SETL Newsletter Number 60 November 4, 1971

SETL Compiled Code: Calls to SETL Procedures

H. S. Warren

General

This newsletter presents a proposed linkage mechanism to be used to invoke a SETL subroutine or function. Newsletter 53 is followed in a very general way, but much of the detail is changed.

The type of linkage proposed is call by value with delayed argument return. Although this type of linkage is logically one of the simplest, there are two points that should be brought to light. Both have to do with the altering of formal parameters by the called procedure: (1) what value is an argument (i.e., actual parameter) to receive if it occurs more than once in an argument list, and (2) what is to be the effect if the called procedure changes a variable that occurs in an argument expression?

Both cases are illustrated by:

```
sub(i,i,a(i));            define sub(x,y,z);
                          x=1;y=2;z=3; return;
                          end sub;
```

The calling sequence to be described performs the assignments in left-to-right order. Thus after the above call, i=2 and a(2)=3.

It is suggested that any expression that is valid as the left-hand side of an assignment statement be a valid value receiver in an argument list, e.g, sub(f{a},<b,c>). The following expressions would then not be valid value receivers: 1, x+y, x+0, (x), +x. Expressions such as these would be valid as arguments that normally receive values, but the value of the formal parameter upon return would be ignored.

The calling sequence to be described is, however, independent of what kinds of expressions are valid value receivers.

As many as possible of the procedure linkage steps are usually placed in the prologue of the called procedure, because this minimizes the total program size (under the assumption that the procedure may be called from several points), and because steps that are dependent upon the characteristics of the called procedure then tend to be in the called procedure, which facilitates independent compilations.

The approach taken here, however, is oriented toward enhancing the execution speed gains that can be achieved with an optimising compiler. The type of optimiser in mind is one that can detect and eliminate "dead" expressions, can remove constant expressions from loops, etc. It is assumed that the optimiser cannot trace flow paths across procedure boundaries, but it does know which items are global, and it may also have a rough idea of how the global items are used in a called procedure (e.g., live on entry, dead on entry, or not used at all).

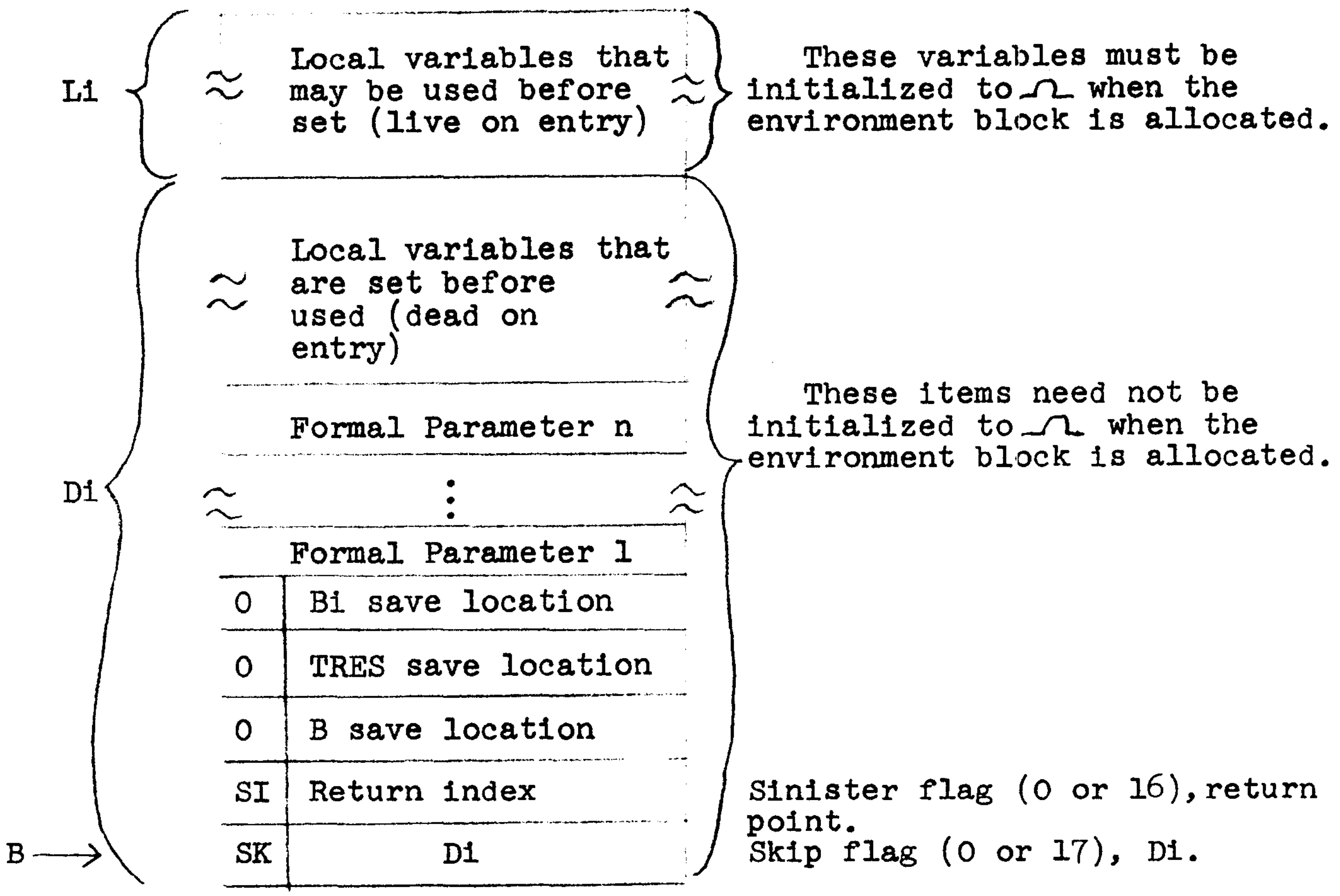
In the design described here, almost all linkage steps are placed at the point of the call. This is to allow the execution speed optimisation of calls that occur within loops, or "implied loops", such as the call to f in "$S=\{x \in A \mid f(x)\ \underline{eq}\ y\}$". The resulting calling sequence is rather large (23 + 3n LITTLE statements for a dexter recursive function call, where n is the number of arguments), but it can be trimmed down to some extent by various optimisation techniques.

Before discussing the calling sequence in detail, the format of the run-time stack will be reviewed.

Environment Blocks

Certain features of the run-time stack and its partitions, or "environment blocks", have been changed from what is described in newsletter 53. The environment blocks are not always chained together. The garbage collector's marking phase consists of a linear scan over the stack, without regard to block boundaries. Any non-SETL objects stored in the stack are coded in such a way that they look to the garbage collector like "short" items (i.e., data items that do not point to data in the heap).

The format of an environment block is shown below.


Li
Local variables that may be used before set (live on entry)
These variables must be initialized to Ω when the environment block is allocated.
Di
Local variables that are set before used (dead on entry)
Formal Parameter n
Formal Parameter 1
0 Bi save location
0 TRES save location
0 B save location
SI Return index
SK Di
B
These items need not be initialized to Ω when the environment block is allocated.
Sinister flag (0 or 16), return point.
Skip flag (0 or 17), Di.

The first word of the environment block will be explained later.

The second word contains a dexter/sinister flag and the return point. These quantities are planted in a called procedure's block by the calling procedure. This word looks to the garbage collector like either a short integer or an undefined atom.

The third word contains the value that B had just before the call. As will be seen later, code optimisation may eliminate its use. It is required, however, in the case of a procedure that calls itself.

The next two words are save areas for a "reserved" pointer, which will be explained later, and the pointer to the environment block of the called procedure that was active just before the call.

The next n words contain the root words of the arguments. Like the return point, these are planted in the called procedure's block by the calling procedure.

The remainder of the environment block contains the procedure's local variables, separated into two classes L and D.

Class L consists of those local variables that are "live on entry" to the procedure: that is, there exists a path (apparently) from the procedure's entry to a use of the variable that does not contain an intervening "set" or "definition" of the variable. These variables:

(1) must be initialized to Ω when the environment block is allocated, and

(2) must be retained in the base level environment block even when the procedure is inactive.

Point (2) refers to the fact that the garbage collector will make a linear pass over the stack, and will encounter the base level environment blocks of all procedures, whether they are active or not. In the marking phase, the garbage collector must trace through the heap for all items in class L, so that the space will not be collected.

Class D consists of those local variables that are "dead on entry" to the procedure: that is, every path from the procedure entry results in a set before a use. These variables:

(1) need not be initialized when the environment block is allocated, and

(2) need not be retained in the base level environment block when the procedure is inactive.

The first word of the environment block will be referred to as the "garbage collector skip word". This word contains a "skip flag" and a count of the number of D-variables, plus the number of formal parameters, plus 5. It is used only in base level environment blocks. The skip flag is set to zero when the procedure is active, and to 17 (an unused type code value) when the procedure is inactive. To the garbage collector, the skip word looks like a short integer when the flag is zero, and the word is then ignored. The garbage collector interprets a code of 17, however, as a signal to skip ahead in the stack by the number of words given. Thus when the procedure is inactive, the heap locations corresponding to dead-on-exit variables and the formal parameters are not marked, and they will be reclaimed.

The initial SETL compiler, which will not have any live dead analysis, might put compiler temporaries in the D area, and all other local variables in the L area. Then, when live dead analysis is added, it will probably be found that the great majority of the local variables can be assigned to the D area.

To review, the advantages of being able to assign a variable to the D area are that (1) procedure calls are faster because there is less initializing to Ω, (2) garbage collection is faster because fewer words in the stack are examined, and (3) garbage collection is more effective because the space occupied by more dead variables is reclaimed.

Reserving Stack Space

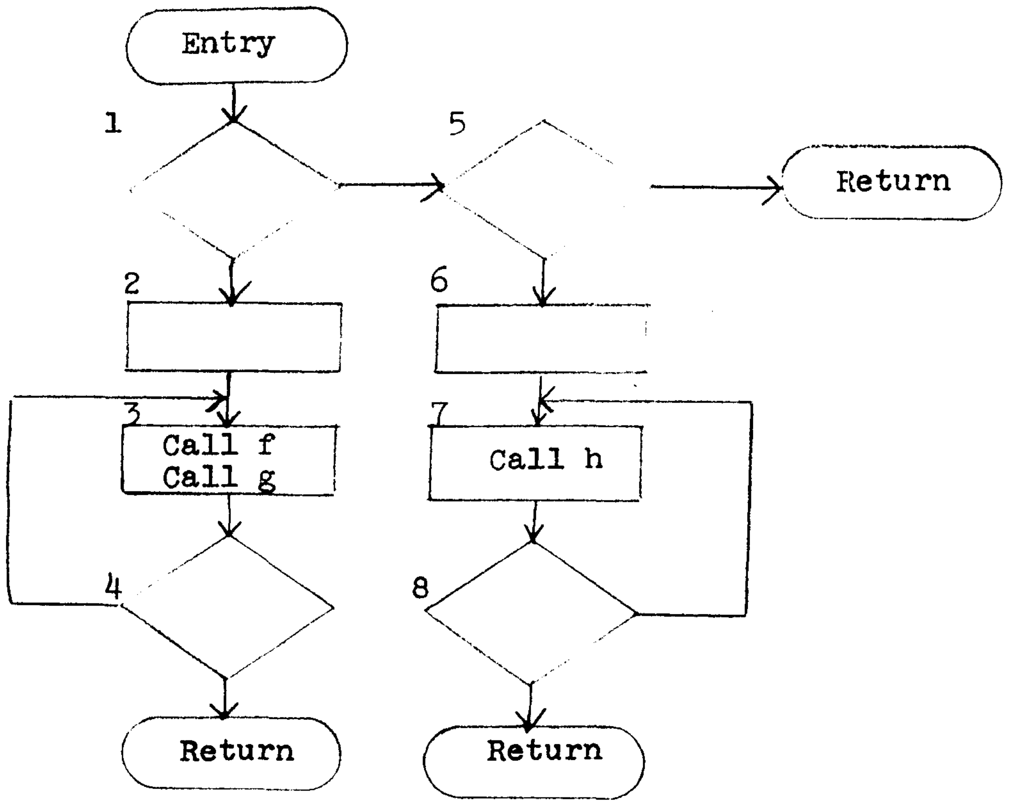
When a procedure is called recursively, space must be allocated on top of the run-time stack for its arguments, local variables, etc. Normally (or at least in PL/I), a procedure allocates its own space using code contained in its prologue. In the calling sequence to be described, however, the space-reserving step is moved back to the point of the call. This is necessary so that arguments and the return point may be planted by the calling procedure.

Furthermore, the amount of space reserved is equal to the amount required for the largest environment block of all called procedures. This is done so that the space-reserving step, which might be something like "CALL RESERVE(108).," in a typical procedure, will be exactly the same for all occurrences in the same procedure.

Finally, the space-reserving step is done before the run-time test to see if the call is recursive.

Now if the optimising compiler is informed that the RESERVE library routine has the property that in two successive calls, the second is a no-operation, then the compiler may move the RESERVE steps to more nearly optimum nodes in the program.

For example, consider the following program:



The unoptimised code would have calls to the RESERVE routine in the calling sequences to f, g, and h. The optimiser can then delete the RESERVE for the call to g, as it always follows the one for f. Then the RESERVE's can be moved from node 3 to node 2, and from node 7 to node 6. It would probably be best not to move the RESERVE's back to node 1, combining them (to save space), because then the program would be worsened for the path Entry-1-5-Return, which might very well be the most common path.

It should be pointed out that the above treatment does worsen some programs, because the RESERVE step was placed ahead of the run-time test for a recursive call. This will cause a RESERVE action in a procedure even when all calls turn out to be non-recursive.

A complication exists because it is not possible to arrive at an 'optimum' set of nodes in which to place the RESERVE steps, because this question boils down to a space-time tradeoff, in general. However, such points need not concern us here. The main point is that the RESERVE is defined in such a way that the optimiser has a great deal of freedom in dealing with it.

The RESERVE action consists of incrementing a globally known pointer TRES, if necessary, so that it is equal to T+M, where T is the pointer to the top of the stack (+1), and M is the number of words to be reserved. Then, if TRES exceeds H (the lower limit of the heap area), the garbage collector is called. That is, it consists of the steps:

```
      temp=T+M.,
      IF (temp.LE.TRES) GO TO ZZZA.,
      TRES=temp.,
      IF (TRES.LE.H) GO TO ZZZA.,
      CALL GARBCOL.,
/ZZZA/CONTINUE.,
```

Compiled Code

Pages 12 to 13 show the code for a calling sequence, a procedure prologue, and a procedure epilogue.

It is assumed that the compiler numbers the procedures it compiles 1, 2, 3,..., in an arbitrary order. The compiler generates several globally known parameters assosiated with each procedure. These are:

Ii	Variable	Invocation count (only required for recursive procedures).
Bi	Variable	Pointer to the i-th procedure's currently active environment block, or to its "base level" environment block when the procedure is inactive (this parameter was called Pi in newsletter 53).
Ni	Macro	Number of formal parameters.
Di	Macro	Number of local variables that are dead on entry to the procedure, +Ni+5.
Li	Macro	Number of local variables that are live on entry to the procedure.
Ei	Macro	Environment block size of i-th procedure (Di+Li).
Mi	Macro	Maximum size environment block of all called procedures.

The items above labeled "macros" are constants that the compiler could generate as absolute numbers or as LITTLE macros, such as + * N3=4 **.

The calling sequence given includes all the steps required for function or subroutine calls, for dexter or sinister calls, and for recursive and non-recursive calls. The columns on the right indicate which lines of code to use in four important special cases. These are the four combinations of (1) the "general" and non-recursive cases, and (2) the dexter and sinister cases. By "general" it is meant that the compiler does not know if the call is recursive or not. By "non-recursive", it is meant that the particular call is known to be non-recursive. If, in addition, the compiler knows that the called procedure is never invoked recursively, then the two statements indicated by (X) may be deleted.

The first line of the calling sequence is RESERVE(M1), which has already been discussed. It is shown as a macro call, to defer the decision as to whether it compiles in-line or as a library routine call.

The next lines evaluate the arguments and place their root words in compiler temporary locations of the calling procedure. Subsequently, these words will be moved to the formal parameter locations of the called procedure. The reason the arguments cannot in general be planted directly in the called procedure's environment block is that to do so without more extensive stack housekeeping would cause wasteful use of stack space for calls of the type "sub(f(x),g(h(x)))." It is hoped that the optimiser can eliminate many of the uses of temporaries through standard techniques, but this may be difficult because of the subscripting involved.

Next the right-hand side is evaluated for sinister calls. The value of the right-hand side is placed in the globally known location RESULT, which is also used for function value returning. Note that the order of expression evaluation in a statement such as $f(e_1,e_2)=e_3$ is e_1,e_2,e_3. This follows PL/I array assignments, and will be easy to remember if, in SETL, we consistently use left-to-right order when an arbitrary ordering is imposed.

The next step saves the pointer to the top of the reserved area, as the called procedure may alter it.

Then the called procedure's invocation count is tested. If it is zero, the call is not recursive, and the next four steps are skipped.

If the call is recursive (I2>0), the called procedure's environment block will be located at the top of the run-time stack. This is set up by saving the current value of B2 and then setting B2 equal to the current top of the stack, T. T is then incremented by the size of the called procedure's environment block (E2), and the first word of the new environment block is initialized.

If the call is not recursive (I2=0), the existing value of B2 is used. This points to the procedure's base level environment block.

The statement at label ZZZA resets the "garbage collector skip flag" to signify that the procedure is active, and the next D2 words may not be bypassed during the garbage collector's marking phase.

Next the sinister flag is set, the index of the return point is stored, B is saved in the new environment block, and the arguments are moved to the new environment block. The called procedure's invocation count is incremented, its environment block is made the currently active one (B=B2), and control is given to the called procedure.

When the called procedure returns, the calling procedure restores its own environment. First its environment block is made the currently active one (B=STACK(B+2)), and the called procedure's invocation count is decremented.

Next, all arguments that are valid value receivers are updated by moving the called procedure's formal parameter root words to the appropriate places. This would usually be the calling procedure's environment block, but it might be another block if an argument appeared in an external statement.

Similarly, the RESULT is moved to the appropriate place, in the case of a dexter function call.

If the call was recursive, the top-of-stack pointer T is restored, as is the called procedure's most recently active environment block pointer, B2. If the call was not recursive, T and B2 are not altered, but the garbage collector skip flag is set to indicate that the "dead on entry" variables may be skipped over during the marking phase.

Finally, the reserved area pointer TRES is restored. This step does <u>not</u> restore TRES to the value it had on entry to procedure 1; that restoration is done by the caller of procedure 1.

LITTLE code for the standard procedure prologue and epilogue is shown on page 13.

In the prologue, a test is made for recursive entry. If recursive, the "live on entry" local variables are initialised to Ω . The steps to do this are indicated by a macro call, which might expand to a library routine call or to an in-line loop. Note that a non-recursive procedure needs no prologue code at all.

In the epilogue, the sinister flag is tested. If not set (dexter call), the globally known location RESULT is set to the value of the return expression. If the call was sinister, the value of RESULT is used as an input to the "corresponding code" of the return expression (see SETL newsletter 30, pages 15-17).

Procedure exit is accomplished by a branch to a globally known location /RETVECT/. The GOBY statement which the compiler places here branches to the return point indicated by the return index that was stored in the called procedure's environment block.

The figure on page 14 depicts the run-time stack during a recursive procedure call. The initial configuration shows the environment block of P1 at the top of the stack, which need not always be the case. TRES has an initial value that was set by the caller of P1.

STANDARD CALLING SEQUENCE
For Procedure 1 Calling Procedure 2

			GENERAL		NON-REC.	
			DEX	SIN	DEX	SIN
	RESERVE(M1)		X	X		
	code to evaluate argument i	Repeat	X	X	X	X
	STACK(B+k_i)=root word for arg. i	N2 times.	X	X	X	X
	code to evaluate r.h.s.			X		X
	RESULT=root word of r.h.s.			X		X
	STACK(B+3)=TRES.,		X	X	X	X
	IF (I2.EQ.0)GO TO ZZZA.,		X	X		
	STACK(B+4)=B2.,		X	X		
	B2=T.,		X	X		
	T=T+E2.,		X	X		
	EVAL STACK(B2)=D2.,		X	X		
/ZZZA/	ESKIPF STACK(B2)=0.,		X	X	X	X
	ESINISTR STACK(B2+1)=0.,		X		X	
	ESINISTR STACK(B2+1)=16.,			X		X
	ERETPT STACK(B2+1)=j.,		X	X	X	X
	STACK(B2+2)=B.,		X	X	X	X
	STACK(B2+i+4)=STACK(B+k_i).,	Repeat N2 times.	X	X	X	X
	I2=I2+1.,		X	X	(X)	(X)
	B=B2.,		X	X	X	X
	GO TO P2.,		X	X	X	X
/RLABj/	B=STACK(B2+2).,		X	X	X	X
	I2=I2-1.,		X	X	(X)	(X)
	STACK(B+v_i)=STACK(B2+i+4).,	As	X	X	X	X
	STACK(B+v)=RESULT.,	required	X		X	
	IF (I2.EQ.0) GO TO ZZZB.,		X	X		
	T=T-E2.,		X	X		
	B2=STACK(B+4).,		X	X		
	GO TO ZZZC.,		X	X		
/ZZZB/	ESKIPF STACK(B2)=17.,		X	X	X	X
/ZZZC/	TRES=STACK(B+3).,		X	X	X	X

STANDARD PROLOGUE
For Procedure 2

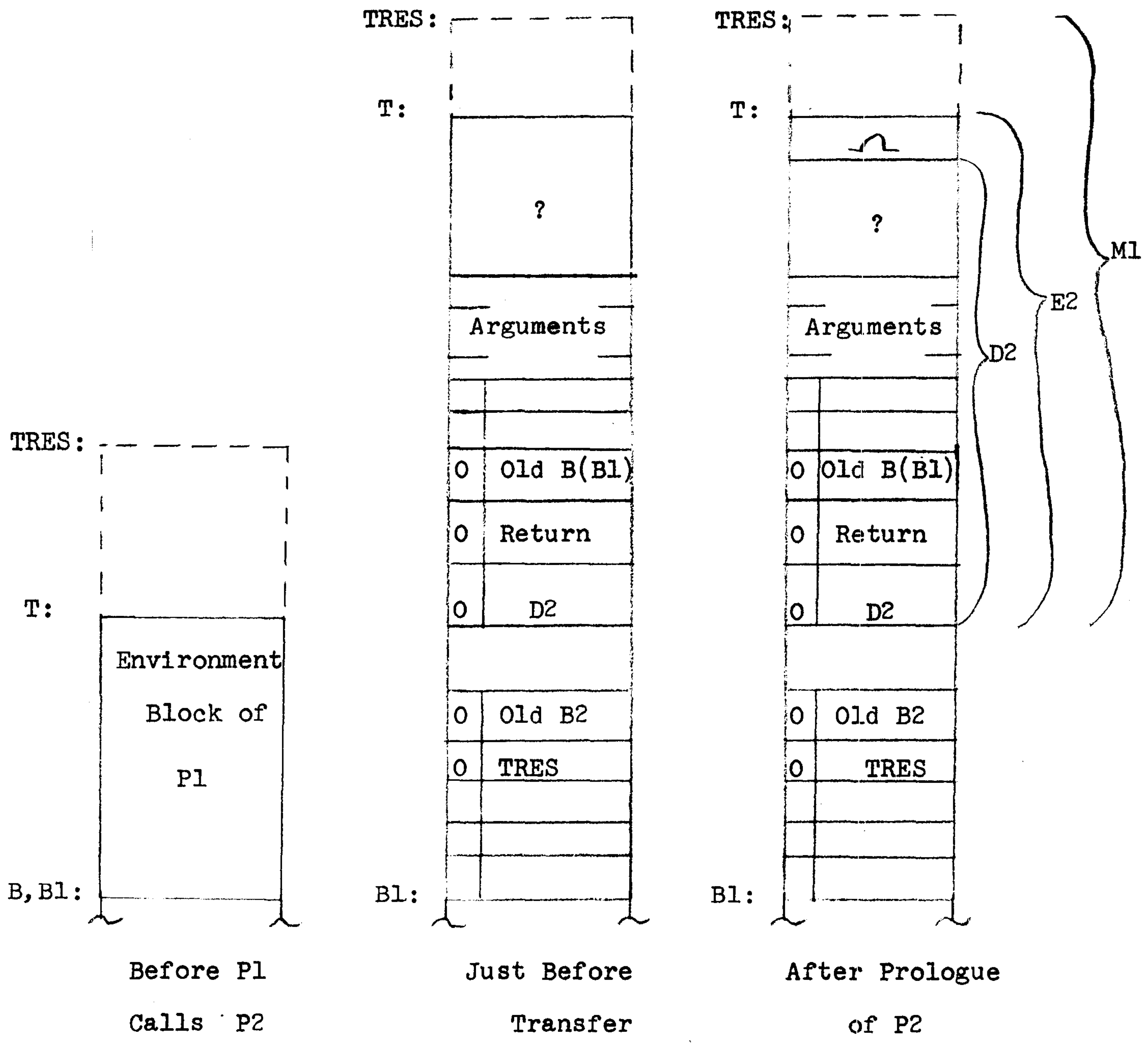
		GEN.	NON-REC.
/P2/	IF (I2.EQ.1) GO TO ZZZA.,	X	
	SETUNDF(B+D2,L2)	X	
/ZZZA/	CONTINUE.,	X	

STANDARD EPILOGUE
For Procedure 2

		GEN.	NON-REC.
	IF (ESINISTR STACK(B).EQ.0) GO TO ZZZA.,	X	X
	corresponding code of return expression (RESULT)	X	X
	GO TO RETVECT.,	X	X
/ZZZA/	code to evaluate return expression	X	X
	RESULT=root word of return expression.,	X	X
	GO TO RETVECT.,	X	X

RETURN VECTOR (GLOBAL)

```
/RETVECT/ GOBY ERETPT STACK(B) (RLAB1,...,RLABm).,
```



Run-time Stack During a Recursive Procedure Call

APPENDIX

It is instructive to consider a simple case with some significant optimisation opportunities, to see what capabilities are needed to optimise procedure calls that occur within loops.

The statement

$$s = \{ x \in p \mid f(x,y) \ \underline{eq} \ z \}$$

will be used. We suppose that f is a SETL routine that has already been compiled and optimised, and the compiler has filed the following information about f, which is available when compiling the above statement:

1. f is a "lowest level" procedure, i.e., it calls no other SETL procedures.

2. f does not modify either of its parameters.

We also assume that all of the above variables (s, x, p, y, and z) are local to the caller, or at any rate are not altered by f by means of an "external" statement.

From (1) above the compiler infers that f is not recursive, and it therefore generates LITTLE code such as the following (which is regarded as being unoptimised):

```
1.     STACK(B+s)=NULLSET.,
2.     STACK(B+x)=UNDEF.,
3./L1/STACK(B+x)=NEXTELT(STACK(B+p),STACK(B+x)).,
4.     IF (STACK(B+x).EQ.UNDEF)GO TO L2.,

5.     STACK(B+k_1)=STACK(B+x).,
6.     STACK(B+k_2)=STACK(B+y).,
7.     STACK(B+3)=TRES.,
8.     ESKIPF STACK(B2)=0.,
9.     ESINISTR STACK(B2+1)=0.,
10.    ERETPT STACK(B2+1)=j.,
11.    STACK(B2+2)=B.,
12.    STACK(B2+5)=STACK(B+k1).,
13.    STACK(B2+6)=STACK(B+k2).,
14.    B=B2.,
15.    GO TO P2.,
16./RLABj/B=STACK(B2+2).,
17.    STACK(B+k3)=RESULT.,
18.    ESKIPF STACK(B2)=17.,
19.    TRES=STACK(B+3).,

20.    STACK(B+k4)=EQUAL(STACK(B+k3),STACK(B+z)).,
21.    IF(.NOT.STACK(B+k4)) GO TO L1.,
22.    CALL AUGMENT(STACK(B+s),STACK(B+x)).,
23.    GO TO L1.,
24./L2/CONTINUE.,
```

As was pointed out, the compiler has already made use of the fact that f is not recursive. Furthermore, in the above code, the compiler made use of the fact that f does not alter its formal parameters, and has therefore suppressed the generation of code to update x and y upon return.

The calling sequence occurs in lines 5 through 19. Lines 3 through 24 constitute an interval which will be optimised.

The first observation is that lines 5 and 6, and the "compiler-temporary" locations denoted by STACK(B+k_1) and STACK(B+k_2), may be deleted by changing lines 12 and 13 to:

```
12'.    STACK(B2+5)=STACK(B+x).,
13'.    STACK(B2+6)=STACK(B+y).,
```

by a process similar to constant propagation and deleting dead assignments. (To simplify the discussion, it is assumed that the compiler has generated four compiler temporary locations. In reality, probably only two would have been used.)

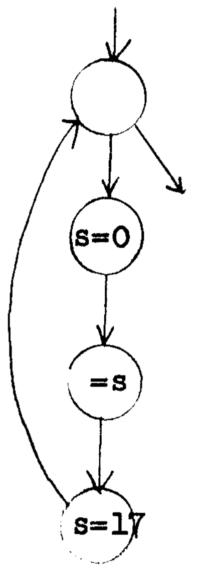
To do the above optimisation and practically anything else, the optimiser must be able to analyse subscript expressions in a fairly sophisticated way.

The next observation is that lines 7 and 19 may similarly be combined, resulting in the deletion of both of them. The fact that they can be combined depends upon the fact that neither STACK(B+3) nor TRES is altered between lines 7 and 19. To see that STACK(B+3) is not altered, the optimiser must somehow be informed that in lines 8-13 and 18, B and B2 point to entirely different areas of the stack (note, however, that within procedure i, B and B_i refer to the same area of the stack except at certain points in a calling sequence when the procedure is calling itself). Furthermore, the optimiser must be informed that procedure 2 (f, in our case) does not alter STACK(B+3) (with the "old" value of B).

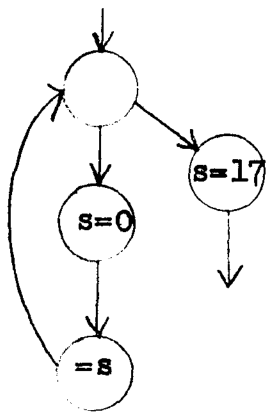
The fact that the global variable TRES is not altered by f may be inferred from the fact that f does not call a SETL procedure. Alternatively, TRES might be explicitly listed as "not set" (and, in fact, not used) in the packet of information about f.

Thus, combining lines 7 and 19 makes lines 7 an assignment to a dead variable and line 19 the no-operation "TRES=TRES".

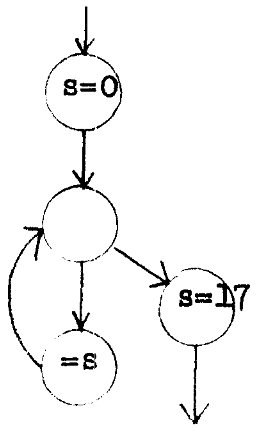
Lines 8 and 18 set the garbage collector skip flag. If the optimiser assumes that this skip flag is used only in procedure f (it is, of course, used in f via f's use of the garbage collector), then lines 8 and 18 could be moved out of the loop as illustrated on the next page.



Original



"s=17" moved forward



"s=0" moved out of loop

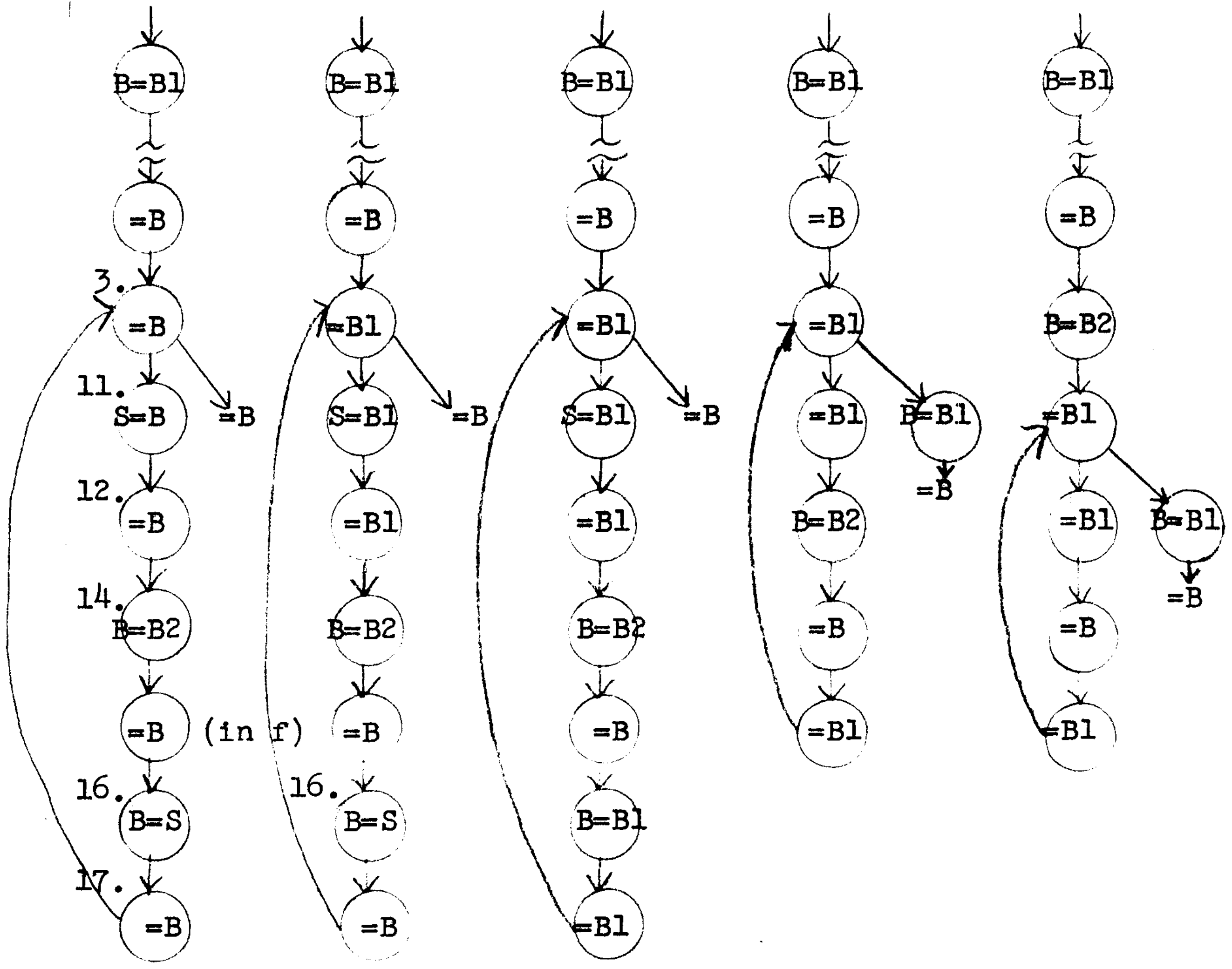
Here "s" denotes the skip flag (ESKIPF STACK(B2)). First "s=17" is moved forward, and the assignment "s=17" occurring before "s=0" (not shown) is deleted. This makes s invariant in the loop, so "s=0" can be moved out.

Although it is tempting to do this optimisation, it probably should not be done because it causes the garbage collector to reclaim less storage (and to take longer to execute) if it is invoked between lines 19 and 23 (in the example, it might be invoked at line 22). Thus the decision is really a space-time tradeoff.

If the decision is made to suppress significant code motion involving changes to the skip flag, then rules must be formulated and the skip flag assignments must be exposed to the optimiser. The situation is analogous to one in which interruptions can occur at certain points (certain library routine calls) which result in uses of certain variables (the skip flags).

Lines 9 and 10 are invariant in the loop and can be factored out. Note that the optimiser must recognize part-word insertions, if for no other reason than to prevent line 9 from looking like an assignment to a dead variable.

The three statements at lines 11, 14, and 16 may be transformed into two statements occurring outside of the loop, by the somewhat involved steps illustrated below.



Original "B=B1" propagated "S=B1" and "B=B1" propagated "S=B1" deleted, "B=B1" moved forward "B=B2" removed from loop

On entry to procedure P1, B has the value B1. The optimiser should be informed of this by placing a "dummy assignment" to this effect at the entry node of P1. This is indicated by the first node "B=B1" in the above graphs.

The first step shows the substitution of B1 for B in the region being optimised, by a process similar to constant propagation. The next step shows a similar propagation of "S=B1" and the "B=B1" that results at node 16. S denotes the expression "STACK(B2+2)".

In the third graph, S has become a dead variable, so the assignment to it can be deleted. The assignment "B=B1" at the bottom of the loop can be moved forward. This places it outside the loop, and also at the beginning of the loop, where it is an assignment to a dead variable and hence may be immediately deleted. The fourth graph shows the structure after these steps.

In the fourth graph, the assignment "B=B2" may be moved back, making B constant in the loop, so the assignment can be factored out. (This could have been done at any point after the second graph.) The final configuration is shown in the fifth graph.

It is possible that this optimisation should not be done, because there may be a facility added to LITTLE to allow a single variable (such as B) to be permanently assigned to a register. In this event, the substitution of B1 for B would be harmful to execution speed.

The key to the above transformation is the substitution of B1 for B. This substitution cannot always be done. This is not evident from the simple case being analysed, but by considering the case of a procedure calling itself (recursively), one finds that the assignments "B1=T" and "B1=STACK(B+4)" occur in the calling sequence. Other optimisations may then be possible, but they won't be dwelled upon here.

Line 13 of the calling sequence, which has been altered by the optimiser already, is now invariant and may be removed from the loop. This corresponds to the fact that y in f(x,y) is invariant.

The final transformed code is shown on the next page. The original calling sequence of 15 statements occurring in a loop has been transformed into five statements in the loop and five statements outside of it.

```
1.           STACK(B+s)=NULLSET.,
2.           STACK(B+x)=UNDEF.,
3.           ESINISTR STACK(B2+1)=0.,
4.           ERETPT STACK(B2+1)=j.,
5.           STACK(B2+6)=STACK(B1+y).,
6.           B=B2.,
7. /L1/      STACK(B1+x)=NEXTELT(STACK(B1+p),STACK(B1+x).,
8.           IF (STACK(B1+x).EQ.UNDEF) GO TO L2.,

9.           ESKIPF STACK(B2)=0.,
10.          STACK(B2+5)=STACK(B1+x).,
11.          GO TO P2.,
12./RLABj/   STACK(B1+$k_3$)=RESULT.,
13.          ESKIPF STACK(B2)=17.,

14.          STACK(B1+$k_4$)=EQUAL(STACK(B1+$k_3$),STACK(B1+z)).,
15.          IF(.NOT.STACK(B1+$k_4$))GO TO L1.,
16.          CALL AUGMENT(STACK(B1+s),STACK(B1+x)).,
17           GO TO L1.,
18./L2/      CONTINUE.,
19.          B=B1.,
```