SETL Newsletter # 113

August 8, 1973

Stephanie Brown

Little Code Generation From The BALM Compiler

This newsletter is intended as a general design specification for a new set of code generator for the BALM Compiler to produce LITTLE.

General Environment

The only part of the BALM compiler we propose to modify is the codegenerator and the tables associated with code generation. We plan to retain the same framework used in the current BALM code generator.

Here follows a brief description of the current BALM code generator.

The control program (CODEGEN) is called whenever a procedure is found in the parsed tree (PROC). Since the appropriate format for defining a procedure in BALM is

X = PROC (ARGS), EXPR END;

CODEGEN has two arguments

1) The name appearing on the left hand side of the equal sign.

2) The parsed and macro expanded tree whose head is PROC. In the case where the BALM expression to be compiled is not a procedure definition, it is made into a procedure and gived the arbitrary name CURCOM. Procedure CODEGEN initializes registers, lists and generates the necessary code to reference the arguments. Next the main code generator routine (COMP) is called with the tree representing the body of the procedure as an argument. Note that the body of a BALM procedure can contain only one expression which may be BEGIN (), ..., ... END. COMP checks for the following cases.

1) its argument is an identifier (name) in which case it generates the appropriate LOAD instruction.
2) its argument is a constant (i.e. not a pair); the appropriate load instruction is generated.
3) its argument is a tree whose head appears on the CODEGENLIST. each entry on this list is a special case and a procedure is associated with it. COMP calls the procedure with the tree as an argument.
4) COMP is called recursively to load the arguments associated with the head of the tree which is COMP's argument. Then

a) if the name at the head of the list is an MBALM op code (appears on OPLIST) the op code is generated.
For example: A + B results in

```
LOAD A
LOAD B
+
```

b) The name is assumed to be a BALM procedure and a call is generated.
For example: FACT (5); results in

```
LOAD  5
LOAD  FACT
CALL  (1 argument)
```

While the same logic will be employed to generate LITTLE, COMP will have a second argument giving the desired destination. This allows us to produce

```
A = GENINT(1);  rather than
RESULT = GENINT(1);
A = RESULT;
```

for A=1. We can also load arguments to SRTL procedures into the correct registers. OPLIST must now contain the following data

1) the name of the SRTL procedure associated with the operation
2) information about where the arguments should be loaded (i.e. ARG1, STACK, etc.)
3) Where the result is returned
4) Whether this is a function or a subroutine.

Some additional bookkeeping facilities must be added. We need an allocator for certain special quantities such as ARG1, ARG2, RESULT,we need to keep track of the contents of the stack and of the various quantities, and we need to keep track of the stack height. Initially all the BALM operations requiring HEAP allocations will be handled by SRTL procedures. This relieves the code generator from the bookkeeping necessary for garbage collection. It does not have to tack HEAP pointers and initialize HEAP locations of partially constructed data structures.

Stack Management

MBALM instructions push and pop a stack. In the simulator this is exactly what is done. The MBALM to COMPASS TRANSLATOR is considerable more efficient in its use of the stack. The height of the stack is computed for each procedure. On entry to the procedure if the necessary number of stack locations are not available a garbage collection takes place. Each instruction which uses the stack has a reference of the following type

STACK(PB+K)

where PB is the height of the stack on entry to the procedure and K is a constant. Whenever the garbage collector is called the current height of the stack is stored.

In the LITTLE code generators we propose to produce code which will handle the stack in a similar fashion. At the beginning of each procedure there will be a

RVSTK(n);

RVSTK or whatever procedure we use to reserve n stack locations will initialize the stack to zero. Then the pointer to the stack top will be increased by n. At the end of the procedure the pointer to the top of the stack will be decremented by n. This method has the advantage that the height of the stack at a given moment need not be kept. This makes the LITTLE code produced somewhat simpler.

The disadvantage is that heap pointers stored in the stack during a procedure may no longer be needed when a garbage collection occurs but the space cannot be recovered until the procedure exits.

Library Usage

A number of BALM operations will be implemented by calls to SRTL procedures or upon utility procedures which must be added to the library. The following is a list of these operations:

CALL
RETURN
APPLY
CFROMV - must call the LITTLE compiler
STKTRACE
+
—
*
/
LIST
VECTOR } - handled as special cases
STRING
: - (cons operation)
VFROMS
SFROMV
INDEX - V[i]
SETINDEX - V[i] = X
IDFROMS
SIZE - NELT
STOP
GARBCOLL
WRLINE
RDLINE
REWIND
BACKSPACE
TIME
END FILE

In-Line LITTLE code

The following is a list of BALM expressions which will be handled as special cases and the LITTLE code produced.

```
PROC(ID1, ID2, ... IDn), X END
     CALL RVSTK (n);
     T=T+n;
     ARG1 = STACK(AB-1);
     STACK(AB-1)=VALTB(id1);  / id1 is an integer constant /
     VALTB (id1) =ARG1;
     .
     .
     .
     ARG1=STACK(AB-n);
     STACK(AB-n)=VALTB(idn);
     VALTB(idn)=ARG1;
     compile code to compute x, the value being in RESULT
     VALTB(id1)=STACK(AB-1);
     .
     .
     VALTB(idn)= STACK(AB-n);
     T=T-n ;
     CALL RETURN;
BEGIN(ID1,ID2, .... ,IDN), X ...,L,...END
     STACK(PB+K)=VB;
     STACK(PB+K1)=NV;
     VB=PB+K1;
     NV=n;
     STACK(VB-1)=VALTB(ID1);
     STACK(VB-2)=ITEMNIL;
     .
     .
     .
```

```
      STACK(VB-n)=VALTB(IDN);
      compile code to stack X
      .
/L/   ?
      .
/RET/VALTB(ID1)=STACK(VB-1);
      .
      .
      VALTB (IDN)=STACK(VB-n);
      VB=STACK(PB+K);
      NV=STACK(PB+K1);
the value of the block is in RESULT

FOR I=(J,K,L) REPEAT X;
if L is a constant 1 and J and K are constants
      I=ROOTSINT;
      DOM(STACK(PB+K),J,K);
      EVAL I = STACK(PB+K);
      compile code to compute X
      EDOM;
if J and K are variables
      I=ROOTSINT;
      STACK(PB+K1)=EVALSINT(J);
      STACK(PB+K2)=EVALSINT(K);
      DOM (STACK(PB+K3),STACK(PB+K1),STACK(PB+K2));
      EVAL I = STACK(PB+K3);
      compile code to compute X
      EDOM;
if L is not a contant 1
      I=ROOTSINT;
      STACK(PB+K1)=EVALSINT(J);
      STACK(PB+K2)=EVALSINT(K);
      STACK(PB+K3)=1;
      IF ESIGNSINT(L) THEN (STACK(PB+K3)= 2); /negative increment/
      STACK(PB+K4)=EVALSINT(L);
      GOBY STACK(PB+K3) (PTEST,NTEST);
```

```
/START/ EVAL I = STACK(PB+K1);
     compile code to compute X
     GOBY STACK(PB+K3) (PINC,NINC);
/PINC/ STACK(PB+K1)=STACK(PB+K1)+STACK(PB+K4);
     GO TO PTEST;
/NINC/ STACK(PB+K1)=STACK(PB+K1)-STACK(PB+K4);
     GO TO NTEST;
/PTEST/ IF(STACK(PB+K1) .LE. STACK(PB+K2)) GO TO START;
     GO TO DONE;
/NTEST/ IF(STACK(PB+K1) .GE. STACK(PB+K2)) GO TO START;
/DONE/ .
WHILE X1 REPEAT X2
     RESULT = ITEMNIL;
/MORE/ compile code for X1 into ARG1
     IF(ARG1 .EQ. FALSE) GO TO NTRUE;
     compile code for X2 into RESULT
     GO TO MORE;
/NTRUE/

RETURN  X
     compile code for X into RESULT
     GO TO RET;
GOTO X
if X is a label
     GO TO X;
if X is not a label
     RESULT = X
     GO TO GGOBY;
/GGOBY/ IF (ETYPE  RESULT) .EQ. LBL) GOTO GGGOBY;
     CALL ERROR;
/GGGOBY/ GOBY (EVAL RESULT (L1, L2, .....Ln);
IF X1 THEN X2 ELSE X3
     IF (X1 .EQ. TRUE)  THEN X2;
     ELSE X3;
     ENDIF;
```

```
X1=X2
     compile code for X2
     X1=RESULT;

HD X1 = X2
     compile code for X2 into RESULT
     compile code for X1 into ARG1
     HEAP(EPTR ARG1) = RESULT;

PAIRQ(X)
     RESULT=ROOTIBOOL;
     EVSBS  =(EYPE X) .EQ. SPECPAIR;

X1  EQ X2
     compiler will check X2 for constant integer, NL. NULT.,etc.
     equality test will be inline for those cases
     otherwise,
     ARG1=X1;
     ARG2=X2;
     CALL EQUAL;
```

In the above code samples it is assumed that whenever LITTLE macros are used that they will exist to the proper depth of nesting. Currently DOM and IF THEN ELSE exist only for 3 levels.

Tasks

There are two sets of tasks necessary for implementing the LITTLE code generators. They can proceed somewhat independently of one another.

1. This task involves programming in LITTLE:
   a) Write the necessary utility routines and routines to interface between the SRTL and the generated LITTLE code. i.e. a RVSTK which initializes the stack, a GENSET which gets arguments from the stack.
   b) Work out the details of the interface so that a LITTLE program may call the LITTLE compiler. LITTLE code is input and output should be a block of machine code. This is necessary to implement CFROMV.

2. This task involves making modifications to BALM:
   a) decide on table layout and encoding for OPLIST, etc. Work out the details of formatting LITTLE code. The code will probably be generated as a list of strings, ID's, numbers and will have to be converted to appropriate LITTLE.
   b) Write the actual BALM code generators, the allocator for registers, ARG1, ARG2, RESULT.