LITTLE Newsletter No. 23 January 17, 1973

D. Shields

NAMESETS: A New Way to Handle Global Variables in LITTLE

We present a hypothetical name-scoping convention for LITTLE which treats "global" variables in a new way. Global variables are realised by defining sets of names, called name-sets. The structure of a LITTLE system using name-sets is also described.

By a global variable we mean a variable which may be accessed in more than one routine; a local variable is a variable accessed by one and only one routine. A routine is a LITTLE subroutine (SUBR) or function (FNCT). Also, any compiler-generated temporary may be viewed as a local variable; and local variables may be viewed as user-defined temporaries.

Global variables are realised in the current LITTLE system by the following simple scheme:

Variable names are not erased from the symbol-table at the end of a routine, and so may be accessed by subsequently compiled routines.

Such global names may be "preserved" across several compilations, since LITTLE currently satisfies the following

a) Storage is allocated in the order in which variables are SIZE-d.

b) All variables defined in a compilation are assigned to a storage block (COMMON block) of the same name as the first routine compiled.

This preservation of names is unwieldy--it assumes knowledge of the internal operation of the compiler, and requires careful monitoring of the routines involved, to insure that the assumed name-to-memory map is never changed (as it would be if a new SIZE statement were inadvertendly placed in wrong location.).

What we need is a scheme for defining global variables which

1) does not depend on the order in which routines compiled (as is current scheme), and
2) is able to support separate compilation of routines in a manner preserving definition of global variables.

Moreover, separate compilation is desirable so that a large system need not be compiled _in toto_ whenever part of it is altered. Unless the user is willing to use the unwieldy scheme sketched above, the current scheme requires total recompilation of large systems. This would not be a problem if the LITTLE compiler were very fast; a rough design goal would be to have a compiler sufficiently fast so that in developing a large system (say twenty thousand LITTLE statements) the compilation time is not significantly greater than the execution time for a typical debugging run. To satisfy this goal requires a compiler capable of compiling several thousand LITTLE statements per second, not a likely prospect. Thus it seems necessary to add to LITTLE the facilities for maintaining libraries of binary modules, i.e., predigested representations of routines which need only be "loaded" to be executed.

Since LITTLE aims to be machine-and system-independent, library support facilities are needed to provide an interface between LITTLE system and the host operating system, in order to avoid expensive conversion from LITTLE formats to host system format. This suggests the inclusion in LITTLE of a pre-loader, which would combine the output of a compilation and libraries produced by previous compilations) to produce a load module for the host operating system. The notion of a pre-loader might also prove useful in the development of an "optimising" assembler, which would perform inter-subprogram optimisations and global register allocation at load-time, when the modules are combined.

These labyrinthine digressions have indicated the factors influencing the design of the name-set scheme; we now define this proposed scheme.

All global variables must be defined with a new type of routine, called a name-definition block, or NAMEDEF. A variable is defined by specifying its size, dimension (if applicable), and any initialisations. This is accomplished by adding to LITTLE a new statement group of the form

```
NAMEDEF name;
... series of SIZE, DIMS,and DATA statements
END;
```

Note that all initialisations (i.e., DATA statements) must occur in the NAMEDEF block in which a variable is defined; otherwise, some standard initialisation, say to zero, is assumed. This is done to insure that variables are not initialised in arbitrary parts of a program (as allowed in the current system), and thus enforces a desired programming discipline. Note further that the NAMEDEF block may contain no executable statements. For those familiar with FORTRAN, we remark that a NAMEDEF block is similar to the FORTRAN BLOCKDATA block, except that no memory map is implied (or forced), as is done when FORTRAN COMMON blocks are declared.

Variables defined in a NAMEDEF block may be accessed from a LITTLE routine in several ways, all of which involve extensions to LITTLE:

a) By using the name-set name as a qualifier, indicated by writing

variable-name → name-set-name

For example,

I → ND = J + K → MD

indicates that the variable I in block ND is to be assigned the sum of J defined in current routine and the variable K in block MD

2) By use of the ACCESS statement, which has the form

ACCESS name-set (list of names);

For example,

ACCESS ND (I, ITOT)

indicates that any (unqualified) use of I or (ITOT) in current routine is to be taken as use of the variable I or (ITOT) in name-set ND. For convenience, default forms of the ACCESS statement are defined as follows:

a) ACCESS name-set$_1$,..., name-set$_k$;

which indicates that references to variables not defined in current routine should be taken as references to variables of same name in any of indicated name-sets.

b) ACCESS;

which is similar to (a), except that all name-sets known to compiler are to be checked. In addition, to provide convenient variable-renaming we allow an "alias" option in the ACCESS statement; e.g.

ACCESS ND (I = IHERE)

indicates that the variable IHERE is the current routine corresponds to the variable I in name-set ND.

Variables may be defined (SIZED) within a routine in the current fashion; all such variables are local, and hence may not be used outside the routine in which they are defined.

(Note that we might extend the ACCESS statement to allow read-only use of global variables, so that a routine could use, but not alter, a global variable. Similarly, a "password" might be required to access a name-set. These considerations might prove useful in designing an operating system; questions of privacy and security of global variables are perhaps best viewed by thinking of NAMEDEF blocks as "files" with well-defined, and protected, access paths.)

A further extension, which would enforce the same discipline on macros which the name-set method enforces on the use of global bariables, would be the addition of a macro-definition block, or MACDEF. Such blocks would contain macros used in more than one routine, and would distinguish system-wide macros from those programmer-defined macros used with a single routine to simplify coding. Such blocks might take the form

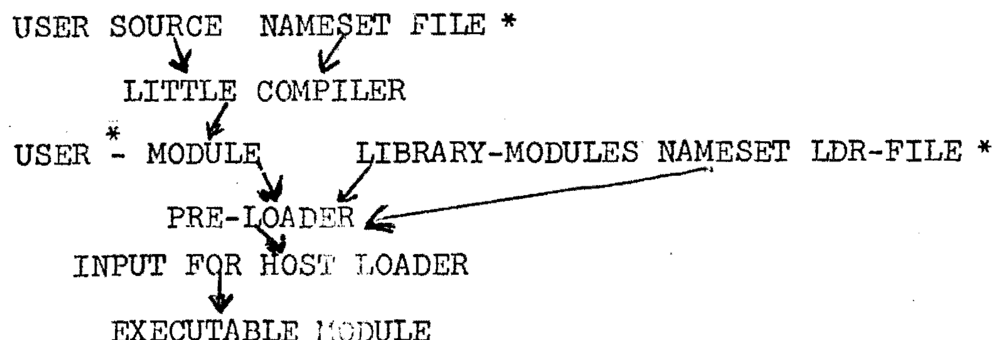
```
MACDEF macro-set;
     ... series of macro definitions...
END;
```

There remains an important class of global names--subroutine and function names. Function names may be elements of name-sets, since they need only be sized. It might be convenient to distinguish a special name-set, SUBRS, to which the compiler adds the name of each routine as it is compiled.

In a projected implementation, name-sets could be maintained in two auxiliary forms:

a) as auxiliary input to the compiler, so that NAMEDEF block need not be included in input source for each compilation. The representation might consists of a pre-hashed symbol table permitting fast determination of name-set membership.

b) as auxiliary input to the pre-loader mentioned previously. The representation might be similar to (a), with possible additional information of interest to loader. A LITTLE job using this system would take the form



* indicates a file produced during previous use of system.

Since the pre-loader has an input only files produced by the LITTLE system, it should admit an efficient implementation. The issues here are the complexity of generating the libraries, the size of the libraries, and the interface problem with the host operating system. The pre-loader should be viewed as a "table-unifier", as is the case with most loaders.

The pre-loader could also provide another useful optimisation by loading (i.e. allocating stovage for) only those global variables which are used. This eliminates needless use of memory.

In summary, we have indicated some of the problems in using current LITTLE system for defining global variables, particularly in developing large systems. We outlined an extension to LITTLE which groups global variables into sets, and suggested an implementation approach which allows separate compilation and testing of subparts of a large system.