Date: 22 January 1997
To: X3J3
From: Van Snyder
Subject: Proposals for Discussion for Fortran 95, 2000 and 2005

The intent of this proposal is to put ideas on the table, to get WG5 and X3J3 thinking about things that haven't been considered before, or about old things that have been forgotten, or just haven't been considered from every reasonable angle.

Most of the changes we propose are independent from each other and from other parts of the language. A few of the changes depend on each other or affect several areas of the language.

Some of the changes we propose can be implemented incrementally, that is, if changes are necessary in several areas in order to gain the full benefit of a proposal, it is nonetheless possible to implement a few of the simpler and possibly less controversial changes, see how they work, and see how difficult they are to implement, before proceeding with others.

We're interested to see the *principles* embodied by the herein proposed changes implemented into Fortran. The details are negotiable. Where details are presented, for example, in the form of concrete editorial proposals, consider them to be examples. We would be pleased to learn that some of the ideas proposed here are already under study.

This document is organized in roughly the order of the probability we attach to getting the proposals implemented, not in order according to the desirability we attach to each proposal.

This document may also be viewed on the world-wide-web at the URL `http://gyre.jpl.nasa.gov/~vsnyder/fortran`. The web contains the same material, but is organized differently. In particular, concrete proposals for change to ISO/IEC 1539:1990 are presented separately for each area of proposal. In this document, all the changes are presented in an appendix, in page-number order.

Contents

1 The "Curse of Compatibility" is not as much of a constraint as it seems

The things that we propose that might not be the same way as in Fortran 77 wouldn't really cause much trouble to users; the changes are designed so that they can be translated by a simple processor.

This is a strategy that X3J3 and WG5 could use in general. X3J3 and WG5 might go so far as to stipulate that a processor system does not conform to the standard if such a translator is not included. (Yes, I know the standard presently applies to the *language*, not to *processors*.)

2 Some of ISO/IEC 1539:1991 Needs Reorganization

Since "host association" applies to derived type definitions, as well as internal procedures and module procedures, Sections 12.1.2.2.1 and 12.1.2.2.2 should be in Chapter 14. Also, since the title of Chapter 14 is "Scope, *association* and definition" one might be tempted to look therein for a discussion of "host association."

If "very local" Variables (see section 19 on page 30) are allowed, there is one more reason to move the discussion to Chapter 14.

The discussions of list-directed and namelist input and output could be simplified by being combined, or by referring from the discussions of namelist input and output to parallel sections concerning list-directed input and output.

3 Intrinsics

3.1 Linear Algebra

Linear algebra needs `MaxAbsLoc` and `MaxAbsVal` operations, e.g. for pivoting. I don't trust the compiler's optimizer to get rid of the temp in `MaxLoc(Abs(A))`, so I'd write

```
I1 = MinLoc(A)
I2 = MaxLoc(A)
IF (ABS(A(I1)) > ABS(A(I2))) I2 = I1 ! I2 = MaxAbsLoc(A) here
```

Intrinsic `MaxAbsLoc`, `MaxAbsVal`, `MinAbsLoc` and `MinAbsVal` would be more efficient, and, for me, clearer.

(`MinAbsVal` and `MinAbsLoc` aren't needed for linear algebra – but they should be included for symmetry.)

3.2 Dealing with I/O trouble

When I try to write a robust, portable program that responds gracefully to trouble with I/O, I put `IOSTAT=` and `ERR=` clauses in I/O statements. But I can't do anything meaningful with the `IOSTAT` result except print it, and admonish the user to look in the manuals. The implementor has graciously provided explanatory error messages, but these are emitted only in the case that I *don't* mention `IOSTAT=` or `ERR=` in the control list.

Can we have a standard intrinsic subroutine that takes an `IOSTAT` value and a unit number, and prints a meaningful related message? Or, a standard intrinsic function that takes an `IOSTAT` value and a unit number, and returns a character variable in which there is a meaningful message related to the `IOSTAT` value? It needn't be the same message that would be printed if `IOSTAT=` or `ERR=` had been absent, because some vendors like to put extra stuff in some messages, but almost *any* message is better than "Error 109. Look in your manual to discover what that means," which is all I can print now, if I want to maintain portability.

3.3 ATAN2 ⇒ ATAN

`ATAN2(Y,X)` is an anachronism. Use `ATAN(Y,X)` instead, and deprecate `ATAN2`.

Also...

Also see "Subtypes of INTEGER Need Regularization" (section 26 on page 45).

4 Gratuitous Constraints

It would simplify the language, and the life of implementors, if gratuitous constraints could be removed. For example, an implementor remarked to me during the F90 standardization that it required extra code in his compiler to prevent list-directed or namelist I/O on internal files (character variables.) So why not allow it?

X3J3 responded to my suggestion during the final public comment on F90 that these constraints would not be removed. The constraint on list-

directed I/O using internal files has been removed, but the constraint on namelist I/O using internal files remains. Other bizarre constraints remain as well: In complex input items in list-directed or namelist input, blanks may appear before or after numeric items, but end-of-line may not appear between a numeric item and a parenthesis.

Similarly, Univac Fortran uses `END=` in a WRITE statement to mean "End of tape" or "End of allocated disk space." The Univac compiler writer assures me it was easier to allow than prohibit (because of the way Univac's OS-1100 returns I/O errors). For implementors who don't or can't detect such things, there's just never a jump to `END=` if it appears in a WRITE statement. That's simpler than prohibiting it.

The language could be simplified if X3J3 were to examine every constraint (both explicit and implied), and replace as many as possible with definitions of what happens, including the possibility that nothing might happen (e.g. `WRITE (3, end=10)` might never jump to statement 10 on a system that doesn't have pre-allocated disk extents) instead of prohibitions.

5 Include Line

The interpretation of the literal on INCLUDE lines is presently SYSTEM DEFINED. It should be changed to USER DEFINED. In correspondence with X3J3 during F90 standardization, I explained how this could be done – I've done it already in preprocessors, so it's not untried methodology. Allowing INCLUDE to remain SYSTEM DEFINED causes portability problems.

6 A More Complete Type System

A more complete type system could correct some oversights or mistakes in earlier Fortran standards, and allow some extensions, especially those having to do with object oriented programming, to be expressed in more natural ways.

Fortran cannot presently construct an array of pointers, except by a clumsy circumlocution.

Fortran cannot presently give a name to a what amounts to a "subtype", and therefore cannot take advantage of several optimizations developed by authors of compilers for languages that have named subtypes.

Consider extending type declarations to allow types to have attribute declarations, using a syntax similar to attribute declarations for variables. For example

```
TYPE, POINTER, INTEGER(KIND(...)) :: PI
TYPE, POINTER, TYPE(FOO) :: PFOO
TYPE(PI), ARRAY(10) :: API
TYPE(PFOO) :: APFOO(10)
```

constructs an array API of 10 pointers to integers, and an array APFOO of 10 pointers to objects of type FOO.

By combining extension to INTEGER KIND specification (see section 26 on page 45), the ability to name attributed types, and the ability to use type names for array index set specifications, one gets a significant benefit noticed by Ada compiler writers: Array bounds checking can be done significantly more completely at compile time.

Consider the following:

```
INTEGER, PARAMETER :: PR1 = SELECTED_INT_KIND(1,10)
TYPE, INTEGER(KIND=PR1) :: TR1
REAL, ARRAY(TR1) :: R
TYPE(TR1) :: IR1
```

This defines

1. A parameter PR1 that is the KIND index of a subtype of integer of which variables can take only values in 1:10.

2. TR1 is specified to be a name of that subtype.

3. R is declared to be an array with the dimension TR1. This specifies the bounds for R, and that R must be subscripted either by a variable of type TR1, or by a constant.

4. IR1 is a variable of type TR1.

If bounds are checked when values are assigned to IR1, then no bounds-checking is needed when IR1 is used as a subscript for R. Furthermore, bounds need not be checked when IR1 gets a value from another value of type TR1, or from a constant – run-time bounds checking is necessary only when IR1 gets a value from an integer of a different subtype.

A complete system of named types, with attribute specifications, allows an improved design of object oriented programming features. In particular, it allows to use the type-safe Ada "class-wide pointer" instead of the type-unsafe C++ "virtual" declaration for run-time dispatching.

Stealing some words from Ada, one might have the following fragment of an example

```
  TYPE, TAGGED :: BASE       ! TAGGED means it can be extended
    ! component declarations
  END TYPE BASE
  TYPE, NEW(BASE) :: DERIVED ! NEW(BASE) means an extension of BASE
    ! component declarations in addition to those from type BASE
  END TYPE DERIVED
  TYPE, POINTER, CLASS(BASE) :: TPCBASE ! Pointer to BASE's class
  TYPE, POINTER, TYPE(BASE) :: TPTBASE  ! Pointer to BASE only
  TYPE(BASE), TARGET :: VBASE
  TYPE(DERIVED), TARGET :: VDERIVED
  TYPE(TPCBASE) :: PCBASE ! Pointer to VBASE or VDERIVED, dispatched
  TYPE(TPTBASE) :: PTBASE ! Pointer to VBASE only, not dispatched.
  PCBASE => VDERIVED ! OK
  PCBASE => VBASE    ! OK
! PTBASE => VDERIVED ! Error
  PTBASE => VBASE    ! OK
```

Words different from TAGGED, NEW and CLASS could be used. The important idea is to be able to give names and attributes to types and subtypes.

Later (maybe never), for more type safety, it may be desirable to have recognizably separate declarations for types and subtypes. For example

```
  TYPE, INTEGER :: T1
  SUBTYPE, INTEGER(KIND(SELECTED_INT_KIND(1,10))) :: S1
```

declares T1 to be a *new type*, and prevents variables of type T1 to be assigned, or argument associated, with variables of any other type (including default INTEGER), or subtype of any type other than T1 (in particular, not S1). But S1 can be assigned, or argument associated, with any variable of type INTEGER or any subtype of INTEGER.

7 WRITE(u...) ⇒ PRINT

Every time I write a library subprogram that does output, I find myself wanting to allow output either to stdout, or to a file, or none at all. So I end up with code blocks of the form

```
  IF (UNIT == 0) THEN
    WRITE (*,100) ... ! or PRINT 100,...
  ELSE IF (UNIT > 0) THEN
    WRITE (UNIT,100) ...
  ENDIF
```

My life would be simplified here if the standard provided a mechanism to write onto a unit and have the effect of `WRITE(*,...)` or `PRINT`, and to read from a unit and have the effect of `READ format` or `READ(*,format)`. I'll propose three mechanisms that provide the functionality that I want, have no impact on other parts of the language, are compatible to earlier versions of standard Fortran, and don't add much burden for developers.

The simplest mechanism would be if the standard defined unit numbers for these purposes. The standard presently prohibits negative unit numbers, so the interpretation of standard-conforming programs would be unchanged if Fortran 2000 were to specify that `READ(-1,format)` has the same effect as `READ format` or `READ(*,format)`, that `WRITE(-1,format)` has the same effect as `PRINT format` or `WRITE(*,format)`, and that `WRITE(-2,format)` doesn't do anything. `READ(-2,format)` could be defined to do nothing, or do the same as `READ(-1,format)` or be an error – I don't care, but somebody else might have an opinion. Or X3J3 could choose different unit numbers.

One might also want to allow different unit numbers for units equivalent to Unix's *stdout* and *stderr*.

Less simple, but also workable, would be for the standard to define a syntax of INQUIRE, say `FILE=*`, that returns a unit number that one can use to get the same effect as READ and PRINT, or provide an intrinsic function to return this value (but, lacking arguments, the intrinsic would be a little goofy). (Some implementors might want to return different unit numbers for READ and PRINT, say 5 and 6. So maybe `FILE=*` and `FILE=**`, or *two* intrinsics could be used.) I haven't given any thought to how one could extend the INQUIRE mechanism to provide a non-functional unit.

Given any of these mechanisms, I could just write

```
WRITE(UNIT,what_format_100_really_is)...
```

and assume that if the user wanted stdout, or no output, UNIT would be set accordingly.

When I proposed this for F90, somebody remarked that it would be "impossible" for some implementors. I think the (unimaginative) complainer had in mind that some implementors don't pre-open units for stdin and stdout (say, 5 and 6), and therefore "couldn't" tell me the unit numbers to use. I don't see any related problem whatsoever with specifying the meaning of unit -1 as suggested above: Just test for the unit number being -1, and jump to the code for READ or PRINT.

Otherwise, one could avoid the "it's impossible" argument outlined above thus: The standard could stipulate that negative unit numbers have system-defined effect. Implementors would thereby be free to return either pre-

opened positive unit numbers for stdin etc. when INQUIRE (or an intrinsic) asks for them, or use system-dependent negative unit numbers. Anybody who puts a negative unit number into a WRITE statement, but one that wasn't gotten from INQUIRE (or an intrinsic) deserves his non-portability.

8 Minor FORMAT Extensions

Minimal width formatting

If I want to use exactly enough space to print an integer, I need to use something like

```
INTEGER J, N
CHARACTER*(15) FMT
DATA FMT /'(...  I  , ...)'/
...
N = 1 + LOG10(MAX(ABS(J), 1))
IF (I .LT. 0) N = N + 1
WRITE (FMT(8:9), '(I2)') N
WRITE (*,FMT) ... , J, ...
```

In Modula-2 I can use `WriteInt(J, 0);` which means "Write J using no more character spaces than necessary."

A simple extension to Fortran is to allow the W part of an I format descriptor to be absent or zero. Instead of the above mess, I could use

```
INTEGER J
WRITE (*, '(...  I, ...)') ... , J
```

It's even more of a mess to write a REAL using F format with as few character spaces as necessary. I usually write it into a character variable using large values of W and D, then scan to remove leading blanks, and trailing blanks and zeroes. Ick.

So it also makes sense to allow W.D to be absent in F descriptors.

An interpretation similar to list-directed input can apply on input: continue reading until a blank, comma, or end-of-line is encountered.

This is not a new idea in Fortran: One can already use an A edit descriptor, with no W given, to mean "Output the corresponding list item using no more character positions than necessary."

I've been informed that I0 and F0.w are in F95. If so, I'll remove this section from my list.

Type conversion during formatted I/O

One is presently allowed to mix numbers of type INTEGER and REAL in expressions. One cannot, however, input or output a REAL with I format, or an INTEGER with E, F or G format.

It would increase the semantic regularity if one could do this, and expect the same conversions to apply as would during mixed-mode intrinsic numeric assignment.

9 Pointers to Procedures

I don't know what's been proposed for pointers to procedures, if anything. But it's important to be careful. One cannot allow storing the address of a procedure into a pointer that is visible when the up-level (host associated) environment for the procedure no longer exists. So don't allow storing the address of an internal procedure into a variable more global than the "containing" procedure. Modula-2 solves this problem by allowing one only to store the address of a non-internal procedure, but I think that's too restrictive.

I hope F2000 will allow multiple levels of internal procedures, and allow them to be arguments. There's no reason to prevent it, other than the potential cost of deep binding. But deep binding is different from shallow binding only when a procedure is passed through a recursive invocation of itself or (one of) its owner(s). So compilers could do the simple efficient thing almost always. If Fortran 2000 allows passing internal procedures as arguments it's important to prevent copying the address of the procedure from an argument into a too-globally-visible variable.

In Ada-95, one can copy a pointer into any variable that has a lifetime no longer than the *type* of the pointed-to thing. This prevents passing internal procedures as arguments (except to other internal procedures at the same level as the passed one), because a procedure could copy an argument that is a pointer-to-procedure into a variable having a lifetime longer than the up-level environment existent at the instant the internal procedure was passed as an actual argument. (This is true because a named type having the same signature as the internal procedure must also be visible to the "owner" of the called procedure.)

What I proposed to cure the problem in Ada was equivalent to an "intent" that imposes the same constraints as Ada's "limited" type attribute. (The "limited" attribute for a type means that intrinsic assignment is undefined.) The result is that the procedure can't do anything with a "limited"

pointer except dereference it or pass it to another procedure argument that's also limited. In particular, it can't store a copy into a pointer variable that might have a longer lifetime than the up-level environment existent at the instant the procedure is passed as an argument. Therefore, it's a simple exercise in induction to prove that the up-level can't vanish between the instants of creating a pointer to a procedure (by using it as an actual argument) and calling it by way of a pointer.

If Fortran 2000 allows both pointers-to-procedures and internal procedures as actual arguments, something similar is needed.

A "limited" intent by itself looks kind of silly. It would also be useful to provide the *limited* attribute for derived types (see section 13 on page 20).

10 Control Structure Extensions / Regularizations

Several extensions to control structures are useful. Some of these regularize the language, and thereby simplify it. Extensions are discussed in the following areas:

- Extend EXIT to allow it to refer to any construct.
- Introduce a BLOCK construct.
- Extensions / Regularizations for CASE.
- GOTO could be tamed.

10.1 Extend EXIT to allow it to refer to any construct

Extend EXIT so it can apply to *any* construct, not just DO's. Of course, to maintain compatibility, one would need to use the "EXIT *construct-label*" syntax. (I tried to get EXIT extended during Fortran 90 standardization, to avoid the latter.) This has no effect on other parts of the language – EXIT from an inner DO to an outer one is already defined, and exit from an inner DO to an enclosing non-DO construct has the *same* effect on the inner DO.

To reduce the number of construct labels one needs to invent, EXIT could be further extended by including the construct type. Some syntaxes might be

EXIT [*construct-label*] [: *construct-type*]
or EXIT [*construct-label*] [(*construct-type*)]
or EXIT [(*construct-type*)] [*construct-label*]

where "[...]" means ... is optional, and *construct-type* could be DO, IF, CASE or BLOCK. This has the additional benefit that it allows to cross-check the construct label and construct type.

If EXIT is thus extended it benefits from introduction of a BLOCK construct.

10.2 Introduce a BLOCK construct

Extending EXIT and CASE, and allowing "very local" variables (see section 19 on page 30) would benefit from introduction of a BLOCK construct.

A BLOCK construct would serve four purposes:

- It provides a place to hang a construct label, to which EXIT can refer.

  Suppose you represent a set by storing elements that are its members in an array. Then to answer the question "Is X absent from S" one could write:

```
B: BLOCK (or a different keyword, say BEGIN)
     DO I = 1, N_Elements_S
       IF (X == S(I)) EXIT B
     END DO
     ! code here to cope with "X is absent from S"
   END BLOCK B
```

  Otherwise, one needs a GO TO, or a LOGICAL variable and a redundant test, or some other circumlocution that obscures the author's intent. BLOCK isn't *necessary* for this purpose, but it's better than

```
B: IF (.true.) THEN
  ...
  EXIT B
```

- It prevents GOTO's originating outside the block from landing inside.
- It can encapsulate "very local" variables (see section 19 on page 30).
- It can extend CASE to express some control strategies more clearly, thereby removing the need either for GOTO's, logical variables and extraneous tests, of gratuitous procedure-ification.

10.3 Extensions / Regularizations for CASE

I propose several extensions to SELECT CASE.

- CASE BLOCKS
- A notation that allows one to specify open or closed ranges is useful on its own, and enables the following proposal.
- REAL CASE ranges would allow eliminating arithmetic-IF without any need for temporary variables, or extraneous re-evaluations of the predicate.
- Allow .AND. *logical-expression* suffix for CASE(...)

10.3.1 CASE BLOCKS

If you have BLOCK, you could use it in conjunction with SELECT CASE in some circumstances to get rid of redundant tests, GO TO's or otherwise gratuitous procedure-ification, by allowing what might be considered improper nesting:

```
S:SELECT CASE (E)
  CASE (1,2)
  ! Blah, Blah
  BLOCK
    "Very local" declarations but no executable statements.
    CASE (3,4)
    ! Blah, Blah
    CASE (5,6)
    ! Blah, Blah
  END BLOCK
  ! Stuff here is done only after E = 3, 4, 5, 6, then
  ! implicitly either EXIT S or EXIT B, where B is
  ! an immediately enclosing BLOCK -- nesting makes
  ! sense.  You'd otherwise need a GO TO, or redundant
  ! tests, or procedure-ification to get to this stuff
  ! only after the 3,4 or 5,6 cases.
  CASE (7,8)
  ! etc
  END SELECT S
```

This extension would have no interaction to other parts of the language (except the necessity to introduce BLOCK).

10.3.2 Open or closed ranges of CASE

The CASE branches of the SELECT CASE construct can specify only closed ranges. Here's a syntax to allow closed or open ranges:

```
SELECT CASE (E)
...
CASE (<lbound> <R1> * <R2> <ubound>)
...
END CASE
```

in which <R1> and <R2> are independently allowed to be .LT. or < or .LE. or <=. Either "<lbound> <R1>" or "<R2> <ubound>" can be elided, but not both. The "*" refers to the value of E, which is only computed once. The present syntax `I:J` is the same as `I <= * <= J`. The present syntax CASE (X) remains, meaning `X == E`.

Here are some examples:

```
CASE (1 <= * < 10)     ! half-open range [1,10)
CASE (1 .lt. * <= 10)  ! half-open range (1,10]
CASE (1 < * .lt. 10)   ! fully open range (1,10)
CASE (1 .le. * <= 10)  ! fully closed range [1,10]
CASE (* < 10)          ! open semi-"infinite" range [-HUGE(E),10)
CASE (10 <= *)         ! closed semi-"infinite" range [10,HUGE(E)]
```

A syntax to denote open or closed ranges allows REAL ranges for the SELECT CASE construct.

10.3.3 Real CASE ranges

Once CASE can specify both open and closed range boundaries, it's not much of a stretch to allow REAL selectors for SELECT case. The standard already specifies the relation between the value *c* of the E in SELECT CASE (E) and the values *low* and *high* of the expressions L and U in CASE (L: U) by reference to relational operators, e.g.

> If the case value range is of the form *low : high*, a match occurs if *low* .LE. *c* .LE. *high* is true.

One could simply replace the two occurrences of ".LE." by "'R1" and "R2", where "R1" and "R2" are independently "<" ".LT.", "<=" or ".LE."), as explained in the argument to allow both open and closed range boundaries.

`CASE(L:U)` is the same as `CASE(L <= * <= U)`, missing `L RL` is the same as `-HUGE(E) <=` and missing `RU U` is the same as `<= HUGE(E)`.

Presently, when the type of E is INTEGER or CHARACTER*1, and the difference between the largest U and the smallest L is small, vendors almost surely produce an equivalent computed GOTO. Otherwise, implementors almost surely transform SELECT CASE (E)... to equivalent IF..., so REAL ranges are not a new code-generation problem for vendors.

I proposed this four times during standardization of Fortran 90. There were three red-herring objections that I demolished:

REAL relationals were a mistake we are not going to repeat.
I can't conceive of how to do mathematical software without them.

REAL DO inductors were a mistake we are not going to repeat.
REAL DO inductors were a mistake, but completely irrelevant to REAL ranges on CASE selectors.

We will not introduce REAL subscripts into Fortran.
It would be a mistake to introduce REAL subscripts into Fortran. Allowing REAL ranges on CASE selectors has nothing to do with REAL subscripts. This objection was apparently based on the incorrect assumption that SELECT CASE must necessarily be translated into computed-GOTO (but see the discussion above).

The last proposal was during the public comment, at which time ANSI mandated an explanation for rejecting each proposal. This one was not explained.

This change would have no effect on other parts of the language. The regularization would make the semantic content of the SELECT CASE construct easier to understand: All intrinsic types (other than COMPLEX, which is neither ordered nor discrete) can be used in SELECT CASE.

10.3.4 CASE() .AND. <logical-expression>

Sometimes, when using SELECT CASE, one wants to go to the selected case, and other times one wants the default action. One cannot, however, write GOTO 1000 inside a CASE branch, in order to transfer control to a label 1000 inside the CASE DEFAULT block.

By extending the CASE statement to allow `.AND.` *`logical-expression`* and defining that when *logical-expression* is .FALSE. the effect is as though the CASE statement were absent, one could selectively cause control to transfer to CASE DEFAULT.

10.4 GOTO could be tamed

One could make GO TO more palatable by introducing a COME FROM statement. Its only purpose is to serve as a target for GO TO statements, and GO TO statements can only transfer control to COME FROM statements that name the labels on the GO TO statements (which must be labelled), and a COME FROM statement can only mention the labels of GO TO statements that transfer to it. Thus:

```
100  GO TO 300
     ...
200  GO TO 300
     ...
300  COME FROM (100, 200)
```

but not

```
     GO TO 300
     ...
     GO TO 300
     ...
300  CONTINUE ! how do we get here?
```

One could retain compatibility by stipulating in the standard "If, and only if, a program unit contains a COME FROM statement, then all GO TO statements must transfer to COME FROM statements." A processor that inserts COME FROM wouldn't be large (see section 1 on page 4).

This has no impact on any other part of the language.

11 Exploiting Modern Processor Architectures

The most efficient implementations of many algorithms in linear algebra, and other disciplines as well, have different structure depending on characteristics of the processor. The characteristics of modern processors that most strongly influence the organization of an algorithm are the cache size, the number of registers, and the size and organization of the pipeline. Compiler writers are making progress in detecting structure in algorithms, and exploiting structure to produce more efficient programs, but at least at present, many algorithms require human insight to produce an efficient implementation.

Two additions to Fortran would allow algorithm authors to develop more efficient implementations of algorithms, and yet maintain portability.

An environmental inquiry intrinsic function that returns the total amount of data cache, in units of the type of its argument, would be useful to specify dimensions of intermediate variables, and parameters for "blocking."

A more complex system, that would allow production of slightly higher performance portable programs, would include an environmental inquiry intrinsic function that returns the number of levels of data cache. In such case, the environmental inquiry intrinsic function that returns the data cache size should also take an argument that denotes the cache level; environmental inquiry intrinsic functions that return the relative speed of the different levels of the cache, and the amount of data transferred, should be provided as well.

Compilers can frequently produce efficient translations by "loop unrolling." The optimal degree of unrolling depends on the number of registers and the pipeline characteristics. The problem that compilers don't address adequately is that sections of the program that produce data to be used in loops that might be fruitfully unrolled, or consume data therefrom, prevent the program from achieving optimal performance if they are not specialized in a way that depends on the degree of unrolling. Failure to so specialize may sometimes prevent a compiler from unrolling a loop.

The important factor to keep in mind is that the compiler's strategy for loop unrolling is a complicated function of the loop body, the size of the register file, the pipeline characteristics, and the investment of the compiler's author. Thus, the best strategy is to ask the compiler how much it unrolls a particular loop, not to use inquiry functions that provide the size of the register file, details of pipeline characteristics, etc., and then try to compute optimal parameters for "blocking" or other problem subdivision strategies.

Revealing the degree of unrolling of each loop would allow algorithm authors to specialize parts of algorithms that communicate with those loops. In particular, some intermediate results should be arrays with sizes that depend on the degree of unrolling, and "outer" loops may also depend thereon.

The following matrix multiplication example, that computes $C = AB$, is too short to illustrate every nuance of the problem, but it does illustrate a little of the flavor of what is needed. It does not illustrate how cache-size information can be exploited, and it's over-simplified in that it assumes dimensions are integer multiples of the unrolling degree – a "real" procedure would need to take care of the surds. The standard should specify exactly what of that is done automatically.

```
REAL A(M,K), B(K,N), C(M,N)
INTEGER, PARAMETER :: NU = UNROLLAMOUNT(U)
REAL S(NU) ! A "register" attribute would help the compiler
```

```
  DO I = 1, M
    DO J = 1, N, NU
      S = 0.0
U:    DO L = 1 , K, NU ! This loop is unrolled NU times
        S = S + A(I, L : L + NU - 1) * B(L: L + NU - 1, J)
      END DO U
      C(I, J: J + NU - 1) = S
    END DO ! J
  END DO ! I
```

The statement `S = S + A(I, L : L + NU - 1) * B(L : L + NU - 1, J)` would be clearer if written `S = S + A(I,L) * B(L,J)`, but then array sizes don't conform. This is something for more competent language designers to debate.

The environmental inquiry intrinsic function UNROLLAMOUNT takes an argument that is an executable construct label, and returns the degree to which the compiler unrolled the loop. The compiler may need to "back patch" U after completing optimization.

12 Character Type Would Benefit From Regularization

By and large, regularizations reduce surprises caused by irregularities, simplify the language description and therefore the teaching of it, and yet expand the language in that they allow one to write meaningful sentences that would otherwise be prohibited.

The scalar character type has operations that are nearly identical to array operations. There are three exceptions that prevent a unified discussion:

- One cannot use only a single expression to denote a single character in a character variable having LENGTH greater than one. One must instead use (N:N) to denote the N'th character.
- The stride in a character substring selector cannot be stated explicitly, and it is assumed to be one.
- The "dimension" of character substring selection has a subscript set that starts at one.

The discussion of character substring selection could be shortened by making it identical to array section selection, and referring the reader to discussion of the latter

It would be fairly easy to carry out the above regularizations.

Although the following changes might ultimately result in a smaller description of a more regular language, ubiquitous and extensive changes and reorganizations of the standard document would be required.

To make numeric and character arrays more similar, note that the character substring selector consists of a subscript in the *row major* position. That is, even though it is the *last* subscript, consecutive values nonetheless refer to consecutive storage units.

First add a LENGTH attribute to every type, that specifies *one* row-major dimension, just as is the case for characters, and allow the subscript and parentheses to be omitted (just as for characters) to denote the section (:). Considering that it's just another dimension, however, one might want instead to keep the "*(*array-spec*) notation, and deprecate LENGTH=. One could go so far as to allow several row-major dimensions.

Then put a brief discussion of row-major subscripting in the section on arrays, and delete separate discussion of character string selection.

13 LIMITED attribute for types and INTENT (LIMITED)

Ada uses a LIMITED attribute for derived types to un-define assignment outside of the package (= module) in which the type is defined. Fortran probably doesn't *need* it, because, unlike Ada, Fortran allows one to redefine intrinsic assignment; one could provide a defined assignment subroutine that prints an error message. On the other hand, a compile-time message would be better.

A LIMITED option for INTENT would also be useful to prevent a procedure from copying an argument, and in particular, from copying an argument that is a pointer by using pointer assignment. Copying the data to which the pointer points using ordinary assignment wouldn't be prohibited, unless the pointer points to a derived type that has the LIMITED attribute.

This prevents one from using a pointer formal argument in any way other than to dereference it (including to call a procedure by way of a pointer to it), or to associate it as an actual argument to a formal argument with INTENT(LIMITED).

This makes Pointers to Procedures (see section 9 on page 11) safe.

INTENT(LIMITED) would have no effect on other parts of the language than those described here.

The LIMITED word is swiped from Ada. A different word could be used.

14 Three Simple Language-size Reductions

14.1 Combine attribute discussions

People gripe that the standard is too big. It could be considerably shortened if separate type and attribute statements were eliminated. One way is to get rid of attribute statements entirely, a la ELF. The "curse of compatibility" militates against this choice (but see section 1 on page 4). Another way is to allow type and attribute specifications in any order before the "::", e.g. "SAVE, REAL :: R". I proposed this in 1987. The argument against it was "I like all my REAL declarations to be the first thing in the declaration." The counter-argument "This change would not *compel* you to abandon that style" seemed to carry little weight. I still think the simplification of the standard document is worth it.

One will also notice that discussions of attributes and corresponding statements are not identical. Is there a special reason for this? Are there any cases where they conflict? If so, which description is definitive?

14.2 Contains

Another simplification would be to get rid of CONTAINS. The only reason for CONTAINS (other than aesthetics) is that there's an ambiguity about

```
INTEGER, PARAMETER :: I
...
REAL FUNCTION F (I)
```

Is this an internal function header, or a declaration of a real array named FUNCTIONF?

"Significant blanks" gets rid of the problem. Alternately, one could be required to declare internal functions (and allowed to declare any functions) by using a syntax for function headers that is more symmetric to other declarations:

```
REAL, FUNCTION, RECURSIVE, RESULT(G) :: F(I)
```

The "curse of compatibility" militates against this choice, too.

But a lot of things that appear to be prohibited by the "curse of compatibility" aren't, really (see section 1 on page 4).

14.3 .EQV. ⇒ .EQ.

Carlie J. Coats, Jr. (`coats@ncsc.org` or `xcc@hilbert.ncsc.org`) has reminded me that Fortran is unique in having .NEQV. and .EQV. operators different from .NE. and .EQ. The language could be simplified by eliminating the former pair, and defining the latter pair to work as expected (like .NEQV. and .EQV.) for LOGICAL operands.

15 Object Oriented Fortran

I understand that OOF probably won't get into F2000 because of the size of the effort. Maybe X3J3 should "sneak up" on full OOF by putting a few OOP-like things into F2000, and delay others, e.g. inheritance and late (dynamic) binding, until later.

No matter when OOF gets off the ground, I think it would be wise to make it more like the Ada-95 object/type/package system than the C++ one. The Ada OOP stuff is fairly well done, while there are too many problems with the way C++ does things – especially the "virtual" mechanism for late binding (as opposed to using class-wide pointers in Ada-95) and multiple inheritance (which language theorists consider never to be necessary). I recommend J. G. P. Barnes's book **Programming in Ada 95**.

Because the Ada-95 object system is built on packages, instead of "classes" there is no need for "friends."

Ada-95 uses the separation of interface and implementation to control visibility upon usage or inheritance in the same way that C++ uses *public*, *private* and *protected* attributes. There are good additional reasons to separate interface and implementation (see section 23 on page 34) in Fortran.

16 INTENT(OUT) for Module Variables

A very small change, that has no effect on other parts of the language, but that could simplify programs, is to allow module variables to have INTENT(out), interpreted to mean that "using" modules can only read the variable, not change it.

When (if ever) a parameterless function F can be referenced without (), that is, by F alone (see section 25.5 on page 44), one could change between a module variable F having INTENT(out) and a parameterless module function F without changing the references.

17 More Can Be Done With Fortran I/O

We describe here several additional facilities that might be accessed via Fortran I/O statements. They have negligible impact on other parts of the language.

Each requires new keywords in OPEN. Most also require addition of a CONTROL statement, with keywords that depend on the characteristics of the unit, as mentioned in the OPEN statement. For some kinds of I/O units, additional keywords may be allowed in READ, WRITE and INQUIRE.

CONTROL could subsume and therefore deprecate REWIND, ENDFILE and BACKSPACE.

The extensions are in six categories:

"Window" units
: Views onto a graphics "canvas" of a specified size. The I/O library or system is responsible for changing the view depending on the window size, stacking order, or scroll-bar manipulations (a la SGI).

"Event" units
: Can be used for notification that a pointing device cursor has entered one of several specified rectangles, or one of the pointing device's buttons has been depressed or released. They are associated with a WINDOW unit.

"Pipe" units
: Can be used for connections to other processes. Processes might be dependent processes on the same system, or independent processes on different systems. This could be used in place of MPI or PVM (but the implementor would be free to build "pipe" units using MPI or PVM) so that when the next improvement on inter-process communication comes out (and there have been at least three so far), codes aren't instantly broken.

Non-synchronous I/O
: Could be supported with minor extensions to READ, WRITE, INQUIRE and CONTROL. Nothing new should be needed in OPEN or CLOSE.

"Associated Variables"
: Variables associated to random-access files are very useful. This gives virtual memory, and persistent data. On some systems (especially Multics, but who uses that any more?), it comes automatically from

the way the memory manager works. A variable that's associated with a file could be a scalar, an array, or of a derived type. In the latter two cases, the upper bound of the last dimension (of the last field) could be "*".

Record Length

Using INQUIRE with an I/O list to get the correct value for the record size always seemed circuitous to me. I would prefer to put an I/O list on an OPEN statement to specify the record length. Something like

```
OPEN (r, FILE='bar', ACCESS='DIRECT', RECL=*) &
      DerivedTypeVar, (A(i), i = j, k)
```

would be easier and clearer than

```
INQUIRE (IOLENGTH=L) DerivedTypeVar,(A(i), i = j, k)
OPEN (r, FILE='bar', ACCESS='DIRECT', RECL=L)
```

17.1 Window units

Fortran I/O could be extended to "Windowing" environments by the following changes.

- Add keywords to OPEN to specify height and width of canvas and initial view, and measurement units thereof (all the same, and restricted to pixels, fractions of the total screen, inches, millimeters, and, maybe, points), colors (background, foreground, border, scroll bars), title.... e.g.

  ```
  OPEN(w, kind="WINDOW", sizes=(/10,200,5,8/), &
    units="INCHES", colors=(/63,0,23,12/), view=(/0,0/) )
  ```

  says "Create a canvas 10 inches wide and 200 inches high, and a view into it that is 5 inches wide and 8 inches high. Background is white, foreground is black, borders are salmon?, scroll bars are blue? and the initial view is at (0,0)."

- Formatted WRITE puts text on the window at the cursor, and scrolls if necessary.

- Unformatted WRITE must provide an even number of numbers (either integer or real). If only two are provided, a point appears in the window. If 2*n are provided, a curve connecting n points appears.

- CONTROL is used to
  - Change the color, line style (solid, dash, dot, long and short dashes, ...), width, smoothness (polygon, Bezier, cubic B spline, ...), closure, fill color and/or pattern, ... of a line output by an unformatted WRITE statement. e.g.

```
CONTROL(w, linestyle=3, width=0.01, &
  smooth='BEZIER', color = 4)
```

  - Change color and cursor for text output by a formatted WRITE statement.
  - Change the title at the top of a window.
  - Change any of the colors previously specified by OPEN.
  - Specify a symbol, its size and color, and whether it's filled or open, to appear at each point on a curve.
  - Delete text at a specified position.
  - More speculative: CONTROL might place a menu bar at the top of a window (but maybe this should be done with WRITE on a different unit – similar to EVENT units (see section 17.2 on page 25), or by WRITE on the same unit, but with special keyword(s)?).
- INQUIRE can be used to
  - Recover any parameter from OPEN.
  - Discover whether the "close" widget has been selected. Remember, scroll bars, resizing, exposing, maximizing and minimizing are all handled by the I/O library or the system, not by the user.
  - If CONTROL can place a menu bar, INQUIRE could inform the program whether anything had been selected, and if so, what had been selected (but maybe this should be done with READ on a different unit – similar to EVENT units (see section 17.2 on page 25), or by READ on the same unit, but with special keyword(s)?).

17.2 Event units

Event units can be used for notification that a pointing device cursor has entered one of several specified rectangles, or one of the pointing device's buttons has been depressed or released. They are associated with a WINDOW unit.

- Keywords in OPEN stipulate the kinds of events to monitor. e.g.

```
OPEN(e, kind="EVENT", window=w, events=(/1,2/) )
```

- WRITE adds to the list of rectangles. A multiple of four numbers must be written.
- REWIND or CONTROL(e,REWIND) erases the list of rectangles.
- INQUIRE can be used to discover if a button has been depressed or released, or if a rectangle entry or exit event has occurred.
- READ can be used to return the pointing device coordinates. One or two variables (or an array of length one or two) must be specified in the I/O list. If just one cell is provided, only X is returned.
- CONTROL can be used to change the color and shape of the pointing device cursor.

Maybe all of this could be done directly with WINDOW units, by using different keywords in READ, WRITE, INQUIRE and CONTROL?

17.3 Pipe units

Pipe units can be used for connections to other processes. Processes might be dependent processes on the same system, or independent processes on different systems. This could be used in place of MPI or PVM (but the implementor would be free to build "pipe" units using MPI or PVM) so that when the next improvement on inter-process communication comes out (and there have been at least three so far), codes aren't instantly broken.

- OPEN starts the asynchronous process, and specifies how it is to be connected – formatted or unformatted. Sequential access is assumed. e.g.

```
OPEN(p, kind="PIPE", command="uncompress <"//file)
```

  would run uncompress with stdin redirected from "file", and the output piped to unit "p". Or

```
OPEN(p, kind="PIPE", port="other.machine:8000")
```

would connect unit "p" to port 8000 on machine "other.machine". "port" could be dependent on the kind of network in use.

```
OPEN(p, kind="PIPE", TO=q)
```

would connect unit "p" to unit "q", so that data written onto unit "p" could be read from unit "q", and vice-versa. This would be useful for connecting different processes in a single program / multiple data computer, e.g. Cray T3D.

- WRITE sends data to the other process's standard input. (What should be done if stdin is redirected in the "command" part?)
- READ reads data from the other process's standard output.
- INQUIRE can be used to determine
  - Whether input is available from the pipe.
  - Whether the pipe is ready for output. Maybe it's never ready if stdin has been redirected in the "command"? (See also "Non-synchronous I/O" in section 17.4 on page 27.)
  - Whether the process is still active.
  - Whether the process is signalling.
- CONTROL can be used to
  - Kill the process.
  - Initiate a signal to the process.
  - Wait for I/O transfers (see also "Non-synchronous I/O" in section 17.4 on page 27.).

17.4 Non-synchronous I/O

Non-synchronous I/O could be supported with minor extensions to READ, WRITE, INQUIRE and CONTROL. Nothing new should be needed in OPEN or CLOSE.

- Allow a new keyword in the control list for READ and WRITE, say SYNCHRONOUS = <LOGICAL VARIABLE> (default .TRUE.)
- Allow a new keyword in INQUIRE that returns a LOGICAL value indicating whether I/O is still in progress. Say, ACTIVE = <LOGICAL VARIABLE>.

- Add a keyword in CONTROL to wait for I/O to finish, say WAIT = <LOGICAL VALUE> (WAIT=.true. means wait, else don't).

Non-synchronous I/O to local devices isn't really different from non-synchronous I/O to other processes (see section 17.3 on page 26), so the two should be considered together.

Non-synchronous I/O isn't really different from I/O in a different thread (see section 18 on page 28). Threads are more general, but the generality may have a price. Some systems may support non-synchronous I/O but not threads, and vice-versa. It's probably OK to have both mechanisms.

The standard could allow implementors to ignore SYNCHRONOUS in a READ or WRITE statement, and then always to return .TRUE. for the variable associated to ACTIVE= in INQUIRE, and never WAIT in a CONTROL statement.

17.5 Associated variables

Implementing associated variables into Fortran wouldn't be a large change in the syntax or semantics. It might be a bit of work for implementors on some systems, and no work at all on others.

Add a new keyword to OPEN, e.g.

```
 OPEN (u, ASSOCIATED=var, FILE='foo')
```

and if "var" is of a parameterized derived type, maybe

```
OPEN (u, ASSOCIATED=var, FILE='foo') P1, P2, ...
```

Thereafter, accesses to `var` are accesses to the file `foo`.

READ, WRITE, REWIND, ENDFILE and CONTROL are not allowed for unit "u" but CLOSE and INQUIRE make sense.

Until CLOSE, `foo` can't be otherwise opened, and `var` can't be otherwise associated.

18 Threads

In addition to making I/O asynchronous (see section 17.4 on page 27), why not just make anything asynchronous? On systems that support threads, one could expect something like the following

```
P: asynchronous
     ! Asynchronous stuff P (including I/O)
```

```
    end asynchronous p
    ! Synchronous stuff
  Q: asynchronous
       ! Asynchronous stuff Q
    end asynchronous p
    ! More Synchronous stuff
    waitfor (P, Q) ! can't appear inside P or Q
                   ! but waitfor (P) could appear inside Q
```

to execute "Asynchronous stuff P" and "Asynchronous stuff Q" in different threads, perhaps on different processors, and piecemeal interleaved with pieces of each other and "Synchronous stuffs." On systems that don't support threads, the effect is as if

```
X: asynchronous
     ...
   end asynchronous X
```

were absent.

Asynchronous blocks could be nested, and there's an implicit WAITFOR of all internal asynchronous blocks at the end of an outer one. The parts of an asynchronous block, except for internal asynchronous blocks, are executed synchronously with respect to each other.

There's an implicit WAITFOR of all asynchronous blocks in a procedure at every RETURN or STOP.

[If the locality of block names is too restrictive, consider using

```
use SYSTEM_ASYNCHRONOUS, ONLY : SYNCH, WAITFOR
type(SYNCH) :: P
asynchronous (DONE = P)
  ! Asynchronous stuff (including I/O)
end asynchronous
! Synchronous stuff
call WAITFOR (P)
```

so P could be exported from a module, or in COMMON, or a dummy argument. Using a derived type for P instead of a LOGICAL variable avoids the hazard that somebody might be tempted to write

```
P = .TRUE.
```

during execution of an asynchronous block, just to see what kind of hell breaks loose.

SYSTEM_ASYNCHRONOUS is a standard intrinsic module that contains private procedures that implement the fork/join mechanism. SYNCH is a PRIVATE type. Furthermore, if LIMITED types are implemented (see section 13 on page 20), then SYNC should be a LIMITED type, so only the fork/join mechanism can change or copy P. Otherwise, a defined assignment subroutine that prints an error message and stops should be provided for type SYNCH (but a compile-time message would be better).

This is still dangerous because somebody might write

```
asynchronous (DONE = P)
 ...
asynchronous (DONE = P)
```

without an intervening WAITFOR, but maybe it would be enough to define "asynchronous" to WAITFOR its DONE argument before proceeding.

Your choice.]

Is there difference between `FORALL I = 1, 10` and

```
FOR I = 1, 10
  asynchronous (DONE=P(i))
    ! blah blah blah
  end asynchronous
END FOR
call WAITFOR(P) ! waits for all P's to indicate their
                ! associated blocks are finished
```

other than that the asynchronous blocks are started in order? If so, and asynchronous blocks are defined, is FORALL really needed?

Variables that are set in asynchronous blocks do not have defined values outside the asynchronous block until a WAITFOR is executed on the block.

One might benefit from an intrinsic BUSY(P) that returns .TRUE. if P hasn't finished, and maybe some signalling stuff, but they're not *necessary* and could be postponed without wrecking the basic idea.

This should have almost no interaction to other parts of the standard, and the standard could allow it to be a NO-OP on systems that don't support threads.

19 "Very Local" Variables

I've found it useful to have "very local" variables. E.g., in Ada, I can write

```
declare
  i: integer
begin
  -- stuff that uses my own private I, not one further away
  -- that may or may not be expected to retain a value for
  -- some later usage.
end
```

Or in C++ I can write {int i; .. } for the same purpose.

This could be accomplished in Fortran simply by relaxing the rules for statement ordering, to allow type, parameter and attribute declarations inside any construct. Then

```
BLOCK ! or IF (foo) THEN, etc.
  integer I, J
  ! Blah, Blah
END BLOCK
```

means that I and J have an existence that extends from their declaration to the end of the construct in which they're declared.

This controls *visibility* but not necessarily *lifetime*, so allowing the SAVE attribute makes sense. No other attributes should be controversial.

"Very local" variables would have no effect on anything else in the language, and almost no effect on the standard – they're a trivial extension of "host association."

20 Is FPP or CoCo Necessary?

A lot of what is proposed for FPP could be done by using IF and CASE with constant predicates. This could conflict with "very local" variables, but one could "export" what would be "very local" variables (see section 19 on page 30) by using a PUBLIC statement. Assume VERSION is a character PARAMETER. Then, e.g.:

```
IF (VERSION == 'INTEGER') THEN
  INTEGER I, J
  PUBLIC
ELSE IF (VERSION == 'REAL') THEN
  REAL I, J
  PUBLIC
END IF
```

or

```
SELECT CASE (VERSION)
  CASE ('INTEGER')
    INTEGER I, J
    PUBLIC
  CASE ('REAL') ...
```

might appear in a sorting routine.

To allow parametric presence of branches of SELECT CASE constructs, one might want to allow:

```
SELECT CASE (E)
  CASE (1)
  ! Blah, Blah
  IF (Case_2_present) THEN
    CASE (2)
    ! Blah, Blah
  END IF
  CASE (3)
  ...
```

but that's kind of wordy. Instead, allow:

```
SELECT CASE (E)
  CASE (1)
  ! Blah, Blah
  CASE (2) .and. Case_2_present
  ! Blah, Blah 2
  CASE (3)
  ...
```

The meaning is "If E == 2 then if Case_2_present do 'Blah, Blah 2' else go to CASE DEFAULT endif endif". If Case_2_present is .FALSE. and constant, the compiler should delete CASE(2) and "Blah, Blah 2" with the effect that when E == 2, CASE DEFAULT gets executed. See also section 10.3.4 on page 16.

21 A SWAP Operator Would Be Useful

Something I'd like to see that is simple and not on your list is a swap operator. Thus

```
A:=:B ! or ??? swaps the contents of A and B.
A<=>B ! or ??? swaps the pointers A and B.
```

This would be allowed anywhere that both A=B and B=A (or A=>B and B=>A) are allowed. One could also restrict A and B to be of the same type without losing any important functionality. This is

- Easy to implement,
- Easy to describe,
- Has negligible interaction with the rest of the language,
- Makes some code more clear (One needn't wonder "is that temp. variable ever used someplace else?"),
- Has frequent use,
- Is more likely to be optimized by a simple compiler.

It should be indivisible when applied to scalars, so it could be used for "test and set" in multiprocessor systems, and it should be indivisible for each component of composite objects.

22 Enumeration Types Would Be Useful

I didn't see enumerated types on the list for F2000. Has that been explicitly prohibited, or just overlooked? I don't see what's hard about them. One could declare their types:

```
    ENUMERATION COLOR (RED, GREEN, BLUE)
or  TYPE, ENUMERATION(RED, GREEN, BLUE) :: COLOR
```

and objects of the type:

```
  TYPE (COLOR) PIXEL
```

and (at least for now) prohibit anything except assignment, test for equality and argument binding.

The standard could specify a representation, e.g. 0, 1, ..., as in Modula-2. Then, other relational operators make sense. An ORD() intrinsic could return the underlying integer representation, and a reference that looks like a function but has the same name as an enumeration type, taking an integer

argument, could go the other way. E.g. ORD(GREEN) returns 1, and COLOR(1) is GREEN.

It is useful to allow enumeration types to be array index sets, but that could be postponed without damaging anything, or risking future incompatibility.

Later (perhaps never) one might allow user-specification of the representation, e.g.

```
TYPE, ENUMERATION(RED, GREEN=12, BLUE) :: COLOR
```

means RED is represented by 0, GREEN by 12, and BLUE by 13.

Literals of the enumerated type behave like integer PARAMETERs, but can only be used with variables and parameters of the type, and variables of the type can not be used with other types. (There is an obscure reason in Ada for considering enumerated type values to be parameterless functions.)

23 Module Interfaces and Implementations Should Be Separated

Both Modula and Ada allow separating module interface and implementation into separate program units. There are several advantages of this approach, as compared to having interface and implementation in a single program unit.

- If one changes the implementation but not the interface, one does not trigger recompilation of dependent modules.
- Vendors of software component libraries who do not wish to publish the source form of the implementation of their product could publish machine-readable source form of the interface without compromising their "trade secrets."
- Several Ada implementors have found that a compiler that reads the source form of the interface is just as fast as a compiler that reads a "pre-compiled" form of the interface. This simplifies the compiler, and further reduces the opportunities for "compilation cascades." (In systems that keep a pre-compiled form of interface modules, compiling the interface module, even if it's not changed, can trigger a "compilation cascade.")

- Ada-95 uses the separation of interface and implementation, together with *private* specifications in the interface module, to provide the analog of C++ *public*, *private* and *protected* visibility control.

- One can construct limited mutual dependencies between modules: Implementation modules A and B can each USE the other's interface module (but recursive dependence between interface modules is prohibited).

- It provides a natural mechanism for private procedures – just don't mention them in the interface module.

Fortran could introduce separation of interface and implementation in a way that is compatible to existing software. Modules that begin with distinguished statements, e.g. INTERFACE MODULE and IMPLEMENTATION MODULE would be just that. An INTERFACE module contains an *interface-body* for every procedure it wishes to publish (but not in an interface block, so it's a module procedure); the full procedure must be declared in the implementation module, with the same characteristics. An implementation module automatically incorporates its corresponding interface module. USE statements refer only to interface modules, or undistinguished modules, never to implementation modules.

This would not be a large change in the standard document, and would be almost completely independent from all other features of the language.

24 Modules Need Initialization Parts

To avoid the need for explicit calls to initialization procedures, or "first-time" flags and associated tests, modules should have initialization parts. This is a small change to the language, and the standard document, and has no effect on other parts of the language.

- Allow executable statements that are not contained in a procedure to be present in a module (or in an implementation module but not an interface module – see section 23 on page 34). These statements are called the *initialization part* of the module.

- The initialization part of every module is executed before the main program.

- If module A USEs module B, then the initialization part of module B is executed before the initialization part of module A. If there is

a mutual dependence between modules A and B (by way of separate implementation and interface modules – see section 23 on page 34), or there is no USE relationship (even by transitivity of USE), then the order of execution is not defined.

25 Fortran Could Have Referential Invariance

The goals of most software engineers are to write programs that are:

- Small in source form.
- Small in executable form.
- Fast.
- As easy as possible, therefore as cheap as possible, to write.
- Clear, therefore as easy and cheap as possible, to understand.
- Not fragile when changes are necessary, therefore as cheap as possible to modify.

Because the total cost of "owning" most programs throughout their lifetimes is most strongly influenced by the last goal, most programming methodologies have focussed on it, at the expense of the others.

In [2] David Parnas showed how one can advance the last goal by putting all references to and manipulations of objects into procedures. This strategy has the advantage of being applicable to any language that has procedures, but the disadvantages:

- One ends up with numerous trivial procedures that have a much higher overhead of reference than of execution – programs consist largely of call and return instruction sequences, and spend most of their time executing them;
- One must write a lot of procedure headers and argument declarations;
- The poor schmo who next reads the program must read numerous procedures, most of which do nothing profound, and keep in mind what each one does (see the next bullet);
- The author's intent is clear in `a = bank_balance(person)` but ambiguous in `call set_bank_balance(person, a)`. Does the latter accomplish `bank_balance(person) = a` or `a = bank_balance(person)`, or something else?)

In [3] Ross, and in [1] Geschke and Mitchell, described how one can design a language so that changes to representation of an object cannot be reflected in changes to usages of the object. Thus, there's no point to encapsulate simple references to and manipulations of objects in trivial procedures. This approach has the disadvantage that one must design a language to have the "Referential Invariance" property, but the advantage that it advances all six goals outlined above:

- Programs are smaller in source form, and therefore easier and cheaper to write and maintain;
- Programs are smaller in executable form;
- Programs execute faster;
- The author's intent is not obscured.
- One needn't trust to the diligence of the programmer to construct programs that are not "fragile" upon changes in object representation **because changes in representation cannot change usages**;

Two fortunate factors have conspired to bring Fortran closer to referential invariance than any other "main stream" language

- Array element selectors and function argument lists are both bracketed by ().
- Pointers are dereferenced automatically.

The barriers to referential invariance in Fortran are

- Although one could replace an array reference A(I) by a function invocation in contexts where the value of A(I) is to be *produced* (called a *right-hand context*, one cannot replace A(I) by a procedure invocation when A(I) is to *absorb* a value, for example on the left side of =. This is called a *left-hand context*. This can be overcome by introducing *Updaters* into Fortran (see section 25.1 on page 38).

  This would be of significant importance when programming for multiprocessor architectures, especially those that use message passing instead of shared memory. Without referential invariance, for *every* reference to a distributed array, one would need explicitly to invoke a message-passing procedure. With referential invariance, one could write `X(I,J) = Y(J,I)`, even if one or both of X and Y are distributed.

The message-passing details would be hidden in the function for Y and/or the updater for X. On shared-memory, or single-processor machines, X and Y could be changed to arrays *without any changes to usages*.

- One can replace neither a right-hand context nor a left-hand context for a scalar by a procedure invocation. This can be overcome by a minor revision of the rules governing the appearance of a function name with no argument list (see section 25.5 on page 44).

- One can replace neither a right-hand context nor a left-hand context for a field of a derived type by a procedure invocation. This can be overcome by at least two mechanisms (see section 25.4 on page 44).

- One cannot use an array section selector, e.g. 3:11:2, as an actual argument. This can be overcome by creating an intrinsic first-class type for array-section selectors (see section 25.3 on page 42).

- One cannot change the representation between a derived type and a scalar or array without changing the references. This probably never happens. If it does, one could write a function and updater to "hide" the "other" representation.

These are all minor changes, orthogonal to the rest of the language, and to each other, and have no impact on existing programs.

Once the concept of updaters is introduced into Fortran, several intrinsic updaters make sense (see section 25.2 on page 42).

25.1 Updaters

I would like to see something like this in Fortran sooner rather than later. I tried in 1986, but was told "it's too late for such major changes." Now, it seems every time I propose it again (for F95, and now for F2000), the answer is still the same. Or is it really too late for F2000?

Suppose you needed to change the representation of a thing called R from an array to a pair of procedures – one to reference R and one to update it. If you used a function named R, the references would still look the same – "R(I)". But what about those places where array elements got updated? They originally were "R(I) = X" or READ or .... You'd need to change them to something like

```
CALL Store_R (R, I, X)
```

and worse for the READ. Ick.

To avoid this problem, the current wisdom is that "R" should be encapsulated in a pair of procedures, e.g.:

```
REAL FUNCTION Get_R(I)
INTEGER I
GET_R = R(I) ! assume R is a module variable
END FUNCTION Get_R
```

and

```
SUBROUTINE Store_R(I, V)
INTEGER I
REAL V
R(I) = V ! assume R is a module variable
END SUBROUTINE Store_R
```

Wouldn't it be nicer not to have these procedures when R is so simple, to be able to write X=R(I) and R(I)=X, and not worry about the possibility that these references might change if it ever becomes necessary to change the representation of R? The ability to change the representation of R, but leave usages intact, has been called "referential invariance."

To move Fortran toward referential invariance, one needs to be able to interpret R(I) as some kind of a procedure invocation, both when it appears where a value gets referenced – that's easy, use a function – and when it's updated – on the left side of an assignment operator, in an I/O list in a READ statement, associated to a formal argument of OUT or INOUT intent, associated to a keyword in an I/O control part that produces a value, e.g. IOSTAT, or used as an internal file. There's also the problem that names that have no punctuation near them might need to be changed to procedures – R is a scalar, or a whole array reference (see section 25.5 on page 44). I don't think you want this for DO inductors, but it may be simpler to allow than prevent.

The languages Mesa and POP-2, and perhaps others, included the notion that a procedure reference can appear where a value gets updated. POP-2 called this kind of procedure an "updater." Think of an updater as complementary to a function: A function uses its arguments to "find" the correct value to return, while an updater uses its arguments to "stash" the function value (a SAVE variable is a good place). An updater/function pair will usually be such that if one invokes the updater with certain values of its arguments, to store a certain value, and then invokes the corresponding

function with the same arguments, the value "saved" by the updater comes back – just like for an array – but that's not *necessary*. A good example is a stack, in which a function reference pops the stack and returns the top element, and an updater reference pushes the stack.

An updater is different from a "left-hand function" as described by Aho et. al., which runs *before* the value is created, and produces a pointer to the place the value is to be stored. An example in which a left-hand function cannot be used, but an updater works, is the case of sparse storage of a vector (see "An Example Where Updater Works and Left-hand Function Doesn't" on page 41))).

I call a function/updater pair an "accessor." This isn't really a radical idea: Array and record field references, and storing into them, are just accessors that the compiler knows how to write. Why not let users write other varieties of accessors?

Adding updaters into Fortran now or soon, however much they are restricted, sets Fortran along a course toward referential invariance. Accessors that can appear without argument lists (see section 25.5 on page 44), or that take section arguments (see section 25.3 on page 42), would move Fortran even closer to referential invariance. One could, however, postpone these two refinements – not having all the details of elemental functions worked out didn't kill array syntax in F90.

I prefer a combined declaration, which would allow function and updater to share variables, especially SAVE variables. It would also emphasize that they must have *exactly* the same calling sequence. Here's a trial balloon, continuing the example above:

```
REAL FUNCTION R(I)
  INTEGER I
  REAL S(MAX_S)
  SAVE S
  ! ...
  R = S(I)
  RETURN
UPDATER U ! this is a new statement
  S(I) = U ! This is a new interpretation: U is the name of
           ! the value being sent _into_ the updater.  E.g.
           ! when one writes "R(3) = B+12.0", the compiler
           ! calls the entry U with I = 3, and a "hidden
           ! argument" equal to B+12.0 that is referenced
           ! internally by writing "U".  The equivalent of a
           ! function result clause is not needed, because U
           ! is never explicitly used to invoke the updater:
           ! R is used for this purpose.  One could use
```

```
  S(I) = R
           ! but this might cause some confusion about whether
           ! to use the value to be stored, or execute the
           ! "function" side if the accessor has no arguments.
           ! In that case, one could put a RESULT(RR) clause
           ! on the FUNCTION header, and use
  S(I) = RR
           ! This is an unimportant detail that the committee
           ! could probably debate endlessly.
  RETURN
END
```

Writing "UPDATER U" instead of only "UPDATER" is just user-provided name-mangling that preserves uniqueness of external names, in exactly the same way INTERFACE does for generics – remember, the compiler must invoke different procedure entry points for value-producing and value-absorbing contexts.

This example is *far too trivial to encapsulate*. The importance of referential invariance is that one could initially write the program with R an array, and use X=R(I) and R(I)=X for all references, without fear that if one *later needs* a less trivial representation, all the usages must change: One can write an accessor *after* the need arises **without changing the usages!**

Ponder the efficiency, clarity, and robustness in the face of change, of programs written using this feature. The first two are well known hallmarks of Fortran. Using "procedural abstraction" instead of just "abstraction" to provide the third *subverts the first two*!

It would be easy to write up a description of the declaration and usage of accessors by extending discussions of functions.

So far, I've described only accessors that are referenced by using the same syntax as arrays. It's also necessary to reference accessors using the same syntax as for derived type members (see section 25.4 on page 44).

25.1.1 An Example Where Updater Works and Left-hand Function Doesn't

Consider the problem of storing a sparse vector.

The idea is that storage is allocated only for values different from a distinguished value, say C (usually 0).

If SA is a sparse array, and SA(i) == C, no storage is allocated for it. Setting SA(i) = C does nothing if SA(i) == C already. If SA(i) /= C, the storage for SA(i) is reclaimed. Finally, referring to SA(i) yields C if no space is allocated. The hard part of this exercise centers around the meaning of

the statement SA(i) = X, since the value of X must be known in order to perform the correct action.

A *left-hand function* as described in numerous textbooks, e.g. by Aho et. al., would require executing the function SA(i) to compute a pointer, and then storing X where the pointer points. The above definition of sparse storage is incompatible with the notion that SA(i) can compute the correct pointer without knowing the value of X.

25.2 Intrinsic Updaters

No matter whether user-defined updaters are added into Fortran, there are some intrinsic updaters that make sense. Consider the following equivalent examples:

```
AIMAG(X) = 2.0        X = CMPLX(REAL(X), 2.0)

REAL(X) = 3.0         X = CMPLX(3.0, AIMAG(X))

ABS(X) = 4.0          X = SIGN(4.0, X)

ABS(X) = 5.0          PHI = ATAN2(AIMAG(X), REAL(X))
                      X = 5.0 * CMPLX(COS(PHI), SIN(PHI))

FRACTION(X) = 0.6     X = SET_EXPONENT(0.6, EXPONENT(X))

EXPONENT(X) = 7       X = SET_EXPONENT(FRACTION(X), 7)
```

Which column do you find clearer?

A complex number C can be considered to be a derived type with "fields" C%REAL and C%AIMAG. Isn't AIMAG(C) clearer than C%AIMAG?. Given this clarity and symmetry, perhaps it's a little more understandable why I and others advocated function-like syntax for derived-type fields.

25.3 Make the Array Section Selector a First–Class Type

An array can take a subscript of the form I:J:K. Thus if one has an array R and uses a section subscript with it, one can't change R into an accessor.

To solve that problem, make the subscript triplet type explicit and first-class. Since the type has more general application than use as a subscript triplet, a more general name may be preferred. Since the subscript triplet denotes a "regularly spaced sequence of integers" (see ISO/IEC 1539:1991,

64:1), and the word SEQUENCE is already used in Fortran, the name SEQUENCE is a possible choice (choose your own type name if you don't like SEQUENCE). I:J:K is a constructor for the type, where I, J and K are arbitrary integer-valued expressions. Intrinsic accessors, say BOUND1, BOUND2 and STRIDE, could examine and change "fields" of variables, and examine fields of constants or parameters.

Variables and expressions of this type could be accessor formal and actual arguments, so you could write

```
X = R(3:11)
```

or

```
R(3:11) = X
```

to refer to

```
FUNCTION R (S)
SEQUENCE S
...
UPDATER U ! for R
...
END FUNCTION R
```

They could also be used for subscripts, or for the control part of explicit or implied DO, e.g.

```
sequence RANGE
RANGE = 3: 11: 2
...
do I = RANGE ! this only works with significant blanks!
  ...
end do
```

and parameters of type SEQUENCE could be used for array bound declarations (only if they have no STRIDE part or unit STRIDE part), or in CASE selectors.

If S is of type SEQUENCE and J is of type INTEGER (or can be coerced thereto) then J = S is prohibited, but S = J means S = J:J:1. This allows an INTEGER to be passed to a dummy argument of type SEQUENCE and INTENT(IN).

One could postpone making SEQUENCE type explicit and first-class, which would compromise but does not demolishing the usefulness of accessors.

25.4 Function References that Look Like Derived Type Field References

At present in Fortran, one can replace neither a right-hand context nor a left-hand context for a field of a derived type by a procedure invocation, except by changing the syntax. There are at least two ways that Fortran might be changed to allow the syntax of derived type field reference and update to be the same as function and updater invocation.

- Allow functions and updaters to be members of derived types, that is, to be declared within the boundaries of TYPE ... END TYPE statements. Thus, if one declares a type T containing a function or updater F, and declares a variable R of type T, R%F is a reference to the function F that is a member of type T, and R is passed into F as a "hidden argument," as in C++. This approach requires either that procedure members of derived types have unique names, or some form of "name mangling." One also requires some distinguished syntax or semantic interpretation to access R. C++ uses `this->`. I would prefer using the type name.

- Define the meaning of R%F, when the type of R has no F field, to be "Invoke a function or updater F, that has an argument of the type of R, with R as its argument, i.e. F(R). Use the *generic* mechanism if necessary." The definition must be extended slightly to include the case that F ought to have additional arguments, e.g. R%F(3): "Invoke the function F with R as its first argument, and remaining parenthesized expressions as subsequent arguments." E.g. F(R,3). This requires neither name mangling, nor a special syntax or semantic interpretation to access R.

  If F is a function that takes an argument of the type of R, and R has no field named F, and G is a function or updater that takes the result type of F, and the result type of F is not a derived type having a field named G, then R%F%G is interpreted to be G(F(R)).

  Similarly, if R is of a derived type that has a field named F, then F(R) is to be interpreted R%F.

25.5 Replacing a Scalar or Whole-array By Procedures

Fortran presently prohibits referencing a function that has no arguments without appending () to its name.

The only context in which a function is allowed to appear without () appended is when it appears as an actual argument.

To move Fortran toward referential invariance, the standard should specify that when a function or accessor (see section 25.1 on page 38) that takes no arguments has an explicit interface, it may be invoked without appending () to its name. This would allow one to change a scalar variable, or to change an array that is referenced as a whole, to an accessor, without changing the references. Similarly, () may be appended to a scalar variable reference. This would allow one to change an accessor or function, that is referenced with () appended, to a scalar variable, without changing the references.

If one wishes to change an array to an accessor, one must define a generic pair of accessors, one with no arguments, to handle the case that the array originally had been referenced as a whole, with no subscripts, and another with as many section arguments (see section 25.3 on page 42) as the original dimensionality of the array.

When a reference to an accessor that takes no arguments appears without appended () as an actual argument to a procedure that does not have explicit interface, the FUNCTION side of the accessor should be run before the procedure call, the result passed to the procedure, and the UPDATER side run after the call.

When a reference to a function (that does not have a corresponding updater) appears without appended () as an actual argument to a procedure that does not have explicit interface, the address of the function should be passed to the procedure.

If one wishes to pass the address of an accessor that takes no arguments, or the result of evaluating a function that has no arguments, the interface of the procedure to which they are passed must be explicit.

26 Subtypes of INTEGER Need Regularization

Fortran has types and subtypes, in much the same spirit as Ada (except for the relation to strong typing). Subtypes of REAL are done well. Subtypes of INTEGER are not. The definition of SELECTED_INT_KIND is the culprit.

SELECTED_INT_KIND should be overloaded to take two arguments, a lower bound and an upper bound. Then SELECTED_INT_KIND(0,65535) *could* select a hardware type that is an unsigned 16 bit integer (but the standard should be silent whether it does).

SELECTED_INT_KIND(100,355) *could* be mapped onto 8 bit unsigned integers by compilers willing to add and subtract 100 whenever necessary

(but the standard should be silent).

References

1. Charles M. Geschke and James G. Mitchell, *On the problem of uniform references to data structures*, **IEEE Transactions on Software Engineering SE-1**, 2 (June 1975) 207–219.

2. David Parnas, *On the criteria for dividing programs into modules*, **Communications of the ACM** (December 1972).

3. D. T. Ross, *Uniform referents: An essential property for a software engineering language*, **Software Engineering 1** (1969) 91–101.

Appendix: Some Concrete Proposals to Change ISO/IEC 1539:1991

This appendix proposes concrete changes to ISO/IEC 1539:1991, with the intent to illustrate one way to implement proposals outlined above. Marginal notes indicate the reason(s) for the proposed change.

At 7:20, 7:22 — Accessor

change *function-subprogram* to *accessor-subprogram*

Add after 7:29 — Accessor

[[*updater-stmt*]
[*execution-part*]]

Add after 9:5 — Comefrom

is *comefrom-stmt*

After 9:13 add — Swap

or *exchange-statement*

After 9:20 add — Swap

or *pointer-exchange-statement*

After 9:49 add — Very-local variables

(4) An IF, DO or BLOCK executable construct, or a *block* introduced by a CASE statement in a *case-construct*, or between a *block-stmt* and the first *case-stmt* in a *case-block*.

At 10:12, add after "result variable." Accessor

A function may have an **updater** part; its invocation receives a value. An updater is complementary to its corresponding function, and has the same name and argument characteristics. The variable that receives the value in an updater is called the **received value variable**, and has the same characteristics and name as the result variable of the corresponding function. A function and updater, taken together, are called an **accessor**.

Add the following column to Figure 2.1 on page 12 Very-local variables

Kind of Scoping Unit	...	Executable Construct
USE Statement		No
ENTRY Statement		No
FORMAT Statement		Yes
Misc. Declarations		Yes
DATA Statement		Yes
Derived-Type Definition		Yes
Interface Block		Yes
Statement Function		???
Executable Statement		Yes
CONTAINS Statement		No

At 22:19-36 Comefrom

Add "COME FROM" to the "Blanks Optional" column.

Add SPECIFICATION MODULE and IMPLEMENTATION MODULE to the "Blank Mandatory" column. Module

Replace 24:32-33 by Include

The interpretation of *char-literal-constant* is user defined.

Replace 26:36 by Sequence

nonnumeric types: Character, Logical and Sequence

After 32:24 add Sequence

4.3.2.3 Sequence type

The **sequence type** is an intrinsic composite type that denotes a regularly spaced sequence of integers. Data entities of the type are

represented by values that consist of three integers, the first bound, the second bound, and the stride.

The type specifier for the sequence type is the keyword SEQUENCE. If the keyword SEQUENCE is specified and the kind type parameter is not specified, the default kind value is the same as that for default integer, the type of the bounds and stride is default integer, and the data entity is of type **default sequence**.

A processor must provide a kind of sequence type corresponding to each kind of integer type provided.

Data objects of sequence type are constructed by using the *sequence-constructor*.

R431 *sequence-constructor*	**is** [*bound*] : [*bound*] ■ ■ [: *stride*]
R432 *bound*	**is** *scalar-int-expr*
R433 *stride*	**is** *scalar-int-expr*

Let B_i and T_i for i = 1:3 be the smallest and largest values of integers of the kind of the first bound, second bound and stride. The kind type parameter value of the sequence constructor is the kind type parameter value for the integer type that includes the minimum of B_i, and the maximum of T_i. If any sub-object has a kind type parameter value different from that of the sequence constructor, the sub-object is converted to the kind type of the sequence constructor.

A data object of sequence type denotes a regular sequence of integers consisting of zero or more values. The third expression is the increment between values, and is called the **stride**.

If the first or second bound is omitted, the values depend on the use of the sequence:

- If a sequence constructor is used as a subscript, or an *array-spec* for a dummy argument array, an omitted first bound is equivalent to the subscript whose value is the lower bound for the array, and an omitted second bound is equivalent to the subscript whose value is the upper bound for the array. The second bound must not be omitted in the last dimension of an assumed-size array.
- Otherwise, an omitted first bound denotes the smallest number represented by the kind of integer of which the sequence is composed, and an omitted second bound denotes the largest number

represented by the kind of integer of which the sequence is composed. *This definition would obviously benefit from precise subtypes of the integer type, e.g. if SELECTED_INT_KIND(100,355) denotes a KIND of integer for which values must be in the closed range [100,355], then in a sequence constructor using that kind, an omitted first bound would denote 100, and an omitted second bound would denote 355.* *Note*

If the stride is omitted it is assumed to be 1. The stride must not have the value zero.

When the stride is positive, the sequence constructor denotes a regularly spaced sequence of integers beginning with the first bound and proceeding in increments of the stride to the largest integer not greater than the second bound; the sequence is empty if the first bound is greater than the second.

When the stride is negative, the sequence constructor denotes a regularly spaced sequence of integers beginning with the first bound and proceeding in increments of the stride to the smallest integer not less than the second bound; the sequence is empty if the first bound is less than the second. For example, the sequence constructor 9:2:-2 denotes the sequence having the four values 9, 7, 5 and 3.

The following is more speculative. If consensus on its value cannot be reached, it can be omitted without serious compromise to the overall proposal. *Note*

A sequence is conformable to an array. Any intrinsic operation defined for arrays may be carried out between an array and a sequence by considering that the sequence denotes an array of rank one and extent equal to the length of the denoted sequence, except that an array may not be assigned to a sequence.

At 32:35-36, 33:7, 247:37 (and elsewhere it presently appears) Sequence

Change "sequence type" to "sequenced type".

At 33:3 and 39:26 Limited

change "*access-spec*" to "*access-spec-list*".

After 33:7 add Limited

Constraint: *access-spec-list* must not contain both PUBLIC and PRIVATE accessibility attributes.

After 33:11 add Limited

Constraint: If a component of a derived type is of a type declared to be limited, the derived type must be limited.

After 34:16 add Limited

If the derived type is limited then no intrinsic operators are defined for the type, intrinsic assignment is not defined for the type, and the components of the type are private exactly as they would be if the type definition included a PRIVATE statement.

After 39:23 add Sequence

or SEQUENCE [*kind-selector*]

After 40:7 add Limited

Constraint: *access-spec-list* must not contain both PUBLIC and PRIVATE accessibility attributes.

After 43:17 add Sequence

5.1.1.7 SEQUENCE

The SEQUENCE type specifier specifies that all entities whose names are declared in this statement are of intrinsic type sequence (4.3.2.3).

The kind selector, if present, specifies the representation method. If the kind selector is absent, the kind type parameter is KIND(1) and the entities declared are of type default sequence.

At 43:18 Sequence

change **5.1.1.7** to **5.1.1.8**

After 44:10 add Limited

or LIMITED

After 44:15 add Limited

If a type is declared with the LIMITED attribute, then no intrinsic operators are defined for the type, and neither intrinsic assignment nor pointer assignment are defined for the type.

If a pointer entity is declared with the LIMITED attribute, then pointer assignment is unavailable for use with that entity.

If any other entity is declared with the LIMITED attribute, then neither intrinsic assignment, nor any intrinsic operators, are available for use with that entity outside the module.

An entity may be limited and private, or limited and public, but it may not be private and public.

Replace 44:23-24 by Limited

or LIMITED

Constraint: The INTENT attribute must not be specified for a dummy argument that is a dummy procedure.

Constraint: Only the INTENT(LIMITED) attribute may be specified for a dummy argument that is a dummy pointer.

After 44:33 add Limited

For a non-pointer dummy argument, the INTENT(LIMITED) attribute specifies that neither intrinsic assignment nor any intrinsic operators are available for use with the dummy argument, and that when the dummy argument is used as an actual argument it can only be argument associated to a dummy argument with INTENT(LIMITED).

For a pointer dummy argument, the INTENT(LIMITED) attribute specifies that pointer assignment is not available for use with the dummy argument, and that when the dummy argument is used as an actual argument that is associated to a pointer dummy argument, the associated dummy argument must also have INTENT(LIMITED).

Replace 45:24-26 by Sequence

R513 *explicit-shape-spec* **is** *sequence-constructor*
or *upper-bound*

R514 *upper-bound* **is** *specification-expr*

Constraint: The sub-objects of the sequence constructor must be specification expressions.

Constraint: The stride of the sequence constructor must have the value 1. *Needed?*

On 57:9, 57:16, 58:32, 59:39 Sequence

Add "default sequence" before "or".

After 61:10 add Accessor

or *updater-reference*

Replace 62:18 by Sequence

R611 *substring-range* **is** *scalar-sequence-expr*

Couldn't we also allow scalar-int-expr? *Note*

Constraint: The *stride* part of *scalar-sequence-expr* must have the value one. *Needed?*

After 64:6 add Sequence

or *scalar-sequence-expr*

At 64:8, 65:32 (and anywhere else it appears) Sequence

Replace *subscript-triplet* by *scalar-sequence-expr*.

Delete 64:10-11 Sequence

Replace 64:14-15 by Sequence

Constraint: The second bound of *sequence-constructor* must not be omitted in the last dimension of an assumed-size array.

After 64:31 add Sequence

It is not necessary for the second bound of a subscript of type sequence to be within the array bounds, so long as all the elements of the denoted sequence are within the array bounds. For example, since the sequence constructor 3:11:7 denotes the sequence (/3, 10/) it may be used to subscript an array having bounds 2:10.

At 65:38 (and anywhere else it appears) Sequence

replace "subscript triples" by "scalar sequence expressions".

At 65:40 (and anywhere else it appears) Sequence

replace "subscript triple" by "scalar sequence expression".

Delete section 6.2.2.3.1. Sequence

Replace 67:17-19 by Sequence

R626 *allocate-shape-spec* **is** *scalar-sequence-expr*

After 67:32 add Sequence

Constraint: The stride sub-object of the scalar sequence expression must be 1. *Needed?*

After 74:40, add Sequence

| S S L |

At 74:51 Sequence

add "S" before "and"

At 74:52 Sequence

add "sequence," after "complex,"

After 76:2 add Sequence

R7.. *sequence-expr* **is** *sequence-constructor*
or *variable*

At 77:33 add Sequence

A **sequence constant expression** is a constant expression whose type is sequence.

After 78:19 add Sequence

R... *sequence-initialization-expr* **is** *sequence-expr*

Constraint: A *sequence-initialization-expr* must be an initialization expression.

After 89:40 add Sequence

| `sequence` | `integer, real, sequence` |

After 90:22 add Sequence

| sequence | INT(*expr*,KIND=KIND(*variable*)):INT(*expr*,KIND=KIND(*variable*)):1 |

After 92:2 add Swap

7.5.1.7 Exchange statement

Two variables may be exchanged by an exchange statement.

R7.. *exchange-statement* **is** *variable-1* :=: *variable-2*

Constraint: *variable-1* and *variable-2* must not be assumed-size arrays.

Constraint: *variable-1* = *variable-2* and *variable-2* = *variable-1* must both be permitted.

Constraint: *variable-1* and *variable-2* must be of the same type.

An exchange statement `variable-1 :=: variable-2` has the same effect as

```
temp = variable-1
variable-1 = variable-2
variable-2 = temp
```

where *temp* has the same type, type parameters and shape as *variable-1* and *variable-2*.

If *variable-1* and *variable-2* are scalars of type integer, real, logical, or character with length 1, then the exchange operation is indivisible. Otherwise, if assignment would be intrinsic, the exchange operation is component-by-component indivisible.

After 92:32 add Swap

7.5.2.1 Pointer exchange

Two pointers may be exchanged by the pointer exchange statement.

R7.. *pointer-exchange-statement* **is** *pointer-object-1* <=> ■
■ *pointer-object-2*

Constraint: Both pointer objects must have the POINTER attribute.

Constraint: Both pointer objects must have the same type, type parameters and rank.

Constraint: Neither pointer object may be an array section with a vector subscript.

A pointer exchange statement

```
pointer-object-1 <=> pointer-object-2
```

has the same effect as

```
temp =>  pointer-object-1
pointer-object-1 => pointer-object-2
pointer-object-2 => temp
```

in which *temp* has the POINTER attribute, and the same type, type parameters and rank as *pointer-object-1* and *pointer-object-2*.

Pointer exchange for a pointer component of a structure may also take place by execution of a derived type exchange statement (7.5.1.7).

Pointer exchange is indivisible.

After 95:10 add Block

(4) Explicit BLOCK Construct

Lines 95:22-23 are incorrect for EXIT, as implemented for Fortran 90.

Replace 95:22-23 by Exit

A statement that refers to the *construct-name* of a construct in which it is contained **belongs** to that construct. Otherwise, an EXIT or CYCLE statement **belongs** to the innermost DO construct in which it appears, and any other statement **belongs** to the innermost construct in which it appears.

Replace 97:34-37 by CASE

R808 *case-construct* **is** *select-case-stmt* BLOCK
[*case-block*] ...
end-select-stmt

After 97:38 add CASE

R8... *case-block* **is** *block-stmt* BLOCK
case-block ...
end-block-stmt
case-group-exec-constructs
or *case-stmt*
block

R8... *case-group-exec-constructs* **is** [*executable-construct*] ...

After 98:3 add Real CASE

or *scalar-real-expr*

Replace 98:6 by CASE .and.

R813 *case-selector* **is** (*case-value-range-list*) ■
■ [.AND. *scalar-logical-expr*]

After 98:8 add CASE .and.

If a *scalar-logical-expr* is present after a *case-selector*, and it is a *scalar-logical-initialization-expr* that has the value false, the *case-stmt* and following block, if any, are considered to be absent.

After 98:12 add CASE

or *case-value case-relational* * range
or * *case-relational case-value*
or *case-value case-relational* * ■
■ *case-relational case-value*

Replace 98:13-19 by Sequence

	or *scalar-sequence-initialization-expr*	
	or *scalar-logical-initialization-expr*	
R815 *case-value*	**is** *scalar-char-initialization-expr*	
	or *scalar-real-initialization-expr*	Real CASE
R8.. *case-relational*	**is** .LT.	CASE range
	or <	
	or .LE.	
	or <=	

Constraint: For a given *case-construct*, if *case-expr* is of integer type, then any *case-value* may be a *scalar-sequence-initialization-expr*; otherwise, each *case-value* must be of the same type as *case-expr*. For character type, length differences are allowed, but the kind type parameters must be the same.

Replace 98:19 by CASE range

Constraint: If *case-expr* is of type logical, the only form of *case-value-range* allowed is a *case-value*.

Replace 98:23-38 by CASE range

Execution of the SELECT CASE statement causes the case expression to be evaluated. The resulting value is called the **case discriminant**, denoted by *c* below. If the *case-selector* includes the optional *scalar-logical-expr*, the *scalar-logical-expr* is called the **case condition**. For a case value range list, a match occurs if the case condition has the value true, or is absent, and *c* matches any of the case value ranges in the list, as follows: CASE .and.

1. If the case value range contains a single value *v*, a match occurs for type logical if the expression *c* .EQV. *v* is true, else a match occurs if the expression *c* .EQ. *v* is true. CASE range
2. If the case value range is of the form *low case-relational* * *case-relational high*, a match occurs if the expression *low case-relational c* .AND. *c case-relational high* is true.
3. If the case value range is of the form *low case-relational* *, a match occurs if the expression *low case-relational c* is true.

4. If the case value range is of the form * *case-relational high*, a match occurs if the expression *c case-relational high* is true.
5. If no other selector matches and a DEFAULT selector is present, it matches the case discriminant.

If no other selector matches and the DEFAULT selector is absent, there is no match.

The forms *low* :, : *high* and *low* : *high* are equivalent to *low* <= *, * <= *high* and *low* <= * <= *high*, respectively.

If a CASE statement matches the case discriminant, execution of the construct is completed in two steps: CASE BLOCK

1. The block following the CASE statement containing the matching selector, if any, is executed. unchanged
2. If the CASE statement is contained within any *case-block*s within the same *case-construct*, *case-group-executable-construct*s, if any, are executed in order from innermost to outermost. CASE BLOCK

At 99:2 Real CASE

replace "An integer" by "A".

After 99:3 add Real CASE

REAL N

Replace 99:5 by CASE

CASE (* <= -1) range

Replace 99:9 by CASE

CASE (1 <= *) range

After 100:35 add Sequence

or [,] *do-variable* = ■
■ *scalar-sequence-expr*

Constraint: In fixed source form, a *scalar-sequence-expr* that consists only of a variable of SEQUENCE type may be used only if the optional comma is present.

After 100:39 add Accessor

This is not necessary until updaters are included in Fortran, and updaters that have no arguments can be referenced by their name alone, without parentheses. *Note*

Constraint: The *do-variable* must not be an updater reference.

After 102:36 add Sequence

If *loop-control* is *scalar-sequence-expr* then m_1, m_2 and m_3, are taken from the first bound, second bound and stride of the sequence type value.

After 106:48 add Block

8.1.5 Explicit BLOCK Construct

The Explicit BLOCK Construct provides the boundaries for a block.

R8... *block-construct* **is** *block-stmt*
block
end-block-stmt

R8... *block-stmt* **is** [*block-construct-name*:] ■
■ BLOCK

R8... *end-block-stmt* **is** END BLOCK ■
■ [*block-construct-name*]

Constraint: If the *block-stmt* of a *block-construct* is identified by a *block-construct-name*, the corresponding *end-block-stmt* must specify the same *block-construct-name*. If the *block-stmt* of the *block-construct* is not identified by a *block-construct-name*, the corresponding *end-block-stmt* must not specify a *block-construct-name*.

8.1.6 EXIT Statement EXIT

R835 *exit-stmt* **is** EXIT [*construct-name*]

Constraint: If an *exit-stmt* refers to a *construct-name* it must be within the range of that construct; otherwise, it must be within the range of at least one *do-construct*.

An EXIT statement belongs to a particular construct. If the EXIT statement refers to a *construct-name* it belongs to that construct; otherwise it belongs to the innermost DO construct in which it appears.

Execution of an EXIT statement terminates execution of the construct to which it belongs.

After 108:11 add Comefrom

8.2.5 COME FROM statement

R8.. *comefrom-stmt* **is** COME FROM (*label-list*)

The COME FROM statement serves as a target for GO TO statements. If, and only if, a program unit contains a COME FROM statement, then all GO TO, ASSIGN and arithmetic IF statements must be labeled; each GO TO or arithmetic IF statement can only transfer

control to a COME FROM statement containing the label of the GO TO or arithmetic IF statement in its *label-list*, and each ASSIGN statement can only mention the label of a COME FROM statement containing the label of the ASSIGN statement in its *label-list*.

Move lines 114:18-19 to be after 114:3 WRITE ⇒ PRINT

Replace 114:4 by WRITE ⇒ PRINT

A scalar integer expression that has the value -1 identifies the same external units as an asterisk.

For purposes of output, a scalar integer expression that has the value -2 identifies an external unit to which error messages may be written, if such a unit exists; otherwise it has the same effect as unit number -1. For purposes of input, it identifies a unit that is always positioned at the end-of-file.

A scalar integer expression that has a value less than or equal to -3 identifies an external unit that neither stores nor displays data written to it, and for purposes of input is always positioned at the end-of-file.

The above specification allows one to "turn off" output to units having magnitude greater than 2 simply by negating the unit number. *Note*

Unit numbers -1 and -2 are preconnected for formatted sequential access.

Otherwise, a scalar integer expression that identifies an external file unit must be zero or positive.

At 148:31 and 152:30 Sequence

Change "character constants and complex constants" to "character, complex and sequence constants".

After 148:42 153:10 add Sequence

When the next effective item is of type sequence, the input form consists of two or three numeric input fields separated by colons. The first numeric input field is the first bound of the sequence, the second is the second bound of the sequence, and the third, if present, is the stride of the sequence. If the third is absent, the stride is 1. Blanks or the end of record may appear either before or after the separating colon(s).

After 150:42 add Sequence

Sequence constants consist of three numeric output fields, separated by colons.

Replace 156:42 by Module

The sequence of *execution-part* statements specifies the actions of the main program during program execution. Execution of an executable program (R201) begins with execution of the initialization parts of modules (11.3), if any, and continues with execution of the first executable construct of the main program.

Replace 157:13-16 by Module

R1104 *module* **is** *undistinguished-module*
or *specification-module*
or *implementation-module*

R11.. *undistinguished-module* **is** *module-stmt*
[*specification-part*]
[*module-initialization-part*]
[*module-subprogram-part*]
end-module-stmt

R11.. *specification-module* **is** SPECIFICATION ■
■ *module-stmt*
[*specification-part*]
[*contains-stmt*
[*interface-body*] ...]
end-module-stmt

R11.. *implementation-module* **is** IMPLEMENTATION ■
■ *module-stmt*
[*specification-part*]
[*module-initialization-part*]
[*module-subprogram-part*]
end-module-stmt

R11.. *module-initialization-part* **is** [*executable-construct*] ...

If an *interface-body* appears in a specification module, the procedure is a *module-subprogram*. The corresponding implementation module must declare the full subprogram, with identical characteristics.

At 157:24, 157:34, 158:5, 158:18, 158:21, 158:22 Module

replace all occurrences of "module" by "undistinguished module or specification module"

At the end of 157:26 add Module

An implementation module must have the same name as the corresponding specification module.

Replace 157:30-32 by Module

A specification module is host to the corresponding implementation module. Entities in the specification module are therefore accessible in the implementation module through host association. A specification module is host to any interface bodies it contains. Entities in the specification module are therefore accessible in the interface bodies through host association. An undistinguished module or implementation module is host to any module procedures (12.1.2.2) it contains. Entities in the module are therefore accessible in module procedures through host association.

11.3.1 Module initialization part

The initialization parts of all modules in a program are executed before the main program. If an undistinguished module or implementation module contains a USE statement (11.3.1, 11.3.2), the initialization part of the module accessed by the USE statement is executed before the initialization part of the module containing the USE statement, except that if two or more implementation modules mutually depend on each other's specification modules, the order of execution of their initialization parts is undefined. If there is no USE relation between two modules, the relative order of execution of their initialization parts is undefined.

11.3.2 Implementation module internal procedures

A procedure that is declared in an implementation module but not in an interface body in the corresponding specification module is an internal procedure of the implementation module. The implementation module is the host of its internal procedures.

11.3.3 Module reference

Replace 158:3 by Module

11.3.4 The USE statement and use association

After 158:41 add Module

The only entities of an implementation module that are made accessible by a USE statement are the module procedures that are declared in the corresponding specification module.

At 159:9-10, 16, 32, 160:3, 26, 161:6, 14, 17 Module

Replace "11.3.3" by "11.3.5"

Replace 163:11-13 by Accessor

The definition of a procedure specifies it to be a function, an updater or a subroutine. A reference to a function either appears explicitly as a primary within an expression, or is implied by a defined operation within an expression. A reference to an updater appears explicitly in a value-receiving context. A reference to a subroutine is a CALL statement or a defined assignment operator (7.5.1.3).

At 163:34 Very-local variables

Replace "or a derived-type definition" by "an executable construct, or a derived-type definition".

A debate is needed to determine the lifetime of a variable declared in an executable construct, with and without the SAVE attribute. My preference is that a variable declared without the SAVE attribute has a lifetime from the instant control enters the block, until the instant control leaves the block, while a variable declared with the SAVE attribute has the same lifetime as the program. *Note*

Replace 165:36-37 by Accessor

The **characteristics of a procedure** are the classification of the procedure as a function, updater or subroutine, the characteristics of its arguments, the characteristics of its result value if it is a function, and the characteristics of its received value if it is an updater.

Add after 166:16 Accessor

12.2.3 Characteristics of updater received value

The characteristics of the value received by an updater are identical to the characteristics of the result value of the corresponding function.

Add after 167:5 Sequence

(f) A dummy argument of type sequence.

(3) The procedure is an accessor. Accessor

Add after 167:20 Accessor

updater-stmt

Add after 171:29 Accessor

R.... *updater-reference* **is** *function-reference*

R.... *accessor-reference* **is** *function-reference*

At 172:11, add Accessor

", an updater reference" after "subroutine reference"

Add after 174:20 Accessor

12.4.2 Evaluation of actual arguments

When a function, updater or subroutine is invoked, actual argument expressions are evaluated, then the arguments are associated, and then the procedure is executed.

If the actual argument expression is an accessor reference, and the procedure has implicit interface, or the procedure has explicit interface and the corresponding dummy argument has INTENT(INOUT), the function part of the accessor is invoked before the procedure is invoked, and the updater part is invoked after the procedure returns.

If the actual argument expression is an accessor reference, the procedure interface is explicit, and the actual argument corresponds to a dummy argument having INTENT(IN), the function part of the accessor is invoked before the procedure is invoked, but the updater part of the accessor is not invoked after the procedure returns.

If the actual argument expression is an accessor reference, the procedure interface is explicit, and the actual argument corresponds to a dummy argument having INTENT(OUT), the function part of the accessor is not invoked before the procedure is invoked, but the updater part of the accessor is invoked after the procedure returns.

At 174:21 change Accessor

12.4.2 to **12.4.3**

At 174:23-24 Accessor

Delete "When ... invoked."

At 174:27 change Accessor

12.4.3 to **12.4.4**

Add after 174:35 Accessor

12.4.5 Updater reference

An updater is invoked by an *updater-reference* in the contexts implied by items (1), (2), (3), (8), (9), (10) and (15) in Section 14.7.4, after the value(s) to be stored is (are) produced. In a masked array assignment, the entire array is received by the updater, after the selected elements have been changed.

12.4.6 Elemental intrinsic updater reference

A reference to an elemental intrinsic updater is an **elemental reference** if one or more actual arguments are arrays and all array arguments have the same shape. The received value must have the same shape as the array arguments and each element of the received value is received by invoking the updater using the scalar arguments and corresponding elements of the array arguments. For example, if X and Y are complex arrays of shape (m,n),

```
REAL(X) = (Y)
```

has the same effect as

```
DO i = 1, m
  DO j = 1, n
    X(i,j) = CMPLX(REAL(Y(i,j)), AIMAG(X(i,j)))
  END DO
END DO
```

At 174:36 change Accessor
12.4.4 to **12.4.7**

At 174:38-39 Accessor
Delete "When ... invoked."

At 175:1 change Accessor
12.4.5 to **12.4.8**

At 175:27 change Accessor
Function to **Accessor**

Replace 175:28 by Accessor
An **accessor subprogram** is a subprogram that has a FUNCTION statement as its first statement and contains an UPDATER statement. A **function subprogram** is a subprogram that has a FUNCTION statement as its first statement and does not contain an UPDATER statement.

At 175:29 change Accessor
function-subprogram to *accessor-subprogram*

Add after 175:31 Accessor

[[*updater-stmt*]
[*execution-part*]]

Add after 175:35 Accessor

R12.. *updater-stmt* **is** UPDATER

Implementors can use the same name for function and updater by placing a jump to the updater's entry a fixed distance before the function's entry. Alternatively, the standard can require the programmer to invent a unique name for the updater, viz.: *Note*

R12.. *updater-stmt* **is** UPDATER *updater-name*

and changing 241:18-20
Program units, common blocks, and external procedures are global entities of an executable program. A function and its corresponding updater are separate external procedures....

At 175:42 change Accessor

"module function" to "module accessor"

At 176:1-2, 30 change Accessor

"function" to "accessor".

After 176:4 add Accessor

The result value of the function is produced by assigning a value to the result variable. The received value of an updater is referenced by using the received value variable. The result variable and the received value variable have the same name and characteristics. The function result variable and updater received value variable are collectively called the **accessor variable**.

At 176:5 change Accessor

"result of the function" to "result of the function or value received by the updater".

At 176:5, 7, 9 change Accessor

"function subprogram" to "accessor subprogram".

At 176:6, 10, 23 change Accessor

"result variable" to "accessor variable".

At 176:9, 11, 20 change Accessor

"function result" to "accessor value".

At 176:10 change Accessor

"function body" to "accessor body".

After 176:12 add Accessor

To illustrate the use of an accessor, consider starting development of a program using:

```
INTEGER, PARAMETER                  :: R_SIZE = 100
REAL, SAVE, DIMENSION(1:R_SIZE)     :: R
```

Then, if requirements change so that one wants, say, subscript checking, or the ability to increase the size of R gracefully as the data grow, replace the declaration or R by the following, without changing any of the usages.

```
REAL FUNCTION R(I)
  INTEGER I
  INTEGER, PARAMETER                 :: R_SIZE = 100
  REAL, SAVE, DIMENSION(1:R_SIZE)    :: R_VALUE
  LOGICAL, SAVE, DIMENSION(1:R_SIZE) :: HAS_VALUE = .FALSE.
  CALL TEST_I
  IF (.NOT. HAS_VALUE(I)) THEN
    WRITE(*,'(1X,A,I0,A)') &
      & 'R(',I,') HAS NOT BEEN GIVEN A VALUE'
    STOP
  END IF
  R = R_VALUE(I)
  RETURN
UPDATER
  CALL TEST_I          ! One could instead make R_VALUE
                       ! and HAS_VALUE allocatable, and
                       ! make new ones twice as big
                       ! whenever I > R_SIZE.
  HAS_VALUE(I) = .TRUE.
  R_VALUE(I) = R
  RETURN
CONTAINS
  SUBROUTINE TEST_I
    IF (I .LT. 1 .OR. I .GT. R_SIZE) THEN
      WRITE(*, '(1X, A, I0)') &
        &  'SUBSCRIPT FOR R OUT-OF-RANGE: ', I
      STOP
    END IF
```

```
  END SUBROUTINE TEST_I
END FUNCTION R
```

Examples of uses of R, either as a variable or an accessor, include

```
DO I = 1, 100
  R(I) = I*I      ! The UPDATER for the R accessor
END DO
...
WRITE (*,*) R(12) ! The FUNCTION for the R accessor
```

One cannot completely carry through the conversion of the representation of R from an array to an accessor, if R is accessed using a whole-array reference, or with a array section selector. These usages require making the type of array section selectors explicit and first class (so variables of the type are allowed), and allowing one to write a pair of accessors, say R_SELECT(I) [I is of type SEQUENCE] and R_ALL, and coupling them into a generic pair by using an interface block named R. One must then be allowed to access R by writing R alone, not R(). *Note*

After 176:17 add Accessor

"The keyword RECURSIVE must be present if the accessor directly or indirectly invokes itself."

At 176:18 change Accessor

"function" to "function and its associated updater".

At 176:19 change Accessor

"result variable of the function" to "accessor variable".

At 176:20, 22 change Accessor

"function name" to "accessor name".

At 176:21 Accessor

Delete "function"

After 176:29 add Accessor

At the initiation of execution of the updater, the value of the accessor variable is the value received by the updater. If the accessor variable has been declared to be a pointer, the shape and association status

of the accessor variable are determined by the shape and association status of the value received by the updater.

It is not permitted to transfer control between the *execution-part* of the function and the *execution-part* of the corresponding updater. *Needed?*

At 177:28 change Accessor
"a function" to "a function, updater"

At the end of 177:29 add Accessor
"A function and its corresponding updater have separate instances."

After177:37 add Accessor
12.5.2.5 UPDATER statement
When the updater part of an accessor is invoked, execution of the updater begins at the first *executable-stmt* following the **UPDATER statement**.

At 177:38 change Accessor
12.5.2.5 to **12.5.2.6**

After 178:4 add Accessor
The following constraint is proposed in consideration of the apparent sentiment eventually to eliminate ENTRY entirely. *Note*

Constraint: An accessor subprogram that contains an UPDATER statement must not contain an ENTRY statement.

At 179:12 change Accessor
12.5.2.6 to **12.5.2.7**

At 179:24 change Accessor
12.5.2.7 to **12.5.2.8**

At 179:28 change Accessor
12.5.2.8 to **12.5.2.9**

At 180:1 change Accessor
12.5.2.9 to **12.5.2.10**

After 189:11 add ATAN2
ATAN(Y, X) Arctangent

Set line 189:12 in small type.

After 191:15 add MaxAbsVal

MAXABSVAL (ARRAY, DIM, MASK)	Maximum absolute value in an array

Optional DIM, MASK

After 191:17 add MinAbsVal

MINABSVAL (ARRAY, DIM, MASK)	Minimum absolute value in an array

Optional DIM, MASK

After 192:8 add MaxAbsLoc

MAXABSLOC (ARRAY, MASK)	Location of a maximum absolute value in an array

Optional MASK

After 192:10 add MinAbsLoc

MINABSLOC (ARRAY, MASK)	Location of a minimum absolute value in an array

Optional MASK

After 192:12 add Sequence

BOUND1(SEQ)	Inquire or update the first bound of a sequence
BOUND2(SEQ)	Inquire or update the second bound of a sequence
STRIDE(SEQ)	Inquire or update the stride of a sequence

After 192:15 add Intrinsic Updater

13.11 Intrinsic updaters

The following intrinsic functions have updater parts, and are therefore **intrinsic accessors**.

ABS(A) Change the absolute value of A to the absolute value of the received value. A may be of any INTEGER, REAL or COMPLEX type.

AIMAG(Z) Change the imaginary part of Z to the received value. Z must be of a COMPLEX type.

EXPONENT(X) Change the exponent part of X, when represented as a model number, to the received value. X must be of a REAL type.

FRACTION(X) Change the fractional part of X, when represented as a model number, to the received value. X must be of a REAL type.

REAL(Z) Change the real part of Z to the received value. Z must be of a COMPLEX type.

After 192:20 add IO_Message

IO_MESSAGE (IOS, UNIT)		Print a message appropriate to the error indicated by IOS, where IOS was set by IOSTAT=IOS during execution of an input/output statement on unit UNIT.

After 193:9 add ATAN2

ATAN(Y,X)	ATAN(Y,X)	default real

Set line 193:10 in small type ATAN2

After 193:25 add ATAN2

DATAN(Y,X)	ATAN(Y,X)	double precision real

Set line 193:26 in small type ATAN2

Copy 199:19-35, changing all instances of ATAN2 to ATAN. ATAN2

Set 199:19-35 in small type. ATAN2

Increase following section numbers in Chapter 13. ATAN2

At 194:38 change Intrinsic Updater

"function" to "accessor".

After 194:43 add Intrinsic Updater

When A is of an INTEGER or REAL type, and ABS is used as an updater, `ABS(A) = B` has the same effect as `A = SIGN(B,A)`

When A is of a COMPLEX type, and ABS is used as an updater, `ABS(A) = B` has the same effect as

```
temp = ATAN2(AIMAG(A), REAL(A))
A = ABS(REAL(B)) * CMPLX(COS(temp), SIN(temp))
```

At 196:4 change Intrinsic
"function" to "accessor". Updater

After 196:8 add Intrinsic
When AIMAG(Z) is used as an updater, `AIMAG(Z) = X` has the same effect as `Z = CMPLX(REAL(Z), X)` Updater

After 200:8 add Sequence

13.13... BOUND1(SEQ)

Description. Return or update the first bound sub-object of a sequence.

Class. Accessor.

Arguments.

SEQ Must be of sequence type.

Accessor Type, Type Parameter, and Shape. The accessor value is a scalar of type integer, with the same kind type parameter as SEQ.

Accessor Value. When BOUND1 is invoked in a value-producing context, the result has a value equal to the first bound of SEQ. When BOUND1 is invoked in a value-receiving context, SEQ has INTENT (OUT) and the first bound of SEQ is updated to be equal to the value received.

Examples. The function reference BOUND1(3:12:2) produces the value 3. The updater reference BOUND1(SEQ)=3 changes the first bound of SEQ to 3.

BOUND1 could be changed to an inquiry function if accessors are not included in the present revision of Fortran. *Note*

13.13... BOUND2(SEQ)

Description. Return or update the second bound sub-object of a sequence.

Class. Accessor.

Arguments.

SEQ Must be of sequence type.

Accessor Type, Type Parameter, and Shape. The accessor value is a scalar of type integer, with the same kind type parameter as SEQ.

Accessor Value. When BOUND2 is invoked in a value-producing context, the result has a value equal to the second bound of SEQ.

When BOUND2 is invoked in a value-receiving context, SEQ has INTENT(OUT) and the second bound of SEQ is updated to be equal to the value received.

Examples. The function reference `BOUND2(3:12:2)` produces the value 12. The updater reference `BOUND2(SEQ)=12` changes the second bound of SEQ to 12.

BOUND2 could be changed to an inquiry function if accessors are not included in the present revision of Fortran. — *Note*

At 208:15 change — Intrinsic Updater

"function" to "accessor".

After 208:23 add — Intrinsic Updater

When EXPONENT(X) is used as an updater, `EXPONENT(X) = I` has the same effect as `X = SET_EXPONENT(FRACTION(X), I)`

At 208:34 change — Intrinsic Updater

"function" to "accessor".

After 209:3 add — Intrinsic Updater

When FRACTION(X) is used as an updater, `FRACTION(X) = Y` has the same effect as `X = SET_EXPONENT(Y, EXPONENT(X))`

After 213:10 add — IO_Message

13.13.xx IO_MESSAGE (IOS, UNIT) — *Is* `UNIT` *needed?*

Description. Print a message appropriate to the error indicated by IOS, where IOS was set by IOSTAT=IOS during execution of an input/output statement on unit UNIT.

Class. Subroutine.

Arguments.

IOS	must be a scalar of type integer. It is an INTENT(IN) argument having a value produced by IOSTAT=IOS during execution of an input/output statement on unit UNIT.
UNIT	must be a scalar of type integer. It is an INTENT(IN) argument giving the unit number on which an input/output statement was executed, that resulted in the value of IOS.

Example.

Execution of the statement

```
READ(10,IOSTAT=IOS)...
```

when unit 10 is positioned at the end of file, followed by execution of the statement
`CALL IO_MESSAGE (IOS, 10)`
might produce the message `End-of-file occurred on unit 10.`

After 218:33 add MaxAbsLoc

13.13.xx MAXABSLOC(ARRAY, MASK)

Optional Argument. MASK

Description. Determine the location of the first element of ARRAY having the maximum absolute value of the elements identified by MASK.

Class. Transformational Function.

Arguments.

ARRAY	must be of type integer or real. It must not be a scalar.
MASK (optional)	must be of type logical and must be conformable with ARRAY. If MASK is absent the effect is as though it were present, conformable with ARRAY, and every element a true value.

Result Type, Type Parameter, and Shape. The result is of type default integer; it is an array of rank one and of size equal to the rank of ARRAY.

Result Value.

The result is a rank-one array whose element values are the values of the subscripts of an element of ARRAY, corresponding to a true element of MASK, whose value equals the maximum absolute value of all such elements of ARRAY. The i^{th} subscript returned lies in the range 1 to e_i, where e_i is the extent of the i^{th} dimension of ARRAY. If more than one such element has the maximum absolute value, the element whose subscripts are returned is the first such element, taken in array element order. If there are no such elements (that is, if ARRAY has size zero or every element of MASK has the value false), the value of the result is processor dependent.

An element of the result is undefined if the processor cannot represent the value as a default integer.

Examples.

Case (i):	The value of MAXABSLOC((/2, -6, 4, 6/)) is (/ 2 /).

Case (ii): If A has the value $\begin{vmatrix} 0 & -5 & 8 & -3 \\ 3 & 4 & -1 & 2 \\ 1 & 5 & -6 & -4 \end{vmatrix}$, MAXABSLOC(A, MASK = A .LT. 6) has the value (/ 3, 3 /). Note that this is true even if A has a declared lower bound other than 1.

13.13.xx MAXABSVAL(ARRAY, DIM, MASK) MaxAbsVal

Optional Argument. DIM, MASK

Description. Determine the maximum absolute value of the elements of ARRAY along dimension DIM corresponding to true values of MASK.

Class. Transformational Function.

Arguments.

ARRAY	must be of type integer or real. It must not be a scalar.
DIM (optional)	must be of type integer with a value in the range 1 .LE. DIM .LE.n, where n is the rank of ARRAY. The corresponding actual argument must not be an optional dummy argument.
MASK (optional)	must be of type logical and must be conformable with ARRAY. If MASK is absent the effect is as though it were present, conformable with ARRAY, and every element a true value.

Result Type, Type Parameter, and Shape. The result is of the same type and kind type parameter as ARRAY. It is scalar if DIM is absent or ARRAY has rank one; otherwise, the result is an array of rank $n - 1$ and of shape (d_1, d_2, ..., d_{DIM-1}, d_{DIM+1}, ..., d_n) where (d_1, d_2, ..., d_n) is the shape of ARRAY.

Result Value.

Case (i) The result of MAXABSVAL(ARRAY) has a value equal to the maximum absolute value of all the elements of ARRAY or has the value of the negative number of the largest magnitude supported by the processor for numbers of the type and kind type parameter of ARRAY if ARRAY has size zero.

Case (ii) The result of MAXABSVAL (ARRAY, MASK=MASK) has a value equal to the maximum absolute value of all the elements of ARRAY corresponding to true elements of MASK or has the value of the negative number of the largest magnitude supported by the processor for numbers of the type and kind type parameter of ARRAY if there are no true elements.

Case (iii) If ARRAY has rank one, MAXABSVAL (ARRAY, DIM [, MASK]) has a value equal to that of MAXABSVAL(ARRAY [, MASK=MASK]). Otherwise, the value of element (s_1, s_2, ..., s_{DIM-1}, s_{DIM+1}, ..., s_n) of MAXABSVAL (ARRAY, DIM [, MASK]) is equal to MAXABSVAL(ARRAY(s_1, s_2, ..., s_{DIM-1}, :, s_{DIM+1}, ..., s_n) [, MASK = MASK(s_1, s_2, ..., s_{DIM-1}, :, s_{DIM+1}, ..., s_n)]).

Examples.

Case (i): The value of MAXABSVAL((/1, -2, 3/)) is 3.

Case (ii): MAXABSVAL(C, MASK = C .LT. 0.0) finds the maximum absolute value of all the negative elements of C.

Case (iii): If B is the array $\begin{vmatrix} 1 & 3 & -5 \\ 2 & -4 & 6 \end{vmatrix}$, MAXABSVAL(B, DIM=1) is (/ 2, 4, 6 /) and MAXABSVAL (B, DIM=2) is (/ 5, 6 /).

After 221:13 add MinAbsLoc

13.13.xx MINABSLOC(ARRAY, MASK)

Optional Argument. MASK

Description. Determine the location of the first element of ARRAY having the minimum absolute value of the elements identified by MASK.

Class. Transformational Function.

Arguments.

ARRAY must be of type integer or real. It must not be a scalar.

MASK (optional) must be of type logical and must be conformable with ARRAY. If MASK is absent the effect is as though it were present, conformable with ARRAY, and every element a true value.

Result Type, Type Parameter, and Shape. The result is of type default integer; it is an array of rank one and of size equal to the rank of ARRAY.

Result Value.

The result is a rank-one array whose element values are the values of the subscripts of an element of ARRAY, corresponding to a true element of MASK, whose value equals the minimum absolute value of all such elements of ARRAY. The i_{th} subscript returned lies in the range 1 to e_i, where e_i is the extent of the i^{th} dimension of ARRAY. If more than one such element has the minimum absolute value, the element whose subscripts are returned is the first such element, taken in array element order. If there are no such elements (that is, if ARRAY has size zero or every element of MASK has the value false), the value of the result is processor dependent.

An element of the result is undefined if the processor cannot represent the value as a default integer.

Examples.

Case (i): The value of MINABSLOC((/2, -6, 4, 6/)) is (/ 1 /).

Case (ii): If A has the value $\begin{vmatrix} 0 & -5 & 8 & -3 \\ 3 & 4 & -1 & 2 \\ 1 & 5 & -6 & -4 \end{vmatrix}$, MINABSLOC(A, MASK = A .LT. 6) has the value (/ 1, 1 /). Note that this is true even if A has a declared lower bound other than 1.

13.13.xx MINABSVAL(ARRAY, DIM, MASK) MinAbsVal

Optional Argument. DIM, MASK

Description. Determine the minimum absolute value of the elements of ARRAY along dimension DIM corresponding to true values of MASK.

Class. Transformational Function.

Arguments.

ARRAY must be of type integer or real. It must not be a scalar.

DIM (optional) must be of type integer with a value in the range 1 .LE. DIM .LE. n, where n is the rank of ARRAY. The corresponding actual argument must not be an optional dummy argument.

MASK (optional) must be of type logical and must be conformable with ARRAY. If MASK is absent the effect is as though it were present, conformable with ARRAY, and every element a true value.

Result Type, Type Parameter, and Shape. The result is of the same type and kind type parameter as ARRAY. It is scalar if DIM is absent or ARRAY has rank one; otherwise, the result is an array of rank $n - 1$ and of shape (d_1, d_2, ..., d_{DIM-1}, d_{DIM+1}, ..., d_n) where (d_1, d_2, ..., d_n) is the shape of ARRAY.

Result Value.

Case (i): The result of MINABSVAL(ARRAY) has a value equal to the minimum absolute value of all the elements of ARRAY or has the value of the positive number of the largest magnitude supported by the processor for numbers of the type and kind type parameter of ARRAY if ARRAY has size zero.

Case (ii): The result of MINABSVAL (ARRAY, MASK = MASK) has a value equal to the minimum absolute value of all the elements of ARRAY corresponding to true elements of MASK or has the value of the positive number of the largest magnitude supported by the processor for numbers of the type and kind type parameter of ARRAY if there are no true elements.

Case (iii): If ARRAY has rank one, MINABSVAL (ARRAY, DIM [, MASK]) has a value equal to that of MINABSVAL (ARRAY [, MASK=MASK]). Otherwise, the value of element (s_1, s_2, ..., s_{DIM-1}, s_{DIM+1}, ..., s_n) of MINABSVAL (ARRAY, DIM [, MASK]) is equal to MINABSVAL(ARRAY(s_1, s_2, ..., s_{DIM-1}, :, s_{DIM+1}, ..., s_n) [, MASK = MASK(s_1, s_2, ..., s_{DIM-1}, :, s_{DIM+1}, ..., s_n)]).

Examples.

Case (i): The value of MINABSVAL((/1, -2, 3/)) is 1.

Case (ii): MINABSVAL(C, MASK = C .LT. 0.0) finds the minimum absolute value of all the negative elements of C.

Case (iii): If B is the array $\begin{vmatrix} 1 & 3 & -5 \\ 2 & -4 & 6 \end{vmatrix}$, MINABSVAL(B,DIM=1) is (/ 1, 3, 5 /) and MINABSVAL(B, DIM=2) is (/ 1, 2 /).

At 228:38 change Intrinsic
"function" to "accessor". Updater

After 229:16 add Intrinsic
When Z is of a COMPLEX type and REAL(Z) is used as an updater, Updater
`REAL(Z) = X` has the same effect as `Z = CMPLX(X, AIMAG(Z))`

After 235:4 add Sequence

13.13... STRIDE(SEQ)

Description. Return or update the stride sub-object of a sequence.

Class. Accessor.

Arguments.

SEQ Must be of sequence type.

Accessor Type, Type Parameter, and Shape. The accessor value is a scalar of type integer, with the same kind type parameter as SEQ.

Accessor Value. When STRIDE is invoked in a value-producing context, the result has a value equal to the stride of SEQ. When STRIDE is invoked in a value-receiving context, SEQ has INTENT(OUT) and the stride of SEQ is updated to be equal to the value received.

Examples. The function reference `STRIDE(3:12:2)` produces the value 2. The updater reference `STRIDE(SEQ)=2` changes the stride of SEQ to 2.

STRIDE could be changed to an inquiry function if accessors are not included in the present revision of Fortran. *Note*

After 247:27 add Sequence

(3) A nonpointer scalar object of type default sequence occupies three consecutive numeric storage units.

At 247:28, 30, 32, 35, 37 and 39. Sequence
Increase the item numbers.

After 265:33 add: Include

C.3.5 Including source text (3.4)

It must be possible to specify a mapping between the *char-literal-constant* on an include line and the text to be included. In many cases, a specification to the processor that *char-literal-constant* is to be interpreted to be a file name will be adequate. Otherwise, the following scheme is one method to provide a mapping from *char-literal-constant* to the text to be included.

A file called a *master include file* specifies how the *char-literal-constant* on an include line is mapped to the text to be included. The master include file contains four kinds of commands, and may contain text to be included.

Let `L` be the *char-literal-constant* from an include line, and `C1` and `C2` be *char-literal-constant*s.

`INCLUDE,F (C1,C2)`
: If `C1` is equal to `L`, then the contents of the file named by `C2` are to be used in place of the include line, else subsequent commands are examined.

`INCLUDE,P (C1,C2)`
: If the filename constructed by `C1//L//C2` exists, then its contents replace the include line, else subsequent commands are examined.

`INCLUDE,T (C1)`
: Text following the `INCLUDE,T` command, up to but not including the next `INCLUDE,[FPT]` or `END INCLUDE` command, is associated with this command. If `C1` is equal to `L`, the associated text replaces the include line, else it is skipped, and subsequent commands are examined.

`END INCLUDE`
: If `END INCLUDE` is present, it indicates the end of the master include file.

The processor should rewind the master include file when its end is reached, not after each include line is processed. In this way, if one provides specifications in the master include file in the same order as they appear in program units, the processor need read the master include file only once per program unit.

After 268:35 add Limited

INTENT(LIMITED) is provided to prevent creating a copy of a pointer to a procedure, thereby preventing a procedure from being executed, by way of a pointer, after its host environment has ceased to exist.

At 274:3 change EXIT

"there is no requirement for the user" to "the user is allowed, but not explicitly required,"

After 274:4 add CASE .and.

The form (*case-value-range-list*) [.AND. *scalar-logical-expr*] allows one to transfer control to the CASE DEFAULT block under conditions that would not otherwise be allowed.