BRON 84 Three Cheers for Procedures Towards a fuller appreciation of the procedural mechanism C. Bron and E.J. Dijkstra University of Kent at Caterbury, 1984

This paper demonstrates the flexibility of procedures when they are treated as first class citizens in a programming language. The point of view of the authors, is that most programming language designs have failed to recognize the procedure object as a very versatile tool. Their enthusiasm for procedural data types comes from their experience with building a build portable operating system in the Modular Pascal programming language. Their experience with system software for this purpose form the basis for the many examples given in the paper.

The advantages of having local procedures in the programming language, namely hierarchical organization of the program and scoping of identifier names, allows implementation of modules using the Pascal-like procedure syntax. Superficially, a module is just a distinguishing procedure that happens to have locally defined procedures. Procedural parameters corresponding to the exported procedures a module are passed to importing modules. If a procedure is not supposed to be visible outside the module, this is accomplished by doing nothing: the local procedure identifiers will be hidden from view. Module block bodies can be used for conventional module initialization, but interface with a bootstrap program and other modules allows one to propagate the linking of modules, and essentially do program linking from the program.

The last chapter of the paper offers an interesting mixture of examples that illustrate various capabilities through the flexible use of procedures.

Some mentioned uses of procedural parameters include keeping algorithms independent of the type of data they manipulate. For example, a generic sort algorithm where the swap and compare operations have been passed as parameters to the sorting procedure, or a selection/searching process, where the action to be taken for each element found is passed as a parameter.

In contrast to the strictly hierarchical nature of modules for program organization, procedural variables are claimed to provide most of the needed escapes from the rigid hierarchy. Two examples are given of how procedural variables can be used to accomplish this.

One example shows how the exception handling mechanism in a programming language can be modeled very simply, if the exception stack contains tuples of the exception name and corresponding exception handler. The following example shows how to build a device independent I/O mechanism by having the file descriptor initialized, when opening the file, with the appropriate procedures for a device type.

The authors discuss how they arrived at the solution of the separate compilation, module dependency problem, and continue discussing some implementation details of especially external, procedure calls.

The paper presents convincing evidence that the philosophy of using procedural parameters and variables for structural organization of a program leads to greater flexibility. Their arguments make one lament that these ideas and capabilities have not been recognized to a much higher degree before.

The lack of development in this direction among modern programming languages can be attributed to a development along other lines toward the same goal; abstraction, modules, independence of algorithm and data, and so on. Unfortunately there might exist a tendency for these languages towards "featurism" (Ada being a prime example). The beauty of Bron and Dijkstra’s approach is that it requires a minimal number of concepts to achieve equivalent expressive power.

One hindrance perhaps to the adoption of the presented form of these ideas, is the reliance of the authors on a mechanism particular to Pascal, local procedures, for simulating the module construct and exception handling. Then again, it disappeared because of discrimination against nested procedures. The LISP language has from its inception had the sought attitude towards procedures in that they were maniputable much like other elements. Nested procedures are rare in LISP, modules tend to look like an amorphous collection of

functions accessed uniformly through a gateway function. This kind of organization is natural in a dynamic scoping environment, and if the ideas in the paper are transferred to this kind of an environment, procedures may yet achieve recognition.