Compiling Techniques for Boolean Expressions and Conditional Statements in ALGOL 60

H. D. Huskey and W. H. Wattenburg

University of California, Berkeley, and Bendix Corporation, Los Angeles, California

1. Introduction

In this note compiling routines will be described for Boolean expressions involving the operators $\vee$, $\wedge$ and $\neg$. The Boolean variables allowed include relational expressions. These routines will then be used in describing procedures for compiling conditional statements and Boolean assignment statements as defined in ALGOL 60.

Any attempt to describe a general compiling algorithm immediately presents the following problem: If an algorithm is constructed to produce efficient object programs for a particular machine it will be a highly specialized routine. For this reason an intermediate language is being used for the object code of the compiling routines being developed at Berkeley. In this note "object code" and "object machine" will imply this intermediate language and a hypothetical machine whose order code is the intermediate language.

Appendix I contains a description of those portions of the intermediate language which are used by the routines to be described in this paper.

2. Compiling System

It is also difficult to describe a particular portion of a compiling system without describing the rest of the system to which it belongs. Hence, it is necessary to present the general characteristics of the compiling system before proceeding with the detail routines for Boolean expressions and conditional statements.

The compiler performs a single scan of the source language. For each identifier, commands are generated according to the delimiters which enclose the identifier. The particular commands generated are determined by two quantities, POP (previous operator) and NOP (next operator). A third register PS (peek symbol) is carried along and is normally equal to POP. It is used in the few cases where it is necessary to examine a delimiter ahead of NOP.

The main entry to the compiler is a block labeled EXECUTIVE. *The entire compiler is local to this block.* Within this block are a number of blocks labeled according to the type of ALGOL statements handled by the logic within them (the ones of interest in this paper will be those labeled BOOLEAN ROUTINE and RELATIONAL ROUTINE). The program immediately following EXECUTIVE determines the block to which control must be transferred to process the portion of source program presently under consideration.

FIND NEXT DELIMITER, BUILD COMMAND, and ARITHMETIC are procedures used by many routines in the compiler. Procedure FIND NEXT DELIMITER performs the scanning of the source program. Whenever it is called it finds the next delimiter and places the code for this delimiter in NOP. The previous value of NOP is assigned to POP and PS. If in scanning forward to find the next delimiter an identifier is encountered, the address for this identifier is assigned to ADDRESS. If no identifier is encountered, ADDRESS is set to zero.

Procedure BUILD COMMAND (OPCOPE, ADDRESS); string OPCODE; integer ADDRESS; places a command in the object program list. Whenever BUILD COMMAND is called with the actual parameter ADDRESS, a command with an absolute address (equal to the value of ADDRESS) is entered in the command list. However, when the actual parameter in place of OPCODE is a type 2 or type 3 command (see Appendix I) a dummy address is used. In this case BUILD COMMAND is called with one of the identifiers A, B, C, D, $\cdots$ in place of ADDRESS.

Procedure ARITHMETIC is called whenever an arithmetic operator is encountered. It is essentially Huskey's algorithm[1] for compiling algebraic expressions. When a non-arithmetic delimiter is encountered it return-transfers with the code for this delimiter in NOP.

The codes assigned to the various delimiters are given in Table 1.

3. Boolean Expressions

A program can easily be compiled to compute the value of a Boolean expression such as

$$A \vee B \wedge C \wedge D \vee \neg (E \wedge F \vee G) \tag{3.1}$$

if one requires all the logical operations to be carried out and the result tested for **true** or **false.** In general, however, this would produce inefficient object programs. For example, the expression (3.1) is completely determined if A is **true**—there is no need to examine the rest of the expression. Likewise, if any one of the variables in the term $B \wedge C \wedge D$ is **false** there is no need to examine the other

[1] HUSKEY, H. D., Compiling techniques for algebraic expressions. *Computer Journal of The British Computer Society*, in press.

TABLE 1

Delimiter	Code	Delimiter	Code	Delimiter	Code
,	10	=	41	**procedure**	62
;	11	≠	42	**own**	63
(	12	>	43	**Boolean**	64
)	13	≧	44	**integer**	65
[	14	<	45	**real**	66
]	15	≦	46	**array**	67
.	16				
		begin	50	**switch**	68
+	21	**end**	51	**string**	69
−	22	**if**	52	**label**	70
*	23	**then**	53	**value**	71
/	24	**else**	54		
÷	25	**for**	55		
↑	26	**step**	56		
		until	57		
∨	31	**while**	58		
∧	32	**do**	59		
¬	33	**comment**	60		
⊃	34	**go to**	61		
≡	35				

two variables. The algorithm presented in the BOOLEAN ROUTINE which follows produces an object program which takes into account such situations. The object programs are optimum in the sense that only the necessary minimum number of variables are tested, assuming that the expression is evaluated in strict left to right order.

The following picture is helpful in describing the algorithm. The source program that follows the Boolean expression contains two entry points to which control is transferred depending on whether the Boolean expression is **true** or **false.** These points are called TRUE EXIT and FALSE EXIT. The rules governing the routine are simple:

If a variable followed by the operator ∨ is **true,** then a transfer should occur to a point which tests the next variable which is at a lower level and is preceded by the operator ∧; if none exists, then to TRUE EXIT. If a variable followed by the operator ∧ is **false** then a transfer should occur to a point which tests the next variable which is at the same or a lower level and is preceded by ∨; if none exists, then to FALSE EXIT.

The appearance of a variable in a Boolean expression results in the command "clear and add variable" in the object program. Each appearance of an operator (∨ or ∧) results in a test of the accumulator by the object program, plus possibly one or more of the type 3 commands (see Appendix I) which properly connect the object program. The test commands are always "transfer on true" (TT) or "transfer on false" (TF) with dummy addresses. In practice these will be "transfer on non-zero" or "transfer on zero" depending upon the particular values used to represent **true** and **false** in a machine. Parentheses within a Boolean expression are taken care of by using the type 3 commands SL and RS which restrict the scope of the loading commands ILB, ILF, etc. Only two different dummy addresses are required for a Boolean expression of any complexity (the dummy addresses A and B are used in the BOOLEAN ROUTINE).

The effect of ¬'s in a Boolean expression is determined according to de Morgan's Theorem. If a variable is effectively complemented then the test is changed (TT becomes TF, TF becomes TT). If an operator is effectively changed then the dummy address of the transfer command is changed (A to B, B to A).

```
BOOLEAN ROUTINE:  begin
        procedure CHANGE NOT FLAG; begin if NOT
          FLAG = 0 then NOT FLAG := 1
        else NOT FLAG := 0 end change not flag;
        procedure LOAD NOT LEVEL; begin CHANGE
          NOT FLAG; NOT LEVEL := LEVEL;
        if NOT LEVEL > NOT HISTORY TABLE [NHTI]
          then NHTI := NHTI + 1;
        NOT HISTORY TABLE [NHTI] := NOT LEVEL
          end load not level,
        procedure REDUCE NOT HISTORY TABLE;
          begin CHANGE NOT FLAG;
        NHTI := NHTI − 1; NOT LEVEL := NOT
          HISTORY TABLE [NHTI] end reduce not
          history table;
        procedure NOT ROUTINE; begin if NOT FLAG
          > 0 then TEST FLAG := 1;
L1:     if LEVEL > NOT LEVEL then go to L3 else
L2:     REDUCE NOT HISTORY TABLE; if NHTI =
          0 then go to L4 else go to L1;
L3:     if NOT FLAG > 0 then OPERATOR FLAG := 1;
L4:     end not routine;
        comment the above procedures are used to keep
          track of the appearance of the Boolean operator ¬
          (not) in a Boolean expression. The NOT FLAG
          (1 or 0) indicates whether the Boolean variable
          under consideration has in effect been comple-
          mented by a previous ¬. However, one must keep
          track of the levels at which previous ¬'s have
          appeared since they could negate entire expres-
          sions. This is accomplished by storing the levels
          at which ¬'s have appeared in a list called NOT
          HISTORY TABLE whose index is NHTI. Before
          any Boolean operator (∧ or ∨) is processed the
          NOT ROUTINE is called, which determines the
          net effect of all ¬'s in the Boolean expression up
          to this operator. The NOT ROUTINE sets two
          flags, the TEST FLAG and the OPERATOR
          FLAG. If the OPERATOR FLAG = 1, then the
          operator presently being processed has in effect
          been changed according to de Morgan's Theorem.
          If the TEST FLAG = 1, then the previous vari-
          able has in effect been complemented according
          to de Morgan's Theorem. Hence, the logic of pro-
          cedure NOT ROUTINE forces the BOOLEAN
          ROUTINE to compile programs for the expanded
          equivalent (according to de Morgan's Theorem)
          of any Boolean expression.
        LEVEL at any point in the source program is
          equal to one plus the number of left parentheses
          encountered minus the number of right parentheses
          encountered. NOT LEVEL is always equal to the
          highest value stored in the NOT HISTORY TABLE
          and as such is always equal to the level of the most
          recent ¬ which can still effect the following pro-
          gram;
ENTRY:  BOOLEAN FLAG := 1;
DISTRIBUTE: if NOP = 31 then go to OR else if NOP =
          32 then go to AND else if NOP = 33 then go to
          NOT else if NOP = 12 then go to LF PAREN-
```

```
        THESIS else if NOP = 13 then go to RT PA-
        RENTHESIS;
      if 21 ≤ NOP ∧ NOP ≤ 26 ∨ 41 ≤ NOP ∧ NOP ≤
        46 then go to EXIT;
      comment if control reaches EXIT from this state-
        ment it means that a relational expression has
        been encountered. EXECUTIVE will transfer
        control to RELATIONAL ROUTINE;
END:  BUILD COMMAND ('CLA', ADDRESS); NOT
        ROUTINE; if TEST FLAG = 1 then BUILD
        COMMAND ('TT', A) else BUILD COMMAND
        ('TF', A); go to EXIT;
      comment if control reaches END the Boolean
        expression has been terminated and only the
        appropriate test for the last variable which
        appeared in the expression is necessary before
        returning to EXECUTIVE.
OR:   if NHTI ≠ 0 then NOT ROUTINE; if ADDRESS
        ≠ 0 then BUILD COMMAND ('CLA', AD-
        DRESS);
      if POP = 13 then go to OR2 else if OPERATOR
        FLAG = 1 then go to OR1 else if TEST FLAG =
        1 then go to C5 else go to C1;
      comment labels C1-C9 are the entries to command
        generators;
OR1:  if TEST FLAG = 1 then go to C6 else go to C2;
OR2:  if OPERATOR FLAG = 1 then go to OR3 else if
        TEST FLAG = 1 then go to C8 else go to C4;
OR3:  if TEST FLAG = 1 then go to C7 else go to C3;
AND:  if NHTI ≠ 0 then NOT ROUTINE else if AD-
        DRESS ≠ 0 then BUILD COMMAND ('CLA',
        ADDRESS);
      if POP = 13 then go to AND 2 else if OPERATOR
        FLAG = 1 then go to AND 1 else if TEST
        FLAG = 1 then go to C6 else go to C2;
AND 1: if TEST FLAG = 1 then go to C5 else go to C1;
AND 2: if OPERATOR FLAG = 1 then go to AND 3
        else if TEST FLAG = 1 then go to C7 else
        go to C3;
AND 3: if TEST FLAG = 1 then go to C8 else go to C4;
      comment the commands which are generated when-
        ever a Boolean operator (∧ or ∨) appears depend
        on only four things: (1) the operator, (2) the TEST
        FLAG, (3) the OPERATOR FLAG and (4)
        whether the previous operator (POP) was a right
        parenthesis. These combinations are produced by
        the command generators following the labels
        C1-C8 (there are only 8 different command
        sequences);
NOT:  LOAD NOT FLAG; FIND NEXT DELIMITER;
        go to DISTRIBUTE;
LF PARENTHESIS: LEVEL := LEVEL + 1; go to C9;
RT PARENTHESIS: LEVEL := LEVEL − 1; FIND NEXT
        DELIMITER; if NOP = 31 ∨ NOP = 32 then
        go to DISTRIBUTE else BUILD COMMAND
        ('RS', A); BUILD COMMAND ('RS', B); go to
        DISTRIBUTE;
C1:   BUILD COMMAND ('TT', B); BUILD COM-
        MAND ('ILB', A); FIND NEXT DELIMITER;
        go to DISTRIBUTE;
C2:   BUILD COMMAND ('TF', A); FIND NEXT
        DELIMITER; go to DISTRIBUTE;
C3:   BUILD COMMAND ('TF', A); BUILD COM-
        MAND ('ILB', B); BUILD COMMAND
        ('RS', A); BUILD COMMAND ('RS', B);
        FIND NEXT DELIMITER; go to DIS-
        TRIBUTE;
C4:   BUILD COMMAND ('RS', A); BUILD COM-
        MAND ('RS', B); BUILD COMMAND
        ('TT', B); BUILD COMMAND ('ILB', A);
        FIND NEXT DELIMITER; go to DIS-
        TRIBUTE;
C5:   BUILD COMMAND ('TF', B); BUILD COM-
        MAND ('ILB', A); FIND NEXT DELIMITER;
        go to DISTRIBUTE;
C6:   BUILD COMMAND ('TT', A); FIND NEXT
        DELIMITER; go to DISTRIBUTE;
C7:   BUILD COMMAND ('TT', A); BUILD COM-
        MAND ('ILB', B); BUILD COMMAND
        ('RS', B); BUILD COMMAND ('RS', A);
        FIND NEXT DELIMITER; go to DIS-
        TRIBUTE;
C8:   BUILD COMMAND ('RS', B); BUILD COM-
        MAND ('RS', A); BUILD COMMAND ('TF', B);
        BUILD COMMAND ('ILB', A); FIND NEXT
        DELIMITER; go to DISTRIBUTE;
C9:   BUILD COMMAND ('SL', A); BUILD COM-
        MAND ('SL', B); FIND NEXT DELIMITER;
        go to DISTRIBUTE;
      comment when BUILD COMMAND is called and
        the actual parameters correspond to a type 2 or
        type 3 IL command the address portion of the
        command is always a dummy address. Only two
        different dummy addresses are needed for the
        commands generated by BOOLEAN ROUTINE
        and these have been indicated by A and B;
EXIT: FIND NEXT DELIMITER end Boolean routine;
        go to EXECUTIVE; . . .

RELATIONAL ROUTINE:
      procedure RELATION TEST; comment after
        the two sides of a relational expression have been
        compared (the left always subtracted from the
        right) the contents of the accumulator must be
        tested. This procedure supplies the appropriate
        test for each of the relational operators. The
        relational operator under consideration is indi-
        cated by the value of RELATION TEST FLAG.
        There are two possible tests for each relational
        operator depending upon whether the delimiter
        which bounds the relational expression on the
        right is a Boolean ∨ or ∧. The delimiter then
        has the same effect as the operator ∧;
      begin integer K; K := RELATION TEST
        FLAG − 40; switch K := EQUAL, NOT
        EQUAL, GREATER THAN, GREATER THAN
        OR EQUAL, LESS THAN, LESS THAN OR
        EQUAL;
EQUAL: if END TEST FLAG = 0 then BUILD COMMAND
        ('TNZ', A) else BUILD COMMAND ('TZ', B);
        go to RETURN;
NOT EQUAL: if END TEST FLAG = 0 then BUILD COM-
        MAND ('TZ', A) else BUILD COMMAND
        ('TNZ', B); go to RETURN;
GREATER THAN: if END TEST FLAG = 0 then BUILD
        COMMAND ('TZP', A) else BUILD COM-
        MAND ('TN', B); go to RETURN;
GREATER THAN OR EQUAL: if END TEST FLAG = 0
        then BUILD COMMAND ('TP', A) else BUILD
        COMMAND ('TZN', B); go to RETURN;
LESS THAN: if END TEST FLAG = 0 then BUILD COM-
        MAND ('TZN', A) else BUILD COMMAND
        ('TP', B); go to RETURN;
LESS THAN OR EQUAL: if END TEST FLAG = 0 then
        BUILD COMMAND ('TN', A) else BUILD
        COMMAND ('TZP', B);
RETURN: end Relation Test procedure;
```

```
MAIN ENTRY: NEGATION FLAG := 1; LEVEL := LEVEL
            + 1; ARITHMETIC; RELATION TEST
            FLAG := NOP; LEVEL := LEVEL - 1;
            NOP := 21; FIND NEXT DELIMITER;
            ARITHMETIC; go to BB;
BB:         if NOP = 31 then go to CC;
            if NOP = 53 then go to DD else RELATION
              TEST; go to RELATION EXIT;
            comment when control reaches BB the delimiter
              which bounds the relational expression on the
              right has been encountered.
CC:         END TEST FLAG := 1; RELATION TEST;
              BUILD COMMAND ('ILB', A); go to RELA-
              TION EXIT;
DD:         RELATION TEST; BUILD COMMAND
              ('ILB', B);
RELATION EXIT: FIND NEXT DELIMITER end Relation
              Routine; go to EXECUTIVE . . .
```

Boolean Assignment Statements. Consider the Boolean assignment statement

(3.2) $\qquad K := M \wedge N \vee P \cdots .$

Assume that the expression on the right of (3.2) has been compiled according to the BOOLEAN ROUTINE. The object program can be completed by adding the following sequence of commands.

```
FALSE EXIT:  ILB   A
             CLA   false
             STO   K
             TC    C
TRUE EXIT:   ILB   B
             CLA   true
             STO   K
             ELB   C
               ⋮
```

The ILB A and ILB B commands load the locations at which they appear into the address portions of the corresponding transfer commands preceding them (in the object program for the Boolean expression M $\wedge$ N $\vee$ P). A third dummy address C is needed to jump over the **true** alternative in case the Boolean expression is false. The ELB C loads the location at which it appears into the address portion of TC C.

Conditional Statements. Consider a conditional statement such as

(3.3) **if** B_1 **then** S_1 **else if** B_2 **then** S_2 **else** S_3; $S_4 \cdots .$

Again we assume that the Boolean expressions B_1 and B_2 are compiled according to BOOLEAN ROUTINE. When a delimiter **if**, **then** or **else** appears, the following commands are entered in the object program:

```
(3.4) if     SL   A
             SL   B

(3.5) then   ILB  B

(3.6) else   TC   C
             ILB  A
             RS   A
             RS   B
```

In addition, the commands

```
(3.7) ;      ILB  C
             ILB  A
```

are entered in the object program when the semicolon following the complete conditional statement is encountered.

The ILB B in the commands for **then** represents the TRUE EXIT. The ILB A in the commands for **else** represents the FALSE EXIT. The ILB C (3.7) connects the TC C (3.6) to provide the necessary jump if the statement following **then** is executed. The ILB A is necessary in the commands for the semicolon (3.7) in case the conditional statement is only an if clause (no **else** appears in the statement); the semicolon then becomes the FALSE EXIT. The SL A and SL B commands for **if** and the RS A and RS B for **else** make the algorithm valid for any level of conditional statement (including conditional statements after **then** which is not presently allowed in the Algol 60 report). These commands have the effect of enclosing the different levels in parenthesis.

It should be noticed that only three different dummy addresses are necessary for any level of conditional statements, including the Boolean expressions within them.

4. Examples

Consider the Boolean expression:

(4.1) $\quad b \vee \neg\,(e \vee \neg\,(x < y \wedge \neg\, g \vee h) \wedge k) \vee m + n = p + q.$

The program compiled by the BOOLEAN ROUTINE and the RELATIONAL ROUTINE for this example is given below. The values of NOT FLAG, TEST FLAG and OPERATOR FLAG are included for each transfer command generated.

Command Location	*Command*		*NOT FLAG*	*TEST FLAG*	*OPERATOR FLAG*
1		CLA b			
2	TT	B	0	0	0
3	SL	A			
3	SL	B			
3		CLA e			
4	TT	A	1	1	1
5	SL	A			
5	SL	B			
5		CLS x			
6		ADD y			
7	TZN	A	0	0	0
8		CLA g			
9	TF	B	1	1	0
10	ILB	A			
10		CLA h			
11	RS	A			
11	RS	B			
11	TT	B			
12	ILB	A			
12		CLA k			
13	RS	A			
13	RS	B			
13	TF	B	1	1	0
14	ILB	A			
14		CLS n			

```
15        SUB  m
16        ADD  p
17        ADD  q
18    TNZ A
19    TC  B
.
.
.
50    ILB  B     (TRUE EXIT)
.
.
.
100   ILB  A     (FALSE EXIT)
```

For clarity, symbols have been used instead of assigning numerical addresses to b, e, x, y, etc., and all commands involving the dummy addresses A and B have been indented. The command location for any type 3 command is the same as that of the next type 1 or type 2 command since the type 3 commands do not appear in the actual machine coding.

The following coding is the result after the type 3 commands have been executed according to their definition in Appendix I.

```
1     CLA  b
2     TT   50
3     CLA  e
4     TT   12
5     CLS  x
6     ADD  y
7     TZN  10
8     CLA  g
9     TF   50
10    CLA  h
11    TT   50
12    CLA  k
13    TF   50
14    CLS  n
15    SUB  m
16    ADD  p
17    ADD  q
18    TNZ  100
19    TC   50
.
.
.
50  (TRUE EXIT)
.
.
.
100 (FALSE EXIT)
```

RELATIONAL ROUTINE forced ARITHMETIC to compile the two relational expressions $x < y$ and $m + n = p + q$ as if they were the arithmetic expressions $-(x) + y$ and $-(m + n) + p + q$.

A good check on the BOOLEAN ROUTINE is that it compiles the same program as above for the expression

$$b \vee (\neg e \wedge (x < y \wedge \neg g \vee h) \vee \neg k) \vee m + n = p + q \tag{4.2}$$

which is the equivalent of (4.1).

Conclusions

The method described in this paper for compiling Boolean expressions is an alternative to the usual method which would compile an object program that performs all logical operations indicated in the expression. The method presented in this paper will compile an object program which tests the minimum number of logical variables in determining the value of the expression.

The BOOLEAN ROUTINE in conjunction with RELATIONAL ROUTINE will compile programs for a Boolean expression of any complexity involving the operators $\vee$, $\wedge$, $\neg$. The logical variables in the expression can also be simple or compound relational expressions of any complexity. The method is especially appropriate for compiling Boolean expressions involving a large number of terms, several levels of logic within the expression or relational expressions.

This method can easily be used within a compiler which produces a particular machine coding rather than the intermediate language described in Appendix I. It simply means that the information retained by the type 2 and type 3 commands in an IL program must now be kept in lists within the compiler. In this case the intermediate language can still be used as a concise and convenient means for describing the compiler routines.

APPENDIX I

The intermediate language (IL) is similar to most symbolic codes in that it consists of a number of individual instructions which indicate operations to be performed on data or on the program. There are five types of commands in the language of which three will be presented here. Types 4 and 5 which concern address modification, looping operations and input-output operations will not be discussed for they are not involved in the BOOLEAN ROUTINE or RELATIONAL ROUTINE.

The type 1 commands perform the arithmetic operations of add, subtract, multiply and divide. It is assumed that the result of any of these operations remains in a storage cell called ACCUMULATOR.

TYPE 1

Command	*Operation*
CLA ADDR	(ADDR) → (AC)
ADD ADDR	(ADDR) + (AC) → (AC)
SUB ADDR	(AC) − (ADDR) → (AC)
ISB ADDR	(ADDR) − (AC) → (AC)
CLS ADDR	− (ADDR) → (AC)
MPY ADDR	(AC) × (ADDR) → (AC)
DIV ADDR	(AC)/(ADDR) → (AC)
IDV ADDR	(ADDR)/(AC) → (AC)

(ADDR) means "the contents of ADDR."

Integer arithmetic is assumed in the above operations. A corresponding set of commands exists for floating point arithmetic.

The type 2 commands include the usual conditional and unconditional transfers of control within a program. However, the commands are always written with a dummy address, rather than an absolute location. These dummy addresses are replaced by absolute locations by the type 3 commands to be described in the next paragraph.

TYPE 2

Command			Operation
TC	DUMMY ADDRESS		Transfer control
TT	"	"	Transfer if (AC) is **true**
TF	"	"	Transfer if (AC) is **false**
TZ	"	"	Transfer if (AC) $= 0$
TNZ	"	"	Transfer if (AC) $\neq 0$
TP	"	"	Transfer if (AC) > 0
TZP	"	"	Transfer if (AC) $\geqq 0$
TN	"	"	Transfer if (AC) < 0
TZN	"	"	Transfer if (AC) $\leqq 0$

The type 3 commands perform the operation of completing the type 2 commands, i.e., they replace the dummy addresses of type 2 commands with absolute locations. The absolute location loaded into type 2 commands by a type 3 command is the location in the program at which the type 3 command appears. A type 3 command always contains a dummy address, and it can operate only on type 2 commands which have the same dummy address.

TYPE 3

Command	Operation
ILB DUMMY ADDRESS	Inclusive Load Backward. The location at which this command appears is loaded into the address portion of *all* previous type 2 commands with the same dummy address as the ILB command.
ILF DUMMY ADDRESS	Inclusive Load Forward. Loads all future type 2 commands with the same dummy address.
ELB DUMMY ADDRESS	Exclusive Load Backward. Loads only the most recent type 2 command with the same dummy address.
ELF DUMMY ADDRESS	Exclusive Load Forward. Loads only the next type 2 command with the same dummy address.
SL DUMMY ADDRESS	Stop Loading. Stops the loading of previous type 2 commands with the same dummy address by ILB or ELB commands. Stops the loading of future type 2 commands with the same dummy address by previous ILF or ELF commands.
RS DUMMY ADDRESS	Remove Stop. Removes the effect of the last previous SL command with the same dummy address.

The type 2 and type 3 commands are used to properly connect a compiled intermediate language program. These commands allow an IL program to retain information normally kept in lists within a compiler. There is an additional feature added by using these commands: type 1 commands can be added or deleted from any IL program without altering the proper connection of the program.

The SLANG System

R. A. Sibley

IBM Systems Center, Bethesda, Maryland

Early in 1960, a project was initiated for the purpose of developing a programming language suited to the task of writing compiler type programs. Through the use of such a language and its associated processors, it was hoped that all of the now classical advantages of a problem-oriented language might be put to work for the compiler writer. A language was developed. Referred to as SLANG, the notation is patterned after ALGOL "58" and has proved to be convenient for expressing many of those processes fundamental to automatic programming. More important, the project has focused attention on what appears to be an important area for programming research. This area of endeavor is concerned with the following observation:

It is possible to describe processes in a machine-independent language which are in themselves machine dependent.

Mathematical techniques are in a large measure machine-independent and thus systems such as FORTRAN have been successful. Compiling techniques have been extremely machine dependent. This has resulted in a great deal of difficulty whenever an attempt has been made to apply the methods of automatic programming to the construction of compilers.

In addition to the beginning of an understanding of the need for new compilation techniques, the work reported here has produced the beginnings of what would seem to be a contribution to the development of such techniques. Indeed, results indicate that it is not only possible to describe the compilation process for a problem-oriented language in a machine-independent manner, but that such a description can be translated into the machine language of any of a class of computers by a single program running on a single computer.

Concerning Machine Independence

Suppose that processors for a series of problem-oriented machine-independent languages POLMI_i, $i = 1, 2, \cdots, \nu$, are required. What advantages are to be gained by de-