11-1-1991

Programming experience with early parameter passing mechanisms using modula-2 and pascal.

Joseph Bergin

SCHOOL OF COMPUTER SCIENCE AND INFORMATION SYSTEMS

TECHNICAL REPORT
Number 46, November 1991

Programming Experience with Early Parameter Passing Mechanisms Using Modula-2 and Pascal

Joseph Bergin
Department of Computer Science
Pace University
New York, NY 10038

Stuart Greenfield
Division of Computer Science and Mathematics
Marist College
Poughkeepsie, NY 12601

PACE UNIVERSITY

* Joseph Bergin is Professor of Computer Science at Pace University. His office is on the New York Campus. Professor Bergin holds a doctorate in mathematics from Michigan State University, and his research interests are in programming languages. This is the fourth issue of Technical Reports in under two years devoted to presenting shorter pieces of his work.

  Professor Bergin's central project during the past couple of years has been writing a textbook introducing data structures and algorithms by way of the object oriented paradigm. We are delighted to report that the manuscript is complete and has been accepted by a publisher. Even better, his publisher has already requested a second version of the text in C++ to be titled something like A Course in Data Abstraction: The Object-Oriented Approach Using C++. (Due to marketing considerations, the publisher wants to hold the first version until the publication of the second version.)

* Stuart Greenfield is Associate Professor of Computer Science at Marist College. He is the author of Invitation to Modula-2, published by Petrocelli Books in Princeton, New Jersey.

==========

"Programming Experience with Early Parameter Passing Mechanisms Using Modula-2 and Pascal" was presented at the Seventh Annual Eastern Small College Computing Conference held at Marymount College on October 11-12, 1991. It is published as pages 124-132 within the Proceedings which comprises the November 1991 issue of The Journal of Computing in Small Colleges (Volume 7, Number 2).

==========

PROGRAMMING EXPERIENCE WITH EARLY PARAMETER PASSING MECHANISMS USING MODULA-2 AND PASCAL

Joseph Bergin
Pace University
New York, NY 10038
BERGINF @ PACEVM.BITNET

Stuart Greenfield
Marist College
Poughkeepsie, NY 12601

INTRODUCTION

Most programming language survey textbooks [1][2][3][4][5][6][7] discuss parameter passing at length. Historically this has been an important topic of research and a number of different parameter passing mechanisms have been employed by various languages. This paper discusses a method by which programmers may simulate various mechanisms using currently popular languages which do not support those mechanisms directly. Programming exercises can thus be used to further the understanding of the issues involved in parameter passing. For example, Modula-2, and some enhanced versions of Pascal (e.g., Turbo Pascal and Think Pascal) and standard C provide the capability to simulate the "pass-by-value-result" and "pass-by-name" mechanisms.

It is assumed that the reader is already familiar with the pass-by-value and pass-by-reference mechanisms. In order to simulate the other mechanisms, we employ features found in Modula-2, namely, procedure types and procedure parameters. Although both Pascal and standard C do not support procedure (function) types, they do support passing procedures(pointers to functions) as parameters and thus those languages may be used similarly to simulate the desired unsupported passing mechanisms.

A study of these mechanisms will generate insight into several subtelties of language design. This is especially true of the effects of the interactions of language features such as the relationship between block structrue and the problem of non-local names, and parameter semantics and the problems of variable access and variable aliasing.

In Modula-2, aliasing of a variable may occur through using the same variable for two different formal parameters. Aliasing may also occur if a global variable is passed as an argument in a situation in which the routine called also references that same global as a non-local name. Aliasing may occur even in the case of value parameters when the types represent pointer variables.

Variable access is an important issue because the time at which a variable is accessed may affect the address and/or the value of the variable to which the access refers. For example, if we make an array subscript reference, as in A[n], the expression may refer to any one of a collection of variables: the cells of the array. In Modula-2, when such an expression is passed as an actual parameter to a value or a reference formal parameter, the expression is evaluated once, at subroutine invocation time and either the value of that cell or the address of the cell is used throughout the body of the subroutine as the value of the formal parameter. This is the case even though that subroutine may have access to n, and may change the value of n. This fixed access to a variable is not necessarily true for the pass-by-value-result or pass-by-name mechanisms.

THE PASS-BY-VALUE-RESULT MECHANISM

Parameter passing using the pass-by-value-result mechanism (also known by the name copy-in-copy-out) is a means by which the "in-out" passing mode may be implemented. However, unlike pass-by-reference, which also implements the "in-out" mode, the called procedure does not have direct access to the actual parameters. Stating merely that a set of parameters are to be passed by value-result leaves us with a few ambiguities. Although the order in which actual parameters are "copied-in" to the called procedure does not have an affect on the returned results, the "copy-out" order of those parameters, in general, does. Thus

the copy back order should be specified -- left-to-right or right-to-left. Furthermore, the address of the actual parameters may be calculated at copy-in time only or may be calculated twice, at copy-in time and copy-out time. Therefore, we cite four different versions of the pass- by-value-result mechanism, namely,

- address calculated once, copy back order left-to-right
- address calculated once, copy back order right-to-left
- address calculated twice, copy back order left-to-right
- address calculated twice, copy back order right-to-left

The above versions will be referred to as **vr1LR**, **vr1RL**, **vr2LR**, and **vr2RL**, respectively, throughout the remainder of this paper.

If addresses are calculated once, copies of the actuals are made and assigned to locally allocated variables (the formal pass-by-value-result parameters) of the called procedure at the time of invocation. The procedure then executes using those local variables. At the time of return, the values of the locals are "copied back" in the appropriate order to the actual parameters whose addresses had been determined at subprogram entry time. If addresses are calculated twice, a recalculation of the actual addresses occurs at return time. In this latter case, changing some variable within the subprogram may affect where the final values are copied back to.

Particular forms of this mechanism have been used in some implementations of Fortran[1][2] and it is employed automatically by some versions of Ada[1][3] to implement the "in-out" passing mode in at least some cases, depending on the structure of the actuals being passed. One advantage of this scheme is that it is possible to implement it in such a way that actual parameters need not be limited to variables but may be expressions. In this later case the "copy-back" to the actual would be omitted by the compiler. Fortran implementations employing the pass-by-value-result mechanism are able to simulate both "in-out" mode and strictly "in" mode passing with but one mechanism in this manner[1]. This leads to a bit more flexibility for the programmer in those cases in which a particular parameter need not be returned to the calling environment.

SIMULATING PASS-BY-VALUE-RESULT

Our simulation of the pass-by-value-result mechanism is a direct translation of its description found in the previous section. We use pass-by-reference formal (physical) parameters, creating new names which do not clash with other names in the context of the called procedure for passing by **vr1LR** and **vr1RL**. Thus we use value-result "pseudo-parameters" (logical parameters) which are locals of the called procedure. The physical parameters are copied to the logical parameters on procedure entry and copied back in the appropriate order upon exit.

The formal parameter names need not be changed for **vr2LR** and **vr2RL** and are passed by value. These parameters are used directly in the called procedure. Just prior to return, the address of each simulated value-result actual parameter is recalculated in the appropriate copy-back order.

A set of four simulations (in Modula-2/Pascal) for the four submechanisms of pass-by-value-result are offered below.

For pass-by-vr1LR and pass-by-vr1RL:

step 1. Pass all value-result parameters by reference employing fresh names not otherwise used in the scope of the called subprogram declaration.
step 2. Declare a local for each such passed parameter.
step 3. Prior to the subprogram code, "copy" the parameter values "in"to, the corresponding locally declared variables.
step 4. Let the subprogram code operate on these local variables.
step 5. After completion of the subprogram code, "copy" the locals "back" to the value-result parameters, left-to-right or right-to-left, appropriately.

For pass-by-vr2LR and pass-by-vr2RL:

step 1. Design a procedure type for the recalculation procedure for each parameter type passed. It needs a value parameter of the same type as the value-result parameter we are simulating.

step 2. Pass all value-result parameters by value. For each such parameter also pass a copy back procedure of the appropriate type.

step 3. Let the subprogram code operate on these formal parameters.

step 4. After completion of the subprogram code, simulate the recalculation of the addresses of the value-result parameters by "copying-back" (assigning) the final values of the value-result parameters to their corresponding actual parameters, left-to- right or right-to-left, appropriately. This is accomplished by calling the copy back procedures in the appropriate order.

step 5. At the point of call, create an actual copy back procedure which copies its parameter to the expression representing the formal procedure of the procedure to be called with value result semantics.

As an example, suppose we wish to simulate (using Modula-2) the following subprogram where the pair of parameters are passed by value-result:

```
PROCEDURE p(VALRES x,y:INTEGER);
BEGIN
...whatever
END p;
```

The simulation for vr1LR (or vr1RL) is:

```
(* step 1 *)    PROCEDURE p(VAR xV,yV:INTEGER);
(* step 2 *)    VAR x,y:INTEGER;
                BEGIN
(* step 3 *)    x:=xV; y:=yV;
(* step 4 *)    ...the same whatever
(* step 5 *)    xV:=x; yV:=y; (or yV:=y; xV:=x;)
                END p;
```

And the simulation for vr2LR (or vr2RL) is:

```
(* step 1 *)    TYPE
                        copyBack = PROCEDURE(INTEGER);
(* step 2 *)    PROCEDURE p(x,y : INTEGER; CX,CY : copyBack);
                BEGIN
(* step 3 *)    ...the same whatever
(* step 4 *)    CX(x);CY(y); (* or  CY(y); CX(x); *)
                END p;
```

To simulate the call p(a,b) we create Ca and Cb as:

```
(* step 5 *)
                PROCEDURE Ca(x:INTEGER);
                BEGIN
                a := x;
                END Ca;

                PROCEDURE Cb(x:INTEGER);
                BEGIN
                b := x;
                END Cb;
```

The actual call is the p(a,b,Ca,Cb);

AN EXAMPLE USING THE PASS-BY-VALUE-RESULT SIMULATION

Although, for the bulk of applications, the use of any of the four versions of the pass-by-value-result mechanism, as well as the pass-by-reference mechanism leaves the computing machine in identical states, we can easily find those that do not. As one such application we offer the program shown in Listing 1, written in Modula-2, which performs its indicated task five times, each employing a different "in-out" passing mode implemented using our four pass-by-value-result simulations and the Modula-2 supported pass- by-reference.

The state in which the memory is left at the return from each call is given in Table 1. Note that only vr1LR leaves the memory in the same state as the pass-by-reference mechanism and the results of employing the four different versions of pass-by-value-result are distinct.

```
MODULE pass;
VAR n:INTEGER;
    a:ARRAY [1..3] OF INTEGER;

    PROCEDURE ref(VAR i,j,k:INTEGER);
    BEGIN
    j:=2;
    k:=i+j;
    END ref;

    PROCEDURE vr1LR(VAR x,y,z:INTEGER);
    VAR i,j,k:INTEGER;
    BEGIN
        i:=x;  j:=y;  k:=z;
    j:=2;
    k:=i+j;
        x:=i;  y:=j;  z:=k;
    END vr1LR;

    PROCEDURE vr1RL(VAR x,y,z:INTEGER);
    VAR i,j,k:INTEGER;
    BEGIN
        i:=x;  j:=y;  k:=z;
    j:=2;
    k:=i+j;
        z:=k;  y:=j;  x:=i;
    END vr1RL;

    TYPE
        copyBackInt = PROCEDURE(INTEGER);

    PROCEDURE CAsubN(x:INTEGER);
    BEGIN
    a[n]:=x
    END CAsubN;

    PROCEDURE CN(x:INTEGER);
    BEGIN
    n := x;
    END CN;
```

```
PROCEDURE vr2LR(i,j,k:INTEGER; Ci,Cj,Ck:copyBackInt);
        BEGIN
        j:=2;
        k:=i+j;
        Ci(i); Cj(j); Ck(k);
        END vr2LR;

        PROCEDURE vr2RL(i,j,k:INTEGER; Ci,Cj,Ck:copyBackInt);
        BEGIN
        j:=2;
        k:=i+j;
        Ck(k); Cj(j); Ci(i);
        END vr2RL;

    BEGIN
        n:=3; a[1]:=2; a[2]:=3; a[3]:=4;
        ref(a[n],n,a[n]);
        n:=3; a[1]:=2; a[2]:=3; a[3]:=4;
        vr1LR(a[n],n,a[n]);
        n:=3; a[1]:=2; a[2]:=3; a[3]:=4;
        vr1RL(a[n],n,a[n]);
        n:=3; a[1]:=2; a[2]:=3; a[3]:=4;
        vr2LR(a[n],n,a[n],CAsubN,Cn,CAsubN);
        n:=3; a[1]:=2; a[2]:=3; a[3]:=4;
        vr2RL(a[n],n,a[n],CAsubN,Cn,CAsubN);
        END pass.
```

----------Listing 1.----------

mechanism	n	a[1]	a[2]	a[3]
ref	2	2	3	6
vr1LR	2	2	3	6
vr1RL	2	2	3	4
vr2LR	2	2	6	4
vr2RL	2	2	4	6
MacroExp	2	2	5	4
Name1	2	2	5	4
Name2	2	2	5	4

----------Table 1.----------

THE PASS-BY-NAME MECHANISM

If a formal parameter has been passed using the pass-by-name mechanism, then each time it is referenced, the address of its corresponding actual parameter is recalculated and is used as the target for the reference. The simplest way to implement pass-by-name is by employing the idea of macro expansion to replace the names representing the formal parameters by the names representing the corresponding actual parameters everywhere in the body of the called procedure. Then execute the resulting code. Certainly such a methodology may require recompilation of the body of the procedure for each call, so that the addresses of non-local names may be recalculated. To simulate this we may simply insert the names of the actuals into the body of the subprogram manually.

An alternative approach involves passing an actual parameter access procedure to the called routine in place of the actual parameter. This access procedure provides a method for repeated access to either the value of an actual parameter or to its address. Such an access procedure has been termed a "thunk" (by Peter Z. Ingerman, an early implementer of Algol)[8]. A thunk is a parameterless procedure that returns the address of the particular

actual parameter for which it is written. In Algol, a language employing pass-by-name, thunks are produced automatically by the compiler. For simulation purposes we may create these procedures and pass them.

SIMULATING PASS-BY-NAME

We offer three ways to simulate the pass-by-name mechanism. The first uses macro expansion as described in the previous section. Although the simplest, such a method may be quite unappealing to the programmer since it is not true to the approach to the pass-by-name mechanism as it has been implemented in Algol. Nonetheless we will demonstrate this method in a subsequent example.

Our second and third approaches make use of the notion of thunks. If we are working in a programming language such as Modula-2, then we can declare the template for a thunk through the use of a procedure type declaration, such as

```
TYPE THUNK = PROCEDURE():ADDRESS;
```

where type ADDRESS holds the address of a memory cell and is assignment compatible with all pointer types.

Now, in order to simulate the pass-by-name mechanism, within the scope of the actual parameter, we create a thunk corresponding to the actual parameter that is to be passed by name and pass the thunk instead. Within the body of the called procedure each access of a parameter passed by name is replaced by an application of the corresponding thunk followed by a dereference of the resulting pointer. The effect of this is the recalculation of the address of the actual parameter each time it is accessed, rather than once at the beginning of the procedure, as is done using pass-by-reference.

Thus to simulate the pass-by-name mechanism in a language which supports procedure type declarations (or at least procedure parameters) we offer the following steps:

step 1. Declare a pointer type corresponding to each of the data types of the actual parameters.
step 2. Declare a procedure type for each unique actual parameter that returns an address. In Modula-2 the ADR standard function may be imported from System. In Pascal there is no such operator, although many implementations provide one. A common method is to redefine the unary operator @ so that it provides the address of its argument. @X represents the address of X, which may be treated (with care) as a pointer.
step 3. Construct the called procedure heading such that the formal name parameters are the corresponding thunks (use procedure parameters if the language of implementation offers doesn't support procedure type declarations).
step 4. Within the called procedure declare a local variable for each actual parameter.
step 5. In the body of the called procedure any reference to a name parameter is replaced by an assignment of the returned address from the corresponding thunk to the corresponding locally declared variable followed by that variable's dereference.
step 6. Code a thunk for each actual name parameter such that it returns the address of that actual parameter. This thunk is created within the name scope of the actual parameter.

As an example, suppose we wish to simulate (using Modula-2) the following subprogram and subprogram call where the pair of parameters are passed by name:

```
PROCEDURE p(NAME x,y:INTEGER);
BEGIN
...whatever
END p;

p(n,m); (* the call *)
```

The simulation is:

```
(* step 1 *)    TYPE intptr = POINTER TO INTEGER;

(* step 2 *)    TYPE THUNK = PROCEDURE():ADDRESS;

(* step 3 *)    PROCEDURE p(xThunk,yThunk:THUNK );
                 (* or, in Pascal:
                        PROCEDURE p(FUNCTION xThunk:intptr;
                                    FUNCTION yThunk:intptr); *)
(* step 4 *)    VAR x,y:intptr;
                        BEGIN
(* step 5 *)            •
                        •
                        x:=xThunk();
                        ...x^...whatever
                        END p;

(* step 6 *)    PROCEDURE nThunk():ADDRESS;
                        BEGIN
                        RETURN ADR(n);
                                (* ADR is the "address of " operator *)
                        END nThunk;

                        PROCEDURE mThunk():ADDRESS;
                        BEGIN
                        RETURN ADR(m);
                        END mThunk;
```

The call is then p(mThunk, mThunk);

If the language of implementation does not support procedure type declarations then the called procedures will have no parameters to simulate pass-by-name and the corresponding coded thunks will have to be employed as globals in the called procedure. The next section will illustrate this approach.

AN EXAMPLE USING THE PASS-BY-NAME SIMULATION

As an example we will use the same program that was used in Listing 1. The three ways of simulating the pass-by-name mechanism are demonstrated in Listing 2 along with the needed additions to the Listing 1 main body.

The state in which the memory is left at the return from each call is given in Table 1. Note that all three approaches yield the same results, as would be expected.

```
PROCEDURE MacroExpansion();
BEGIN
n:=2;
a[n]:=a[n]+n;
END MacroExpansion;

PROCEDURE aSubnThunk():ADDRESS;
BEGIN
RETURN ADR(a[n]);
END aSubnThunk;
```

```
PROCEDURE nThunk():ADDRESS;
BEGIN
RETURN ADR(n);
END nThunk;

TYPE intptr=POINTER TO INTEGER;
THUNK=PROCEDURE():ADDRESS;

PROCEDURE Name1(x,y,z:THUNK);
VAR  i,j,k:intptr;
BEGIN
j:=y();    j^:=2;
i:=x();j:=y();k:=z();   k^:=i^+j^;
END Name1;

PROCEDURE Name2();  (* using only globals *)
VAR  i,j,k:intptr;
BEGIN
j:=nThunk();  j^:=2;
i:=aSubnThunk();
j:=nThunk();
k:=aSubnThunk();  k^:=i^+j^;
END Name2;

(* main body additions *)
BEGIN    .    .
n:=3;  a[1]:=2;  a[2]:=3;  a[3]:=4;
MacroExpansion();
n:=3;  a[1]:=2;  a[2]:=3;  a[3]:=4;
Name1(aSubnThunk,nThunk,aSubnThunk);
n:=3;  a[1]:=2;  a[2]:=3;  a[3]:=4;
Name2();
END pass.
```

----------Listing 2.----------

Pass by name has more than historical interest. While it is complicated and difficult to use without error, it is a very powerful device. Using it, it is possible to write very general code to manipulate data structures, including recursively defined structures. Jensen's device applies a process to an array, permitting great flexibility, including specification of the operation to be performed and even of the structure of the array. In particular the same process may be made to operate on an array of scalars as an array of arrays.

REFERENCES

[1] MacLennan, B., "Principles of Programming Languages", 2nd Ed., Holt, Rinehart and Winston, 1987
[2] Sebesta, R., "Concepts of Programming Languages", Benjamin/Cummings, 1989
[3] Sethi, R., "Programming Languages: Concepts and Constructs", Addison- Wesley, 1989
[4] Ghezzi, C. & M. Jazayeri, "Programming Language Concepts", 2nd Ed., Wiley, 1987
[5] Horowitz, E., "Fundamentals of Programming Languages", 2nd Ed., Computer Science Press, 1984
[6] Ledgard, H. & M. Marcotty, "The Programming Language Landscape", 2nd Ed.(?), SRA, 198?
[7] Pratt, T., "Programming Languages: Design and Implementation", 2nd Ed., Prentice-Hall, 1984
[8] Ingerman, P., "Thunks", Communications of the ACM, Vol.4 No.1, 1961
[9] Wexelblat, R. (editor), "History of Programming Languages", Academic Press, 1981

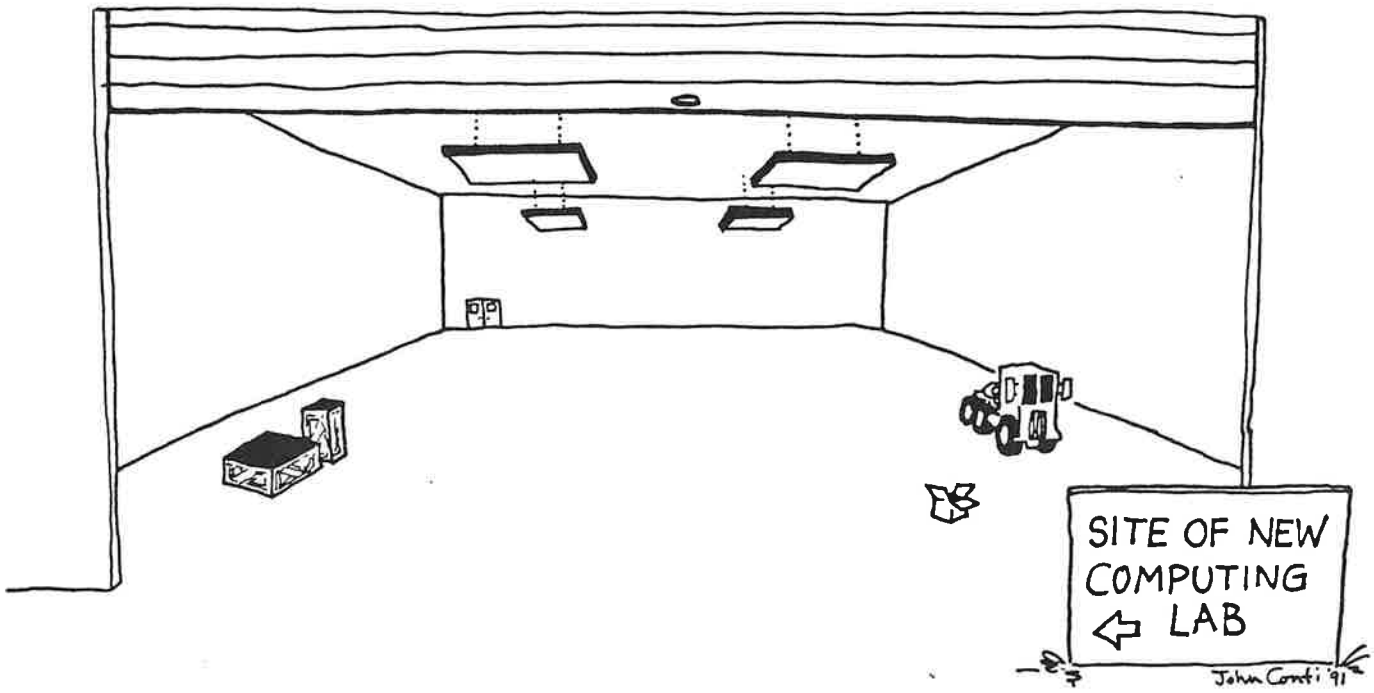
```
PROCEDURE nThunk():ADDRESS;
BEGIN
RETURN ADR(n);
END nThunk;

TYPE intptr=POINTER TO INTEGER;
THUNK=PROCEDURE():ADDRESS;

PROCEDURE Name1(x,y,z:THUNK);
VAR  i,j,k:intptr;
BEGIN
j:=y();   j^:=2;
i:=x();j:=y();k:=z();   k^:=i^+j^;
END Name1;

PROCEDURE Name2();  (* using only globals *)
VAR  i,j,k:intptr;
BEGIN
j:=nThunk();  j^:=2;
i:=aSubnThunk();
j:=nThunk();
k:=aSubnThunk();  k^:=i^+j^;
END Name2;

(* main body additions *)
BEGIN    .    .
n:=3;  a[1]:=2;  a[2]:=3;  a[3]:=4;
MacroExpansion();
n:=3;  a[1]:=2;  a[2]:=3;  a[3]:=4;
Name1(aSubnThunk,nThunk,aSubnThunk);
n:=3;  a[1]:=2;  a[2]:=3;  a[3]:=4;
Name2();
END pass.
```

----------Listing 2.----------

Pass by name has more than historical interest. While it is complicated and difficult to use without error, it is a very powerful device. Using it, it is possible to write very general code to manipulate data structures, including recursively defined structures. Jensen's device applies a process to an array, permitting great flexibility, including specification of the operation to be performed and even of the structure of the array. In particular the same process may be made to operate on an array of scalars as an array of arrays.

REFERENCES

[1] MacLennan, B., "Principles of Programming Languages", 2nd Ed., Holt, Rinehart and Winston, 1987
[2] Sebesta, R., "Concepts of Programming Languages", Benjamin/Cummings, 1989
[3] Sethi, R., "Programming Languages: Concepts and Constructs", Addison- Wesley, 1989
[4] Ghezzi, C. & M. Jazayeri, "Programming Language Concepts", 2nd Ed., Wiley, 1987
[5] Horowitz, E., "Fundamentals of Programming Languages", 2nd Ed., Computer Science Press, 1984
[6] Ledgard, H. & M. Marcotty, "The Programming Language Landscape", 2nd Ed.(?), SRA, 198?
[7] Pratt, T., "Programming Languages: Design and Implementation", 2nd Ed., Prentice-Hall, 1984
[8] Ingerman, P., "Thunks", Communications of the ACM, Vol.4 No.1, 1961
[9] Wexelblat, R. (editor), "History of Programming Languages", Academic Press, 1981



... Of course, our CPU fits on the head of a pin, but we figured, "GO FOR IT!"