LITTLE Newsletter Number 3 November 1, 1971

Possible future extensions to LITTLE J. T. Schwartz

Various semantic extensions to LITTLE, aimed principally at certain basic issues crucial in the creation of systems and large programs, could greatly improve its value as a systems writing language. This newsletter is intended to begin the process of getting some of these things on paper. The main areas which deserve to be treated are as follows:

1. Memory hierarchy management.
2. Interrupt handling.
3. Debugging features.
4. Syntactic extensions.

In this newsletter, areas 1 and 4 are discussed. The other areas will be addressed in another newsletter.

1. Memory hierarchy management.

Under this heading various desirable possibilities may be contemplated.

A. Virtual memories and associated paging structures.

B. Special types of memory structures, allowing extension, movability, and particular special types of paging.

C. Paging of sections of code; physical grouping of code sections and data items in a manner permitting efficient paging.

1A. Virtual memory structures.

Of the many possible approaches to this interesting area, we choose the following, which serves at least to fix our attention. One-dimensional arrays of a given item SIZE (in the LITTLE sense) are provided. These arrays are dimensioned; some, as presently in LITTLE, with fixed dimensions; others with 'contingent' dimensions, in a manner to be explained below. We think of physical storage as providing a number of arrays, having fixed maximum possible dimensions, but from the point of view of any particular program having dimensions which are 'contingent' but which cannot exceed these physical maxima. These 'physical storage arrays' might have such names as CENTRALMEM, DRUM, DISC1, TAPEDRIVE3, STRIPFILE10, etc., depending on the actual collection of physical devices available. Each of these physical arrays will then have its own particular physical limits and performance characteristics.

We now introduce a family of declarations which allow special storage treatment to be declared for an array A. If no special declaration is made for A, it is stored in central memory in the standard fashion. However, if a declaration is made for A, then A will be stored on some secondary storage medium, certain pages of A being dynamically brought to higher storage levels as required.

The form of the declarations which we contemplate, and their semantic effect, will now be explained.

The general form of a storage declaration is as follows:

```
(1)  STORE  <array name>  PAGESIZE  <integer>
                               ON <target array name> <(optional) page
                                                        level list>.,

     Example:
(2)  STORE A PAGESIZE 512 ON DISC, PAGES
         3 ON CENTRALMEM, 10 ON DRUM.,
```

In general, <array name> in (1) is the name of an array for which a declaration is being made; <integer> declares the number of words (of machine-dependent standard SIZE) in a single 'page' of the array A; <target array name> declares the 'target array' within which A is stored. This target array may either be some programmer declared array, itself the subject of a storage declaration, or may be some 'physical' array like 'central memory', 'drum', 'disc', etc. If in (1) a <page level list> is given, it will have the form

(3) ,PAGES <pagelt>, <pagelt>,...

where <pagelt> has the form

(4) <integer> ON <target array name>,

and states the number of pages of the array A which are to be held within some specific target array.

For a set of declarations having the form (1) to be valid, we require that they be non-recursive. More specifically, it must be possible to assign indices to the arrays which occur in these declarations in such a way that each <target array> occurring in the declaration of an array A has an index higher than the index of A. Certain 'top level' arrays A will then not be target arrays of anything; dimensions should be declared for these. All other arrays B will be of 'contingent' dimension, B having a size deducible from the sizes of all the arrays stored on B. Certain 'bottom level' arrays will have no declared target arrays; these are ultimate parameters of the program in which they occur and must be assigned to available physical devices when this program is enabled for execution.

Note that the syntactic style proposed in the preceding paragraphs allows the arrays regarded as 'physical' in an initial program version to be treated as 'logical' if this becomes necessary. For example, a program containing the declaration (2) can be run on a configuration containing no drum by adding some such declaration as

STORE DRUM PAGESIZE 1024 ON DISC.,

to the program.

If no <page level list> is included in the declaration (1) defining the manner in which a given array is to be stored, the system will append some standard default list.

The appearance of an <array name> A in a declaration (1) implies the creation of a subsidiary 'index array' for A; we refer to this subsidiary array as A.INDEX. The nominal number of entries in A.INDEX is equal to the number of entries in A, divided by the page size of A. It should also be possible to declare the storage treatment to be accorded A.INDEX. Suppose for example that a program uses two large arrays, one, A, in a relatively dynamic manner; the other, B, as an 'archival' backup and less dynamically. In this hypothetical case, some such declarations as the following might be used.

```
DIMS  A(100  000), B(10  000  000).,
STORE A PAGESIZE 500 ON DISC,
    PAGES 20 ON DRUM, 5 ON CENTRALMEM.,
STORE B PAGESIZE 4000 ON STRIPFILE,
    PAGES 20 ON BREAKUP.,
STORE BREAKUP PAGESIZE 500 ON DISC,
    PAGES 5 ON DRUM, 2 ON CENTRALMEM.,
STORE B.INDEX PAGESIZE 500 ON BREAKUP.,
STORE A.INDEX PAGESIZE 200 ON CENTRALMEM.,
STORE BREAKUP.INDEX PAGESIZE 200 ON CENTRALMEM.,
```

B. <u>Special types of memory structures, allowing extension, moving, and particular special types of paging</u>.

The proposal made above allows an array for which growth to large size is anticipated to be declared with a very large dimension; most of the array can reside on a secondary medium, with parts being paged in. For the effective use of this technique it is important, however, that the declarations described above should allow a certain degree of dynamic variability. For example, if an array A used in a program grows while another B fails to do so, we may wish to increase the number of pages of A stored on a high-grade storage array C while diminishing the number of pages of B so stored. It may be desirable to allow several arrays, or portions of several arrays, to be stored within C, and to move the boundaries between the parts of C devoted to storing these arrays, depending upon the amount to which each array has grown or shrunk since the last allocation was made. At certain points in the execution of a program it may be the case that certain arrays A lose their significance; for example, during a compilation a 'generated code' array loses its significance the first time a fatal compilation error occurs. It is then desirable to be able to relase for other use all the space in a high-grade storage array G which such an array A formerly occupied.

Certain arrays will be accessed in a pattern showing regular trends: perhaps scanned always from low addresses to high addresses, perhaps active after the manner of a pushdown stack in which access moves regularly up and down the stack, etc. For use in such cases, one may wish to provide mechanisms which assure anticipatory paging of data blocks whose imminent use can be anticipated.

A general, though possibly over-expensive, scheme for use in these cases is as follows. Make it possible for a programmer defined trap to be set on each attempt to reference an array address nominally not present in central memory. The code at such a point can drop blocks apt not to be needed, and issue references to blocks likely to be needed, thereby forcing them to load.

This may imply the provision of additional statements such as

```
PUSH   A(J)  TO <target array>,
```

which would initiate a series of background actions eventually resulting in the page containing the array element A(J) being moved to a higher or lower storage level. A "$-\infty$" storage level might then be equivalent to erasure.

This whole rather important issue deserves careful semantic and syntactic exploration.

C. <u>Paging of sections of code, physical grouping of code sections and data items in a manner permitting efficient paging.</u>

Storage-management mechanisms like those described in the preceding paragraphs might do most of what is necessary for the paging of code, provided that a method for assigning particular sections of code to particular code-storage arrays is made available. To this end, it might be sufficient to provide a declaration having the form

STORECODE <target array name>

Such a declaration will force the section of code running from its occurrence to the next following STORECODE declaration to be kept in a given target array. Thus, for example, code producing exceptional error printouts, together with the format information and message text these require, etc., can be kept in some array normally held in secondary memory, etc.

Note that for this application mechanisms like those described in the preceding section, which permit the parameters occurring in storage declarations to be varied dynamically, can be particularly valuable.

It is also appropriate that blocks of information declared to be stored in a target array A should be arranged serially within A in the order of their declaration. This permits one to keep together blocks of code and data likely to be exercised in close temporal proximity.

A quite different technique, but one which also addresses the problem of code storage, may be addressed here. Code stored in special interpretable format can allow a higher density of packing than normal machine code (at a substantial cost in efficiency). This density comes from the possibility of using short address fields keyed to the variable names and transfer labels occurring in a given code section, and from the suppression of temporary variable names. An expanded operation code set, allowing frequently occurring operation sequences to be represented densely, can also be incorporated to advantage. The prototypical statement I=I+1 can be represented interpretively as

I, 1, STORE, I ,

which allowing 1 byte/items is 4 bytes. In full machine code for a standard machine the same statement might be

LOAD R1, I; ADD IMMEDIATE 1, R1; STORE R1, I

which, allowing 4 bytes for a fullword instruction, would be approximately 10 bytes of code. Thus, the use of an interpretive format may yield a 2-1 reduction in code size. An interpreter for an average machine might run to some 8000 bytes of storage with a speed loss of 100-1. Therefore, the segregation into interpretive format of 1-2000 statements of code, whose execution should occupy less than 1 percent of the running time of a total program, should begin to achieve storage economies at a relatively limited cost in speed. For large programs with a very scattered pattern of execution, this may be a better technique than more straightforward paging.

A possible syntactic convention in which this technique could be embodied is as follows: allow an array A used for the storage of code to be designated as

INTERPRETIVE A.,

Blocks of code stored in A would then have compressed interpretive format, and would be interpreted when their execution was called for.

2. Syntactic extensions.

When a first LITTLE compiler is completed, it will be appropriate to extend its translator section considerably. As long as a given extension does not change the semantics of the language, that is, as long as the extended language has a straightforward translation into the unextended language, this will not affect the 'middle' and 'back' portions of LITTLE, i.e., its optimiser and code-generator sections. The following constructions, many proposed for SETL, are desirable for LITTLE; some even more in LITTLE than in SETL.

A. If-then-else constructions.

IF (<expression>) <statement>

IF (<expression>) THEN <block>.,

IF (<expression>) THEN <block> ELSE <block'>.,

IF (<expression>) THEN <block> ELSE IF
(<expression'>) THEN <block'>.,

and so forth. A <block> is a sequence of <statement>'s. The SETL scope terminators END IF., etc., are also desirable.

B. The IFF-statement.

This construction is of great value in making complicated sets of tests more transparent, and ought to be included in LITTLE, perhaps with somewhat restricted rules.

C. 'While' iterations.

(WHILE <condition>) <block>.,

(WHILE <condition> DOING <block'>) <block>., etc. This subsumes the FORTRAN-like DO-loops, which can be written

```
J=1., (WHILE J.LE.LIM DOING J=J+1) <block>.,
```

A form even closertto the FORTRAN 'DO' can then be obtained readily using the macro features which will be available.

In this connection the SETL

```
QUIT.,
```

and the statement

```
ITERATE.,
```

corresponding in meaning to the SETL 'continue' should be available. Note that 'CONTINUE' in LITTLE has a different meaning.

D. <u>'At' constructions.</u>

As in SETL, a statement having the form

```
AT <label>  <block>.,
```

could be useful for the physical concentration of logically related code. Labels referenced in AT statements might be restricted to have names beginning with 'at', or some such.

E. <u>Local name scopes.</u>

The present SETL name-scoping rules have their horrible side; note in particular that the various subroutines comprising a total program can at present not always be rearranged without serious semantic consequences ensuing. To relieve some of these problems, it is proposed to add a new declaration.

LOCAL <sizeelement>,..., <sizelement>.,

This would act in much the same way as the present

SIZE <sizelement>,..., <sizelement>.,

except that names declared as LOCAL would without further declaration not be known outside the subroutine within which they were declared.

F. Improved data statements.

The present

DATA <var> = <const>,...,<const'>.,

should also allow

DATA <var> = <constlist>/...,

where <constlist> is a comma-separated list of <constelt>'s, and <constelt> has the syntax

<constelt> = <constexpn> | <constexpn> (<constlist>)

A <constexpn> is any expression containing only constants, no variable names.

In the second construction, the <constexpn> signifies a number of repetitions. Thus to zero all the locations of a 1000 element array A we may write

DATA A = 1000(0).,

G. Still further extended macros.

A number of relatively small extensions to the present macroprocessor can extend its power significantly. These are as follows.

G1. <u>Macro-expansion output will pass through the macro-definition detector.</u>

This will permit macro definitions to be imbedded in macros, albeit in a somewhat roundabout way. (Here we prefer a roundabout method to the direct method which might be made available, since allowing such combinations as

```
+* A = +* B = ...
```

can lead to errors.)

Our roundabout method is as follows. We write

```
+* Q3(W1,W2,W3) = W1  W2  W3**
```

To include a macro definition within a macro, we may then write some such construction as

```
+* DEFINE(WD,TEXT) = Q3(+,*WD=TEXT *,*)**
```

For an example of the use of this type of construction, note that by writing

```
++DO (J,A,B) = J=A.,/ZZZA/
  IF (J.GT.(B)) GO TO ZZZB
Q3(+,*AZZZ=ZZZA*,*)
Q3(+,*BZZZ=ZZZB*,*)
Q3(+,*CZZZ=J*,*)
```

and by writing

```
+* ENDO = CZZZ = CZZZ+1., GO TO AZZZ.,
  /BZZZ/CONTINUE **
```

we may then employ simple (but not nested)DO-loops having the easy form

```
DO(J, 1, N).,
text
...
ENDO.,
```

G2. <u>An alternate form for generated symbols.</u>

A convention distinctly superior to that presently available is as follows. Permit macro-definitions in the form

```
+* NAME(ARG1,...,ARGn/XARG1,...,XARGm) = text **
```

The 'normal arguments' ARG1,...,ARGn will behave like the present macro-arguments, and are to be supplied when the macro is called. The 'extra arguments' XARG1,...,XARGm are not to be supplied, but will be generated by the macro expander when the macro is called. In this new style, the preceding 'DO' macro could be written as

```
+*DO(J,A,B/ZZZA,ZZZB) = J=A.,/ZZZA/
             ...etc.      **
```

This convention simplifies the present code and removes a number of the technical pitfalls presently afflicting the nested use of macros containing generated symbols.

G3. <u>Iterative macro-arguments.</u>

Suppose that we allow a '-' to be prefixed to certain of the formal arguments of a macro definition, as in

+*NAME(ARG1,-ARG2,-ARG3) = text **.

The semantic intent of this syntactic marking is as follows. Call a formal argument iterative if it is marked with a '-' sign. If the macro is called with a non-parenthesised string in the place of one of its iterative arguments, then the argument is treated in the ordinary way. If a parenthesised string consisting of substrings separated by commas is supplied in place of an iterative argument, then the parentheses will be removed, and macro-expansion will be performed repeatedly, a separate expansion occurring for each substring. For example, the definition

+* PHRASE(-WD) = THE WORD IS WD. **

and the call

PHRASE((YES,NO,MAYBE))

will together result in the expansion

THE WORD IS YES.
THE WORD IS NO.
THE WORD IS MAYBE.

If several iterative arguments consisting of parenthesised strings consisting of comma-delimited substrings are simultaneously supplied, the macro expansion will advance from one substring to another for every iterative argument in each iteration of its expansion. This permits a certain type of 'respectively' construction. For example, the definition

+* EXPLAIN (LANG,-WD1,-WD2) =
WD1 IS THE LANG FOR W2. **

and the call

EXPLAIN (GERMAN,(JA,NEIN,VEILLEICHT),

(YES,NO,MAYBE))

will lead to the expansion

JA   IS THE GERMAN FOR YES.
NEIN IS THE GERMAN FOR NO.
VIELEICHT IS THE GERMAN FOR MAYBE.

Suppose next that a macro with several iterative formal arguments is called with parenthesised, comma delimited, substrings as actual arguments, but that more substrings are given for an argument A then for another argument B. In this case, the final substring of A will be repeated as often as necessary for iteration over all the substrings of B to take place. Thus, for example, the expansion of

EXPLAIN (GERMAN,(NEIN,JA), (NO,YES,OK))

is

NEIN IS THE GERMAN FOR NO.
JA   IS THE GERMAN FOR YES.
JA   IS THE GERMAN FOR OK.

The expansion of a macro is complete when every substring of each of its iterative arguments has been appropriately substituted in the prototype macro body.

G4. Rudimentary pattern matching.

By supplying the LITTLE macro-processor with a pattern matching facility, we can gain the useful ability to call macros in men-standard form. As a lexical form for the invocation of this facility, we propose

(5) ++ <name> <(optional)argument list> <(optional)separator list> <(optional)terminator string>=<text>

Before explaining the detailed syntax and semantics of this type of declaration, we give an illustrative example of its use. By declaring the pattern

    ++DO (J)/(,)/., = DO J =

we ensure that an occurrence of DO<*name> = will be translated into a macro call whose initial part has the form DO(J,... . Following the initial occurrence of DO <*name> = , commas will be taken as delimiting arguments, and ., will be taken as terminating the macro call. Thus, we may write

    DO J = A-1, B+C.,

and have it translated into

    DO (J,A-1,B+C)

which by further macro expansion can give as the code desirable for a do-loop head. Another example is as follows. By declaring

    ++CALLSUB / (,)/)., = CALL SUB(

we make it possible for every (closed) call of a subroutine 'SUB' present in an original text to be transformed into a corresponding macro-call, which may then be expanded in some appropriate manner. This can facilitate a hand-optimisation useful once a program has been debugged.

We now explain the proposed syntax and semantics of (5) more systematically. In (5), <name> is the name of a macro, which is called when the pattern present in <text> is detected in a source string of tokens. If some of the tokens in <text> are not parts of the pattern, but are to be transmitted as arguments of the macro call, an <argument list> should be present in the declaration (5) and these tokens should appear in it. The syntactic form of an argument list, when one is present, is

(< name>,..., < name>).

Once the presence of some particular macro call has been established (by the occurrence of an appropriate sequence of tokens), further macro arguments will be delimited by the occurrence of some token in the <separator list>. If no <separator list> is declared, the comma will act as separator. The macro will be terminated by the occurrence of the pattern of tokens appearing in the <terminator string> of (5); if no <terminator string> is declared, an unbalanced right parenthesis, or an exposed instance of ., , will terminate the macro.

G5. Expansion-time calculations, conditional macro expansions.

A proposal for adding these powerful additional macro-processor features will be made in a later newsletter.