SETL NEWSLETTER NO. 128
A LITTLE Written Translator
from SETL to LITTLE.

Ed. Schonberg
Stephanie Brown
April 25, 1974

We examine here the feasibility of a direct SETL to LITTLE translation as the next phase of development of the SETL system. The interest of such a direct translation is two-fold:

a) by eliminating the BALMSETL link from the present system, many of the annoying semantic restrictions imposed by BALM on SETLB can be removed. In particular, the implementation of the proposed SETL namescoping scheme becomes possible. Current restrictions on the use of labels and recursive calls within iteration loops can also be removed.

b) Bypassing BALM as an intermediate language streamlines the SETL system and simplifies its maintenance. Incrementality of the language can actually be assured more easily than with a SETL-to BALM-to LITTLE system, where tables of variables, procedures and linkages have to be maintained in two places simultaneously (the BALM and SRTL environment).

The SETL to LITTLE translation scheme we want to propose is relatively simple to implement because the now complete SETL run time library offers a rich semantic environment in which to perform the required parsing and code generation. The scheme is based upon the following three modules:

a) A preparsing program which receives SETL as input and produces a parse tree as output. This program uses the LITTLE lexical scanner and the table driven topdown parser. It also creates the symbol table need for resolution of namescoping and for incrementality.

b) A series of tree walking routines which produce LITTLE code. These routines will initially be written in BALM and hand translated into SRTL-SETL.

c) The LITTLE compiler which will compile the code produced in step b).

Step a) is already implemented as a means of eliminating the SETLB preprocessor.

In this newsletter we focus on incrementality, the implementation of the SETL namescoping scheme, and the mechanisms for procedure linkage, as these questions are closely related and their resolution greatly affects future implementations.

INCREMENTALITY

Experience with the current SETLB system indicates that savefiles are extremely useful, and we feel that some form of incremental compilation ought to be provided in the next SETL system. The scheme described in newsletter No. 60 precludes this, by compiling a SETL program into a single LITTLE procedure. We propose, therefore, that every SETL procedure be compiled as a separate LITTLE procedure. Compiled procedures can then be kept in the form of a user library, and collected from it on successive runs. As LITTLE does not produce movable code, compiled procedures will be loaded together with SRTL, and their code will not appear in dynamic storage. No specific provision for saving the state of dynamic storage is contemplated here, beyond what is available at the operating system level (i.e. the cataloging of the full binary file of a program before beginning execution or upon termination).

To allow the accessing of global variables belonging to precompiled namescopes, a symbol table will be saved together with the user library. The symbol table will be used and extended during the preparse phase to resolve variable names implicit in namescoping declarations. This is described below in detail.

Some examples of namescoping are included to clarify the proposed subset of SETL namescoping.

1. If there is no name scope one is automatically generated and all procedure names are automatically made global.

```
define subl(a,b);
        :
       sub2(a+10,n):
        :
end subl;

define sub2;
        :
end sub2;

subl(<:1,2,3,4>, <10,1>);
```

The only global variables are subl and sub2 and they are known to the entire program.

2. If the user wishes variables other than procedure names to be global they must be explicitly declared within a user defined name scope.

```
scope naml;
     global gl,g2,g3;
     define subl(x,y);
             :
        a = gl;
     end subl;
     gl = <10,20>;
     g2 = 5;
     g3 = <: 1,2,3,4 > ;
     subl(5,40);
end naml;
```

3. If a variable is to be global but must be stacked when a procedure is called recursively it must be explicitly declared global and owned.

```
scope nam1;
     global Y; owns sub2(y);
     define sub1(a,b);
          :
       if y lt 1 then sub2(y,b);
          :
     end sub1;

     define sub2(x);
          :
       y = y+1;  n = y;
       sub2(n);
          :
     end sub2;

     sub2(40);
end nam1;
```

4. Several name scopes may be declared to permit additional flexibility. Note that procedures are only known within the namescope in which they are defined. Include statements are used as follows.

```
scope nam1;
     global a,b,c;  owns sub1(c);
     define sub1;
          :
     end sub1;

     define sub2;
          :
     end sub2;
end nam1;

scope nam2;
     include nam1(c,sub1);
          :
     sub1(x,y);
end nam2;
scope nam3;
     include nam1(a);
          :
end nam3;
```

NAMESCOPING

We propose to implement at first only a subset of the full namescoping features described in newsletter 76. The following seems to be a minimal self-contained set which is sufficiently interesting in itself to serve as a test of the usefulness of the full namescoping scheme. The proposed implementation is modular enough to allow for gradual inclusion of other features as need arises and/or manpower becomes available. Specifically, we propose to incorporate the following features into the system:

a) Namescope declarations will be available at a single level. Namescopes will include procedures but a procedure will not contain several namescopes. As a result, all namescopes will be known to each other, and within a namescope all procedures therein will be able to call each other.

b) Global statements will always have their default levels, i.e. that of the namescope in which they appear. Procedure names within a namescope will be automatically global and will not have to be declared so explicitly.

c) Owns statement will be provided.

d) Include statements will be provided but without any aliasing facilities. Furthermore, include declarations will not propagate their effects: if namescope a accesses some global variables of namescope b, and the declaration

include a (all);

appears within namescope c, this will only make accessible within c those variables declared global in a. This will simplify considerably the design of the preparsing phase.

e) A variable referenced by an include statement takes on its current value, i.e. the value it has within the procedure which owns it.

Features a)-e) can be incorporated without undue strain within the environment block scheme described in newsletter 60. We review briefly the organization of environment blocks, before describing the symbol table and attacking the question of procedure linkage.

ENVIRONMENT BLOCKS

To each namescope and procedure we allocate a section of stack: its base-environment block. This allocation is performed at the beginning of execution by an initialization procedure. In the case of a recursive procedure, a new environment block is created upon each recursive entry, and released on return. The environment block of each procedure stores its arguments, local variables, and some pointers needed for procedure linkage (a detailed description is given in newsletter 60). In addition the base environment contains some information which needs to appear in a fixed location (i.e. known at compile-time);

a) The number of arguments in the procedure

b) The number of local variables

c) The total size of the block

d) The invocation count of the procedure

e) A pointer to the first variable in the current environment block

f) A pointer to the entry of the binary code of the procedure

A namescope block stores the variables declared global to the namescope, and not owned by any procedure. The only additional information attached to such

a block is the number of such variables. The address of each global variable which is not owned, is known, therefore, at compile time, and can be transmitted to all namescopes that access it, via the symbol table. The address of owned variables is calculated at run time by adding a constant offset to the address of the currently active environment block of its owner. The location of this latter address (within the base environment block of the owner) is known at compile time and it will appear in the symbol table entry for that variable.

We describe now the organization of the symbol table.

SYMBOL TABLE

The symbol table serves two functions:

a) At compile time it collects all namescoping information and is used for address resolution of each variable in the program.

b) It serves as a library directory when previously compiled procedures are loaded and reinitialized.

Purpose b is served simply by storing in the symbol table the allocating information described above for namescopes and procedures. The names of namescopes, variables and procedures, are also stored in the symbol table. As they are not used at run time except to provide debugging traces, they do not have to be kept as SETL objects, but can be stored in packed form (10 characters per word on the 6600) or as LITTLE self-defining strings.

Purpose a) requires that the tree structure implicit in the namescoping scheme be appropriately encoded. It also requires the address of the actual values of a variable or procedure. The entry for any item in the symbol table will contain, therefore

a) The index in the symbol table of the item which brackets it

b) an index indicating the order of appearance within that bracket.

For a global variable or a procedure, the bracket is the namescope in which it appears. For an owned variable, it is the procedure which owns it. The same can be applied to embedded namescopes, when implemented.

The first step of the preparsing phase collects all namescoping declarations and gathers all the information needed for a) and b) above. It is then easy to calculate the address of each environment block to be allocated. From this the address of each global variable is obtained and stored into its symbol table entry (as a stack index). For owned variables, it is a location containing the address of the current environment block of the owner which has to be stored (the offset was obtained in the previous step).

In the case of procedures the symbol table entry is its value itself, in the form of a SETL object in the correct SRTL format. Besides containing the index of its bracketting namescope, this entry contains a pointer to the base environment block.

Notice in this connection that the assignment statement:

```
x = f;
```

where f is a procedure, assigns to x the same environment block currently pointed at by f. This assignment is not equivalent to the creation of a new instance of f, but amounts only to a renaming. This departure from strict value semantics in the case of procedures seems unimportant. If the need arises to create several instances of a given procedure, this can be achieved by an explicit call to the COPY function and to the environment block allocation procedure.

PROCEDURE LINKAGE

PROCEDURE LINKAGE

Once the SETL program is compiled, it has to be run as any other LITTLE program. Procedure START will be executed first, it will call upon the block allocation procedure, and then call the main procedure in the SETL program. It is understood that one namescope will contain executable code not bracketted within a procedure. The compiler will assign to it a standard name, say MAIN, and START will contain the statement CALL MAIN; .

During execution of the SETL program every function retrieval has to be resolved to determine whether it corresponds to an indexed retrieval on a SETL object, or to a procedure call. If the latter, then the addresses of the arguments have to be saved so that delayed value returns can take place. The following scheme suggests itself:

a) At compile time it is possible to evaluate those procedure arguments which can be value-receiving (for the first version of the system, this will mean variables only). Then code is emitted that will stack the addresses of these variables before the procedure is called. If an argument is not value receiving, om. will be stacked instead.

b) On return, a procedure retvals will scan the list of addresses to reassign those arguments that are value receiving. It is possible to determine at compile time whether a procedure actually modifies its argument. If it doesn't the stacking of addresses and the call to retvals can be bypassed. Because of this possible optimization, the addresses will not be placed in the environment block of the procedure itself.

c) The simplest, although not the most efficient, way of extending the scheme to general value receiving expressions, is to expand them at the SETL level to list explicitly the temporaries generated and the sequence of retrievals and storages. A more efficient scheme might make use of dope vectors. We will postpone decision on this point until later.