Subject: Coroutines (hopefully done)
From: Van Snyder

1 Number

TBD

2 Title

Coroutines.

3 Submitted By

J3

4 Status

For consideration.

5 Basic Functionality

Provide for coroutines.

6 Rationale

In many cases when a "library" procedure needs access to user-provided code, the user-provided code needs access to data of which the library procedure is unaware. There are at least three ways by which the user-provided code can gain access to these entities:

- The user-provided code can be implemented as a procedure that is invoked by the library procedure, with the extra data stored in globally-accessible variables.
- The user-provided code can be implemented as a procedure that takes a dummy argument of extensible type, which procedure is invoked by the library procedure, with the extra entities in an extension of that type.
- The library procedure can provide for *reverse communication*, that is, when it needs access to user-provided code it returns instead of calling a procedure. When the user-provided code reinvokes the library procedure, it somehow finds its way back to the appropriate place.

Each of these solutions has drawbacks. Entities that are needlessly public increase maintenance expense. If the user-provided procedure expects to find its extra data in an extension of the type of an argument passed through the library procedure, the dummy argument has to be polymorphic, and the user-provided code has to execute a SELECT TYPE construct to access the extension. Reverse communication causes a mess that requires GO TO statements to resume the library procedure where it left off, which in turn requires one to simulate conventional control structures using GO TO statements. This reduces reliability and increases development and maintenance costs.

Reverse communication is, however, a blunt-force simulation of a well-behaved control structure that has been well-known to computer scientists for decades: The *coroutine*. Coroutines would allow user-provided code needed by library procedures more easily to gain access to data of which the library procedure is unaware, without causing the disruption of the control structure of the library procedure that reverse communication now causes.

Coroutines are useful to implement *iterator* procedures, that can be used both to enumerate the elements of a data structure and to control iteration of a loop that is processing those elements.

7 Estimated Impact

Minor additions to Subclause 2.3.4 and Section 12. Estimated at J3 meeting 169 to be at 5 on the JKR scale.

8 Detailed Specification

Provide two new statements, which we here call SUSPEND and RESUME.

Provide a new form of subprogram, the *coroutine*, that cannot contain an ENTRY statement, and is the only subprogram in which a SUSPEND statement is allowed. A coroutine requires an explicit interface.

Coroutines can stand on their own, or be type-bound procedures or actual arguments. They can be procedure pointer targets, provided the pointer has explicit interface. Generic coroutines are allowed, provided the *generic-spec* is *generic-name*. Recursive and internal coroutines are allowed.

When a coroutine is invoked by a CALL statement, execution continues with the coroutine's first executable construct. When a coroutine executes a SUSPEND statement, execution continues after the CALL or RESUME statement that initiated or resumed its execution; when a RESUME statement is executed, execution resumes after the SUSPEND statement. When a coroutine executes a RETURN or END statement, execution continues after the CALL or RESUME statement that initiated or resumed its execution, and it is an error if one later attempts to RESUME it without first calling it.

A type-bound coroutine shall be initiated using a variable, and resumed using the same variable. A coroutine that is initiated using a pointer shall be resumed using the same pointer. Otherwise, a coroutine shall be resumed from the same scoping unit in which it is initiated.

8.1 Data entities

Variables within a coroutine can have the SAVE attribute, with the usual implications.

Unsaved local variables within a coroutine retain their definition status and values from SUSPEND to RESUME. Automatic objects in addition retain their bounds and length parameter values. The specification part is not elaborated upon resumption. If a coroutine references a module or common block, it is considered to continue to reference it between SUSPEND and RESUME.

Argument association does not survive execution of a SUSPEND statement. New actual arguments shall be supplied on the RESUME statement. If the dummy argument has neither the ALLOCATABLE nor POINTER attribute, the corresponding actual argument supplied on the RESUME statement shall have the same dynamic type, length type parameters and bounds as the corresponding actual argument on the CALL statement that initiated execution of the coroutine. An optional argument shall have the same present/absent state. The value of any argument might be different, and the allocation or association status of an allocatable or pointer argument might change. A change in the value of a variable between SUSPEND and RESUME does not affect the bounds or length parameter values of automatic variables within the coroutine.

8.2 Activation records

The above rules guarantee that coroutines can be reentrant. The following paragraphs suggest one way to implement those rules.

When a coroutine suspends execution by executing a SUSPEND statement, its activation record is saved. When a coroutine is resumed, its activation record is restored. Therefore there is no restriction on where a SUSPEND statement is allowed to appear among the executable constructs.

A type-bound coroutine's activation record is saved in and restored from an extension of the variable by which its execution is initiated or resumed, i.e., in `X`, if it is referenced as `X%C`. If a coroutine is accessed by a procedure pointer, its activation record is saved in the pointer. Otherwise, the processor stores the activation record locally (so CALL and RESUME have to be in the same scoping unit).

9 History

03-258r1, section 1.1	m166
04-149r1	m167
04-345	m169
04-380r1	m170