

### Introduction

This newsletter discusses the possibility of compiling SETL programs into LITTLE code, without using the "SETL machine" approach. The motivation for this is run-time efficiency on conventional (non-microprogrammable) machines, while maintaining portability.

A primary goal is to produce, by early 1972, the first version of a SETL compiler capable of compiling a preliminary SETL language, but probably without optimizations. The implementation in early 1972 should be able to handle most SETL features, such as recursive procedures, arbitrarily large integers, the compile statement, SETL name-scoping rules, and SETL rules for matching arguments to formal parameters.

### Bootstrapping

An executable SETL-to-LITTLE compiler can be produced as follows:

1. Code the SETL-to-LITTLE compiler in BALM-SETL.
2. Code an equivalent version of (1) in pure SETL.
3. By compiling (2) with (1), generate a SETL-to-LITTLE compiler coded in LITTLE.

At this point, we have a self-compiling SETL compiler which can be executed in a non-interpretative manner. BALM is no longer needed.

The LITTLE code which is generated from SETL source programs will have a high density of calls to SETL "library" functions, e.g., PLUS. A sufficient number of these to execute the compiler itself must initially be coded in LITTLE. Functions not used by the compiler (e.g., "pow", "npow", possibly the

equality test for sets, etc.) could be coded in SETL, but doing so might only be practical if SETL allows separately compiled modules. After the compiler is first running, all of the library functions could be coded in SETL if desired.

The resulting compiler may be transported to new machines by transporting LITTLE.

### Data encoding

In this implementation, space for almost all data is allocated from a single storage pool. Two forms of storage allocation from this pool are provided. In one form, storage is allocated and explicitly freed in a stack-like manner. In the other form, storage is not explicitly freed. Instead, a compacting garbage collector is invoked when necessary.

The two kinds of storage allocation cause blocks (of arbitrary length) to be obtained from opposite ends of the pool. These ends are referred to as the "stack" area and the "heap" area. The garbage collector is invoked when a storage request would cause the stack and the heap to overlap. The garbage collector compacts only the heap area, but it uses the stack area to determine which words in the heap are in use.

For compaction, all data in the heap is assumed to be relocatable without the aid of an explicit relocation directory. The handling of the compile statement has not yet been addressed.

All data items have a "root" which would generally occupy a single computer word, and which contains the data item's type and either its value or a pointer to its value. [Note: the "copy problem" of SETL has not been considered. The formats being described will probably be expanded to include information such as a reference count.]

Thus, all items can be classified as either "short" or "long". The long form is arbitrarily long, being limited only by available main storage. The long form is used for large

integers, reals, long strings, all tuples, the sub-structures of sets, and procedures created by compile.

Figure 1 depicts an encoding of SETL data items. The item roots are assumed to be 60 bits long for the CDC 6600, and 32 bits long for the IBM System/360. Five bits suffice for the "type" field, but it is suggested that six be used on the 6600, and eight be used on the System/360, with the high order one or three bits always zero. The "value" field (54/24 bits) holds the item's value if it will fit, and otherwise it contains an integer which is used as a word-index into the storage pool.

For efficiency, it is suggested that the implementation be "locked in" to a particular value-encoding of the type field. The encoding shown permits the implementation routines to do an indexed branch on the type field, it permits a test of a single bit to determine whether or not the "value" field is a pointer, and items of similar types are grouped together.

A type code of zero is used for short integers in the hope that the optimizing compiler may in some cases be able to generate in-line code without field extractions (e.g., for loop control in  $(1 \leq i \leq 10)$ , and when the SETL data strategy elaborations are used).

Type codes from 20 on up are intended for use by the various representations of sets which may be implemented in the future.

It is possible to represent the hashed set in a way that closely resembles a tuple. This point should be addressed when the garbage collector is designed.

On the 6600, the short form of an integer would probably be used only if the integer requires 48 bits or less (47 magnitude plus a sign). The short form is then compatible with existing software and permits simple multiply and divide operations. Six bits in the word are never used. On the System/360, the short form would probably be used for up to 24 bits.

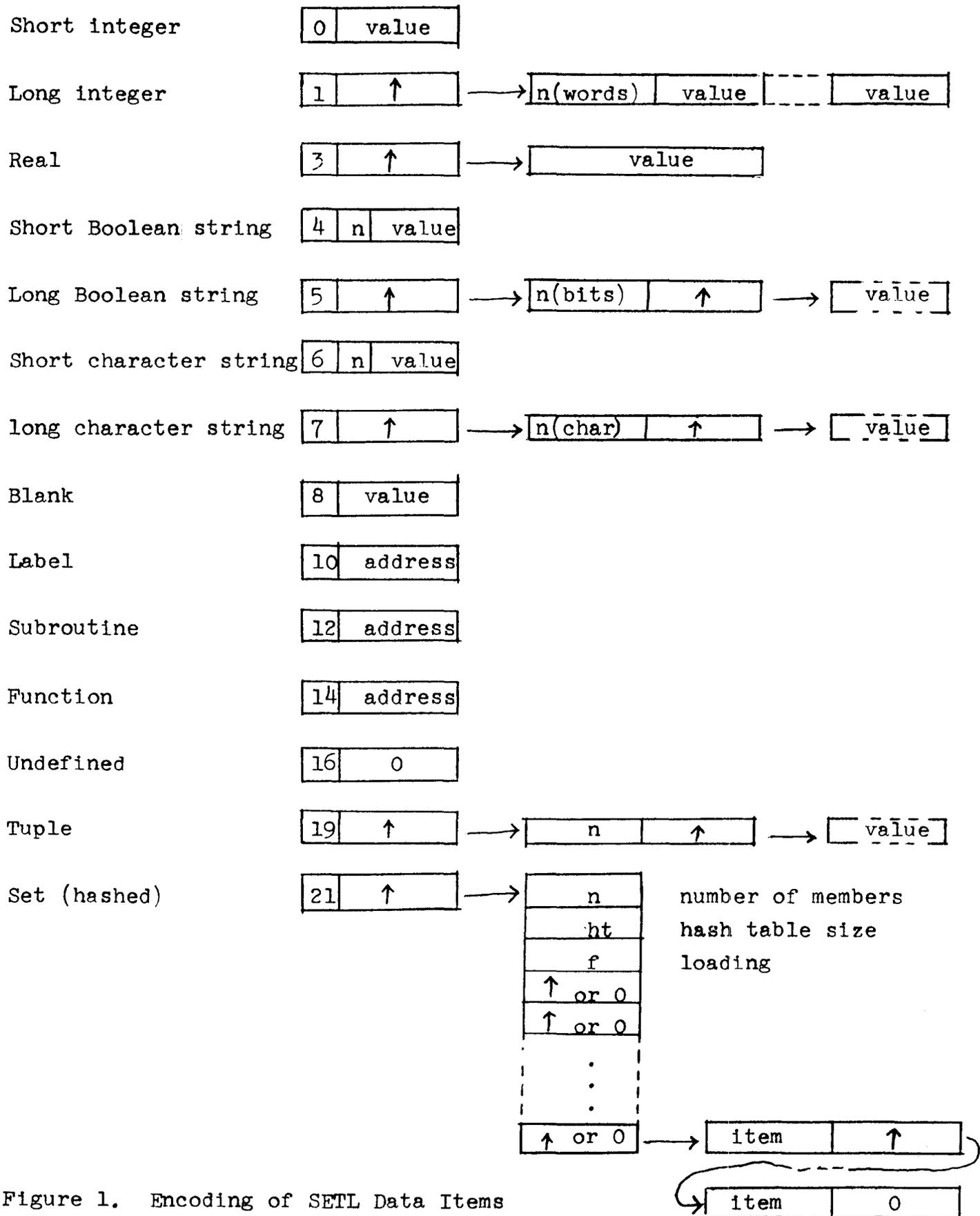


Figure 1. Encoding of SETL Data Items

The long form of an integer permits the representation of any integer that will fit in main storage. (It is assumed that the computer's word length exceeds the number of bits required for addressing, by five. Otherwise, multiple words must be used for some of the items being discussed.) On the 6600, both the length of the long integer and its starting location could be packed into the root word, but this is not recommended because it would cause difficulties in transferring SETL to a variety of machines.

It is suggested that long integers be stored in the "forward" direction (most significant word first) because this is conceptually the simplest and it is compatible with the way strings would probably be stored (consider "integer as bstring"). It is probably debatable whether forwards or backwards leads to the simplest and fastest SETL programs (forwards is best for comparison and division, and backwards is best for addition, subtraction, multiplication, and reducing the size of the integer when possible).

For short Boolean strings, six bits are required for "n" (the length of the string) on the 6600, leaving 48 bits for the string itself. On the System/360, it is convenient to use eight bits for "n", leaving 16 bits for the string. The string should be right-justified in its field, to facilitate left-padding with zeros for and and or operations on different length strings (this applies to both short and long Boolean strings).

For uniformity, it is suggested that the short character string also have six (6600) or eight (System/360) bits for the length field. This leaves room for six seven-bit characters on the 6600, and for two eight-bit characters on the System/360. It is suggested that the characters be left justified for both short and long character strings, as this is preferable on

computers with variable field length instructions.

Long strings (and all tuples) have an extra level of indirection involved. This is to permit subpart extraction without copying when the context allows it. The word-pair  $\langle n, \text{pointer} \rangle$  may be expanded to include a reference count, word count of "value", etc.

It is probably not necessary to have a long form of the blank atom. On the System/360, 16,777,216 ( $2^{24}$ ) values of newat can be generated. Although an algorithm could be written that would exhaust these values in a minute or so, it does seem reasonably safe.

The value of a label, subroutine, or function variable is simply the machine address of the machine code corresponding to the label, subroutine, or function name. (SETL does not alter the run-time stack as a result of a "go to".)

A tuple is stored in contiguous words as shown. Each component of the tuple is in the form of a root word. Thus, the components of a tuple can be directly addressed through indexing (even though the components may be rather complex data structures).

The representation of a hashed set closely follows that described in newsletter 49. The members of a set are stored as simple one-way lists, in no particular order. Each item in the list has the format of a root word. The manner of storing tuples in sets (for fast functional evaluation) is not yet addressed, but it may differ from that of newsletter 49.

In this implementation, it is suggested that hash codes be computed on demand, rather than always computed and saved. This saves space and also the overhead of computing hash codes when they are never used. This, incidentally, will be more significant when non-hashed sets are introduced.

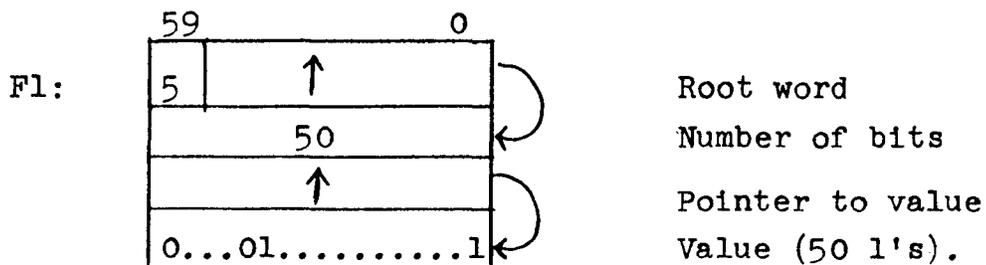
Fixed and dynamic storage

In this implementation, the SETL compiler assigns constants and certain external variables to fixed storage. The root words of these items are referred to by the compiler-generated names F0, F1, F2,....

Variables other than certain external variables are stored in "dynamic" storage (stack or heap), and are referenced by a term of the form D(p+k). Here "p" is an integer variable designating a block of storage which has been allocated from array D, and k is a constant giving the offset of the variable. For example, if a SETL routine has a local variable x and a variable y with the declaration "sub external y", then x might be referenced as D(P+5) and y as D(P1+6). Here it is assumed that P points to the currently active block (top of the stack), and P1 points to the block associated with the most recent invocation of routine "sub".

For a constant such as the SETL 101b, the compiler might generate the LITTLE statement DATA F0=B04030...05 for the 6600. The generation of LITTLE code for long constants is more complicated because of the "pointer" field. One would like to be able to supply the pointer field without using executable code (for compactness of the object program).

For example, a string of 50 1-bits should be compiled into fixed storage as shown below (for the 6600):



To specify this in LITTLE would require a significant increase in the capabilities of the DATA statement, so that the compiled code could be something like:

```
SIZE F1(60).,  
DIMS F1(4).,  
DATA F1=5x255 + A(F1(1))-A(D(1))+2,50,  
      A(F1(1))-A(D(1))+4,B000377...7.,
```

where A(x) denotes the word address of x. Then, when F1(1) bits 17-0 are used as an index into the "D" array, the item actually located is F1(2) (i.e., the "50" above). This problem is not yet resolved.

Variables declared as "blank external" are assigned to fixed storage. If the statement "proc external x,y,..." occurs in procedure "proc", then x,y,... are also assigned to fixed storage. Otherwise, procedure "proc" must be examined by the compiler to determine the storage class for x,y,....

### Overall program structure

This implementation assumes an overall program structure that is slightly different from that in the notes (p.72). We take the view that all SETL procedures must begin with either a "define" or a "definef" statement. The total program is assumed to be contained in a procedure that is entirely compiler-supplied. This outermost procedure is known to be non-recursive. A variable declared as "blank external" is taken to refer to the compiler-supplied main procedure, and hence it may be placed in fixed storage. Such a variable also takes on the name scope of the compiler-supplied main procedure.

It is believed that this approach leads to somewhat simpler procedure prologues and epilogues. For example, the "return" statement always compiles in the same way. It also seems to slightly increase the power of the "external" statement, and permits particularly efficient references to "blank external"

variables.

If the first statement is not "define", then the compiler may of course supply a statement such as "define noname". In case the corresponding "end" statement is also missing, it is suggested that one be supplied at the very end of the total program, rather than before the first supplied "define" statement. (Name scoping is affected by the choice. The reason for suggesting that it be supplied at the end is that the compiler cannot know that an "end" statement is missing until it has read the entire program, so it may be simplest to put it at the end.)

#### External statement; stacking of variables

In this newsletter, no particular name-scoping rules for SETL are assumed. However, to seriously contemplate the compilation of SETL programs, it does seem to be necessary to have detailed knowledge of the relation between the "external" statement and the stacking of variables.

The effect of the "local" statement (see newsletter 34, page 7) is identical to an appropriate "external" statement. For example, the statement "sub local a,b,c,..." is equivalent to "sub external z,y,x,..." where z,y,x... is a list of all non-external variables in the procedure containing the "local" statement. Thus, the "local" statement will not be discussed further.

The details given here on the meaning of the "external" statement are intended to be an elaboration of the material in the notes. The gaps are filled in with an implementation in mind, but no changes are intended.

The basic requirement is that the statement "suba external x,y,z" defines x,y,z "as having references which are the same as those which identical names occurring in the subroutine suba would have" (notes, p.71).

Thus, x,y,z have the same name scope as the x,y,z in "suba", and, at any point in time, have the same values.

If "suba" is entered recursively, then x,y, and z are stacked and new instances are created, having the value  (assume that x, y, and z are local to "suba"). The names x,y, and z in the procedure containing the "external" statement should then refer to the new instances.

If "suba" is not active at all, then the x, y, and z are taken to refer to a "base level" of the variables in "suba", which will be used when "suba" is entered. This point is unconventional (or at any rate has no counterpart in Algol or PL/I), and is somewhat anomalous, but nevertheless it does allow a very natural and entirely flexible use of the "external" statement in a non-recursive environment. It is also easily implemented.

With this interpretation of the "external" statement, the stacking of local variables may be implemented as follows.

Before beginning execution of a SETL program, a compiler-supplied main routine creates a storage block for the local variables for each procedure, and initializes a set of pointers to point to these blocks. There is one block, and one pointer, for each SETL procedure. "Local variable" here means all variables mentioned in the SETL procedure except those that are declared "external", regardless of the procedure name (if any) on the "external" statements. It is assumed that the "local" statement, if any, has been converted into an appropriate "external" statement.

Associated with each procedure is an "invocation count". The compiler-supplied main routine initializes all invocation counts to zero.

The prologue of each procedure includes a test of its invocation count. If zero, the procedure uses the pre-allocated, or "base" instance of its local variables. If non-zero (recursive entry), the procedure stacks the current instance of its local variables, and allocates a new instance. In any event, the invocation count is then incremented.

The epilogue (return) of each procedure decrements the procedure's invocation count. If the result is zero, the procedure simply returns. If the result is non-zero, the procedure pops up the older instance of its local variables, and then returns.

In a non-recursive environment, no storage allocation and freeing is necessary. This speed advantage is attained, however, at the expense that space for all local variables is allocated all of the time.

The compile-time rules for resolving the references in the statement "sub external x" are summarized below.

1. If this statement occurs in subroutine "sub", then assign "x" to fixed storage. Use normal name-scoping rules for "x".

2. If "sub" is omitted, then assign "x" to fixed storage. For name-scoping purposes, regard "x" as being internal to the outermost (compiler-supplied) subroutine.

3. In all other cases ("sub" is the name of another subroutine) consider this "x" to be the same as the "x" in subroutine "sub".

- a. If "sub" contains no occurrence of "x", then treat it as if "sub" contained the statement "x=x" (i.e., use (c) below).
- b. If "sub" contains a statement such as "sub external x", "external x", or "further external x" then apply (1), (2), or (3) respectively ("x" has acquired the scope of the "x" in "sub").
- c. In other cases ("x" is internal to "sub"; possibly a formal parameter) then assign "x" to the dynamic storage location of the "x" in "sub".



Compiled code for a simple SETL procedure

The appendix shows a simple nonsensical program coded in SETL, as an example of what sort of LITTLE compiled code might be possible.

The program consists of a procedure "illustrate", which has an internal function-procedure "f". As there are two procedures, the compiler generates two integer variables I1 and I2 to refer to their invocation counts, and two integer variables P1 and P2 to refer to the top of their "environment" stacks. The variable P is used to refer to the currently executing procedure's environment.

If a variable x is local to a procedure Pi, then the compiled code for Procedure Pi could reference x by either D(Pi+k) or D(P+k). Though the variable P may seem unnecessary, it is needed for garbage collection, is convenient to use in connection with procedure linkage, and will probably be found useful as a debugging aid.

The LITTLE code shown in the appendix is based on the format of an "environment" block shown below.

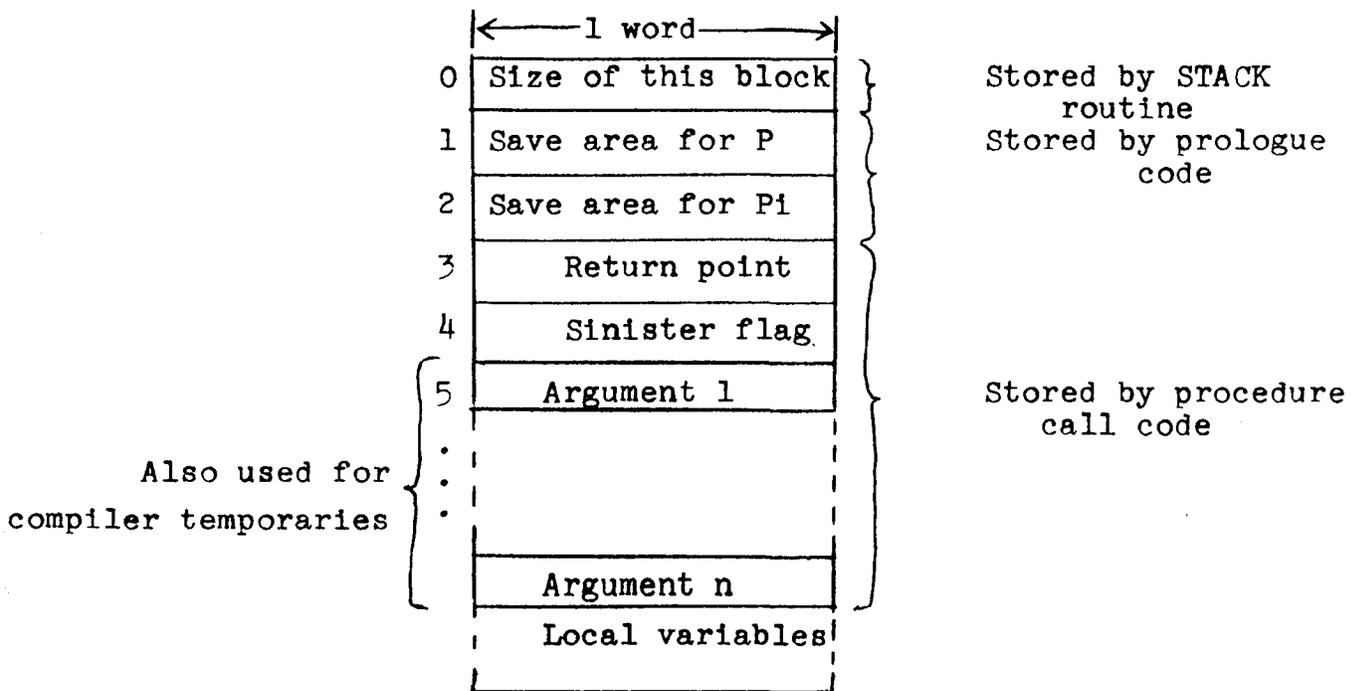


Figure 2. Layout of an Environment Block

The first block of code generated by the compiler is a copy of the complete run-time library (PLUS routine, etc.). In essence, this is supplied in source (LITTLE) form. It includes declarations of variables such as P and array D that are used both in the library and in compiler-generated code. Thus these "common" variables need not be passed explicitly as arguments. If a library routine "inadvertently" uses a non-shared name such as I1, however, then no harm is done because these are re-declared in the compiled code.

Since the run-time library may eventually involve some 10,000 LITTLE statements, it will actually be inserted in some sort of compressed form. This might be simply a matter of eliminating comments and redundant blanks, or some partially compiled form might be used, or possibly only its symbol table need be inserted.

The entire SETL source program is compiled into a single LITTLE subroutine called CMPCODE (compiled code). Thus all compiler-generated names (such as P1, P2, etc.) are known everywhere.

The statement DATA I1=0/I2=0 initializes the invocation counts of the two compiled procedures "illustrate" and "f".

The next statement, CALL DINIT, effects any initialization that might be required for dynamic storage allocation. This routine might allocate a portion of the dynamic storage array D for the garbage collector's bit table(s), initialize pointers to the first words of the stack and the heap, etc.

Incidentally, the exact manner of setting the size of array D has not been worked out. It is assumed that a value is passed as a parameter to the compiler, and the compiler somehow communicates it to the run-time library block of code (which contains the DIMS D statement).

The next three statements obtain and initialize a five-word block which serves as the "environment" block for the compiler-supplied main routine. Only the first three words are

initialized at this point. Note that the statement "P=STACK(5)" is used to request five contiguous words from the "stack" portion of dynamic storage. The value of the STACK library function is an integer which points to the block allocated.

The STACK routine initializes all words in the allocated area to  $\perp$ . This is done so that a procedure's local variables will all be equal to  $\perp$  when the procedure is entered for the first time or is entered recursively.

The next four statements allocate the base level blocks for the two SETL procedures. The "Pi save areas" are initialized to zero. The size of an environment block is  $5+t+v$  words, where  $t$  is the maximum number of "compiler temporaries" needed at any point in time, and  $v$  is the number of local variables in the procedure. Arguments are passed in the compiler temporary area, so  $t$  must be at least as large as the largest number of arguments passed to any called procedure.

The next statement sets PD to point to the first available word in the dynamic stack area (i.e., the stack area above the base allocations). This pointer is used to control allocation and freeing of environment blocks when recursion occurs. It is initialized by DINIT, is increased by STACK, and is decreased as part of the compiled code for the "return" statement.

The next four LITTLE statements provide information needed by the garbage collector. The garbage collector receives an input from the vector GLIST, and to explain how it is used it is necessary to look ahead and consider the compiled code for procedure prologues.

When the  $i$ 'th SETL procedure is invoked recursively, it allocates a new environment block and stores the old value of P<sub>i</sub> in the "Pi save location" of the new block. It then sets P<sub>i</sub> to point to the new block. The procedure also stores the old value of P in the "P save location" of the new block, and sets P to point to the new block. This has the effect of making the new block the head of two one-way chains. If one views the blocks

as being chained by the P save location, then the chain acts as a stack that reflects the dynamic nesting of procedure calls. If one views the blocks as being chained by the Pi save location, then the blocks form a set of stacks, with each stack reflecting the depth of recursion of a corresponding procedure. The P-stack and the Pi-stacks end with zero pointers.

The garbage collector is invoked from STACK. It "knows" the current value of P. By starting with the block pointed to by P, and tracing through the blocks using the P save location, the garbage collector can locate the environment blocks of all active procedures (an "active" procedure is one that has been invoked more times than it has returned). In addition, the garbage collector has the values of pointers to all the base level environment blocks. These values were given to it by the vector GLIST and its dimension GLISTSZ. By scanning through all these environment blocks, the garbage collector can locate all currently used words in the heap.

It is assumed that the garbage collector "knows" the formats of the environment blocks and all SETL data items. Note that the size of each environment block is stored in the block.

The garbage collection algorithm will not be trivial and its design has not yet been studied to any depth. A major point is that to move an item (for compaction), the garbage collector must be able to locate all pointers to the item, hopefully in an efficient manner. Thus garbage collection might be related to the copying problem.

Getting back to the LITTLE code, the next three statements cause a transfer to the main SETL routine, "illustrate". This is compiled as a normal procedure call to a subroutine that has no arguments. The form of the compiled code for a procedure call is:

```
D(P+3) = i (i refers to Li below)
D(P+4) = 0 or 1 (sinister flag)
code to evaluate argument 1
D(P+5) = root word of argument 1
.
.
.
code to evaluate argument n
D(P+4+n) = root word of argument n
GO TO label constant
Li CONTINUE
D(P+k) = RESULT (for dexter function calls only)
```

The first statement above stores the return point, in effect. The next statement stores a dexter/sinister indication (the compilation of sinister calls has not yet been studied to any extent). The next n statements evaluate and store the arguments. This is followed by a GO TO referring to the procedure. The GO TO would be replaced by something more complicated in the case of a variable procedure name. The CONTINUE statement serves only as a place to attach the compiler-generated label Li.

The call to the main SETL routine is followed by a RETURN which terminates execution. This is followed by a "switch" which is used to accomplish a transfer to a label variable. In an unoptimized compiler, this switch includes a list of all the statement labels in the compiled code, both programmer-supplied and compiler-generated. The effect of "variable=label constant;...;GO TO variable" is accomplished by "variable=label number;...;S=variable;GO TO SWITCH".

Next the compiled code for program "illustrate" begins. The prologue includes a test of the procedure's invocation count and the stacking of the procedure's environment if it is entered recursively, as has been explained.

The remainder of the compiled code can probably be followed without further comment, but a few remarks will be made about the passing of arguments to procedures.

It has been assumed that the library routines (PLUS, LESS, etc.) always return with a unique copy of the result. The value of these routines and of a compiled function is a "root word", which may point to a structure in the storage heap. The point is that the structure in the heap is not shared by other root words.

For a procedure call such as  $f(1,s,x+1)$ , the arguments are always regarded as expressions to be evaluated in the calling routine. Thus, in the absence of optimization, what is passed to  $f$  in this example is a copy of the constant 1, a copy of  $s$ , whatever  $s$  may be, and a unique copy of the result of the calculation " $x+1$ ". This is why the compiled code for " $a=f(1);$ " passes "1" by means of the statement  $D(P+5)=COPY(F0)$ . The non-optimizing compiler allows for the possibility that "1" is a long integer whose value may be altered by  $f$ .

The type of linkage used here, and suggested for SETL subroutines and functions, is call by value with delayed argument return (for all data types). This type of linkage is conceptually simple and it allows the compiled code for referencing formal parameters in the body of a procedure to be identical to the compiled code for referencing other local variables. This, in turn, will probably enhance certain optimization opportunities for SETL. For example, common expressions involving formal parameters and external variables can be factored. That they cannot always be factored for other types of linkages is illustrated below, where the common expression is " $y+1$ ".

Case 1. Shows "call by  
value" is required.

```
sub(a,a);
.
.
define sub(x,y);
x=y+1;
g=y+1;
.
.
.
```

Case 2. Shows "delayed  
argument return" is also  
required.

```
define main;
.
.
sub(y);
.
define sub(x);
main external y;
x=y+1;
g=y+1;
```

Concluding remarks

Based on this look at LITTLE as a target language for SETL, it appears that the target language should compile subscripting references as efficiently as possible. A reference to  $D(P+i)$ , where  $i$  is a constant, should compile without an explicit addition for the "+1". It would also be helpful if a reference to  $P(i)$  were just as efficient as a reference to a non-subscripted variable. Then the variables  $P_1, P_2, \dots$  would instead be  $P(1), P(2), \dots$ , which form is more convenient for the garbage collector.

The addition of an EQUIVALENCE statement might accomplish this, as the compiler could generate the statement:

EQUIVALENCE ( $D_1, D(1)$ ), ( $D_2, D(2)$ ), ( $D_3, D(3)$ ), ..., and then reference  $D(P+1), D(P+2)$ , etc., as  $D_2(P), D_3(P)$ , etc.

It will also help if LITTLE is capable of assigning a ubiquitous variable such as  $P$  to a register, particularly since it is most often used as a subscript. For our purposes, it would be adequate if it were possible to explicitly instruct LITTLE to keep a variable in a register, by supplying a statement such as REGISTER  $P$ . This type of problem is compounded on the System/360,

as on that machine one really wants to keep 4\*P in a register.

A better approach than indexing for referencing dynamic storage would be to add to LITTLE a capability similar to the STRUCTURE statement of the IBM System/360 FORTRAN IV (H) compiler (reference IBM form Y28-6642 Appendix J: Facilities Used by the Compiler). This feature is similar to the PL/I BASED attribute, but simpler. It is not necessary to introduce the "address" as a data type. Integers suffice, and this in turn means that arithmetic can be done on the address values, which is often desirable. It is a simple matter to generate efficient code for referencing structured variables. There is no addition as indexing normally implies, and there is no multiplication by four on the System/360.

It will be helpful if LITTLE eventually can compile in-line code for simple operations on floating point numbers and variable length strings.

Another feature that would be of value is the ability to handle procedure calls with a variable number of arguments (with no predetermined limit). This would be used for {a,b,c,...} and <a,b,c,...>, for example.

It may be more reasonable to use FORTRAN as a target language, rather than LITTLE. Some of the pros and cons are listed below.

Advantages of FORTRAN over LITTLE:

1. FORTRAN is already running and debugged on the machines likely to be of interest to SETL.
2. FORTRAN is more familiar than LITTLE, which simplifies local SETL installation and maintenance.
3. It would probably be easier to transport SETL, as hand-coding a few field extraction routines may be easier than defining a complete "machine block".
4. Most FORTRAN compilers will probably be quite fast in compiling and will probably generate good code (particularly of value for subscripting).

5. Miscellaneous features such as the assigned GO TO, EQUIVALENCE, and ENTRY (of FORTRAN IV, which would be heavily used in library routines) will no doubt be found helpful.

6. Some FORTRAN compilers already have a fair amount of global optimization. If we could rely on this, then we would be free to concentrate on the optimization problems that are more or less unique to SETL, such as the copying problem and the various representations of sets.

Advantages of LITTLE over FORTRAN:

1. Increased machine independence.
2. In-line field extractions.
3. Local control of the design of the language (of course, this is also a burden, but it's nice to know that we can add the STRUCTURE statement if we really need it).
4. Probable gain in efficiency in subroutine linking (due largely to the name-scoping of LITTLE).

APPENDIX

```
define illustrate;  
a = f(1);  
return;  
    definef f(x);  
    external e;  
    e = x+1;  
    a = 2;  
    y = f(3);  
    return 4;  
    end f;  
end illustrate;
```

Illustrative SETL Program

|   | Fixed Storage | Main Routine | Subroutine "Illustrate" | Function "f" |               |
|---|---------------|--------------|-------------------------|--------------|---------------|
| 0 | 1             | 5            | 7                       | 9            | size          |
| 1 | e             | 0            |                         |              | P save        |
| 2 | 2             | 0            |                         |              | Pi save       |
| 3 | 3             | 1(L1)        |                         |              | return point  |
| 4 | 4             | 0            |                         |              | sinister flag |
| 5 |               |              | arg(1)                  | arg(3)       |               |
| 6 |               |              | a                       | x            |               |
| 7 |               |              |                         | a            |               |
| 8 |               |              |                         | y            |               |

Data Storage (fixed and stack) for  
Illustrative SETL Program

LITTLE Code Resulting from Compilation of the  
Illustrative SETL Program

```
COMM      SETL LIBRARY.      CMND.,
          The complete library (PLUS, LESS, etc.) is inserted here.
          SUBR      CMPCODE.,
          Size declarations for non-"common" variables.
COMM      COMPILER-SUPPLIED MAIN ROUTINE.      CMND.,
          DATA I1=0/I2=0.,      Preset invocation counts.

          CALL DINIT.,      Initialize dynamic storage.

          P=STACK(5)..,      Allocate 5-word environment block.
          D(P+1)=0.,      Clear "P save" location.
          D(P+2)=0.,      Clear "Pi save" location.

          P1=STACK(7)..,      Base level allocation.
          D(P1+2)=0.,      Clear Pi save location.
          P2=STACK(9)..,      Base level allocation.
          D(P2+2)=0.,      Clear Pi save location.
          PD=P2+9.,      Set pointer to dynamic stack area.
          DIMS GLIST(2)..,      Garbage collection list.
          GLIST(1)=P1.,      Tell garbage collector where
          GLIST(2)=P2.,      base level allocations are.
          DATA GLISTSZ=2.,

COMM      CALL MAIN SETL PROCEDURE.      CMND.,
          D(P+3)=1.,      Store return point (L1).
          D(P+4)=0.,      Turn off sinister flag.
          GO TO L2.,      Go to main SETL routine.
/L1/      CONTINUE.,
          RETURN.,      To operating system.
/SWITCH/  GOBY S(L1,L2,L3,L4,L5,L6,L7,L8,L9)..
```

```
COMM      define illustrate;  CMND.,
COMM      PROLOGUE FOR "illustrate".  CMND.,
/L2/      IF (I1 .EQ. 0) GO TO L3.,  Branch if not a recursive entry.
          TEMP=STACK(7).,  Allocate an environment block.
          D(TEMP+2)=P1.,  Save P1.
          P1=TEMP.,  Set new value of P1.
/L3/      I1=I1+1.,  Increment invocation count.
          D(P1+1)=P.,  Save P.
          P=P1.,  Set new value of P.
COMM BODY OF MAIN SETL ROUTINE.  CMND.,
COMMa     a=f(1);  CMND.,
          D(P+3)=4.,  Store return point (L4).
          D(P+4)=0.,  Turn off sinister flag.
          D(P+5)=COPY(F0).,  Set argument for f.
          GO TO L6.,  Go to f.
/L4/      CONTINUE.,
          D(P+6)=RESULT.,  Move result to "a".
COMM      return;  CMND.,
          P=D(P+1).,  Pop up caller's environment.
          I1=I1-1.,  Decrement invocation count.
          IF (I1 .EQ. 0) GO TO L5.,  Branch if non-recursive entry.
          P1=D(P1+2).,  Restore P1.
          PD=PD-7.,  Free environment block.
/L5/      S=D(P+3).,  Get return point index.
          GO TO SWITCH.,  Return.

COMM      define f(x);  CMND.,
COMM      Prologue for routine f.  CMND.,
/L6/      IF (I2.EQ.0) GO TO L7.,  Branch if not a recursive entry.
          TEMP=STACK(9).,  Allocate an environment block.
          D(TEMP+2)=P2.,  Save P2.
          P2=TEMP.,  Set new value of P2.
```

```
/L7/      I2=I2+1.,          Increment invocation count.
          D(P2+6)=COPY(D(P+5))., Get argument x.
          D(P2+1)=P.,      Save P.
          P=P2.,          Set new value of P.
COMM      Body of procedure "f".  CMND.,
COMM      external e;  CMND.,
COMM      e=x+1;  CMND.,
          F1=PLUS(D(P+6),F0)..,
COMM      a=2;  CMND.,
          D(P+7)=COPY(F2)..,  or "D(P1+6)=COPY(F2)"
COMM      y=f(3);  CMND.,
          D(P+3)=8.,          Store return point (L8).
          D(P+4)=0.,          Turn off sinister flag.
          D(P+5)=COPY(F3)..,  Store argument (3).
          GO TO L6.,          Go to procedure "f".

/L8/      CONTINUE.,
          D(P+8)=RESULT.,    Move result to "y".
COMM      return 4;  CMND.,
          RESULT=COPY(F4)..,  Set value of function.
          P=D(P+1)..,          Restore caller's environment.
          D(P+5)=D(P2+6)..,    Pass back parameter x.
          I2=I2-1.,          Decrement invocation count.
          IF (I2 .EQ. 0) GO TO L9., Branch if non-recursive entry.
          P2=D(P2+2)..,          Restore P2.
          PD=PD-9.,          Free environment block.

/L9/      S=D(P+3)..,          Get return point index.
          GO TO SWITCH.,      Return.
COMM      end f;  CMND.,
COMM      end illustrate;  CMND.,
          END.,
```