrix must have extent 1 in one dimension. The result is a vector having the elements along the variable extent.

SCALAR(OBJECT)

OBJECT must have total size of 1. This is for use in situations which explicitly call for SCALAR arguments.

FULL(MATRIX)

The result is a FULL matrix with of the same size as the argument. This is used in cases where a subroutine requires a FULL format actual argument, and the programmer wants to use a packed format. Depending upon context, this may or may not result in copying into and padding an intermediate FULL matrix.

BAND_SYMMETRIC(MATRIX,NBANDS)

Here and below assume N is equal to the smaller extent of MATRIX. The result is an N by N BAND SYMMETRIC matrix with elements taken from the diagonal and lower NBANDS bands of the upper left N by N submatrix of the source MATRIX. We must have NBANDS <= N-1. If the extents of MATRIX are too large, the routine should be called with subrange notations on the source. For example, the result of BAND_SYMMETRIC(F(1:10,1:13),3) is a 10 by 10 BAND SYMMETRIC matrix with 5 bands with the diagonal and two lower bands selected from the upper left corner of F and the two upper bands symmetric copies of those.

SYMMETRIC(MATRIX)

The result is an N by N SYMMETRIC matrix with elements taken from the **lower** triangular half matrix of MATRIX where N = the smallest extent of MATRIX.

BAND(MATRIX,NUP,NLOW)

The result is an N by N BAND matrix with elements taken from the diagonal and surrounding bands of MATRIX. We must have NUP and NLOW <= N-1 where N = the smallest extent of MATRIX.

HERMITIAN(MATRIX)

The result is an N by N HERMITIAN matrix with elements determined from the lower triangular half matrix of MATRIX. Again, N = the smallest extent of MATRIX.

In cases where the programmer wants to use the upper triangle or upper bands as the source, TRANSPOSE(MATRIX) can be supplied as the argument.

If one of the above operators is not applied to make the meaning of the assignment clear, the assignment is carried out in a fashion as if it were supplied – i.e. all data comes from the **bottom** part of the source. The size is taken to be the minimum of the current size (range) of the source and the upper size bound of the target.

4.4 A Matrix Interface for Procedures.

Because of our atomicity requirements for scalar-vector-matrix operations, there should be a methodology for passing and receiving vectors and matrices as single entities. In the exisiting PROTRAN, one must separately pass the matrix storage, actual dimensions and range sizes when calling a **user defined** subroutine with a matrix argument. In the receiving subroutine these arguments must be collected to reformulate a matrix declaration.[9] For example, the subroutine interface

```
SUBROUTINE OLDSUB(MATSTORE,NSTORE,MSTORE,NRANGE,MRANGE)
$DECLARATION
REAL MATRIX MATSTORE(N=NSTORE,M=MSTORE)
N = NRANGE
M = MRANGE
$
```

[9]See [Rice 83] p. 465

is required to send a single FULL matrix as an argument.

We wish for the range, static size and storage format to be passed to subroutines without explicit effort by the programmer – simply giving the matrix name, section, expression, etc. should suffice as actual argument. The programmer should be able to specify the nature of the dummy argument in two seemingly contradictory fashions:

- When desired, the exact format may be specified. This may be done by explicitly giving the format on the PROTRAN declaration line corresponding to the dummy argument in question. Thus,

```
SUBROUTINE SUB(MAT)
REAL, SYMMETRIC, MATRIX MAT(N)
```

  declares a subroutine accepting only SYMMETRIC matrices as an argument. Also, an old-style range variable is declared. It assumes the range of the actual argument as its initial value. Similarly,

```
SUBROUTINE SUB(MAT)
REAL FULL MATRIX MAT(N,:)
```

  declares the dummy argument to be a FULL matrix with one old-style range variable. In this case, a coercion operator should be used to pass, say, a symmetric matrix as argument, resulting most likely in creating and filling a temporary FULL matrix[10].

- The alternative is to leave the format of the dummy argument indeterminate, and to assume the format of the actual. To induce this, the format part is not given in the argument declaration. Thus,

```
SUBROUTINE SUB(MAT)
REAL, MATRIX MAT(N,M)
```

  declares that the format of the dummy argument matrix MAT will be that of the actual argument. This is not to say that the format will be unknown, only that it is not known at compile time. The format will be fixed throughout an invokation of the subroutine, but will not be known until that invokation. The programmer must exercise caution when using *assumed-format* matrices, for the legality of their use in certain situations changes dynamically.

[10]Recall that the copy-restore method of argument passing works, officially.

The preprocessor should be prepared to handle matrices of this format, which will be distinguished from the rest and referred to as *assumed-format* matrices. We have created special logical-valued inquiry functions with which the actual format may be tested, if necessary.

The real motivation for introducing *assumed-format* matrices is efficiency. In a procedure where a matrix dummy argument is used in a generic fashion, i.e. no format-specific operations are called for by the programmer, the natural inclination is to declare the dummy argument to be a FULL matrix, providing maximum generality. However, if a calling routine wishes to pass a specially formatted matrix, say a BAND SYMMETRIC matrix, a tremendous amount of copying must first take place to provide a FULL matrix to the called routine. The use of an assumed-format dummy allows the actual-argument to be passed under maximal packing. The generated code must be more complicated, for format checks must be made dynamically, whenever an *assumed-format* matrix is used.

Note that we retain object code compatibility for we may still pass matrices to old fashion routines by using the STORE conversion operator to pass the storage area, and the ROWRANGE, COLRANGE, ROWSIZE and COLSIZE operators to explicitly pass the current range. For example, if OLDSUB is written under PROTRAN I, with an interface of form

```
SUBROUTINE OLDSUB(MAT,Rsize,Csize,Rrange,Crange)
REAL MATRIX MAT(N=Rsize,M=Csize)
N = Rrange
M = Crange
```

then we may always call it from a modern PROTRAN II program:

```
CALL OLDSUB(STORE(F),ROWSIZE(F),COLSIZE(F),ROWRANGE(F),COLRANGE(F))
```

where F is a full matrix expression.

Further, we may receive matrices in the old fashion by recieving the intended storage array, ranges, and static sizes as dummy arguments, then initially setting the ranges and copying the intended storage array to the actual matrix storage array. Schematically, this appears

```
SUBROUTINE OLDSUB(MATSTORE,NSTORE,MSTORE,NRANGE,MRANGE)
REAL FULL MATRIX MAT(N=NSTORE,M=MSTORE)
MAT = FULL(MATSTORE)
N = NRANGE
M = MRANGE
```

We have used the FULL format here only for illustration – we could have received a matrix of any format. Note that MAT is a local matrix of the procedure, so we may have to restore the array MATSTORE before return:

```
MATSTORE = STORE(MAT)
NRANGE = N
MRANGE = M
```

and perhaps

```
SET RANGE MATSTORE(N,M) .
```

This procedure could be used as an intermediary to a fully automated procedure (i.e. `CALL SUB(MAT)`) as in section 2.4.

We do not, however, retain source code compatibility, that is, old PROTRAN programs may not be translated with the new pre-processor and retain their old meaning. Specifically, when a matrix name is passed as an argument, the new protocol will be used, and all information about the matrix is passed as an aggregate. Formerly, the matrix name was simply a reference to the storage array when used in a call to a user-supplied function. Thus, if the sample code given at the beginning of this section is read by the PROTRAN II preprocessor, the arguments actually generated would differ from that of the old PROTRAN preprocessor.

Argument Association

The range of dummy arguments is determined by the "size" of the actual. The actual argument associated with a matrix dummy argument may be a general vector-matrix expression, alias object, vector-matrix section, or function result. In this case, the "size" of the actual argument is resulting size(s) of the expression. If the actual argument is a vector or matrix name, the "size" of the actual is deemed to be the current range. There are three sets of quantities to be determined.

1. The **leading dimension** of the storage of the actual argument.
2. The **upper bounds** imposed on each range of the dummy.
3. The **range values** of the dummy.

This size of the actual determines the upper bound on the range of the dummy, even though it may not be the leading dimension of the storage of the actual. The ranges are initialized to these upper bounds. If we uniformly assumed that leading dimensions are the same as range upper bounds, there would be a need to "copy" vector-matrix actual arguments into smaller temporaries at call. Copying is necessary for expressions, sections, aliases, and function results, but is wasteful when applied to matrix names.

4.5 Manipulating Object Shape.

In this section, we describe the commands which change the shape of PROTRAN II data objects.

The SET RANGE statement.

As we remarked earlier, all PROTRAN II data objects are range-able. The range may be changed implicitly through an assignment or problem solving statement, or explicitly by changing an independent range variable, or executing a SET RANGE command. This command can update both range variables of matrices at once,and possibly updates the ranges of several matrices and/or vectors at once. Recall that the range values must always remain within the upper bounds established at the point of initialization. The SET RANGE provides a visible point for **safety checks** on range values. This is necessary in light of independent ranging, where there is no hope of the compiler or pre-processor detecting all range changes for such checks. The single argument forms of the statement are:

SET RANGE *matrix-name*(*range-specifier*, *range-specifier*)

or

SET RANGE *vector-or-matrix-name*(*range-specifier*)

range-specifier is *integer-expression*

or :

The first form is for matrices with two range values, such as FULL and ALIAS format matrices. The second form is for vectors and matrices which are symmetric, such as BAND, HERMITIAN, BAND SYMMETRIC, and SYMMETRIC. If a colon is used as the specifier, the range is unchanged but its value is checked. We can also use the following multiple argument forms:

SET RANGE(*range-specifier*, *range-specifier*) :: *list*

or

SET RANGE(*range-specifier*) :: *list*

where *list* is a list of matrix or vector or *range-group* names. If a SET RANGE(*spec1*,*spec2*) is performed on an object with one range variable, such as a symmetric matrix, the *spec* value actually used is processor dependent. This is useful when dealing with assumed format objects, where the exact format of the dummy argument, and thus the appropriate number of range values, is not actually known. Thus, if S is a symmetric matrix SET RANGE S(50) effectively renders S to be a 50 by 50 matrix, while SET RANGE S(50,100) could set the range to either 50 or 100, but does not violate symmetry by making S into a 50 by 100 FULL matrix. The band count on banded or band symmetric matrices is not a range-able quantity. If it were, the semantics of ranging would be extended to causing certain elements to appear and disappear. Adjusting the ranges of a dummy argument has no effect upon the ranges of the actual. The execution of a SET RANGE upon an un-allocated matrix or vector, or an unidentified alias object may change the value of any associated independent range values, but has no effect upon the allocation status of the object. Thus, if we declare

```
REAL, FULL, ALLOCATABLE, MATRIX A(Nrange,:)
```

and then execute

```
SET RANGE A(50,100)
```

the value of the scalar `Nrange` is set to 50, but `A` is still unallocated.

The IDENTIFY statement.

The data mapping for ALIAS objects is set up using this statement. The form is

IDENTIFY(*general-alias-element* = *parent-object-expression* , *subscript-ranges*)

The subscript ranges must begin at 1 and have step 1, consistent with our numbering requirements. The *parent-object-expression* may be a *single-parent-object* or a case construct of the form:

CASE (*logical-expression*) *single-parent-object*
CASE (*logical-expression*) *single-parent-object*
...
DEFAULT *single-parent-object*

The *logical-expressions* must not recursively depend upon values in the alias object. Another alternative is to use a series of subranges:

FOR (*subscript-ranges*) *single-parent-object*
FOR (*subscript-ranges*) *single-parent-object*
...
DEFAULT *single-parent-object*

The subscripts must be those of the *general-alias-element*. If the `subscript-ranges` overlap, or more than one logical expression holds for an index value, the first is used. The ranges and upper range bounds are re-initialized at this point. The semantics of the call is to completely re-identify the alias object, not just a portion of it. For example, in the code fragment

```
IDENTIFY( A(I,J) = B(I,J), I=1:50, J = 1:100 )
IDENTIFY( A(I,J) = C(I) , I=1:50, J = 1:1 )
```

the effect of the second statement is to redeclare `A` to be a one column matrix corresponding to `C`, instead of simply superimposing `C` on the first column of `B`. This would have been accomplished with:

```
IDENTIFY( A(I,J) =
         CASE( J > 1 ) B(I,J)
         CASE( J > 0 ) C(I)
         I=1:50, J = 1:100 )
```

Notice that the second logical expression holds, but the first case is chose for the majority of the elements. The ranges of A are set to 50 and 100, respectively.

The Fortran 8X standard requires that the same parent objects be involved in all identifications for a single alias element. We would either lift this requirement altogether or simultaneously enforce it and do away with multiple-parent identifications. There may be good reasons found at the implementation phase for doing so.

Allocation and Deallocation.

The allocation and deallocation elements are simple:

ALLOCATE(*allocation-list*) DEALLOCATE(*deallocation-list*)

For allocation each element of the *allocation-list* is of the form:

matrix(*integer-expression*, *integer-expression*),

matrix(*integer-expression*),

or

vector(*integer-expression*)

For deallocation, the object name suffices. The second matrix form is for packed-format matrices. The band size is considered static, so it is not specified in the allocation. Allocation of an already allocated object is equivalent to first deallocating the space and then performing the allocation. The allocation initializes the ranges and upper bounds for the dimensions.

4.6 New Problem Solving Statements

The BUILD Matrix/Array PROTRAN II Statement

This is motivated by the addition of array constructors. This is a generalization on that theme and incorporates ideas from several array shaping intrinsic procedures in Fortran 8X. First, we give some examples.

```
1)  BUILD ARRAY A
          WITH ( I = 1:N, J = 2:M ) A(I,J) = cos(I*J + 1.2) / (I+J-1)
```

```
2)  BUILD MATRIX B =    (1.2 , 3.7*cos(x) )
                        (4.8 , exp(2*p-q) )

3)  BUILD ARRAY  A
          RANGE /Arange/ = (1:7, 12:40); TYPE = DOUBLE  PRECISION
          PAD = 0.0
          WITH ( I = 1:7:2, J = 12:20:3, X(I,J) < = 12.0 * I + cos(J) )
                 A(I,J) = 1.3 /(I+J-1) *B(I,J-4)
          AT ( I=1, J=21 ) C
```

Here, arrays B,C,X already exist. The **BUILD ARRAY** code must check the validity of the construction.

```
4) BUILD ARRAY A(I,:,J) = (X,Y,Z)
                          (R2,R3)
                          (R4,R5,R6,R7)
   WITH (L=N1:N2, M = N3:N4, K = 1:NALL )
        X(L,K,M) = L*M/(cos(K+2) + 2.5)
   WITH (L=1:72, K=1:NALL, M=N3:N4 ) R7(L,K,M) = 1
   WITH Y = 7*Z - 3*R2
```

The arrays Z,R2,R3,R4,R5 and R6 already exist. The arrays X,Y, and R7 are temporarily created. Figure 1 is the picture of A.

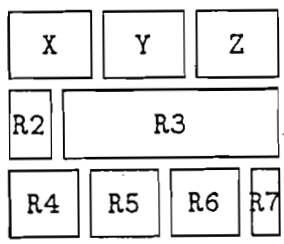


Fig. 1. The array A. All arrays extend into the third dimension.

A formal definition is as follows. Items in bold are keywords, items in *italics* are optional.

BUILD ARRAY <name> <*(subscripts)*> = <*1D or 2D list*>
BUILD MATRIX <name> <*(subscripts)*> = <*1D or 2D list*>

We require the subscript ranges of matrices to begin at 1.
The following are optional keyword phrases

RANGE / <name> / = <vector>
PAD <value>
TYPE <specification of elements>
FORMAT <specification of **BAND, SYMMETRIC**, etc.> For matrices only
WITH (<ranges>, *mask*) < array elements > = <expression>
or
WITH < array name > = < array expression >
AT (< array location >) < array expression >
Puts in < array expression > with corner at < array location >

4.7 Array to Matrix Conversion Operators.

As we remarked above, the storage areas for PROTRAN II matrices and vectors are not accessible. Thus, there is no direct way to make assignments to vector-matrix storage from an array or array expression. However, the programmer does have at his disposal an assortment of operators which "convert" an array into a matrix. An assignment to a matrix using these operators thus affects the matrix storage area, indirectly. We describe these operators in this section. Essentially the idea is to interpret the array argument as the storage area of a matrix. Notice that these operators are distinguished from the earlier restriction operators by the type of the argument.

VECTOR(A)

1. A must be a rank one array or rank two array in which only one dimension has extent greater than one.
2. The result is a PROTRAN II VECTOR having the same length as the source array.

SCALAR(A)

1. A must be a scalar Fortran object or an array of length one.
2. The result is a PROTRAN II SCALAR of the same type.

FULL(A)

1. A must be a rank two array. If A is ALLOCATABLE the A must be allocated.

2. The result is a PROTRAN II FULL matrix of the same type having the same elements as A except that the numbering is interpreted to begin at 1 and have step 1. The row and column sizes of this matrix are same as the sizes of A.

BAND(N,NUP,NLOW,A)

1. A must be a rank two array which is currently allocated.

2. We must have NUP + NLOW + 1 <= min(N, column size of A), and N <= row size of A.

3. The result is a BAND matrix with the leftmost columns of A serving as as the bands. The first column of A represents the lowest band of the result, the second column the next higher band, etc. The dimension of the resulting matrix is N.

BAND_SYMMETRIC(N,NBANDS,A)

1. A must be a rank two array which is currently allocated.

2. We must have 2*NBANDS + 1 <= N <= row size of A, and NBANDS + 1 <= column size of A.

3. The result is a BAND SYMMETRIC matrix with the leftmost columns of A as the lower bands. The first column of A represents the lowest band of the result, the second column the next higher band, etc., and the NBANDS + 1st column as the diagonal. The dimension of the resulting matrix is N.

SYMMETRIC(N,A)

1. A must be a rank one array which is allocated.

2. The result is a square symmetric matrix of dimension N. We must have N(N+1)/2 <= length of A.

3. The (i,j) element of the result is given by the $\frac{i(i-1)}{2} + j$ entry of A, if $j <= i$, and by the $\frac{j(j-1)}{2} + i$ entry otherwise.

HERMITIAN(N,A)

1. A must be a rank one array which is allocated.

2. A must have type COMPLEX.

3. The result is a square hermitian matrix of dimension N. We must have N(N+1)/2 <= length of A.

4. The (i,j) element of the result is given by the $\frac{i(i-1)}{2}+j$ entry of A, if $j <= i$, and by the complex conjugate of the $\frac{j(j-1)}{2}+i$ entry otherwise.

4.8 Matrix to Array Conversion.

We provide two operators for this purpose.

STORE(M)

M may be either a matrix or vector. The result is the packed array which stores the matrix. If the argument is a matrix expression, the minimally conformant type of the expression determines the type of the storage arrangement.

ARRAY(M)

This is shorthand for STORE(FULL(M)). In other words, it is an array containing a "snapshot" of the matrix, including off band zeros and symmetric elements.

For example, if M is the band symmetric matrix

$$M = \begin{bmatrix} 1 & 2 & 0 & 0 \\ 2 & 3 & 4 & 0 \\ 0 & 4 & 5 & 6 \\ 0 & 0 & 6 & 7 \end{bmatrix}$$

then STORE(M) is the array

$$\begin{bmatrix} 2 & 1 \\ 4 & 3 \\ 6 & 5 \\ 0 & 7 \end{bmatrix}$$

and ARRAY(M) is the array

$$\begin{bmatrix} 1 & 2 & 0 & 0 \\ 2 & 3 & 4 & 0 \\ 0 & 4 & 5 & 6 \\ 0 & 0 & 6 & 7 \end{bmatrix}$$

. As another example, suppose we wish to use the columns of a matrix M (the format is immaterial) as the bands of a band symmetric matrix. The expression is BAND_SYMMETRIC(*exp*, *exp*, ARRAY(M)). Note that this different from BAND_SYMMETRIC(*exp*, *exp*, STORE(M)) and the restriction operator BAND_SYMMETRIC(*exp*, *exp*, M).

4.9 PROTRAN II Intrinsic Functions.

Fortran 8X provides the programmer with a variety of intrinsic functions for manipulating arrays. We describe a set of functions here which operate on scalar-vector-matrix objects.

```
NORM1(OBJECT)
NORM2(OBJECT)
NORMINF(OBJECT)
NORMFROB(OBJECT)
```

- The OBJECT must be a PROTRAN II matrix or vector which is not of type logical.
- The result is the respective norm of the object, depending upon whether it is a vector or matrix.

An alternate syntax would be

```
|| ... ||1
|| ... ||2
|| ... ||INF
|| ... ||FROB
```

A pair of vertical bars would behave just like parentheses.

The following logical-valued functions may be called on assumed format objects to determine the exact nature of the argument association:

FULL_FORMAT(M)
BAND_FORMAT(M)
SYMMETRIC_FORMAT(M)
BAND_SYMMETRIC_FORMAT(M)
HERMITIAN_FORMAT(M)

The following operators return information about the shape of PROTRAN II objects. Their meanings are clear.

ROWRANGE(M)
COLRANGE(M)
ROWSIZE(M)
COLSIZE(M)
NBANDS(M)
SUPER_BANDS(M)
SUB_BANDS(M)

4.10 Fortran-Like Intrinsic Functions.

Recall that one of our objectives was to incorporate the array facilities of Fortran 8X into PROTRAN. Part of the Fortran 8X convenience is a set of intrinsic functions provided to manipulate arrays as atomic entities. For the sake of symmetry between Fortran 8x and PROTRAN II, we include the same intrinsics in the PROTRAN II definition. We do this even though the utility of such an intrinsic may be questionable in a mathematical computation. How often, for example, does a scientist have occasion to perform a circular shift on the columns of a matrix? (This is done in Fortran 8X by applying the CSHIFT function to an array argument.) In this section we specify how these Fortran 8X intrinsic functions may be used on PROTRAN II data types. We have used the same name for the matrix-operator version of the intrinsic. In the final implementation it may be deemed wise to use a slightly different name to clarify the distinction (e.g. MPACK for the matrix version instead of PACK) In the spirit of the problem solving nature of PROTRAN, we have duplicated each of the services as a problem solving statement.

An effort has been made to keep the arguments purely mathematical, even requiring, for example, that certain integer arguments be declared PROTRAN II scalars. These restrictions are not real impediments to computing with arrays and matrices in a mixed fashion, but the simply force the programmer to observe the type restrictions and to bear in mind the real difference between arrays and matrices. If necessary , the programmer may use coercion operators upon arguments,

providing a double check on type compatibilities. This stricture also simplifies the job of the pre-processor, if only a minute amount. Finally, we point out that there may be some duplication of services, in the sense that there may be other PROTRAN II matrix-specific operators or problem solving statements which do similar or identical things. The inclusion of these operators is motivated primarily by the need for consistency with Fortran 8X, and not by consideration of PROTRAN II as a self contained language. This latter consideration motivates the creation of special PROTRAN II problem solving facilities.

PRESENT(M)

M must be an OPTIONAL dummy argument of the enclosing procedure. The result is a logical value which is true iff the argument has been passed.

MERGE(TOBJECT,FOBJECT,MASK)

1. All three arguments must be of the same structure (SCALAR, VECTOR, MATRIX) and range.
2. TOBJECT and FOBJECT must be of the same type. MASK must be of type LOGICAL.
3. The result has the same structure, range, and type as TOBJECT and FOBJECT. Its format is the minimally-conforming format of TOBJECT and FOBJECT. It agrees with TOBJECT in all indices for which MASK has value TRUE and agrees with FOBJECT in all other indices.

PACK(OBJECT,MASK,VECTOR)

1. OBJECT and MASK must be of the same structure and range.
2. MASK must be of type LOGICAL.
3. VECTOR must be a PROTRAN II vector of the same type as OBJECT.
4. The length of VECTOR must be >= the total number of entries in MASK which are TRUE.
5. The result is a PROTRAN II VECTOR of range (length) the same as VECTOR and type that of OBJECT and VECTOR. The ith entry of VECTOR is the ith element of OBJECT for which MASK is TRUE. The remaining elements of the result are the same as the corresponding elements of VECTOR.

SPREAD(SOURCE, DIM, NCOPIES)

1. SOURCE must be a PROTRAN II VECTOR.
2. DIM must be a PROTRAN II SCALAR of type INTEGER with value of either 1 or 2.
3. NCOPIES must be a PROTRAN II SCALAR of type INTEGER of positive value.
4. If DIM = 1 (2), NCOPIES of SOURCE form the columns (rows) of the result, which has structure MATRIX, format FULL, and type equal to that of SOURCE.

UNPACK(VECTOR,MASK,FIELD)

1. VECTOR and FIELD must be of the same type.
2. VECTOR must be a PROTRAN II VECTOR.
3. MASK must be of type LOGICAL and of same structure (SCALAR, VECTOR, or MATRIX) and range as FIELD.
4. The length of VECTOR must be >= the total number of TRUE entries in MASK.
5. The result is the same type as FIELD and VECTOR same structure and range as FIELD and of the minimally conformant format of FIELD and MASK. Corresponding to FALSE entries in MASK, the result contains the same element of FIELD. For the ith TRUE element of MASK in standard order, the result contains the ith entry of VECTOR.

RESHAPE(MOLD,SOURCE,PAD,ORDER)

1. MOLD must be a PROTRAN II VECTOR of type INTEGER and length = 2. All entries must be 1 or greater.
2. ORDER must be a PROTRAN II VECTOR of type INTEGER and range (length) = 2. This is optional. ORDER must be the vector [1,2] or [2,1].
3. SOURCE must be a PROTRAN II VECTOR. If PAD is missing, SOURCE must have total size at least that of the product of the entries in MOLD.
4. PAD is optional and must be a PROTRAN II VECTOR of same type as SOURCE and we must have range(PAD) + range(SOURCE) >= product of entries in MOLD.

5. The elements of SOURCE used to fill the result, which is a PROTRAN II FULL MATRIX of the same type as SOURCE and PAD. These elements fill the result in column (row) first order if ORDER = [1,2] ([2,1]).

EOSHIFT(OBJECT,DIM,SHIFT,BOUNDARY)
CSHIFT(OBJECT,DIM,SHIFT)

1. OBJECT must be of VECTOR or MATRIX structure. DIM must be a PROTRAN II INTEGER SCALAR.

2. SHIFT must be of INTEGER type. If OBJECT is a MATRIX, then SHIFT must be either a PROTRAN II VECTOR or SCALAR. If OBJECT is a VECTOR SHIFT must be a SCALAR.

3. BOUNDARY is optional. Same requirements as on SHIFT except that the type must match that of OBJECT.

4. Result is the same as in the array version – each element is moved SHIFT locations in the DIM direction. In the case of EOSHIFT "shifted off" elements are replaced with either zero or the value in BOUNDARY and in the case of CSHIFT the shift is circular. The result has the same type and structure as OBJECT and format is FULL.

MAXLOC(OBJECT,MASK)
MINLOC(OBJECT,MASK)

1. MASK is an optional PROTRAN II LOGICAL object of the same structure as OBJECT.

2. Result is a PROTRAN II VECTOR of typeINTEGER containing the indices of the largest (smallest, respectively) element of OBJECT for which MASK is TRUE. If there are multiple largest (smallest resp.) elements, the result could be the indices of any of them. If OBJECT is a VECTOR (MATRIX), the result vector has length (range) = 1 (2).

TRANSPOSE(MATRIX)

1. Result has same type, structure (must be MATRIX), and format as MATRIX with the (i,j) entry now equal to the (j,i) entry. In the case of SYMMETRIC or BAND SYMMETRIC matrices, no action is taken. For BAND matrices, the number of upper and lower diagonals are swapped, and elements are swapped. For FULL matrices, the elements are swapped, as well as the row and column ranges.

2. Depending upon the sophistication of the preprocessor, this call need not necessarily result in actual copying, but may simply be used as a notational device in a calculation.

The following operators are straightforward and the descriptions are brief:

```
DOTPRODUCT(V1,V2)
```

Arguments must be PROTRAN II vectors of same length. Result is a SACLAR.

```
MATMUL(OBJ1,OBJ2)
```

Shapes must make sense for matrix multiplication. Structure of arguments and result can be vectors and/or matrices.

```
ANY(MASK,DIM)
ALL(MASK,DIM)
COUNT(MASK,DIM)
MAXVAL(OBJECT,DIM,MASK)
MINVAL(OBJECT,DIM,MASK)
PRODUCT(OBJECT,DIM,MASK)
SUM(OBJECT,DIM,MASK)
ALLOCATED(ALIAS_OBJ)
```

Same as Fortran 8X versions.

1. DLBOUND(OBJ,DIM)
2. ELBOUND(OBJ,DIM)

3. DUBOUND(OBJ,DIM)
4. DSHAPE(OBJ)
5. DSIZE(OBJ,DIM)

6. EUBOUND(OBJ,DIM)
7. ESHAPE(OBJ)
8. ESIZE(OBJ,DIM)

The first two are included for consistency with Fortran 8X only, for the lower bounds of all mathematical objects is always 1. 3,4, and 5 give the initialization values and 6,7,8 return current range values.

The lengthy list of element-wise numerical and mathematical intrinsic operators of Fortran 8X should be duplicated as well.

References

[Rice 83] J. R. Rice. *Numerical Methods, Software, and Analysis.* McGraw Hill, New York, 1983.

[Smith 87] Brian T. Smith, ed. *A Review and Analysis of Fortran 8X.* Argonne National Laboratory Report ANL-87-40, 1987.

[MR 87] Michael Metcalf and John Reid. *Fortran 8X Explained.* Oxford University Press, New York, 1987.

[Fortran 8X] ANSI X3J3 Committee. *Proposed Fortran 8X Definition.* June 1987.

Appendix 1: PROTRAN

The PROTRAN variables are declared following the keyword **SDECLARATION.** As in Fortran, variable types are specified by a keyword followed by a list of variables. The PROTRAN declarations are more complex than Fortran and of the following form:

type datastructure list

where *type* is one of INTEGER, REAL, DOUBLE PRECISION, COMPLEX, DOUBLE COMPLEX, or LOGICAL; *datastructure* is one of SCALAR, VECTOR, or *format* MATRIX; and *list* is a list of variable names with range variables (optionally) and storage size specified. There are various storage formats for matrices, and *format* specifies which is to be used. The normal case for *format* is FULL (this is the default if nothing is said). For the present discussion, we assume all matrices are in FULL format; Section 15.2.C gives a discussion of the other special storage formats, SYMMETRIC, BAND, BAND SYMMETRIC, and HERMITIAN.

The declarations of vectors and matrices involve the usual Fortran specification for storage plus the optional specification of the range variable. Each item in the *list* is of the form *vname* (*rangevar* = *number*) or *mname* (*rowrangevar* = *numberrow*, *colrangevar* = *numbercol*) where *vname* and *mname* are names of vectors or matrices, respectively. The values *number*, *numberrow*, and *numbercol* specify the storage size (the usual Fortran declaration), and *rangevar*, etc., which are optional, define range variables. Thus in REAL VECTOR R (N = 5) we have 5 places of storage set aside for the elements of R, and the value of N is the current working or actual size of R. The range variable is initially the storage size.

We list several simple but important facts about range variables.

1. *The range variable is initially set equal to the storage size specification.* Thus the N = 5 in the declaration corresponds to an actual assignment of a value to N.

2. *The range variable can be changed at any point by:*

   (A) *Ordinary Fortran statements.* Thus one can read a value for N or set N = 3.

   (B) *Assigning a value to the vector or matrix.* Suppose A is declared by MATRIX A(N = 5,N = 5); then N is 5 at the start. If A is assigned as 3 by 3 value (as in the sample program given above), then the row range and column range of A become 3 and the value of N becomes 3. Even if the range variables of A are not named, PROTRAN introduces them and their values would become 3.

   (C) *The result of a problem-solving statement.* Suppose one had declared X(M = 5) instead of X(N = 5) in the previous program and executed $LINSYS A*X = R with N = 3. Before this statement M is 5, but the result of solving the linear system gives three elements in X. Thus the range of X, that is, M, is changed to 3. **If a problem-solving statement has a numerical failure, then the range variables of the answers are set to zero.**

   Note that this convention in PROTRAN can cause unexpected results when one variable is used as the range variable for several arrays. If the $LINSYS statement of the example program above has a numerical failure (the matrix A is singular), then the range variable N is set to zero. This simultaneously sets the working size of A and R to zero.

3. *The current value of the range variable is always known to PROTRAN and governs any use of the variable.* Thus in $PRINT A or $PLOT A or $ASSIGN A(I,J) = 0.0, the action is taken for all indices up to the values of the range variables.

4. *Every vector and matrix has range variables.* If the programmer does not specify one, then PROTRAN creates a range variable. It is hidden from the programmer, but it can still be changed as in rules 2B and 2C above. So the statements

```
$DECLARATIONS
 VECTOR ERROR(5)
$ASSIGN ERROR = (1., 2., 3.)
$PRINT ERROR
```

result in three elements of ERROR being printed because the ASSIGN has changed the range of ERROR to three.

General Purpose PROTRAN Statements

These PROTRAN statements are general purpose: assignment of values, output, options, changing to Fortran, and ending programs. PROTRAN words are in boldface, variable names or expressons are in italics; types are indicated under parts of some statements. Default values are listed at the right.

ASSIGN Assignment of values to scalars, vectors, and matrices.

VARIB = *expression*
scalar Fortran expression

or

VECT = $(e_1, e_2, \ldots, e_n)$ *MATR* = $(e_{11}, \ldots, e_{1n})$ $\ldots$ $(e_{m1}, \ldots, e_{mn})$
vector list of Fortran expressions
list of Fortran expressions

or

VECT(I) = *expression in I*
vector Fortran expression

MATR(I,J) = *expression in I and J*
matrix Fortran expression

Note: Values are assigned within the row and column ranges of the variables.

or

VMNAME = *vector-matrix expression*
vector-matrix simple expression: $A+B$, $A-B$, $A*B$,
transpose (') may appear also

DECLARATIONS Following statements are PROTRAN declarations. See Section 15.2.A; this statement, if present, must precede all PROTRAN statements except possibly FORSPEC and FORDATA.

END End of program, same as Fortran STOP. END (RETURN, END in subprograms).

FORDATA Following statements are Fortran DATA statements and arithmetic statement functions.

FORSPEC Following statements are Fortran declarations.

FORTRAN Following statements are Fortran. A.S at the end of line also means Fortran.

OPTIONS Select options and set parameters of the PROTRAN system

list of options separated by semicolons

WORKSPACE	: Set working space size	5000
PRINT (*LIST*)	: Set levels for PROTRAN messages *LIST* : PREPROCESSOR, NUMERICAL, or SETUP *LEVEL*: WARN, WARN/FIX, FATAL, or NONE	WARN
QUIT(*LIST*)	: Set levels to terminate run Same as PRINT	FATAL
NAME5, NAME6	: Fifth and sixth characters of PROTRAN temporary variables	V.7
ERRVAR	: Name of global error variable for IMSL library routines	IER
LABELSTART	: First label used by PROTRAN	30000
PAGEWD	: Page width for various output (80 or 129)	80

Each problem-solving statement keyword is followed by the minimum required information (in some cases there are two forms). This is followed by a list of the optional phrases, their definition, and default values. PROTRAN words are in boldface, variable names are in italics; types are indicated under parts of the statement. Only those statements used in this book are included in this table. Default values are listed at the right.

APPROXIMATE Approximate or smooth discrete data (least squares)

DATA ; VS *POINTS* ; **BY** *FORMULA*
vector vector function name

or

DATA ; VS *POINTS* ; AT *OUTPTS* ; IS *VALUES*
vector vector vector vector

NPOINTS	:	Number of data points in *DATA*	Range (*POINTS*)
NOUTPUT	:	Number of output points in *OUTPTS*	Range (*OUTPTS*)
USING	:	Type of approximation for the data, choices are	SPLINES
		SPLINES — cubic splines	
		POLYNOMIALS — polynomials of degree NBASIS-1	
		F(K,X), K, X basic functions *F(X,K)* for *K* = 1,2,. . .,NBASIS	
		If **BY** is given, *F* can only depend on *K* and *X*.	
NBASIS	:	Number of basis functions used	Range (*POINTS*)
ERROR	:	Estimate of standard deviation for SPLINE approximation	output value
COEFFICIENTS	:	Coefficients of polynomials or basis functions	output vector

DERIVATIVE Compute derivative of Fortran function or data set

function ; D(*X*) ; AT *POINTS* ; IS *VALUES*
Fortran expression scalar vector vector

or

DATA ; VS *X* ; AT *OUTPTS* ; IS *VALUES*
vector vector vector vector

NPOINTS	:	Number of data points in *DATA*	Range (*X*)
NOUTPUT	:	Number of output points in *OUTPTS*	Range (*OUTPTS*)
ORDER	:	Order of derivative, 1 or 2	1
STEPSIZE	:	Initial stepsize for differentiating expression	1.0
ERRTARGET	:	Error target used for differentiating expression	0.001
DEFINE	:	Defines more complicated functions for differentiating	absent

DIFEQU Solve a system of ordinary differential equations

Y' = *F(X, Y)* ; **ON** (*XA, XB*) ; **INITIAL** = *YSTART* ; **DEFINE**
scalars *F* defined in Fortran

EQUATIONS	:	Number of differential equations	Range (*YSTART*)
ERRTARGET	:	Error target used in solving differential equation	0.001
ABSOLUTE	:	If present, ERRTARGET applies to absolute error	absent
SOLUTION	:	Specify how solution is returned, choices are	absent
		MVNAME Name of matrix or vector with table of solution	
		FNAME Name of function subprogram *FNAME(X,J)* for the Jth solution	
		absent The solution is printed out	
NOUTPUT	:	Number of points where *Y* is printed, stored, or saved	100

Note: One must have VECTOR *F, Y, YSTART* and MATRIX *MVNAME*
or SCALAR *F, Y, YSTART* and VECTOR *MVNAME*

EIGSYS Solve for eigenvalues and eigenvectors of a matrix

A * *X* = *LAMBDA* * *X*
matrix matrix vector matrix

NOVECTORS	:	Do not compute eigenvectors, *X* can be a dummy name	absent
ORDER	:	Order of the matrix *A*	Row Range (*A*)
NOSAVE	:	Value of matrix *A* may be changed	absent
SYMMETRIC	:	*A* is assumed symmetric even if not so declared	absent
HERMITIAN	:	*A* is assumed Hermitian even if not so declared	absent

Note: *A* cannot be a BAND or BAND SYMMETRIC matrix

INTEGRAL Compute the integral of a Fortran function or data set

function ; **FOR** (*X* = *A*, *B*)

Fortran expression or name scalars

or

DATA ; VS *POINTS*

vector vector

IS	:	Value of integral	value printed
ERRTARGET	:	Error target used in estimating integral	0.001
ABSOLUTE	:	If present, **ERRTARGET** applies to absolute error	absent
NPOINTS	:	Number of data points in *DATA*	Range (*POINTS*)
DEFINE	:	Defines more complicated functions for integration Only the function name is given after **INTEGRAL**	absent

INTERPOLATE Interpolate values of discrete data set

DATA ; VS *POINTS* ; BY *FORMULA*

vector vector function name

or

DATA ; VS *POINTS* ; AT *OUTPTS* ; IS *VALUES*

vector vector vector vector

NPOINTS	:	Number of data points in *DATA*	Range (*POINTS*)
NOUTPUT	:	Number of output points in *OUTPTS*	Range (*OUTPTS*)
USING	:	Type of interpolation for the data: choices are	SPLINES
		SPLINES cubic splines	
		HERMITES Hermite cubics	
		POLYNOMIALS polynomials of degree NPOINTS-1	
		F(*K*,*X*), *K*,*X* basis functions *F*(*K*,*X*) for *K* = 1,2,. . . ,NPOINTS	
		If **BY** is given, *F* can only depend on *K* and *X*	
COEFFICIENTS	:	Coefficients of polynomials or basis functions	output value

LINSYS Solve a system of linear algebraic equations

A * *X* = *B*

matrix vector vector

or

A * *X* = *B*

matrix matrix matrix

EQUATIONS	:	Number of equations and unknowns	Row Range (*B*)
RHS	:	Number of right-hand sides (columns of *B*)	Column Range (*B*)
POSDEF	:	Assume that *A* is positive definite	absent
NOSAVE	:	Value of matrix *A* may be changed	absent
HIGHACCURACY	:	Use iterative refinement for high accuracy	absent

NONLIN Solve a system of nonlinear equations

F(*X*) = 0 ; **GUESS** = *XSTART* ; **DEFINE**

vectors vector *F* defined in Fortran

or

F(*X*) = 0 ; **IN**(*XLOW*, *XHIGH*) ; **DEFINE**

scalar scalars *F* defined in Fortran

EQUATIONS	:	Number of equations and unknowns **IN** is used only for a single equation	Range (*XSTART*) 1
ERRTARGET	:	Error target used in solving equations	0.001
SOLUTION	:	Name of solution of equations	values printed

POLYNOMIAL Compute roots of a polynomial

COEFS

vector (coefficients in order of ascending powers)

ROOTS	:	Name of the vector of roots of the polynomial	roots printed
DEGREE	:	Degree of the polynomial	Range (*COEFS*)

Appendix 2: Array Facilities in Fortran 8X

An **array** is a set of data, all of the same type and type parameters, whose individual elements are arranged in a rectangular pattern. An **array element** is one of the individual elements in the array and is a scalar. An **array section** is a subset of the elements of an array and is itself an array.

An array with a name has one subscript for each dimension of the pattern. The pattern may have up to seven dimensions, and any **extent** (size) in any dimension. The **rank** of the array is the number of dimensions, and its **size** is the total number of elements, which is equal to the product of the extents. Arrays may have zero size. The **shape** of an array is determined by its rank and its extent in each dimension, and may be represented as a rank-one array whose elements are the extents. The rank of a scalar is zero. All named arrays must be declared, and the rank of a named array is specified in its declaration. The rank of a named array, once declared, is constant and the extents may be constant also. However, the extents may vary during execution for dummy argument arrays, automatic arrays, alias arrays, ranged arrays, and allocatable arrays.

Two arrays are said to be **conformable** if they have the same effective shape. A scalar is deemed to be conformable with any array. Any operation defined for scalar objects may be applied to conformable objects. Such operations are performed element-by-element to produce a resultant array conformable with the array operands. Element-by-element operation means corresponding elements of the operand arrays are involved in a "scalar-like" operation to produce the corresponding element in the result array, and all such element operations may be performed simultaneously.

A rank-one array may be constructed from scalars and other rank-one arrays and may be reshaped into any allowable array shape.

Construction of Array Values. An **array constructor** is defined as a sequence of specified scalar values and interpreted as a rank-one array whose element values are those specified in the sequence. The sequence of values may be specified by any combination of individual scalar values, ranges of values, rank-one arrays, and other array constructors.

If every expression in an array constructor is a constant expression, the array constructor is a constant expression. An example is:

```
REAL X (3)
X = [ 3.2, 4.01, 6.5 ]
```

A one-dimensional array may be reshaped into any allowable array shape using the RESHAPE intrinsic function.

```
Y = RESHAPE (MOLD = [3, 2], SOURCE = [2.0, 2 [4.5], X])
```

This results in Y having the 3 × 2 array of values:

```
2.0   3.2
4.5   4.01
4.5   6.5
```

Assumed-Shape Array. An **assumed-shape array** is a dummy argument array that takes its shape from the associated actual argument array.

R517 *assumed-shape-spec* **is** [*lower-bound*] :

The rank is equal to the number of colons in the *assumed-shape-spec-list*.

The size of a dimension of an assumed-shape array is the size of the corresponding dimension of the associated actual argument array. If the lower bound value is d and the size of the corresponding dimension of the associated actual argument array is s, then the value of the declared upper bound is $s + d - 1$. If the lower bound is omitted, the default value is 1. The declared lower bound is *lower-bound*, if present, and 1 otherwise.

Deferred-Shape Array. A **deferred-shape array** is an allocatable array or an alias array. An **allocatable array** is a named array whose type, type parameters, name, and rank are specified in a type declaration statement containing an ALLOCATABLE attribute, but whose declared bounds, and hence declared shape, are determined when space is allocated for the array by execution of an ALLOCATE statement (6.2.2).

An **alias array** is a named array whose type, type parameters, name, and rank are specified in a type declaration statement containing an ALIAS attribute, but whose bounds, and hence shape, are declared when it is alias associated to an array by execution of an IDENTIFY statement (6.2.6).

R518 *deferred-shape-spec* **is** :

The rank is equal to the number of colons in the *deferred-shape-spec-list*.

The size, bounds, and shape of an unallocated allocatable array are undefined, and no reference may be made to any part of it, nor may any part of it be defined. The declared lower and upper bounds of each dimension are those specified in the ALLOCATE statement when the array is allocated.

The bounds of the allocated array are unaffected by any subsequent redefinition or undefinition of specification expression variables involved in the bounds.

The size, bounds, and shape of an alias array that is not alias associated are undefined. No reference may be made to any part of it except by using certain intrinsic functions, nor may any part of it be defined. The declared lower and upper bounds of each dimension are those specified in the IDENTIFY statement when the array is alias associated.

An allocatable dummy array argument may be associated only with an allocatable actual argument. An actual argument that is an allocated array may be associated with a nonallocatable array dummy argument. An array-valued function may declare its result to be an allocatable array.

Assumed-Size Array. An **assumed-size array** is a dummy array whose size is assumed from that of an associated actual argument. The rank and extents may differ for the actual and dummy arrays; only the size of the actual array is assumed by the dummy array.

Array Elements and Array Sections.

array-element **is** *parent-array* (*subscript-list*)

Constraint: The number of subscripts must equal the declared rank of the array.

array-section **is** *parent-array* (*section-subscript-list*) [(*substring-range*)]

Constraint: If *substring-range* is present, *parent-array* must be of type character.

Constraint: At least one *section-subscript* must be a *subscript-triplet*.

Constraint: The number of *section-subscripts* must equal the declared rank of the array.

parent-array **is** *array-name*
or *structure-component*

Constraint: A *structure-component* may appear only if the component specified is an array.

subscript **is** *scalar-int-expr*

section-subscript **is** *subscript*
or *subscript-triplet*

subscript-triplet **is** [*subscript*] : [*subscript*] [: *stride*]

stride **is** *scalar-int-expr*

An array element is a scalar. An array section is an array. If a *substring-range* is present in an *array-section*, the object is an array of the shape specified by the *section-subscript-list* and each element is the designated substring of the corresponding element of the array section. For example, with the declarations:

```
REAL A (10, 10)
CHARACTER (LEN = 10) B (5, 5, 5)
```

A (1, 2) is an array element, A (1:N:2, M) is a rank-one array section, and B (:, :, :) (2:3) is an array of shape [5, 5, 5] whose elements are substrings of length 2 of the corresponding elements of B.

When the stride is positive, the subscripts specified by a triplet form a regularly spaced sequence of integers beginning with the first subscript and proceeding in increments of the stride to the largest such integer not exceeding the second subscript; the sequence is empty if the first subscript exceeds the second.

The stride must not be zero.

When the stride is negative, the sequence begins with the first subscript and proceeds in increments of the stride down to the smallest such integer equal to or exceeding the second subscript; the sequence is empty if the second subscript exceeds the first.

For example, if an array is declared as B (10), the array section B (3 : 11 : 7) is the array of shape [2] consisting of the elements B (3) and B (10), in that order. The section B (9 : 1 : −2) is the array of shape [5] whose elements are B (9), B (7), B (5), B (3), and B (1), in that order.

For another example, suppose an array is declared as A (5, 4, 3). The section A (3 : 5, 2, 1 : 2) is the array of shape [3, 2] shown below:

A (3, 2, 1) A (3, 2, 2)
A (4, 2, 1) A (4, 2, 2)
A (5, 2, 1) A (5, 2, 2)

SET RANGE Statement. Execution of a **SET RANGE statement** establishes the effective ranges for the arrays in the array name list or for the members of the range list specified by the range list name (5.2.8).

set-range-stmt	**is** SET RANGE ([*effective-range-list*]) *array-name-list* **or** SET RANGE ([*effective-range-list*]) / *range-list-name* /
effective-range	**is** *explicit-shape-spec* **or** [*lower-bound*] : [*upper-bound*]

The RANGE attribute allows arrays to have a declared upper and lower bound as in FORTRAN 77 and additionally to have a changeable effective lower and upper bound. The effective bounds provide a concise way to set the working bounds on a group of arrays and to improve the readability of the statements. For example, the following statements using the triplet notation

```
A(J:K+1, J-1:K) = B(J:K+1, J-1:K) + C(J:K+1, J-1:K) + C(J:K+1, J:K+1)
A(J:K+1, J-1:K) = A(J:K+1, J-1:K) + A(J:K+1, J-1:K)
```

may be written as follows if the RANGE attribute and SET RANGE statement are used:

```
SET RANGE (J:K+1, J-1:K) A, B, C
A = B + C + C (:,J:K+1)
A = A + A
```

Note that the declared bounds of A, B, and C are not changed by the SET RANGE statement. The only change is to the bounds used when a whole array is referenced or an array section with omitted lower bounds is referenced.

An explicit subscripted reference to an array element outside the effective bounds is allowed and is not an error. Subscript references to elements outside the declared bounds remains undefined as in FORTRAN 77.

Array IDENTIFY Statement. The array IDENTIFY statement establishes an element-by-element association between an alias-array object and an array-parent object. Such an alias has properties similar to those of an array section, but can specify a greater variety of subsets of the array elements of the parent. For example, an alias may be the diagonal of an array of rank two, or may have one subscript selecting elements of an array of derived type and another indexing a component of the array elements (Examples 1 and 2 below).

identify-array-stmt	**is** IDENTIFY (*alias-element* = ■ ■ *parent-array-element* [(*substring-range*)] , ■ ■ *alias-bounds-spec-list*)
alias-element	**is** *alias-name* (*subscript-name-list*)
parent-array-element	**is** *parent-array* (*mapping-subscript-list*)
alias-bound-spec	**is** *subscript-name* = *lower-bound* : *upper-bound*

An alias array is alias associated with a parent array following a valid execution of an IDENTIFY statement. An alias array must not be referenced or defined unless it is alias associated with a parent that may be referenced or defined. Note that an array that is alias associated must be directly or indirectly associated with a nonalias object.

Execution of an IDENTIFY statement for an alias array completely determines the declared bounds of the alias array. These are the bounds specified by *alias-bound-specs*. If the alias array also has the RANGE attribute, the effective bounds are set equal to the declared bounds.

If the same *alias-name* appears in more than one IDENTIFY statement, it must always have the same parent array unless the parent arrays are subobjects of the same named object. If the parent array is an alias array, it must be alias associated. If the parent array is an allocatable array, it must be currently allocated. Whenever an allocatable array is deallocated, all aliases associated with it cease to be alias associated.

If an alias array is alias associated, it may be used according to the rules that govern the use of data objects except for certain restrictions

The following are examples of array IDENTIFY statements:

(1) Skew section

```
IDENTIFY (DIAG (I) = ARRAY (I, I), I = 1:N)
IDENTIFY (DIAG1 (I) = ARRAY2 (3, I, I), I = 1:N)
```

(2) Array of structure components

```
IDENTIFY (PART (I) = STRUCTURE % ARRAY (I), I = 1:N)
IDENTIFY (PATTERN (I, J) = STRUCTURE (I) % ARRAY (J), I = 1:M, J = 1:N)
```

Masked Array Assignment—WHERE. The masked array assignment is used to mask the evaluation of expressions and assignment of values in array assignment statements, according to the value of a logical array expression.

General Form of the Masked Array Assignment. A **masked array assignment** is either a WHERE statement or WHERE construct.

where-stmt **is** WHERE (*mask-expr*) *assignment-stmt*

where-construct **is** *where-construct-stmt*
[*assignment-stmt*]...
[*elsewhere-stmt*
[*assignment-stmt*]...]
end-where-stmt

where-construct-stmt **is** WHERE (*mask-expr*)

mask-expr **is** *logical-expr*

elsewhere-stmt **is** ELSEWHERE

end-where-stmt **is** END WHERE

Constraint: In each *assignment-stmt*, the *mask-expr* and the variable being defined must be arrays of the same effective shape.

Examples of a masked array assignment are:

```
WHERE (TEMP > 100.0) TEMP = TEMP - REDUCE_TEMP

WHERE (PRESSURE <= 1.0)
   PRESSURE = PRESSURE + INC_PRESSURE
   TEMP = TEMP - 5.0
ELSEWHERE
   RAINING = .TRUE.
END WHERE
```

Interpretation of Masked Array Assignments. When the *assignment-stmt* in a *where-stmt* is executed, the *expr* is evaluated for all the elements where *mask-expr* is true and the result is assigned to the corresponding elements of *variable*. When a *where-block* is executed, the *mask-expr* is evaluated and the result saved by the processor. Each *assignment-stmt* in the where block is evaluated, in sequence, as if it were WHERE (*expr*) *assignment-stmt* and each *assignment-stmt* in the ELSEWHERE block is evaluated, in sequence, as if it were WHERE (.NOT. *expr*) *assignment-stmt*.

If a nonelemental function reference occurs in *expr*, the function is evaluated without any masked control by the *mask-expr*; that is, all of its argument expressions are fully evaluated and the function is fully evaluated. If the result is an array, elements corresponding to true values in *mask-expr* (false in the *expr* after ELSEWHERE) are selected for use in evaluating each *expr*.

If an elemental function reference occurs in *expr* and is not an actual argument of a nonelemental function reference, the function is evaluated only for the elements corresponding to true values in *mask-expr* (false values after ELSEWHERE).

There is a significant difference between the argument association allowed in this standard and that supported by FORTRAN 77 and FORTRAN 66. In FORTRAN 77 and 66, actual arguments were limited to consecutive storage units. With the exception of assumed length character dummy arguments, the structure imposed on that sequence of storage units was always determined in the invoked procedure and not taken from the actual argument. Thus it was possible to implement FORTRAN 66 and FORTRAN 77 argument association by supplying only the location of the first storage unit (except for character arguments, where the length would also have to be supplied). On the other hand, this standard allows arguments that do not reside in consecutive storage locations (for example, an array section), and dummy arguments that assume additional structural information from the actual argument (for example, assumed-shape dummy arguments). Thus, the mechanism to implement the argument association allowed in this standard must be more general.

Because there are practical advantages to a processor that can support references to and from procedures defined by a FORTRAN 77 processor, requirements for explicit interfaces have been added to make it possible to determine whether a simple (FORTRAN 66/FORTRAN 77) argument association implementation mechanism is sufficient or whether the more general mechanism is necessary (12.3.1.1). Thus a processor can be implemented whose procedures expect the simple mechanism to be used whenever the procedure's interface is one which uses only FORTRAN 77 features and which expects the more general mechanism otherwise (for example, if there are assumed-shape or optional arguments). At the point of reference, the appropriate mechanism can be determined from the interface if it is explicit and can be assumed to be the simple mechanism if it is not. Note that if the simple mechanism is determined to be what the procedure expects, it may be necessary for the processor to allocate consecutive temporary storage for the actual argument, copy the actual argument to the temporary storage, reference the procedure using the temporary storage in place of the actual argument, copy the contents of the actual argument, and deallocate the temporary storage.

Summary of Features. This section is a summary of the principal array features.

Whole Array Expressions and Assignments. An important extension is that whole array expressions and assignments are permitted. For example, the statement

```
A = B + C * SIN (D)
```

where A, B, C, and D are arrays of the same shape, is permitted. It is interpreted element-by-element; that is, the sine function is taken on each element of D, each result is multiplied by the corresponding element of C, added to the corresponding element of B, and assigned to the corresponding element of A. Functions, including user-written functions, may be array valued and may overload scalar versions having the same name. All arrays in an expression or across an assignment must "conform"; that is, have exactly the same "rank" (number of dimensions) and "shape" (set of lengths in each dimension), but scalars may be included freely and these are interpreted as being broadcast to a conforming array. Expressions are evaluated before any assignment takes place.

Array Sections. Whenever whole arrays may be used, it is also possible to use rectangular slices called "sections". For example:

```
A(:, 1:N, 2, 3:1:-1)
```

consists of a subarray containing the whole of the first effective dimension, positions 1 to N of the second dimension, position 2 of the third dimension and positions 1 to 3 in reverse order for the fourth dimension. This is an artificial example chosen to illustrate the different forms. Of course, the most common use is to select a row or column of an array, for example:

```
A (:, J)
```

WHERE Statement. The WHERE statement applies a conforming logical array as a mask on the individual operations in the expression and in the assignment. For example:

```
WHERE (A .GT. 0) B = LOG (A)
```

takes the logarithm only for positive components of A and makes assignments only in these positions.

The WHERE statement also has a block form (WHERE construct).

Automatic and Allocatable Arrays. A major advance for writing modular software is the presence of automatic arrays, created on entry to a subprogram and destroyed on return, and allocatable arrays whose rank is fixed but whose actual size and lifetime is fully under the programmer's control through explicit ALLOCATE and DEALLOCATE statements. The declarations

```
SUBROUTINE X (N, A, B)
REAL WORK (N, N); REAL, ALLOCATABLE :: HEAP (:, :)
```

specify an automatic array WORK and an allocatable array HEAP. Note that a stack is an adequate storage mechanism for the implementation of automatic arrays, but a heap will be needed for allocatable arrays.

Array Constructors. Arrays, and in particular array constants, may be constructed with array constructors exemplified by:

```
[1.0, 3.0, 7.2]
```

which is a rank-one array of size 3,

```
[10[1.3,2.7], 7.1]
```

which is a rank-one array of size 21 and contains [1.3,2.7] repeated 10 times followed by 7.1, and

```
[1:N]
```

which contains the integers 1, 2, ..., N. Only rank-one arrays may be constructed in this way, but higher dimensional arrays may be made from them by means of the intrinsic function RESHAPE.

Appendix 3: Fortran 8X Intrinsic Functions for Arrays

Vector and Matrix Multiply Functions.

DOTPRODUCT (VECTOR_A, VECTOR_B)	Dot product of two arrays
MATMUL (MATRIX_A, MATRIX-B)	Matrix multiplication

Array Reduction Functions.

ALL (ARRAY, DIM)	True if all values are true
ANY (ARRAY, DIM)	True if any value is true
COUNT (ARRAY, DIM)	Number of true elements in an array.
MAXVAL (ARRAY, DIM, MASK)	Maximum value in an array
MINVAL (ARRAY, DIM, MASK)	Minimum value in an array
PRODUCT (ARRAY, DIM, MASK)	Product of array elements
SUM (ARRAY, DIM, MASK)	Sum of array elements

Array Inquiry Functions.

ALLOCATED (ARRAY)	Space allocation query
DLBOUND (ARRAY, DIM)	Declared lower dimension bounds of an array
DSHAPE (SOURCE)	Declared shape of an array or scalar
DSIZE (ARRAY, DIM)	Declared total number of array elements
DUBOUND (ARRAY, DIM)	Declared upper dimension bounds of an array
ELBOUND (ARRAY, DIM)	Effective lower dimension bounds of an array
ESHAPE (SOURCE)	Effective shape of an array or scalar
ESIZE (ARRAY, DIM)	Effective total number of array elements
EUBOUND (ARRAY, DIM)	Effective upper dimension bounds of an array

Array Construction Functions.

MERGE (TSOURCE, FSOURCE, MASK)	Merge under mask
PACK (ARRAY, MASK, VECTOR)	Pack an array into a vector under a mask
RESHAPE (MOLD, SOURCE, PAD, ORDER)	Reshape an array
SPREAD (SOURCE, DIM, NCOPIES)	Replicates an array by adding a dimension
UNPACK (VECTOR, MASK, FIELD)	Unpack a vector into an array under a mask

Array Manipulation Functions.

CSHIFT (ARRAY, DIM, SHIFT)	Circular shift
EOSHIFT (ARRAY, DIM, SHIFT, BOUNDARY)	End-off shift
TRANSPOSE (MATRIX)	Transpose of matrix