BOUNDARY CONTEXT TRANSLATION

Robert M. Graham
Computation Center
Massachusetts Institute of Technology
Cambridge, Mass.

All translators are syntax-directed in the sense that the translator must obviously recognize the various syntactic structures and the output of the translator is a function of the syntax of the language. The term syntax-directed is usually applied to a translator which contains a direct encoding of the syntax of the language, this direct encoding being used by the translator as data. The companion paper by Cheatham and Sattley is concerned with this type of translation.1 In the other class of translators the syntax is essentially buried in the coding of the translator. Most of the algebraic languages in use are precedence grammars,2,3 or close enough so that the properties of precedence grammars are useful. Using the properties of precedence grammars, bounded context translation is possible. At each step in the scan of an expression in bounded context translation the decision as to what action to take next is a function of the symbol currently under scan and of N symbols on either side (where N is fixed for the particular language).

Most translators produce an intermediate form of the program before producing the final machine code. Once this intermediate form is produced, the distinction between syntax-directed translation and other methods disappears. The primary goal of the intermediate form is to encode the program in a form which is easily and efficiently processed by the computer. Most optimization algorithms, such as elimination of common subexpressions and optimum evaluation of Boolean expressions, are much simpler when applied to some intermediate form rather than to the original expression or the final machine language version. The intermediate form exhibits the structure (i.e., which subexpressions are the operands of each operator) and the order of evaluation of the subexpressions.

In this paper we will restrict ourselves to the source language defined in Appendix A. For the sake of brevity when we say expression we will mean either an expression, as defined in Appendix A, or an assignment statement. This language has no constants and identifiers are single letters, thus avoiding the problem of recognition. A translator for a language such as ALGOL would have a recognizer which would scan a statement and which, each time it was called, would recognize the next element in the input string. After recognizing an identifier the recognizer would store it in a symbol table if it were not already there. The symbol table would also contain other information pertaining to the identifier such as type, address, dimension, etc. The recognizer would then return a pointer to the place in the symbol table where that identifier and its properties were stored. A constant, after recognition, would be stored in a table of constants, given an internal identifier, and treated from that point on just as any other identifier would be. When an operator was recognized, a pointer to the place...
in the operator table where the properties of that operator were stored would be returned. Thus we see that the recognizer would effectively reduce an expression of a language such as ALGOL to an expression of our language; that is, each element would be reduced to a single symbol. In general, the intermediate form does not contain the identifiers or operators but the pointers supplied by the recognizer. However, for clarity, in the examples we will use the actual identifiers and operators. The recognizer is heavily dependent upon the details of the particular language being translated and such characteristics of the computer as character representation, word size, etc. The algorithms to be discussed in this paper depend only upon a few very general properties of the source language. Most of the common compiler languages have these properties so the techniques discussed here are essentially language independent. While the algorithms which we will describe here apply only to the simple expressions of Appendix A, several of them may be trivially generalized to translate more complex expressions and even the bulk of the ALGOL statements.  

Some representation of the tree form of an expression is used in several translators. Figure 1 shows the tree form of the expression A := B*C + D*(E - F) and the matrix representation of the tree 1, 2, 3. The first column is the operator, the second column is the left hand operand, and the third column is the right-hand operand. An integer represents the result of the row with that number. The subexpressions are to be evaluated in the sequence in which the rows are written. Figure 2 shows the tree form used by Ross 12. The first three columns in the representation are the same as in Figure 1. There are two additional columns which explicitly state the sequence of subexpression evaluation. The fourth column is the minor evaluation string pointer (dashed arrows in the tree) and the fifth column is the major evaluation string pointer (solid arrows in the tree). In the example the evaluation is to start with row 2. The rules for following the evaluation strings for the row under examination are:

1. If this is the first time this row has been examined and,
   a. If the minor string pointer is not empty; proceed to the row named by it.
   b. If the minor string pointer is empty; evaluate this row and proceed to the row named by the major string pointer.

2. If this is the second time this row has been examined, evaluate this row and proceed to the row named by the major string pointer.

In this example the ‘(’ and ‘)’ symbols should be treated as “do nothing” operators when their evaluation is called for by the rules. Ross’s representation is part of a more general system and, hence, some features of the representation are not pertinent to this discussion. A representation similar to Ross’s based on threaded lists is described in Ref. 9.

Evans 4 bases his intermediate form on postfix notation. The expression A := B*C + D*(E − F) in postfix form is ABCDEF * + :=:. In this form both the structure and the order of subexpression evaluation are implicit. The right-hand operand of an operator is the first complete expression to the left of the operator and
the left-hand operand is the second complete expression to the left of the operator. In the example the right-hand operand of the '+' is 'DEF-*' and the left hand operand is 'BC*'. The subexpressions are evaluated in left to right order.

The transformation of a completely parenthesized expression into matrix form is relatively simple. The expression is scanned from left to right. The following rules are applied to each symbol as it is encountered (a variable name is considered as a single symbol):

1. If the symbol is not a ')'; continue the scan.
2. If the symbol is a ')' and the expression is properly formed then the four symbols to the left of the ')' should be of the form 's1#s2', where 's1' and 's2' stand for variable names or integers (row numbers) and '#' is any operator; write 's1#s2' as the next row of the matrix and replace '(s1#2)' in the expression by the number of the row just written in the matrix. Continue the scan.

In Figure 3 the changes in the expression are shown on the left, the small arrow under the expression indicating the points, during the scan, at which a row is written in the matrix. The row actually written at that point is shown on the right.

Completely parenthesized expressions have such an overabundance of parentheses that they are difficult to read; hence, languages such as ALGOL have precedence rules making it unnecessary to write completely parenthesized expressions. The language is a precedence grammar if the precedence rules are such that given two adjacent operators it is unambiguous which is to be evaluated first. Thus if a language is a precedence grammar it is possible to construct a table of the type shown in Figure 4. To determine which of two adjacent operators to evaluate first, find the intersection of the row labeled with the left operator and the column labeled with the right operator. In the context of transforming an expression into matrix form, the order of evaluation of the operators is to be interpreted as the order of their entry into the matrix. A subexpression enclosed by parentheses is to be completely evaluated before any consideration is given to the operators on either side of it. Applying these considerations we have the following rules for transforming an expression to matrix form. We enclose the expression on the left by '[-' and on the right by ']-'. Again we scan from left to right, applying the rules to each symbol in turn:

1. If the symbol is an operator, #1, and the left context is,
   a. '[-s1'; continue the scan.
   b. '(s1'; continue the scan.
   c. 's2#s1'; look up #2#1 in the table.
      i. 'right'; continue the scan.
      ii. 'left'; write 's2#s1' as the next row of the matrix, replace 's2#s1' in the expression by the number of the row just written in the matrix, and apply rule 1 again.

2. If the symbol is ')' and the left context is,
   a. 's2#s1'; write 's2#s1' as the next row of the matrix, replace 's2#s1' in
the expression by the number of the row just written in the matrix, and apply rule 2 again.

b. `(s'); replace `(s)' by 's' and continue the scan.

3. If the symbol is `|—|' and the left context is,
   a. `s2#s1'; write `# s2 s1' as the next row of the matrix, replace `s2#s1' in the expression by the number of the row just written in the matrix, and apply rule 3 again.
   b. `|— s'; the expression has been completely transformed into matrix form.

This is the essence of bounded context translation. The rules just stated show that N = 3 for the precedence grammar of appendix A. That is, in deciding what action to take next, at most only the three symbols immediately to the left of the symbol currently under scan need be examined regardless of the length of the expression being translated.

Before examining some ways that bounded context analysis has actually been implemented let us restate, precisely, the above rules in the form of a flow chart (Figure 5). In the flow charts of this paper the true exit of a decision box is marked t and the false exit is marked f. We consider the expression being analyzed as a string of symbols, S (indexed by k), bounded on the left by `|—|' and on the right by `|—|'. We will distinguish three classes of symbols:

1. I, the class of identifiers: any variable name.
2. R, the class of matrix row numbers: any integer.
3. θ, the class of operator symbols: θ = {+, —, *, /, ::=, ( ), |—, —|}. For future use we distinguish the subclass of arithmetic operators, θ° = {+, —, *, /}.

Symbols from the input string are transferred to an auxiliary list, L (indexed by j). This will avoid the problem of gaps when replacements are made in the input string. M is the matrix (with i the row index) and T(x, y) is a function whose arguments are a pair of operators and whose value is the label found at the intersection of the x-row and the y-column in Figure

6. The label ERR has been filled in for all illegal operator pairs. If one of these occurs then the expression is incorrectly formed. The questions of the detection of incorrectly formed expressions and what action to take when errors are detected is very important in actual translator construction. These questions will not, however, be discussed in this paper. We will assume here that all expressions are properly formed. When the function T(x, y) takes on the value END then the matrix form of the expression is complete.
It is obvious from the table in Figure 6 that most operator pairs cause one of two actions to take place: 1) the operator currently under scan is put on the list, or 2) rows are written in the matrix until an operator pair is found which calls for some other action. The only exceptions are the removal of parentheses and termination of the transformation. If a precedence function is defined for each operator then a table of all operator pairs is not necessary.

A precedence function, $P(x)$, is defined such that, given a pair of operators #1#2, if $P(#1) > P(#2)$ then the operator #1 is evaluated before the operator #2. A precedence function for the operators used in the previous examples is defined in Figure 7. The algorithm for transforming an expression into matrix form that is used in GAT5 and MAD67 uses a precedence function. The flow chart in Figure 8 gives an algorithm for generating the matrix form of the expression. This algorithm differs from the GAT-MAD algorithm only in the direction of scanning the expression. Notice that, assuming all expressions are properly formed and since $P(\text{'('}) < P(\text{')'})$ and $P(\text{|--'}) < P(\text{'-'})$, when the test $P(L(j-1)) \geq P(S(k))$ fails then $S(k) = \text{'\text{')'}}$ implies $L(j-1) = \text{'\text{')'}}$ and $S(k) = \text{|--'}$ implies $L(j-1) = \text{|--'}$.

Bauer and Samelson10 use two lists, one, L, in the translator and one, N, in the object program. L, called the "symbols cellar," is used for storing operators which can not be evaluated yet, just as in the previous algorithms. N, called the "numbers cellar," is used by the object program during execution for the temporary storage of partial results and values of identifiers. Their algorithm does not generate an intermediate form, but immediately generates machine code. In the examples of this paper, the machine language used will be that described in Appendix B. This language is very similar to the FAP symbolic machine language used on the IBM 709-90-94 computers. C (indexed by m) will be a matrix where generated machine instructions are stored. The first column will contain the operation code and the second column the address. Figure 9 is the table, for Bauer and Samelson's algorithm, corresponding to the table in Figure 6, and Figure 10 is their algorithm. Whenever an identifier is encountered in the scan, instructions are generated to move its value onto the N list. Bauer and Samelson give, in the same paper, modifications to this algorithm which will generate more efficient machine code.

### Figure 7

<table>
<thead>
<tr>
<th>x</th>
<th>P(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>/</td>
<td>7</td>
</tr>
<tr>
<td>*</td>
<td>7</td>
</tr>
<tr>
<td>+</td>
<td>6</td>
</tr>
<tr>
<td>-</td>
<td>6</td>
</tr>
<tr>
<td>:=</td>
<td>5</td>
</tr>
<tr>
<td>)</td>
<td>4</td>
</tr>
<tr>
<td>(</td>
<td>3</td>
</tr>
<tr>
<td>-</td>
<td>2</td>
</tr>
<tr>
<td>+</td>
<td>1</td>
</tr>
</tbody>
</table>

### Figure 8

![Flow chart](image)
Ross's algorithm, as given in Ref. 12, transforms more than just expressions into tree form. In the version of his algorithm given in Figures 11 and 12 the machinery not needed to transform expressions has been omitted. A minor evaluation string pointer is set whenever the right operand of an operator is set and both operands are variables, or whenever a left operand is set and the operator is a modifier. The only modifier in an expression is the '('.

The minor evaluation string is forced to pass through modifiers since a modifier may change the interpretation of the right operand. For example, the right operand of '(' may be either a normal subexpression or the argument of a function depending upon whether the left argument of the '(' is empty or is an identifier. A major evaluation string pointer is set whenever a right or left operand is set and the operand is a row number.
Evans uses an algorithm based on Floyd's productions. Instead of generating machine code directly as Floyd did in his paper, Evans transforms the expression into postfix form. This algorithm is probably the most versatile of the algorithms which we have described here. The central idea here is a set of productions which determine the course of action at each step in the translation process. A production consists of five parts:

1. A top of stack configuration.
2. A replacement for the configuration of part 1.
3. The name of an action routine.
4. Scan, no scan flag.
5. Which production to consider next.

Figure 14 gives the productions for transforming an expression into postfix form. The expression is scanned from left to right. As each new character is scanned it is put on the top of the pushdown stack and the productions are then searched for the first one whose part 1 matches the present top of the stack (when a class symbol such as \( \theta \) appears, any member of that class will give a match). When a match is found that portion of the top of the stack which matched part 1 of the production is replaced by part 2 of the production. If part 2 is empty, the replacement degenerates to the deletion of the symbols, which matched part 1, from the top of the stack. The action routine named in part 3 is then executed. After the action routine has been executed the productions are again searched for a match with the top of the stack; however, the search is started with the production whose line number is given in part 5 of the last interpreted production.

<table>
<thead>
<tr>
<th>x</th>
<th>P(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>7</td>
</tr>
<tr>
<td>/</td>
<td>7</td>
</tr>
<tr>
<td>+</td>
<td>6</td>
</tr>
<tr>
<td>-</td>
<td>6</td>
</tr>
<tr>
<td>:=</td>
<td>5</td>
</tr>
<tr>
<td>)</td>
<td>4</td>
</tr>
<tr>
<td>-1</td>
<td>3</td>
</tr>
<tr>
<td>(</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 14.

Figure 13.

\[
\begin{array}{|c|c|c|}
\hline
1 & 2 & 3 \\
\hline
1 & I & OUT + 1 \\
2 & ( & NOP * 1 \\
3 & C & COMP * 1 \\
4 & := & OUTP("\theta") + 1 \\
5 & := & COMP END \\
6 & ) & COMP 7 \\
7 & ( & NOP + 5 \\
\hline
\end{array}
\]

P-Table

Figure 15.
a "*" appears in part 4, a new symbol is scanned from the input string before the search continues. The productions given in Figure 13 are sufficient only to transform the simple expressions of this paper. The productions that Evans gives will transform all the statements of ALGOL into postfix form. Figure 15 gives an algorithm for transforming an expression into postfix form using the productions of Figure 13. The action routine OUT puts the last identifier scanned into the output string, M. The action routine OUTP(x) puts its argument, x, into the output string. The productions cause the unary operator 'loc' to be introduced into the output string following a variable which is on the left side of '==' which indicates that a location (where a value is to be stored) is involved rather than a value. The action routine COMP uses the precedence function, defined in Figure 14, to determine when operators are placed in the output string (i.e., to determine the sequence of evaluation of the operators).

Once the matrix form of the expression has been generated, the final translation to symbolic machine code is relatively simple. Corresponding to each operator is a set of machine instructions, a pattern. Figure 16 gives the patterns for the operators of our language; the first column is the operation code and the second column is the address. The matrix is translated one row at a time, in sequence. Row i of the matrix, M, is translated by making a copy, in the code output matrix, of the pattern corresponding to the operator M(i, 1), replacing all occurrences of '1' by the left operand, M(i, 2), all occurrences of 'r' by the right operand, M(i, 3), and all occurrences of 't' by the row number, i. The row numbers (integers) which appear in the code are to be interpreted as the names of temporary storage locations. Figure 17 is an algorithm for performing this translation. N is the number of rows in the matrix, M, C is the code output matrix, PAT1(x) a function whose value is the index of the first line of the pattern for the operator x, and PAT2(x) a function whose value is the index of the last line of the pattern for x (both these functions are defined in Figure 18). The translation of the matrix in Figure 1 is shown in Figure 19.

It is immediately obvious that very inefficient machine code is produced by this algorithm.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CLA</td>
<td>1</td>
<td>+</td>
</tr>
<tr>
<td>2</td>
<td>FAD</td>
<td>r</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>STO</td>
<td>t</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>CLA</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>FSB</td>
<td>r</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>STD</td>
<td>t</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>LOQ</td>
<td>1</td>
<td>+</td>
</tr>
<tr>
<td>8</td>
<td>FMP</td>
<td>r</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>STD</td>
<td>t</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>CLA</td>
<td>1</td>
<td>/</td>
</tr>
<tr>
<td>11</td>
<td>FDP</td>
<td>r</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>STO</td>
<td>t</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>CLA</td>
<td>r</td>
<td>:=</td>
</tr>
<tr>
<td>14</td>
<td>STO</td>
<td>t</td>
<td></td>
</tr>
</tbody>
</table>

Pattern Table

Figure 16.
Once we begin to consider the production of efficient machine code, the algorithms rapidly get very complicated. We can make some improvement, however, without a great deal of complication. In the example are several redundant store and fetch instructions. These can be eliminated if we keep track of the contents of the special machine registers. We then insert store instructions only when the current contents of a special register is not one of the operands of the next operator and fetch instructions only when the operand is not already in a special register. To implement this we generalize the idea of patterns. Corresponding to each operator is a variable pattern, that is, the instructions which are actually copied into the code output matrix depend upon the contents of the special registers.

The method used in the MAD translator is general enough for the machine structure assumed in this paper. The problem of generating efficient machine code is a very difficult one and is yet unsolved. There are methods, undocumented, other than the one to be described but none which can claim to produce highly efficient code in all circumstances. The patterns will be arranged so that the result of a row is, in general, not stored, i.e., it will be left in one of the registers AC or MQ. The machine code produced when a row of the matrix is translated will depend on the values of four Boolean variables. These variables are named AC, MQ, LO, and RO. Suppose we are ready to translate the ith row of the matrix, then these variables have the following meanings:

1. If AC is true, the result produced by row i-1 is in the AC.
2. If MQ is true, the result produced by row i-1 is in the MQ.
3. If LO is true, the left operand in row i is i-1 (i.e., the result of row i-1).

Figure 18.

<table>
<thead>
<tr>
<th>x</th>
<th>PAT1(x)</th>
<th>PAT2(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>-</td>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>*</td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>/</td>
<td>10</td>
<td>12</td>
</tr>
<tr>
<td>:=</td>
<td>13</td>
<td>14</td>
</tr>
</tbody>
</table>

Figure 19.

Once we begin to consider the production of efficient machine code, the algorithms rapidly get very complicated. We can make some improvement, however, without a great deal of complication. In the example are several redundant store and fetch instructions. These can be eliminated if we keep track of the contents of the special machine registers. We then insert store instructions only when the current contents of a special register is not one of the operands of the next operator and fetch instructions only when the operand is not already in a special register. To implement this we generalize the idea of patterns. Corresponding to each operator is a variable pattern, that is, the instructions which are actually copied into the code output matrix depend upon the contents of the special registers.

The method used in the MAD translator is general enough for the machine structure assumed in this paper. The problem of generating efficient machine code is a very difficult one and is yet unsolved. There are methods, undocumented, other than the one to be described but none which can claim to produce highly efficient code in all circumstances. The patterns will be arranged so that the result of a row is, in general, not stored, i.e., it will be left in one of the registers AC or MQ. The machine code produced when a row of the matrix is translated will depend on the values of four Boolean variables. These variables are named AC, MQ, LO, and RO. Suppose we are ready to translate the ith row of the matrix, then these variables have the following meanings:

1. If AC is true, the result produced by row i-1 is in the AC.
2. If MQ is true, the result produced by row i-1 is in the MQ.
3. If LO is true, the left operand in row i is i-1 (i.e., the result of row i-1).

Figure 19.

Once we begin to consider the production of efficient machine code, the algorithms rapidly get very complicated. We can make some improvement, however, without a great deal of complication. In the example are several redundant store and fetch instructions. These can be eliminated if we keep track of the contents of the special machine registers. We then insert store instructions only when the current contents of a special register is not one of the operands of the next operator and fetch instructions only when the operand is not already in a special register. To implement this we generalize the idea of patterns. Corresponding to each operator is a variable pattern, that is, the instructions which are actually copied into the code output matrix depend upon the contents of the special registers.

The method used in the MAD translator is general enough for the machine structure assumed in this paper. The problem of generating efficient machine code is a very difficult one and is yet unsolved. There are methods, undocumented, other than the one to be described but none which can claim to produce highly efficient code in all circumstances. The patterns will be arranged so that the result of a row is, in general, not stored, i.e., it will be left in one of the registers AC or MQ. The machine code produced when a row of the matrix is translated will depend on the values of four Boolean variables. These variables are named AC, MQ, LO, and RO. Suppose we are ready to translate the ith row of the matrix, then these variables have the following meanings:

1. If AC is true, the result produced by row i-1 is in the AC.
2. If MQ is true, the result produced by row i-1 is in the MQ.
3. If LO is true, the left operand in row i is i-1 (i.e., the result of row i-1).
4. If RO is true, the right operand in row i is i-1.

Instead of regarding the patterns as a fixed set of machine code to be produced in translating a row of the matrix, we now take the view that the pattern is really a small program which, when executed, produces machine code. Viewing it in this light, we need a language in which to write the program.

<table>
<thead>
<tr>
<th>operation code</th>
<th>first operand</th>
<th>second operand</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>inst</td>
<td>a</td>
<td>compile the machine instruction inst with a in its address field (a may be I, r, t, or blank)</td>
</tr>
<tr>
<td>[AC]</td>
<td>l1</td>
<td>l2</td>
<td>if [AC] = 'true' transfer to line 11, otherwise transfer to line l2</td>
</tr>
<tr>
<td>MQ</td>
<td></td>
<td></td>
<td>set the value of [AC] to val</td>
</tr>
<tr>
<td>LO</td>
<td></td>
<td></td>
<td>transfer to line 1</td>
</tr>
<tr>
<td>RO</td>
<td></td>
<td></td>
<td>halt</td>
</tr>
<tr>
<td>S</td>
<td>[AC]</td>
<td>val</td>
<td></td>
</tr>
<tr>
<td>J</td>
<td>l</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In our language we will need the following types of instructions: produce a machine instruction, branch on the truth or falsity of one of the Boolean variables, absolute transfer, set the value of one of the Boolean variables, and halt. Figure 20 is a flow chart for a program to produce the code for ‘+’, where ‘1’ and ‘r’ have the same meanings as in Figure 16, but ‘t’ now refers to the temporary used to store the result of the previous row. Notice that if there is a result in the AC or MQ and it is not either of the operands then instructions to store it and fetch one of the operands are generated. If one of the operands is in the wrong special register an exchange instruction is generated. The word COMPILE means, here, write the instruction into the code output matrix.

A command in our pattern programming language will have a line number, an operation

\[
\begin{array}{c|c|c|c|c|c|c|c|c}
\text{line} & \text{AC} & \text{RO} & \text{MQ} & \text{LO} & \text{XCA} & \text{STO} & \text{FRP} & \text{CLA} \\
01 & 2 & 6 & 24 & 20 & 16 & 10 & 52 & 56 & 60 \\
02 & 1 & 9 & 37 & 28 & 49 & 53 & 62 & 66 & 70 \\
03 & 1 & 4 & 26 & 34 & 35 & 51 & 75 & 79 & 83 \\
04 & 1 & 3 & 30 & 47 & 69 & 91 & 97 & 101 & 105 \\
05 & 2 & 6 & 38 & 40 & 52 & 64 & 76 & 79 & 83 \\
06 & 1 & 5 & 51 & 53 & 65 & 67 & 79 & 81 & 85 \\
07 & 1 & 4 & 61 & 63 & 75 & 77 & 89 & 91 & 95 \\
08 & 1 & 3 & 29 & 31 & 43 & 45 & 57 & 59 & 63 \\
09 & 1 & 2 & 18 & 20 & 32 & 34 & 46 & 48 & 52 \\
10 & 2 & 6 & 47 & 49 & 61 & 63 & 75 & 77 & 81 \\
11 & 1 & 5 & 61 & 63 & 75 & 77 & 89 & 91 & 93 \\
12 & 1 & 4 & 12 & 14 & 26 & 28 & 40 & 42 & 44 \\
13 & 1 & 3 & 8 & 11 & 23 & 25 & 37 & 39 & 41 \\
14 & 1 & 2 & 9 & 12 & 24 & 26 & 38 & 40 & 42 \\
15 & 2 & 6 & 28 & 30 & 42 & 44 & 55 & 57 & 61 \\
16 & 1 & 5 & 10 & 13 & 25 & 28 & 40 & 43 & 46 \\
17 & 1 & 4 & 4 & 7 & 19 & 22 & 34 & 37 & 40 \\
18 & 1 & 3 & 7 & 10 & 22 & 25 & 37 & 40 & 43 \\
19 & 1 & 2 & 6 & 9 & 18 & 21 & 33 & 36 & 39 \\
20 & 2 & 6 & 16 & 18 & 30 & 32 & 44 & 46 & 48 \\
21 & 1 & 5 & 11 & 14 & 26 & 29 & 41 & 44 & 47 \\
22 & 1 & 4 & 3 & 5 & 15 & 17 & 29 & 31 & 33 \\
23 & 1 & 3 & 2 & 4 & 10 & 12 & 24 & 26 & 28 \\
24 & 1 & 2 & 1 & 3 & 9 & 11 & 23 & 25 & 27 \\
25 & 2 & 6 & 2 & 4 & 10 & 12 & 24 & 26 & 28 \\
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c|c|c|c}
\text{x} & 1 & \text{PAT(x)} & 20 & 37 & 54 & 68 \\
\hline
\text{+} & 1 & \text{+} & \text{+} & \text{+} & \text{+} & \text{+} \\
\text{-} & \text{-} & \text{-} & \text{-} & \text{-} & \text{-} & \text{-} \\
\text{*} & \text{*} & \text{*} & \text{*} & \text{*} & \text{*} & \text{*} \\
\text{/} & \text{/} & \text{/} & \text{/} & \text{/} & \text{/} & \text{/} \\
\text{=} & \text{=} & \text{=} & \text{=} & \text{=} & \text{=} & \text{=} \\
\end{array}
\]

Figure 21. Figure 22.
Figure 23.

From the collection of the Computer History Museum (www.computerhistory.org)
The program for translating the matrix into machine code now becomes an interpreter which executes (interpretively), for each row of the matrix the appropriate pattern program. The pattern programs will be stored in a matrix, P (indexed by k), the first column is the operation code, the second column is the first operand, and the third column is the second operand. As before, we have a function PAT(x) whose value is the index of the first line of the pattern program for the operator x. The values of LO and RO are set by examining the operands in a row before executing the pattern program for that row. The function PAT(x) is defined by the table of Figure 21. Figure 22 gives the pattern programs (the P matrix), and Figure 23 is the interpreter for the pattern programs, i.e., the algorithm for translating the matrix into machine code. OP(y) is a function, defined in Figure 24, whose argument, y, is one of the operation codes of the pattern programming language and whose value is a label, the label of that portion of the algorithm which interprets the operation code y. Figure 25 shows the translation of the matrix in Figure 1 into machine code using the algorithm of Figure 23. For each row of the matrix is shown the machine code produced for that row and the status of the four Boolean variables after translating that row and just before considering the next row, that is, at point 3 in the flow chart of Figure 23. Notice that just this simple consideration of the contents of the special registers gives us a saving of five instructions when compared to the instructions produced by the algorithm of Figure 17.

It is obvious that only trivial modifications are necessary to be able to use the pattern program interpreter with Ross's intermediate form. Instead of considering the rows of the matrix in sequence, the minor and major evaluation strings are followed. When the rules for following the evaluation strings call for evaluation of a row, the appropriate pattern program is interpreted. Evans uses an algorithm very similar to the pattern program one which we have described.

No consideration has been given to the type (real, integer, etc.) of the identifiers involved. In many computers there are different instructions for floating decimal (real) and integer arithmetic. This is easily taken care of by having, for example, two pattern programs for '+' one to be interpreted when the operands are real, and the other to be interpreted when the operands are integer. Finally, it is clear that the interpreter instead of generating machine instructions could actually execute them, thus turning the entire translator itself into

<table>
<thead>
<tr>
<th>x</th>
<th>OP(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>MOP</td>
</tr>
<tr>
<td>AC</td>
<td>ACOP</td>
</tr>
<tr>
<td>MQ</td>
<td>MQOP</td>
</tr>
<tr>
<td>LO</td>
<td>LOOP</td>
</tr>
<tr>
<td>RO</td>
<td>ROOP</td>
</tr>
<tr>
<td>S</td>
<td>SOP</td>
</tr>
<tr>
<td>J</td>
<td>JOP</td>
</tr>
<tr>
<td>H</td>
<td>HOP</td>
</tr>
</tbody>
</table>

Figure 24.

<table>
<thead>
<tr>
<th></th>
<th>H</th>
<th>AC</th>
<th>MQ</th>
<th>LO</th>
<th>RO</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>*</td>
<td>B</td>
<td>C</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>E</td>
<td>F</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>3</td>
<td>*</td>
<td>D</td>
<td>2</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>4</td>
<td>+</td>
<td>1</td>
<td>3</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>5</td>
<td>=</td>
<td>A</td>
<td>4</td>
<td>true</td>
<td>false</td>
<td>true</td>
</tr>
</tbody>
</table>

Figure 25.
an interpreter which could execute (interpre­
tively) programs written in the source lan­
guage.

BIBLIOGRAPHY


APPENDIX A

Definition of the Language Used in the Examples

<identifier> ::= A | B | C | D | E | F | G | H
<primary> ::= <identifier> | <expression>
<mop> ::= + | ~ | * | /
<aop> ::= + | ~
<term> ::= <primary> | <term><mop>
<expression> ::= <term> | <expression><aop><term>
<assignment statement> ::= <identifier> ::= <expression>

APPENDIX B

The Machine Language Used in the Examples

There are two special registers, the AC and the MQ. Instructions are single address. The meaning of the instructions is expressed as a short ALGOL program. The '?' in the context 'MQ:=' means that the contents of the MQ is indeterminate. When the name of a register does not appear to the left of an '::::' in the description of an instruction, then the value of that register is unchanged.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLA X</td>
<td>AC::=X</td>
</tr>
<tr>
<td>FAD X</td>
<td>AC::=AC+X; MQ:=?</td>
</tr>
<tr>
<td>LDQ X</td>
<td>MQ::=X</td>
</tr>
<tr>
<td>FMP X</td>
<td>AC::=MQ*X; MQ:=?</td>
</tr>
<tr>
<td>FDP X</td>
<td>MQ::=AC/X; AC:=?</td>
</tr>
<tr>
<td>FSB X</td>
<td>AC::=AC-X; MQ:=?</td>
</tr>
<tr>
<td>STO X</td>
<td>X:=AC</td>
</tr>
<tr>
<td>STQ X</td>
<td>X:=MQ</td>
</tr>
<tr>
<td>CHS</td>
<td>AC::=-AC</td>
</tr>
<tr>
<td>XCA</td>
<td>TMP::=AC; AC::=MQ; MQ:=TMP</td>
</tr>
</tbody>
</table>