LITTLE Newsletter # 41 R. Dewar
December 1, 1975

Dynamic Arrays in LITTLE

This newsletter contains two suggestions for extension to the LITTLE language. The first adds a dynamic array facility, the second deals with variable size item handling.

One of the advantages of requiring all indirect references to be in base+offset form (e.g., indexed array operations) rather than direct pointer form is that flexible arrays in the ALGOL-68 sense are easily implemented. This is because the operation of increasing the size of an array can be implemented by copying the array to another location in memory without worrying about instances of pointers to the original copy.

The need for such a feature is clear, even at the LITTLE level. All modern computer systems (that I know of) allow dynamic determination and use of available memory. Many systems also allow dynamic allocation and deallocation of memory at run time (e.g, CDC 6600, UNIVAC 1100, DEC 10, but not the IBM 370). The failure of LITTLE to provide such a feature results in objectionable constraints leading to (for example) the phenomena of fixed table sizes in the LITTLE compiler itself and the small-medium-large versions of SETL. Without this feature the implementation of heap languages (e.g., ALGOL-68, SIMULA-67, SNOBOL4) is quite unsatisfactory in a LITTLE environment.

It appears that this feature can be added to LITTLE with a minimum of effort and in an efficient manner. The remainder of this note details a specific suggestion.

1. Declaration of Dynamic Arrays

An array would be declared dynamic by omitting the bounds specification, e.g.

```
DIMS X( )
```

Such declaration could appear anywhere that DIMS declarations are allowed.

2. Allocation of Dynamic Arrays

The allocation or reallocation (change of upper bound) of an array is effected by:

```
SETDIMS (arrayname, newupperbound)
```

Elements with common subscripts would be retained. New element values would be undefined (or in general, treated the same way as initial values of static arrays in the absence of DATA statements).

The initial allocation gives an upper bound of zero, prohibiting references until a SETDIMS occurs and also prohibiting <u>compile</u> time DATA statements for dynamic arrays. (There is a temptation to provide a feature for initial allocation specification but it would be hard to implement).

3. Use in SUBROUTINE Calls

The use of SETDIMS while any dynamic array is currently bound as a parameter causes unacceptable implementation difficulties. There are two possibilities:

(a) Prohibit use of SETDIMS while any dynamic array is bound as a parameter.

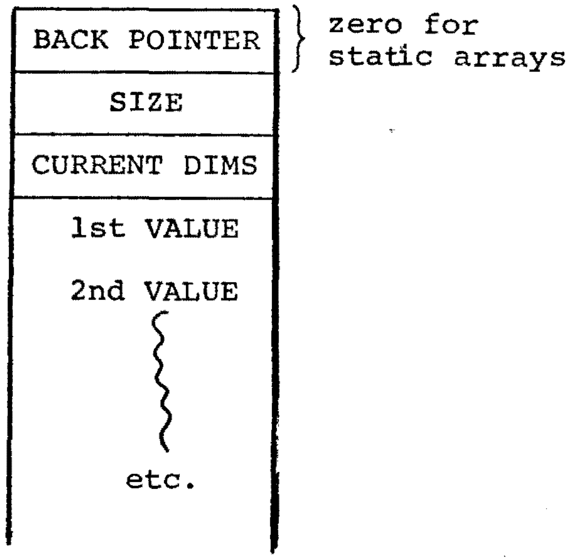
(b) Prohibit the use of dynamic arrays as parameters.

(b) is much clearer but is probably over restrictive, but this choice is left open.

<u>IMPLEMENTATION</u>

The following describes a general implementation scheme which includes provision for subscript checking. This latter provision is generally applicable to static as well as dynamic arrays.

An "array value" appears as follows



A dynamic array declaration allocates a static word whose value is a pointer to the current array value (the pointer probably points to the first value for historical consistency). The initial pointer points to a dummy array value whose current DIMS is set to zero and whose backpointer points to the static word.

At the start of execution a heap area is allocated. The mechanism varies from system to system, but all systems I know of cater to this requirement. A next-available location pointer is set to point to the start of the area.

On execution of a SETDIMS, there are two cases:

1) The array is currently not allocated, as indicated by the fact that its value address is not in the heap. In this case a new value is allocated at the end of the heap; and the static pointer (accessible through the back pointer in the dummy value) is made to point to the new array.

2) The array is currently on the heap. In this case, its size is changed by moving any arrays beyond it up or down (their backpointers allow corresponding adjustment of their static pointer cells). Note that this approach obviates the "double storage" required by reallocation schemes.

Reference to elements of dynamic arrays generate code entirely analogous to that for formal parameters, the array address being taken from the static pointer location instead of the parameter list. The penalty for such accesses will vary from machine to machine. Note in particular that the arbitrary choice of ones origin is unfortunate; it may be worth changing the dynamic array pointers to be zero origin at the expense of some inconsistency between such pointers and subroutine parameter pointers. This latter choice is even more attractive if dynamic arrays are not allowed as subroutine parameters.

On some machines (e.g., UNIVAC 1100, CDC 6600), which are typically those for which the access penalty is highest, it is possible to access one array (the one at the start of the heap) directly and avoid the penalty. One possibility is to treat the first dynamic array in the START routine in this manner and tell users that this is the case. This would allow the heap of a SETL, SNOBOL4 etc., implementation to be handled in this more efficient manner.

The current DIMS of an array is always behind the first element and this may be used to check subscripts, even for formal parameters of subroutines. It could also be used to implement a GETDIMS(array) function which would give the current size of a (static or dynamic) array.

If it is decided to allow dynamic arrays to be passed as parameters, then the SETDIMS restriction can be checked by maintaining a global counter incremented before such a call and decremented after, SETDIMS being allowed only when zero.

The size field of arrays can be used to check parameter consistency, although such a check is much safer if the base address of an array is the most significant (rather than the least significant) word of the first element.

Given the introduction of the DIMS X( ) notation, it might as well be used for subroutine parameters, eliminating the current integer which has syntactic but not semantic significance. SETDIMS is of course not allowed for subroutine parameters in any case.

Handling variable size items in LITTLE

Although not stated in the manual, it seems reasonable that the size of a subroutine parameter must match the size of the corresponding argument. The current LITTLE compiler will in general generate wrong code if this is not the case, where wrong means uninterpretable by the published rules.

There is however a need for handling multi-word items in a more dynamic manner. For example, the system routines LOR, LAND disobey the SIZE matching requirement and work only because the meaning of undefined is "known" in this case. The fact that "cheating" is going on here is made clear by the fact that these particular routines failed in one recent transportation effort which would otherwise have worked.

The following is an attempt to define and codify the cheat used in LOR and LAND.

The meaning of SIZE X(N) where X is a parameter will be that N is the largest possible size which the argument may have.

If the parameter is only used with field extractors or as an argument in a further call, no temporaries are needed anyway (this is the case in LAND and LOR). In this case an effectively infinite value can be used for N to indicate that arbitrary size items can be handled.
Alternately an extra notation

```
SIZE X( );
```

might be introduced to correspond to this situation, in which case accesses would be limited to these two types (field extraction and calls).

Where other kinds of references are present (assignments, direct operations) temporaries of size N are allocated, as is appropriate to handle the largest possible item.

Note That extracting a field outside the <u>actual</u> size (even if within the declared size) is undefined, just like referencing an argument vector outside its actual bounds.