Fortran 2000 requirement R3 has at various times been referred to as a requirement for procedure pointers or procedure variables. A description that does not prejudge this distinction might be that it is a requirement for a "changeable procedure identity" (cpi) — a named entity to which one can assign the identity of a procedure such that one can later invoke that procedure through that named entity. The main feature that distinguishes a cpi from a dummy procedure (which also fits the previous sentence) is that the duration of "assignment" to the cpi is entirely under program control and not forced to correspond to the lifetime of a procedure execution. Also, a cpi can be used to transitively define another cpi and can be compared with another cpi or a fixed procedure identity.

I believe it is fair to say that, in the judgment of the /data subgroup, the same semantics can ultimately be achieved whether one calls a cpi a procedure variable, a procedure pointer, or something else (e.g., a procedure accessor). The issues all have to do with the notation that one uses to express these semantics and the extent to which one's expectations from data variables and pointers helps or hinders one in remembering this notation and the associated rules on using it. In this document, I have attempted to summarize the specific issues brought out in subgroup discussions. I have attempted to present these issues in an evenhanded manner, but since I do have an opinion on this matter, it is entirely possible that I have not fully succeeded.

What's "Natural"?

There are conflicting arguments about which approach is most intuitive. Proponents of the procedure pointer approach have pointed out that the likely representation of a cpi is the machine address of the code implementing the procedure and that we typically call such addresses pointers. In such a view, a "procedure variable" would be one in which one actually stores the code to implement a procedure. Proponents of the procedure variable approach counter that allocatable arrays and ISO varying strings are examples where the direct representation contains a pointer to the "real" representation and that the collective procedures of a program might be thought of a defining a giant enumeration type. It is strange to have pointers to *X* if we do not have *X* variables and that variables have to be declared targets, but procedures are targets automatically. Proponents of "something else" can argue that both analogies are flawed and that it would be cleaner to start from scratch.

Declaration Syntax

A fundamental problem in the declaration area is that with ordinary data

```
REAL :: a_real_variable
REAL, PARAMETER :: a_real_constant=1.0
```

the simpler declaration syntax describes something that is modifiable and one adds an extra attribute to get something fixed, but that with procedures

```
EXTERNAL :: a_subroutine
REAL, EXTERNAL :: a_real_function
```

the simplest available syntax has already been made to mean fixed identities in existing standards, so it became necessary to add an attribute (this example for the pointer approach)

```
REAL, EXTERNAL, POINTER :: a_changeabe_real_function_identity
```

This was found especially objectionable in the case of derived components, where modifiable things are the only things allowed. There was the further complication, that in some cases one needed to add this extra attribute to statements that don't support multiple attributes in Fortran 90/95. In an attempt to deal with both of these problems, the proponents of procedure variables suggested that their extra attribute (`VARIABLE`) be assumed in derived types. Although slightly more defensible than making a comparable suggestion for the `POINTER` attribute, this had the effect of making component declaration syntax mean something different from what that same syntax would mean outside a derived type definition, so that was unpopular, as well.

To solve the problem of multi-statement declarations (interface blocks) in derived type definitions, a method had been invented to give a name to a set of procedure characteristics, so that a single statement could be used to declare a procedure identity whose interface includes those characteristics. Current subgroup thinking is to treat the latter statement analogously to a type statement with the simplest form declaring changeable procedure identities and the addition of another attribute to denote those that are fixed:

```
PROCEDURE(real_function) :: a_changeable_real_function_identity
PROCEDURE(real_function), EXTERNAL :: a_fixed_external_real_function
```

Note that this declaration approach can be used whether our changeable identities can be called pointers or variables.

"No Procedure" Value

Past experience with existing languages that support some kind of cpi is that many applications require some means of indicating "no procedure" rather than a specific procedure (somewhat analogous to an optional dummy procedure). It would be possible, to simply require programmers to create their own "no procedure" identities by creating appropriately named procedures, but most people seem to prefer the idea of a language supplied identity for this purpose. For the procedure pointer approach, the value `NULL()` is available at no cost. For the procedure variable or "something else" approaches, there is a small cost – it is necessary either to create a separate method of creating a "no procedure" value or to explicitly extend NULL to generate such non-pointer special values. (The latter approach appears to be the one currently favored by procedure variable proponents.)

Integration with Dummy Procedures

With the procedure variable approach, it seems almost inevitable that one say that the existing feature called a "dummy procedure" is nothing more than a "dummy variable"

where the variable is a "procedure variable" (i.e., a "dummy procedure variable"). Given the similarity between a dummy procedure and a cpi, this has the desirable effect of making them the same, so the rules for assignment/association need be written only once. However, the need to be completely compatible with the existing feature also creates complications: Dummy procedures are, in effect, `INTENT(IN)`, and many implementations take advantage of this and use a method of passing procedures that does not allow for changing the input. Thus, to allow Fortran 2000 implementations to be object-compatible with existing Fortran 90/95 implementations, we would need rules like

- Procedure variables are `INTENT(IN)` by default (as opposed to the unspecified intent for all other variables).
- If a dummy procedure variable is given an intent other than `INTENT(IN)`, the interface must be explicit where it is called (as opposed to intent of all other variables having no effect on explicitness).

With the procedure pointer approach, one could do this integration (albeit, with more text to justify associating a fixed procedure identity with a cpi), but one also reasonably has the option of not doing this integration (to avoid the extra explicitness and intent rules), at the cost of having to explain the interaction between dummy procedures and procedure pointers. The something else approach is in about the same situation as the pointer approach.

Assignment

The procedure variable approach suggests

```
cpi = procid
```

The procedure pointer approach suggests

```
cpi => procid
```

The something else approach suggests

```
cpi := procid
```

or

```
call proc_assign(cpi,procid)
```

In the procedure variable approach, will users have problems with the difference between the following two statements?

```
name1 = procid    ! This would be "name1 => procid" in the pointer approach
name2 = procid()  ! This is an ordinary assignment in both approaches
```

This is, of course, the same distinction that has to be made in the following:

```
call sub1(procid)
call sub2(procid())
```

In the absence of new rules, the procedure variable approach could allow

```
cpi = 3.14159265358979323846264 3
```

invoking a defined-assignment procedure. Is this what we want? There is some suggestion that the procedure pointer approach may also allow this, but not

```
cpi => 3.14159265358979323846264 3
```

Comparison

Procedure variables:

```
IF (cpi /= NULL()) CALL cpi
IF (cpi2 /= taboo_function) x = cpi2(y)
```

Procedure pointers:

```
IF (ASSOCIATED(cpi)) CALL cpi
IF (.NOT.ASSOCIATED(cpi2,taboo_function)) x=cpi2(y)
```

Something else

```
IF (PASSOCIATED(cpi)) CALL cpi
IF (.NOT.PASSOCIATED(cpi2,taboo_function)) x=cpi2(y)
```

Mostly, this is a question of what you find "prettiest", but there may be some concern in the procedure variable approach whether there would be confusion among

```
IF (cpi==cpi2) CALL do_something          ! are these the same procedure?
IF (cpi()==cpi2()) CALL do_something      ! do they return the same result?
IF (cpi==cpi2()) CALL do_something        ! ?! cpi2 is a procedure-valued
            ! Is cpi the same procedure as the procedure returned by cpi2
```

This is certainly unambiguous to the compiler, so this is "merely" a question of whether programmers might be confused. In the procedure pointer approach, these three statement would look like the following:

```
IF (ASSOCIATED(cpi, cpi2)) CALL do_something
IF (cpi()==cpi2()) CALL do_something
IF (ASSOCIATED(cpi, cpi2())) CALL do_something
```

Arrays of Procedures

With the procedure variable approach, an obvious question is that if we have scalar procedure variables, can we also have array procedure values? If so, we can ask whether one can use elements of such arrays directly

```
x = proc_array(subscript)(arguments)
```

or only indirectly

```
scalar_proc = proc_array(subscript)
x = scalar_proc(arguments)
```

If direct notation is allowed, how does it interact with our notion of array expressions? Do we allow things like the following?

```
array(:) = proc_array(:)(arguments)
```

If so, would

```
array(1:n) = proc_array(1:n)(array2(1:n))
```

be more like

```
do (i=1,n); array(i) = proc_array(i)(array2(i)) ; end do
```

or

```
do (i=1,n); array(i) = proc_array(i)(array2(1:n))) ; end do
```

or could it be like either depending on the explicit interface of `proc_array`? In a slightly different direction, could elemental procedures now accept procedure array arguments, as in the following?

```
result(1:n)=integrate(function(1:n),start_point(1:n),end_point(1:n))
```

In yet another direction, it becomes visually ambiguous whether

```
IF (cpi1(i)==cpi2(i)) CALL do_something
```

is comparing the results of two procedures or the identities of two procedure array elements?

The extent of such questions suggests that at least some of these options should be prohibited.

For all three approaches, one can use the circumvention of putting a cpi in a derived type and then declaring an array of that derived type:

```
TYPE wrapper
 PROCEDURE(real_function) :: proc
END TYPE wrapper

TYPE(wrapper), DIMENSION(10) :: proc_array
```

This allows syntactic expression equivalent to many of the previous questions. E.g.

```
x = proc_array(subscript)%proc(arguments)
```

In the procedure pointer approach, many of the hard questions are moot because referencing a pointer component of an array parent is already prohibited. In the procedure variable or something else approaches, a similar restriction could be added (as a slight bump in the language).

Pointer to cpi

In the procedure variable approach, the obvious expectation is that one can make such a variable a target and then have a pointer to such variables (or allocate them). This makes it necessary to distinguish the following:

```
ptr_to_proc_var = NULL()      ! set the proc variable to "no procedure"
ptr_to_proc_var => NULL()     ! indicate there is no procedure variable
```

With the other approaches, one once again uses the wrapper type, so these would be written

```
ptr_to_cpi%proc => NULL()     ! set the cpi to "no procedure"
ptr_to_cpi => NULL()          ! set the ptr to indicate no cpi
```

which seems less syntactically similar.

NULLIFY

Is the `NULLIFY` statement allowed as a synonym for setting a cpi to `NULL()`? (If this is allowed in the procedure variable approach, there is a true ambiguity when one tries to apply `NULLIFY` to a pointer to a procedure variable.)

Procedure-valued Functions

I have heard some concern expressed about whether the procedure variable approach can engender extra confusion in the case of a function which returns procedure variable and which uses the function name as its result variable. I have been unsuccessful in constructing an example that seems any more confusing than examples that have nothing to do with procedure variables/pointers/whatever.

Conclusion (Mine, Not the Subgroup's)

None of the costs of the differences described above are individually very large. The issue for me is the collective weight of these costs.

Ω