ON THE PRACTICABILITY OF MANNA'S
METHOD OF VERIFYING THE TERMINATION
AND CORRECTNESS OF PROGRAMS

by

S. Droste

Department of Computer Science
Livingston College
Rutgers University
The State University of New Jersey,
New Brunswick, New Jersey 08903

Department of Computer Science Technical Report #11
October, 1971
I. INTRODUCTION

Program testing or debugging is one of the most time consuming tasks in the development of a computer program; sometimes requiring as much as 90% of the total time [15]. Errors occur in every phase of the development, ranging from errors in the syntax of the programming language (often simply a misplaced comma) to a logical error in the problem specification. Language processors adequately locate and flag the syntax errors; we will not discuss them here. We are much more interested in carefully written programs which are ready for computer testing. Debugging programs exist to aid the programmer in locating infinite loops, printing intermediate results, tracing the sequence of execution, etc., but these can really be considered computer-assisted console debugging procedures. In a batch environment, the use of these techniques requires foresight as to what might go wrong as the requests are made prior to execution; it is not always easy to locate the proper check points, nor is it easy to know what information to request.

A general procedure followed in program testing is to apply a set of simple test data to the program and to use hand calculations to check the results. The selection of this set of test data is often a non-trivial problem. Further, a successful run does not guarantee that the program will terminate and give correct results for all values of the input data. All possible branches in the program should be checked with a variety of data. For larger programs, thorough testing of every branch is impractical, if not impossible. It is quite common to find that a "proven" program will run successfully many times and
then suddenly fail for a given input. A program is rarely known to be absolutely correct. For this reason program maintenance continues to consume a great deal of programmer and computer time.

Clearly, a method of proving the termination and correctness of programs would be desirable. We would like a program which would accept as input the program to be tested and its specifications, and which would determine if, for all inputs:

1) the program terminates and delivers correct results,
2) the program terminates, but delivers incorrect results,
3) the program terminates, but delivers no results, or
4) the program does not terminate.

Ideally this program would also, in the case of error, locate the offending portion of the program. We would further require that the computer time required for this procedure be reasonable.

The possibility of the development of such a verifying program is suggested by the work of Manna [9, 10], in which he relates the problems of termination and correctness of programs to equivalent problems in first-order logic. He has translated the question of the termination and correctness of programs into the question of the satisfiability of formulas of the first-order predicate calculus (or, equivalently, the validity). He has described an algorithm which will construct, for a given program (whose syntax and notion of execution (semantics) has been defined) or abstract program, a first-order formula characterizing its execution. He shows that the problem of proving the termination and correctness of the program or abstract program is equivalent to proving the validity of the formula. The validity problem is undecidable. However, it is semidecidable. (There are subclasses of formulas
for which validity is decidable and Manna shows that there exist subclasses of abstract programs which are equivalent to the decidable subclasses of formulas.) Any decision or semidecision procedure that applies to formulas in first-order logic can be applied to prove the termination and correctness of programs and abstract programs. One such procedure is based on the resolution principle [11, 13].

The purpose of this essay is to examine the practicality of this method for proving the termination and correctness of programs by determining the ease of constructing the first-order formula and the applicability of a semidecision procedure to this formula. Because a large number of "tested" Fortran programs in varying degrees of complexity were readily available, Fortran was selected as the programming language. By "tested", we mean simply that the programs terminated and gave correct results for one or more sets of input data. Resolution was selected as the semidecision procedure because it is machine-oriented and programs utilizing this principle exist.

The discussion is limited to the termination problem of programs. Manna has extended his results to include correctness and equivalence; he claims that they are reducible to the termination problem. The programs considered are limited to simple iterative programs written in Fortran.

To make this paper self-contained, Chapter II contains the description of a first-order predicate calculus and a description of Robinson's resolution procedure [11, 13]. Chapter III contains a summary of Manna's work, his principal definitions and theorems [9].

If the resolution procedure is to be applicable, the time required to reach a solution must be reasonable; in general, this cannot be determined from the formula itself. However, the first
level of the procedure is to obtain all possible resolvents from the original set of clauses, making no substitutions for individual variables. The number of steps of resolution required for this first level is determined by the number of pairs of matching complementary literals contained in the original set of clauses. This provides a first estimate on the amount of time required. The minimum number of steps will be taken if the empty clause is a member of the set of first resolvents. The time required to obtain this set of first resolvents is therefore among the first factors used in evaluating the practicality of this method of verifying the termination of programs.

In Chapter IV a Fortran program is examined in detail; rules for constructing the graph of the program from the Fortran source statements are presented. An algorithm to count the number of matching complementary literals appearing in the formula of a program without constructing the actual program is presented. A number of Fortran programs were then examined; the number of steps of resolution required for the first level was determined for each program. In all cases the number of steps and therefore the time required is reasonable. However, it is shown that, except for exactly one program, a solution in this first level is not possible (i.e., the empty clause will not be the resolvent of two original clauses).

The question of the applicability of resolution to these formulas is not fully answered; additional information is required. An experiment to study the problem further and to collect the necessary data is described.
II. THE FIRST-ORDER PREDICATE CALCULUS

This chapter contains the primitive basis of a first-order predicate calculus with arithmetic and identity; resolution is the rule of inference. The system described here is similar to those of Church [2], Thomason [16], and Robinson [11, 13].

20 The primitive basis.

20.1 Definitions. The syntactic units from which the language is constructed are the following:

1) Improper symbols:
   , ( ) & = \lor \exists \equiv

2) Individual variables:
   x x_1 x_2 \ldots
   y y_1 y_2 \ldots
   z z_1 z_2 \ldots

3) Function variables, of degree n for each n \geq 0:
   f^n_1 f^n_2 \ldots
   \varepsilon^n_1 \varepsilon^n_2 \ldots
   \eta^n_1 \eta^n_2 \ldots

4) Function constant: The successor function is denoted by:
   \sigma

5) Individual constants:
   a a_1 a_2 \ldots
   b b_1 b_2 \ldots
   c c_1 c_2 \ldots

0 is an individual constant, \sigma(0) is 1, \sigma(1) is 2, etc.
6) **Predicate variables**, of degree $n$, $n > 0$:

\[
p^n, p_1^n, p_2^n, \ldots \\
q^n, q_1^n, q_2^n, \ldots \\
r^n, r_1^n, r_2^n, \ldots
\]

**20.2 Convention:** Bold-face letters serve as metavariables whose range is the set of the corresponding primitive symbols of the language. In addition, the bold-face capitals $A$, $B$, and $C$ serve as metavariables whose range is the set of well-formed formulas of the object language. Hereafter, $A$ and $B$ are used as metavariables whose range is terms. Parenthesis, superscripts, and subscripts are omitted whenever their omission causes no confusion. Church's dot notation and left association are used in the usual way.

**20.3 Definition.** The class of *terms* is defined inductively as follows:

1) Each individual variable and each individual constant is a term.

2) If $t_1, t_2, \ldots, t_n$ ($n > 1$) are terms, then so is $t_i^n(t_1, t_2, \ldots, t_n)$.

**20.4 Definition.** The class of *atomic formulas* is defined inductively as follows:

1) The following are atomic formulas:

\[
T, F, p_1^0, q_1^0, r_1^0
\]

2) If $t_1, t_2, \ldots, t_n$ ($n > 1$) are terms, then the expressions

\[
p_1^n(t_1, t_2, \ldots, t_n), q_1^n(t_1, t_2, \ldots, t_n), r_1^n(t_1, t_2, \ldots, t_n)
\]

are atomic formulas.

3) If $t_1$ and $t_2$ are terms, the expression

\[
t_1 = t_2
\]

is an atomic formula.
20.5 Definition. The class of well-formed formulas (wffs) is
defined inductively as follows:
1) An atomic formula is a well-formed formula (wff).
2) If $\mathcal{A}$ is a wff, then so are
   $\neg \mathcal{A}$
   $(\exists \mathcal{A})$ (\( \chi \) is said to be existentially quantified)
   $(\forall \mathcal{A})$ (\( \chi \) is said to be universally quantified)
3) If $\mathcal{A}$ and $\mathcal{B}$ are wffs, then so are
   $(\mathcal{A} \in \mathcal{B})$  $(\mathcal{A} \supset \mathcal{B})$  $(\mathcal{A} \land \mathcal{B})$  $(\mathcal{A} \equiv \mathcal{B})$

20.6 Definition. An occurrence of \( \chi \) in a wff \( \mathcal{A} \) is a bound
occurrence if it is in a part of \( \mathcal{A} \) which is a wff of the form $(\exists \mathcal{A})_\beta$
or $(\forall \mathcal{A})_\beta$. An occurrence of \( \chi \) which is not bound is called a free
occurrence. \( \chi \) is free in \( \mathcal{A} \) if it has at least one free occurrence
in \( \mathcal{A} \).

20.7 Substitution and Replacement [16].
1) Let \( \chi \) be an expression, and let $\mathcal{Z}$ and $\mathcal{T}$ be any terms. The
result $\chi[\mathcal{Z}/\mathcal{T}]$ of substituting $\mathcal{Z}$ for $\mathcal{T}$ in $\chi$ is the expression obtained
by putting an occurrence of $\mathcal{Z}$ for every free occurrence of $\mathcal{T}$ in $\chi$.
Furthermore, let $\mathcal{X}$ be any individual variable; then the result $\chi[\mathcal{X}/\mathcal{T}]$ of
substituting $\mathcal{X}$ for $\mathcal{T}$ in $\chi$ is the expression obtained by putting an
occurrence of $\mathcal{X}$ for every free occurrence of $\mathcal{T}$ in $\chi$.
2) Let $\mathcal{A}$ be a formula and $\mathcal{Z}$ and $\mathcal{T}$ be terms. A formula $\mathcal{B}$ is the
result $\mathcal{A}[\mathcal{Z}/\mathcal{T}]$ of replacing $\mathcal{T}$ by $\mathcal{Z}$ in $\mathcal{A}$ if $\mathcal{B}$ differs from $\mathcal{A}$ only in
exhibiting occurrences of $\mathcal{Z}$ at zero or more places where $\mathcal{A}$ exhibits
free occurrences of $\mathcal{T}$.

20.8 Example. Let $\mathcal{A}$ be the formula $p(x, c) \supset (y)(y = x \supset p(y, c))$.
Then, $p(a, c) \supset (y)(y = a \supset p(y, c))$ is the result of substituting $a$ for $x$. 
in \( A \); and \( p(a, c) \equiv (y)(y = x \equiv p(y, c)) \) is a result of replacing \( x \) by \( a \) in \( A \).

20.9 Axioms. The system has infinitely many axioms, among them all instances of the following schemata:

1. \( A \rightarrow B \rightarrow A \)
2. \( (A \rightarrow B = C) \rightarrow (A \rightarrow B = A \rightarrow C) \)
3. \( \neg A \rightarrow \neg B \rightarrow A \rightarrow B \)
4. \( (x)(A \rightarrow B) = (x)A \rightarrow (x)B \) provided \( x \) is free in \( B \) and there are no free occurrences of \( x \) in \( A \).
5. \( (x)(A \rightarrow B) = (x)A \rightarrow (x)B \) provided \( x \) is free in \( A \) and \( B \).
6. \( (x)A = A[x/\xi] \) provided all variables in \( \xi \) occur free in \( A \) in exactly those places where \( x \) occurs free in \( A \).
7. \( x = L \)
8. \( x = x = A = A[y/x] \)

Also, any universalization of any instance of the above.

In addition, the following axioms from Peano [5]:

9. \( \neg(\sigma(x) = 0) \)
10. \( \sigma(x) = 0 \rightarrow x = 0 \)
11. \( \mathbb{P}(0) \& (x)(\mathbb{P}(x) \rightarrow \mathbb{P}(\sigma(x))) = (x)\mathbb{P}(x) \)

(the axiom of induction)

Lastly, the axiom of extensionality:

12. \( (x)(p(x) = q(x)) \rightarrow p = q \)

20.10 Definitions.

A wff is called quantifier-free if it contains no occurrence of \( (x) \) or \( (\exists x) \).

A wff is in prenex normal form if it begins with a sequence of quantifiers \( (x) \) and \( (\exists x) \) in which no variable occurs more than once (called the prefix), and if the sequence is followed by a quantifier-free wff (called the matrix).
A wff is in **functional form** if its prenex normal form has only universal quantifiers in the prefix. To convert to this form, each existential quantifier in the prefix is removed and each occurrence of an existentially quantified variable in the matrix is replaced by an occurrence of a determinable function constant which depends on the universally quantified variables that preceded the existential quantifier in the prefix.

21 **Resolution** [11, 15].

In order to state the rule of resolution, it is necessary to give some specific definitions. The rule of resolution is given in 21.2.

21.1 **Definitions** [15].

1) An **atom** is an atomic formula.

2) A **literal** is either an atom or the negation of an atom. The literals $\bar{A}$ and $\neg \bar{A}$ are **complementary**; each is the **complement** of the other.

3) A **clause** is a disjunction of literals; the empty clause is denoted by $\Box$. The clause is considered simply as the set of its disjuncts because the order and multiplicity of the disjuncts in a disjunction are immaterial.

4) A **substitution** is an operation $\theta$ which can be performed on an expression $E$ to obtain another expression $E^\theta$; the operation consists of replacing each occurrence in $E$ of each of a list $x_1, \ldots, x_n$ of distinct individual variables by an occurrence of the corresponding term in a list $t_1, \ldots, t_n$ of (not necessarily distinct) terms. It is always assumed that $t_i$ is different from $x_i$. We write $\theta = [t_1/x_1, \ldots, t_n/x_n]$. The null substitution is allowed. If $C$ is a set of expressions,
\( C \) represents the set of all expressions \( E \), for \( E \) in \( C \).

The composition \( \circ \lambda \) of two substitutions is the substitution \( \mu \) such that \( E \mu = (E \circ \lambda) \lambda \) for all \( E \). Composition of substitutions is associative.

5) An expression \( Y \) is an instance of an expression \( X \) is \( Y = X \theta \) for some substitution \( \theta \).

6) A substitution \( \theta = [x_1/x_1, \ldots, x_n/x_n] \) is invertible, and has the inverse \( \theta^{-1} = [x_1/x_1, \ldots, x_n/x_n] \), if \( [x_1/x_1, \ldots, x_n/x_n] \) is a substitution, that is, if \( x_1, \ldots, x_n \) are distinct individual variables.

7) An expression \( Y \) is a variant of an expression \( X \) if \( Y \) is an instance \( X \theta \) of \( X \) for some invertible \( \theta \) such that \( X = Y \theta^{-1} \). [13]

8) Let \( X \) be a set of expressions and \( \theta \) a substitution. Let \( P \) be the partition of \( X \) determined by the rule that \( E \) and \( F \) are in the same block of \( P \) if and only if \( EF = FE \). We say that \( P \) is induced in \( X \) by \( \theta \).

9) Let \( X \) be a set of expressions and \( P \) a partition of \( X \). We say that \( P \) is a unifiable partition of \( X \) if and only if there is some substitution \( \theta \) which induces \( P \) in \( X \).

10) Let \( X \) be a finite set of expressions. A set \( \{ \theta_1, \ldots, \theta_n \} \) of substitutions is said to be a basis of \( X \) if for each unifiable partition \( P \) of \( X \) there is exactly one \( \theta_1 \) in the set which induces \( P \) in \( X \).

11) Let \( X \) be a finite set of expressions. A basis \( \{ \theta_1, \ldots, \theta_n \} \) of \( X \) is said to be a prime basis of \( X \) if, for any basis \( \{ \theta_1, \ldots, \theta_n \} \) of \( X \), we have

\[ \{ \theta_1, \ldots, \theta_n \} = \{ \theta_1 \lambda_1, \ldots, \theta_n \lambda_n \} \]

for some set \( \{ \lambda_1, \ldots, \lambda_n \} \) of substitutions.
12) Every finite set $X$ of expressions has a prime basis; moreover, given $X$, we can compute a prime basis of $X$, and conversely. These computations are made by means of the prime basis algorithm [c.f., 15, pp. 90, 91]; we do not present this in detail here; the preceding definitions and the algorithm are required to state the generalized resolution principle in 21.2.

13) If $\{x_1^{n_1}, \ldots, x_{n_k}^{n_k}\} = Y$, then we can find a clause $X$ such that $\{x_1^{n_1}, \ldots, x_{n_k}^{n_k}\} \supset X$ and a substitution $\lambda$ such that $Y = X\lambda$.

21.2 The single rule of inference is the generalized resolution principle [13]:

From $(A_1 \lor B_1), \ldots, (A_n \lor B_n)$ one may infer the resolvent $(A_1' \lor \ldots \lor A_n' \lor B_1')$, provided that $(B_1', \ldots, B_n') \supset B'$, where $(A_1', \ldots, A_n')$, $(B_1', \ldots, B_n')$ are variants of the clauses $(A_1 \lor B_1), \ldots, (A_n \lor B_n)$, $\delta$ is a member of a prime basis of the set $T$ of all terms which appear in $(A_1' \lor B_1'), \ldots, (A_n' \lor B_n')$, and $B'$, where $B$ is a disjunction.

21.3 Resolution is a machine-oriented inference principle for the first-order predicate calculus that is effective, sound, and complete [11]. It is based on notions of unsatisfiability and refutation rather than validity and proof. Robinson's proof procedure is based on the resolution formula:

$$[(p \lor A) \land (\neg p \lor B) \supset (A \lor B)]$$

where $A$ and $B$ are disjunctions of literals. To apply this procedure, we assume the wff to be in prenex normal form with the quantifier-free matrix in conjunctive normal form. The quantifier prefix consists only of universal quantifiers binding each variable in the matrix. The matrix, then, is a conjunction of clauses, which will simply be considered a set of clauses, each conjunct one member of the set.
21.4 Definition [13]. If \( S \) is any set of clauses, the resolution of \( S \), denoted by \( R(S) \), is the set of clauses consisting of the members of \( S \) together with all resolvents of all pairs of members of \( S \). The \( n^{th} \) resolution of \( S \), denoted by \( R^n(S) \), is defined for each \( n \geq 0 \) as follows:

1) \( R^0(S) = S \)
2) \( R^{n+1}(S) = R(R^n(S)) \) for \( n \geq 0 \).

The first resolution of \( S \), for the purposes of this paper, is defined as the set of clauses consisting of the members of \( S \) together with all ground resolvents of all pairs of members of \( S \). The ground resolution principle is:

From \( (A_1 \lor B_1), \ldots, (A_k \lor B_k) \) one may infer the ground resolvent:

\[
(A_1 \lor \ldots \lor A_k), \text{ whenever } \{B_1, \ldots, B_k\} \supset \emptyset
\]

In other words, the ground resolvent consists of the set of all literals from the clauses except the matching complementary pairs. No substitutions are made for the individual variables.

21.5 Definition. A refutation of a set \( S \) of clauses is a finite sequence \( B_1, \ldots, B_n \) of clauses such that:

1) each \( B_i, 1 \leq i \leq n \), is either in \( S \), is an axiom, or is a resolvent of two earlier clauses in the sequence, and
2) \( B_n \) is \( \emptyset \).

21.6 Theorem. Robinson's generalized resolution theorem is:

For any finite set \( X \) of clauses; if \( X \supset \emptyset \), then, for some \( k \geq 1 \)

\[
\{X_1 \theta_1, \ldots, X_k \theta_k\} \supset \emptyset, \text{ where } X_1, \ldots, X_k \text{ are in } X \text{ and } \theta_1, \ldots, \theta_k \text{ are substitutions.}
\]

And his completeness theorem for the generalized resolution principle is:

For any finite set \( X \) of clauses; if \( X \supset \emptyset \), then there is a proof of \( X \supset \emptyset \) in which each inference is an application of the generalized resolution principle.
21.7 Robinson shows that it is immediate from the resolution theorem that a finite set $S$ of clauses is unsatisfiable if and only if there is a refutation of $S$.

21.8 For this particular study we are interested in the number of steps of resolution required to obtain the first resolution; if the empty clause occurs in this set, a minimum solution has been reached. A step of resolution is defined as the following procedure:

1) select a pair of matching complementary literals, and

2) apply the ground resolution principle to all possible pairs of clauses which contain this pair of complements (i.e., $k=2$).

21.9 The resolution procedure therefore consists of computing, given the finite set $S$ of clauses as input, the sequence of sets $S$, $R(S)$, $R^1(S)$, ... until one is encountered which either contains $\Box$, in which case the set $S$ is unsatisfiable, or is equal to its predecessor, in which case the set $S$ is satisfiable. But it is known that for some inputs, this procedure will not terminate in either of these two ways, but will continue computing forever. In addition to this theoretical limitation, there are two pragmatic limitations associated with resolution:

1) a memory limitation as the list of clauses to be retained grows, and

2) the time required to complete the search successfully (if it can be completed).

Various strategies have been developed to retard the growth of the list of clauses and to decrease the search time [7, 8, 17, 18, 19].

21.10 An example of the use of the resolution procedure is given in 34.14.
III. A SUMMARY OF MANNA'S WORK [9].

30. Introduction.

30.1 Manna relates the problems of termination and correctness of programs and abstract programs to equivalent problems in first-order predicate calculus. He describes an algorithm which will construct for a given abstract program or program, a first-order formula characterizing its execution. The program terminates if and only if this formula is unsatisfiable. Known proof procedures for the first-order predicate calculus can thus be applied to prove the termination of programs and abstract programs. Manna claims that the correctness and equivalence problems of programs is reducible to the termination problem.

31 Manna's principal definition.

31.1 Definition [9]. An abstract program (or program schema) consists of:

1) A finite directed graph \(<V,L,A>\) (V (vertices), L (labels), and A (arcs) are nonempty finite sets; \(A \subseteq V \times L \times V\),) such that:
   a) there exists exactly one vertex \(S \in V\) with in-degree 0 (i.e., with no arcs leading to \(S\)), called the start vertex;
   b) there exists exactly one vertex \(H \in V\) with out-degree 0 (i.e., with no arcs leading from \(H\)), called the halt vertex; and
   c) every vertex \(v \in V\) is on some path that joins \(S\) and \(H\).

2) a) A set of \(m (m \geq 0)\) distinct individual variables
   \[ \hat{\gamma} = \{\gamma_1, \gamma_2, \ldots, \gamma_m\}, \] called input variables; and
b) A set of \( n \) (\( n \geq 1 \)) distinct individual variables
\[ \bar{x} = (x_1, x_2, \ldots, x_n) \] called program variables.

3) With each \( \alpha = (v, \bar{x}, v') \in \mathcal{A} \) there is associated

a) a wff \( \phi_{\alpha} \) called the test predicate of \( \alpha \); and
b) an \( n \)-tuple \( \bar{t}_{\alpha} = (t_1^{(\alpha)}, t_2^{(\alpha)}, \ldots, t_n^{(\alpha)}) \) of terms called
the assignment function of \( \alpha \). The intended interpretation is: \( v \): if \( \phi_{\alpha} \) then [replace simultaneously each variable
\( x_i \) by \( t_i^{(\alpha)} \) and go to \( v' \)].

The wff \( \phi_{\alpha} \) does not contain any predicate variables. In addition, the
wff \( \phi_{\alpha} \) and the terms \( t_i^{(\alpha)} \) do not contain individual variables other than
\( \bar{x} \) and \( \bar{y} \). If \( v = S \) (i.e., \( \alpha \) is an arc leading from the start vertex),
the wff \( \phi_{\alpha} \) and the terms \( t_i^{(\alpha)} \) do not contain the program variables \( \bar{x} \).

In addition, an abstract program should satisfy the following restriction:

4) For every vertex \( v \) (\( v \neq H \)), if \( \alpha_1, \alpha_2, \ldots, \alpha_n \) is the set of
all arcs leading from \( v \), the set of the test predicates
\( \phi_{\alpha_1}, \phi_{\alpha_2}, \ldots, \phi_{\alpha_n} \) is:

a) complete, i.e., \( (\exists \bar{x})(\exists \bar{y})[\phi_{\alpha_1} \vee \phi_{\alpha_2} \vee \ldots \vee \phi_{\alpha_n}] \) is valid, and
b) mutually exclusive, i.e., \( (\exists \bar{x})(\exists \bar{y})[\phi_{\alpha_i} \land \phi_{\alpha_j}] \) is
unsatisfiable for every pair \((i,j)\), \( i \neq j \).

Thus, an abstract program may be thought of as representing a family of
(real) programs as none of the functions, predicates, or constants are
specified. An example of an abstract program is given in 31.2. By
specifying an interpretation for the symbols, a program of this family
is obtained.

31.2 Example [9]. Figure 1 represents an abstract program \( AP^* \).
This could be represented as a "Fortran" program in which the number and
type of statements is known, but the expressions are given in "general"
form. An example of such a representation accompanies the abstract
program \( AP^* \) in Figure 1.
IF (exp_1) GO TO 6
X=exp_2
IF (exp_1) GO TO 4
X=exp_3
IF (exp_1) GO TO 5
STOP
6 X=exp_4
4 IF (exp_1) STOP
X=exp_3
GO TO 4
5 X=exp_2
GO TO 4

Figure 1. The abstract program AP* and the corresponding Fortran "general" program.
31.3 Definition [9]. An interpretation $I$ of an abstract program $AP$ consists of a nonempty set of elements $D_I$ (called the domain of the interpretation) and assignments to the variables of $AP$:

1) to each function variable $f^n_i$ which occurs in $AP$ we assign a total function mapping $(D_I)^n$ into $D_I$, and

2) to each predicate variable $P^n_i$ which occurs in $AP$ we assign a total function mapping $(D_I)^n$ into $\{T,F\}$. 

Let $AP$ be an abstract program and $I$ an interpretation of $AP$. The pair $(AP,I)$ is called a program. Thus, a program $(AP,I)$ corresponds to the code that a programmer would write. The vertices of the directed graph correspond to the IF statements of the program and the set of test predicates at the vertex are taken from the expression (logical or arithmetic) of the statement. Vertices and test predicates are also described by DO-loops and computed GO TO statements. A simple continuation vertex with the single test predicate $T$ is caused by certain conditions among a sequence of assignment statements. The assignment functions correspond to the calculations performed by a sequence of assignment statements. A more detailed description of the relationship between the graph and the statements of the Fortran program will be found in Chapter IV. An example of a program $(AP^*,I^*)$ is given in 31.4.

31.4 Example [9]. Figure 2 represents a program $(AP^*,I^*)$ which was obtained from the abstract program $AP^*$ by making the following assignments to the symbols:

- $D$ is the integers
- $f(x)$: $x+1$
- $p(x)$: $x=0$
- $a$: $-1$
Figure 2. The program \((AP^*, I^*)\).
A corresponding Fortran program is:

```
IF (Y.EQ.0) GO TO 3
X = -1
IF (X.EQ.0) GO TO 4
X = X + 1
IF (S.EQ.0) GO TO 5
STOP
3 X = Y
4 IF (X.EQ.0) STOP
   X = X + 1
   GO TO 4
5 X = -1
   GO TO 4
```

Another program of this family (AP*,I**) can be obtained by making the following assignments to the symbols of the abstract program AP*:

- D is the integers
- f(x): 2x-1
- p(x): x<0
- a: 5

A corresponding Fortran program is:

```
IF (Y.LT.0) GO TO 3
X = 5
IF (X.LT.0) GO TO 4
X = 2 * X - 1
IF (X.LT.0) GO TO 5
STOP
3 X = Y
4 IF (X.LT.0) STOP
   X = 2 * X - 1
   GO TO 4
5 X = 5
   GO TO 4
```

The representation of (AP*,I**) is the same as that of (AP*,I*) with the obvious changes of predicate, function, and constant assignments.

31.5 Definition [9]. Let (AP,I) be a program. Then the result obtained by assigning values \( \bar{\gamma} \), \( \bar{\gamma} \in (D_I)^m \) for the input variables \( \bar{y} \) of the program is called the interpreted program \((AP,1,\bar{\gamma})\). Programs with no input variables (i.e., m=0) will be considered interpreted programs.
The interpreted program \((AP, I, \tilde{\gamma})\) defines an execution sequence \(<AP, I, \tilde{\gamma}>\) which is a (finite or infinite) sequence of triples
\[ (s^{(1)}, v^{(1)}, x^{(1)}), (s^{(2)}, v^{(2)}, x^{(2)}), (s^{(3)}, v^{(3)}, x^{(3)}), \ldots \]
where:

1) \((s^{(j)}, v^{(j)}, x^{(j)}) \in L \times V \times (D \cup \{T\})^n\) for every \(j \geq 1\),

2) \((s^{(1)}, v^{(1)}, x^{(1)})\) is the first triple in the sequence if and only if there exists an arc \(a = (s^{(1)}, v^{(1)}) \in A\) such that
\[ \phi_a(\tilde{\gamma}) = T \text{ and } x^{(1)} = \tilde{t}_a(\tilde{\gamma}). \]
\(\phi_a(\tilde{\gamma})\) and \(\tilde{t}_a(\tilde{\gamma})\) stand for the result of substituting \(\tilde{\gamma}\) for \(\tilde{\gamma}\) in \(\phi_a\) and \(\tilde{t}_a\).

3) \((s^{(j)}, v^{(j)}, x^{(j)})\) and \((s^{(j+1)}, v^{(j+1)}, x^{(j+1)})\) are two successive triples in the sequence if and only if there exists an arc \(a = (v^{(j)}, x^{(j+1)}, v^{(j+1)}) \in A\) such that
\[ \phi_a(\tilde{x}^{(j)}, \tilde{\gamma}) = T \text{ and } x^{(j+1)} = \tilde{t}_a(\tilde{x}^{(j)}, \tilde{\gamma}). \]
\(\phi_a(\tilde{x}^{(j)}, \tilde{\gamma})\) and \(\tilde{t}_a(\tilde{x}^{(j)}, \tilde{\gamma})\) stand for the result of substituting \(\tilde{x}^{(j)}\) for \(x^{(j)}\) and \(\tilde{\gamma}\) for \(\tilde{\gamma}\) in \(\phi_a\) and \(\tilde{t}_a\).

4) The sequence is finite, of length \(k \geq 1\), if and only if \(v^{(k)} = H\). In this case \(x^{(k)}\) is called the value of the execution sequence \(<AP, I, \tilde{\gamma}>\) and is denoted by \(val<AP, I, \tilde{\gamma}>\).

In other words, execution always starts at the start vertex, \(S\). On execution of the \(j^{th}\) step, \(j \geq 1\), control moves along the arc
\[ a = (v^{(j-1)}, x^{(j)}, v^{(j)})\], where \(v^{(0)} = S\), and \(\phi_a\) represents the condition that this arc is entered. The value of each program variable \(x_{\alpha}\) is replaced in the \(j^{th}\) step by the current value of \(t_{\alpha}^{(a)}\), simultaneously. So \(x^{(j)}\) represents the current value of the program variables \(\tilde{x}\) after executing the \(j^{th}\) step. Execution stops whenever control reaches the halt vertex, \(H\). Thus, an interpreted program corresponds to a single execution of the program and the value of the execution sequence contains the output of the program (which may or may not be transmitted in its entirety to an output device). An example of an interpreted program \((AP^*, I^*, 1)\) is given in 31.6.
31.6 Example [9]. By assigning the value 1 to the input variable y of the program \((A^*,I^*)\) of Section 31.4, we obtain the interpreted program \((A^*,I^*,1)\) represented in Figure 5. This defines the execution sequence \(<A^*,I^*,1>:\n\)

\[(1,1,-1), (3,2,0), (5,3,-1), (7,3,0), (8,4,1).\]

By assigning the value 2 to the input variable y of the program \((A^*,I^{**})\) of Section 31.4, we obtain the interpreted program \((A^*,I^{**},2)\), which defines the execution sequence \(<A^*,I^{**},2>:\n\)

\[(1,1,5), (3,2,9), (6,4,9).\]

The representation of \((A^*,I^{**},2)\) is that of \((A^*,I^*,1)\) with the obvious changes in predicate, function, constant, and input variable assignments.

31.7 Definition [9]. Let \((A^*,I,\gamma)\) be an interpreted program, and let \(v \in V\) be any vertex of \(A^*\). Let \(\delta\) be a specified total predicate from \((D_f^n)\) into \((T,F)\). Then

1) if for all \(\xi \in (D_f^n)\) vertex \(v\) is reached in state \(\xi\), then \(\delta(\xi) = T\), and \(\delta\) is called a valid predicate of \(v\) for \((A^*,I,\gamma)\).
2) if for all \(\xi \in (D_f^n)\) vertex \(v\) is reached in state \(\xi\) just in the case that \(\delta(\xi) = T\), then \(\delta\) is called a minimal valid predicate of \(v\) for \((A^*,I,\gamma)\).

A valid predicate is true of the values that arise during execution and also of other values. A minimal valid predicate is true only of the values that actually arise during execution.

31.8 Example [9]. The predicate \(x \leq 0\) is a valid predicate and the predicate \(x = -1\) is the minimal valid predicate of the vertex \(1\) for the interpreted program \((A^*,I^*,1)\).
Figure 5. The interpreted program \((AP^*, I^*, 1)\)
32 Manna's termination algorithm.

37.1 This section contains Manna's algorithm to construct a first-order formula from the abstract program AP or the program (ap,l). This formula is built from free-variable formulas, each representing the traversal of an arc in the abstract program, and the formula of the program AP is the universal quantification over the program variables of the conjunction of these formulas. Following this construction and the definition of termination, he proves that the abstract program or the program terminates if and only if this formula is unsatisfiable. His theorems are presented without proofs. An example of the construction is given using the program AP* of 31.2.

32.2 Algorithm [9]. Let AP be any abstract program with program variables \( \bar{x} = (x_1, \ldots, x_n) \), \( n \geq 1 \), and input variables \( \bar{y} = (y_1, \ldots, y_m) \), \( m \geq 0 \). Associate with every vertex \( v_i \) of AP a distinct n-adic predicate variable \( q_i \). For each arc \( \alpha = (v_i, e, v_j) \), define \( W_\alpha \) as:

\[
(q_i(\bar{x}) \& \phi_\alpha) = q_j(\bar{e}_\alpha)
\]

However, if \( v_i = S \) (i.e., \( v_i \) is the start vertex of AP), replace the occurrence of \( q_i(\bar{x}) \) in \( W \) by \( T \), and if \( v_j = H \) (i.e., \( v_j \) is the halt vertex of AP), replace the occurrence of \( q_j(\bar{e}_\alpha) \) in \( W \) by \( F \). Let \( a_1, a_2, \ldots, a_N \) be the set of all the arcs of AP. Then define \( W_{AP} \) as:

\[
(\bar{x})[ W_{a_1} \& W_{a_2} \& \ldots \& W_{a_N} ]
\]

The input variables \( \bar{y} \) are free variables in \( W_{AP} \). The predicate variable \( q_i \) is interpreted as a state predicate, where state is taken to mean the current values of all program variables. Let \( \xi \) be the state. Then the interpretation of the wff of an arc \( \alpha = (v_i, e, v_j) \): \( (q_i(\xi) \& \phi_\alpha) \Rightarrow q_j(\xi') \) is the following:

1) \( q_i(\xi) = T \) iff vertex \( v_i \) is reached with state \( \xi \) and \( q_i \) is a valid predicate of \( v_i \),

2) \( \phi_\alpha \) represents the condition that the arc is entered, and
3) $q_j(\xi') = \top$ iff $\xi'$ is the result of replacing each program variable with its assignment function, and $q_j$ is a valid predicate of $V_j$.

32.3 **Definition** [9]. The program $(AP, I)$ is said to terminate if for all $\gamma \in (D_I)^m$, the execution sequence $<AP, I, \gamma>$ is finite.

32.3.1 **Definition**. Let $(AP, I)$ be a program. Let $W_{AP}$ be the formula of the abstract program $AP$. Then $(W_{AP}, I)$ is defined as the formula of the program $(AP, I)$ and is obtained from $W_{AP}$ by replacing the predicate and function variables by their assignments under the interpretation $I$.

32.4 **Theorem** [9]. The program $(AP, I)$ terminates if and only if $(W_{AP}, I)$ is unsatisfiable (or equivalently, $(\neg W_{AP}, I)$ is valid).

32.5 **Definition** [9]. An abstract program $AP$ is said to terminate if for every interpretation $I$, the program $(AP, I)$ terminates.

32.6 **Theorem** [9]. The following theorem follows from Theorem 32.4 and Definition 32.5:

An abstract program $AP$ terminates if and only if $W_{AP}$ is unsatisfiable (or equivalently, $(\neg W_{AP}$ is valid).

Manna claims this theorem transforms completely the problem of the termination of abstract programs into an equivalent problem in logic. This enables us to obtain many results about the problem of the termination of abstract programs just by using well-known results in logic.

32.7 **Example** [9]. The wff $W_{AP^*}$ of the abstract program $AP^*$ of 31.2 is $(x) (\forall i \geq 1 W_i)$, where:

$W_1$: \[ T \quad \neg \neg p(y) \Rightarrow q_1(a) \]

$W_2$: \[ T \quad \neg p(y) \Rightarrow q_5(y) \]

$W_3$: \[ q_1(x) \quad \neg \neg \neg p(x) \Rightarrow q_2(f(x)) \]

$W_4$: \[ q_1(x) \quad p(x) \Rightarrow q_3(x) \]

$W_5$: \[ q_2(x) \quad p(x) \Rightarrow q_5(a) \]

$W_6$: \[ q_2(x) \quad \neg p(x) \Rightarrow F \]

$W_7$: \[ q_3(x) \quad \neg p(x) \Rightarrow q_3(f(x)) \]

$W_8$: \[ q_3(x) \quad p(x) \Rightarrow F \]
32.8 Correctness and Equivalence [9]. Manna claims that the correctness and equivalence problems of abstract programs is reducible to the termination problem. The following is an informal discussion; the formal definitions and theorems can be found in [9].

32.9 Correctness [9]. To reduce the question of correctness to the question of termination, add 4 arcs and 2 vertices to the abstract program AP as shown in Figure 4. This produces the abstract program AP' with start vertex S' and halt vertex H'. The wff \( \phi(\gamma) \) is a wff (with no free variables other than \( \gamma \)) which represents the specification on the input variables. The wff \( \psi(\gamma, x) \) is a wff (with no free variables other than \( x \) and \( \gamma \)) which represents the relation between the input variables and the program variables at the termination of the program AP. The definition of correctness requires that for:

1) every interpretation I that includes assignments for all the variables that occur in AP, \( \phi \), \( \psi \), and

2) every \( \gamma \) such that \( \phi(\gamma) = T \),

\( \psi(\gamma, \text{val AP}, I, \gamma) = T \) and the execution sequence \( \langle AP, I, \gamma \rangle \) is finite. His theorem states there exists an abstract program AP' such that AP is correct with respect to \( \phi \) and \( \psi \) if and only if AP' terminates. In this way the specifications on the input variables and the relation between the input variables and the program variables are made to perform the same function as the test predicate of an arc.

32.10 Equivalence [9]. A similar argument is made for the equivalence question. The sets of input and program variables of two abstract programs are made to agree. The definition of equivalence requires both execution sequences to be finite and the values of the execution sequences to be equal. The theorem states the existence of an abstract program AP'' such that AP and AP' are equivalent with respect to \( \phi \) if and only if AP'' terminates. The specifications on the input values serves the function
Figure 4. The abstract program $AP'$. 
Figure 5. The abstract program $AP''$. 
of a test predicate, and an additional predicate is used to assure
the equality of the values of the execution sequences. An example of
an abstract program AP is given in Figure 5.

33 Results from Gödel [5]

33.1 Before discussing the application of resolution, it is
necessary to mention a few results due to Gödel. In particular:

33.2 Definition. The primitive recursive functions are defined
inductively as follows:

1) [Zero function] \( Z(x) = 0 \) is a primitive recursive function,

2) [Successor function] \( S(x) = x + 1 \) is a primitive recursive
function,

3) [Projection functions] \( U_i^n(x_1, \ldots, x_n) = x_i \) are primitive
recursive functions,

4) [Composition] If \( h, g_1, \ldots, g_m \) are primitive recursive
functions, so is the function \( f \) defined by:
\[
f(x_1, \ldots, x_n) = h(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n))
\]

5) [Primitive recursion] If \( g \) and \( h \) are primitive recursive
functions, so is the function \( f \) defined by:
\[
f(0, x_2, \ldots, x_n) = g(x_2, \ldots, x_n)
\]
\[
f(k+1, x_2, \ldots, x_n) = h(k, f(k, x_2, \ldots, x_n), x_2, \ldots, x_n)
\]

33.3 Definition. A relation \( R(x_1, \ldots, x_n) \) between natural numbers
is said to be primitive recursive if there is a primitive recursive
function \( f(x_1, \ldots, x_n) \) such that for all \( x_1, x_2, \ldots, x_n \):
\[
R(x_1, \ldots, x_n) \leftrightarrow [f(x_1, \ldots, x_n) = 0]
\]

33.4 Every function (relation) obtained from primitive recursive
functions (relations) by substitution of primitive recursive functions
for the variables is primitive recursive; so is every function (relation)
obtained from primitive recursive functions (relations) by primitive recursive
deinition according to 5) above. Hereafter, boldface letters \( R, S, \) and \( T \)
will be used as metavariables whose range is primitive recursive relations.
33.5 If \( R \) and \( S \) are primitive recursive relations, so are \( \neg R \), \( R \lor S \), and \( R \land S \).

33.6 If the functions \( g(x_1, \ldots, x_n) \) and \( h(x_1, \ldots, x_n) \) are primitive recursive, so is the relation \( g(x_1, \ldots, x_n) = h(x_1, \ldots, x_n) \).

33.7 If the function \( g(x_1, \ldots, x_n) \) and the relation \( R(x, y_1, \ldots, y_n) \) are primitive recursive, so are the relations \( S \) and \( T \) defined by:

\[
S(x_1, \ldots, x_n, y_1, \ldots, y_n) \Leftrightarrow (\exists x)[x \leq g(x_1, \ldots, x_n) \land R(x, y_1, \ldots, y_n)]
\]

and

\[
T(x_1, \ldots, x_n, y_1, \ldots, y_n) \Leftrightarrow (\exists x)[x \leq g(x_1, \ldots, x_n) = R(x, y_1, \ldots, y_n)]
\]

as well as the function \( \psi \) defined by:

\[
\psi(x_1, \ldots, x_n, y_1, \ldots, y_n) = \text{lex}[x \leq g(x_1, \ldots, x_n) \land R(x, y_1, \ldots, y_n)]
\]

where \( \text{lex}(x) \) means the least number \( x \) for which \( P(x) \) holds and 0 in case there is no such number. This is the bounded least number operator.

33.8 The following functions are primitive recursive:

\[
x + y
\]
\[
x \times y
\]
\[
x^y
\]
\[
x \div y
\]
\[
x - y \text{ (proper subtraction)}
\]

as well as the relations:

\[
x < y
\]
\[
x = y
\]

33.9 For every primitive recursive relation \( R(x_1, \ldots, x_n) \) there exists an \( n \)-place free variable formula \( A \) (with free variables \( y_1, \ldots, y_n \)) such that for all \( n \)-tuples \( (x_1, \ldots, x_n) \) we have:

\[
A[x_i/y_i] \text{ for all } i, 1 \leq i \leq n,
\]

is provable just in the case that the relation \( R(x_1, \ldots, x_n) \) holds, and

\[
\neg A[x_i/y_i] \text{ for all } i, 1 \leq i \leq n,
\]

is provable just in the case that the relation \( R(x_1, \ldots, x_n) \) does not hold.
34 The application of resolution.

34.1 This section describes the application of the resolution procedure to the formula of an abstract program $W_{AP}$ or that of a program $(W_{AP}, I)$. The latter is the more interesting case because such formulas would be constructed from the source programs that programmers write. It is shown that the formula of the program as constructed can be made to conform to the specifications required by the resolution procedure. An example of the use of resolution to prove the termination of the abstract program $AP^*$ of 31.2 is given. The formula of this abstract program was constructed in 32.7.

34.2 From the construction of the wff $(W_{AP}, I)$, it appears to be very nearly in the form required by the resolution procedure. By construction, the wff $(W_{AP}, I)$ is the universal quantification over the program variables of the conjunction of the wffs of the arcs. The wffs of the arcs are composed of state predicates and test predicates. The input variables are free variables in the formula.

34.3 The resolution procedure requires that the formula be the universal quantification over all variables of a matrix (quantifier-free wff) in conjunctive normal form. Each clause in the matrix is a disjunction of literals; literals are either atomic formulas or the negation of atomic formulas.

34.4 There are three factors which prevent the direct application of resolution; they are:

1) the construction does not require that the test and state predicates be atomic formulas,

2) the wffs of the arcs are not disjunctions (if the components are literals, the conversion is trivial and does not add any clauses),

3) the input variables are free variables in the wff of the program.
34.5 If each test predicate and each state predicate could be considered a literal for the resolution procedure, then the first two limitations are removed by converting each wff of an arc to its disjunctive normal form. From the results of Gödel, especially 33.9, if the test predicates and the assignment functions are limited to primitive recursive relations and functions, each test predicate and each state predicate can be expressed as a literal for the resolution procedure. In the remainder of this paper we will continue to use the symbols $\phi_a$ and $q_i(\xi)$ to represent this Gödel form of the test and state predicates; it should be clear from the context which form is meant.

34.6 In the preceding theoretical framework, the input variables remained constant throughout the program. They were available as arguments for the test predicates and as terms in assignment functions. The input variables received a value only under the assignments for an interpreted program $(AP, I, Y)$, i.e., at execution time. In writing real programs, these restrictions are not met; it is common practice to alter the contents of a memory location into which an input value was placed. The theoretical restrictions are met, while permitting standard programming techniques, by the following:

1) to the set of program variables $\bar{x}$, add $m$ distinct individual variables, where $m$ is the number of input variables in the program,

2) add the following assignment function

$$x_{m+1} = y_1, \ 1 \leq i \leq m,$$

after each assignment to an input variable, and

3) replace each occurrence of $y_1$ by an individual constant.
34.7 There are no output variables. The output value is the value of the execution sequence when the program terminates. Common programming techniques permit "output on the fly" and the use of the same memory locations for successive values. To bring this into the theoretical framework and to indicate that an assignment has been made, add to the set of program variables, \( n_1 \) distinct individual variables, where \( n_1 \) is the number of distinct variables whose values are to be transmitted to an output device.

34.8 The wff of the program is now the universal quantification over all variables. In the remainder of this paper, we will use \( \xi \) to represent the augmented set of program variables:

\[
\{x_1, \ldots, x_n, x_{n+1}, \ldots, x_{n+m}, x_{n+m+1}, \ldots, x_{n+m+n_1}\}
\]

We will use the symbols

\( y_1, \ldots, y_m \) to represent the input variables,

\( x_1, \ldots, x_n \) to represent the program variables, and

\( z_1, \ldots, z_n \) to represent the output variables

simply to distinguish their function.

34.9 A step of resolution is the process of generating all possible resolvents for a matching pair of complementary literals. The literals \( A \) and \( \neg A \) are matching complementary literals; except for the negation symbol, the symbols in each must agree in number and position. For example, consider the following set of clauses:

\[
\{p(x), q(a)\}, \{\neg p(x), r(b)\}, \{p(x), q(b), p(y)\}
\]

\( p(x) \) and \( \neg p(x) \) are matching complementary literals and the resolvent clauses are:

\[
\{q(a), r(b)\}, \{r(b), q(b), p(y)\}
\]

34.10 No substitutions are made for individual variables; all possible resolvents for all matching pairs of complementary literals
are generated. Later in the procedure, substitutions are made for the individual variables, and the steps repeated using the clause instances.

34.11 The number of steps required to generate the \( n \)th resolution is therefore \( M/2 \) steps, where \( M \) is the number of matching complementary literals contained in the set of clauses of the \( (n-1) \)st resolution. Obviously, if the empty clause, \( \square \), appears in the first resolution, this takes the minimum number of steps, namely \( M/2 \), where \( M \) is the number of matching complementary literals contained in the original set of clauses.

34.12 After replacing each component in the \( \text{wff} \) of each arc by its corresponding Gödel form and converting each to disjunctive normal form, the \( \text{wff} \) of the program \( (W_{AP',I}) \) is in the form required by the resolution procedure. The minimum solution is achieved if the empty clause occurs in the first resolution. Therefore, the minimum number of steps of resolution required for a solution can be determined by counting the number of matching complementary literals in the \( \text{wff} \) of the program. Since the only literals in this \( \text{wff} \) are present by virtue of being a component of the \( \text{wff} \) of an arc, examination of the structure of these \( \text{wffs} \) leads to the set of distinct literals in the \( \text{wff} \) of the program; from this set the matching complementary literals are counted.

34.13 Example [9]. Using Theorem 32.6, we prove that the abstract program \( AP^x \) terminates by proving \( W_{AP^x} \) is unsatisfiable. \( W_{AP^x} \) was constructed in 32.7. After changing each conjunct to disjunctive form and replacing \( y \) by \( b \), the list of clauses, \( S \), for the resolution procedure are:
1. \( p(b), q_1(a) \)
2. \( \neg p(b), q_3(b) \)
3. \( \neg q_1(x), p(x), q_2(f(x)) \)
4. \( \neg q_1(x), \neg p(x), q_3(x) \)
5. \( \neg q_2(x), \neg p(x), q_3(a) \)
6. \( \neg q_2(x), p(x) \)
7. \( \neg q_3(x), p(x), q_3(f(x)) \)
8. \( \neg q_3(x), \neg p(x) \)

34.14 The set of distinct literals is:
\[
\{p(b), \neg p(b), p(x), \neg p(x), \neg q_1(x), \neg q_2(x), \neg q_3(x), q_1(a), q_3(b), q_2(f(x)), q_3(x), q_3(a), q_3(f(x))\}
\]
and the set of matching complementary literals is:
\[
\{p(b), \neg p(b), p(x), \neg p(x), q_3(x), \neg q_3(x)\}
\]

Therefore, the first resolution will require three steps. The empty clause does not occur in the first resolution and it is then necessary to generate clause instances, and again obtain the set of matching complementary literals. This is the method used by resolution programs; it is very lengthy but complete. Hand resolution on such a simple example is done in a different manner. One takes an educated guess as to which clauses will quickly resolve to a single literal, making substitutions for individual variables where necessary. When a resolvent clause is obtained with a single literal, an effort is made to find a resolvent which contains its complement as a single literal. These two then resolve to the empty clause. This is a trial and error method and often many resolvents are generated which are errant to the final result. When printing the final copy, these extraneous resolvents are omitted.
34.15 The following clauses were obtained by hand resolution of the above set of 8 clauses:

9. \( \neg p(b) \)  
   by 2 and 8 (with \( x=b \))

10. \( q_1(a) \)  
    by 1 and 9

11. \( \neg q_1(x), q_2(f(x)), q_3(x) \)  
    by 3 and 4

12. \( q_2(f(a)), q_3(a) \)  
    by 10 and 11 (with \( x=a \))

13. \( \neg q_2(x), q_3(a) \)  
    by 5 and 6

14. \( q_3(a) \)  
    by 12 and 13 (with \( x=f(a) \))

15. \( \neg q_3(x), q_3(f(x)) \)  
    by 7 and 8

16. \( q_3(f(a)) \)  
    by 14 and 15 (with \( x=a \))

17. \( p(a), q_2(f(a)) \)  
    by 5 (with \( x=a \)) and 10

18. \( p(a), p(f(a)) \)  
    by 6 (with \( x=f(a) \)) and 17

19. \( \neg q_3(a), p(f(a)) \)  
    by 8 (with \( x=a \)) and 18

20. \( \neg q_3(a), \neg q_3(f(a)) \)  
    by 8 (with \( x=f(a) \)) and 19

21. \( \neg q_3(a) \)  
    by 16 and 20

22. \( \square \)  
    by 14 and 21

34.16 Thus, we have a refutation of the set of clauses \( S \) (clauses 1-8), hence by Robinson's Theorem, the set is unsatisfiable, and by Theorem 32.6, the program AP* terminates.

34.17 This example illustrates the procedure; it is obviously trivial. A real program would contain more than a single input and a single output variable; test predicates and assignment functions are more complex and more numerous. Thus, locating complementary pairs becomes more difficult. An example of this is given in 41.6. In the next chapter a short Fortran program is used to illustrate the application of Manna's method to actual programs.
IV. APPLICATION OF MANNA'S METHOD.

40 Introduction.

40.1 In this chapter, a Fortran program chosen from an undergraduate text [3] is presented. The abstract program AP, the program (AP, I), and the formula \( W_{AP} \) of the program are constructed according to Manna. The relationships between the graphical representation and the Fortran source program are explained informally.

40.2 Rules for constructing the graph of a program from the Fortran source statements are given and it is shown that this graph conforms to Manna's definition of a program (AP, I).

40.3 The goal is to count the number of matching complementary literals in the wff of the program in order to arrive at the number of steps of resolution required to obtain the first resolution. The factors influencing this count are the test predicates, the state predicates, and the assignment functions of the arcs. After examining the graphs of the program and the structure of the wffs of the arcs, a simple tabular representation is given which facilitates the counting and which would be suitable for machine implementation. Examination of this table provides the set of distinct test predicates and the set of distinct state predicates, from which the set of matching complementary literals is obtained. It is shown that this constructively counts all matching complementary literals.

41 Example [3].

41.1 The following example was taken from an undergraduate text in Fortran programming.
41.2 The problem statement. Ninety students have taken a quiz.

The possible scores on the quiz are 0, 5, 10, 15, 20, and 25. The input data consists of ninety cards keypunched with student number (columns 1-5) and quiz score (columns 7-8). Write a program to determine which of the six possible scores most students received, and how many students received that score.

41.3 The Fortran source program.

```fortran
DIMENSION KOUNT (6)
C
C INITIALIZE ARRAY ELEMENTS TO ZERO.
DO 8, I=1,6
7 KOUNT(I)=0
DO 1 I=1,90
READ 2, MARK
C FORMAT(6X12)

C SCALE TEST SCORES TO A VALUE BETWEEN 1 AND 6 INCLUSIVE.
MARK =MARK/5+1
C USE SCALED SCORE AS A SUBSCRIPT AND ADD 1 TO PROPER ARRAY ELEMENT.
1 KOUNT(MARK)=KOUNT(MARK)+1
C DETERMINE SCORE MOST OFTEN ATTAINED AND NUMBER OF STUDENTS RECEIVING IT.
MOST=KOUNT(I)
DO 3 I=2,6
IF(MOST-KOUNT(I)) 6,3,3
6 MOST=KOUNT(I)
3 CONTINUE
C CONVERT SCALED TEST SCORE TO ACTUAL SCORE.
C PRINT SCORE(S) RECEIVED BY MOST STUDENTS AND NUMBER OF STUDENTS.
DO 11 I=1,6
IF(MOST-KOUNT(I)) 11,13,11
13 L=I*5-5
PRINT 12,MOST,L
12 FORMAT(1X13,30H STUDENTS RECEIVED A SCORE OF 13)
11 CONTINUE
C
C CALL EXIT
END
```

41.4 The abstract program AP. The abstract program AP in Figure 6 represents the family of programs of which the above Fortran program is a member. By making assignments to the predicate and function variables of the abstract program according to the source statements, the program (AP, I) is obtained. These are listed below.

41.5 The assignments. The following assignments are made to the predicate and function variables of the abstract program.
Note: A is $U_{y_1}(x_1, \ldots, x_6)$
B is $f_4(x_8, U_{x_7}(x_1, \ldots, x_6))$
C is $U_{x_7}(x_1, \ldots, x_6)$

Figure 6. The abstract program.
1) The domain of the interpretation is the set of positive integers.

2) The set of input variables \( \{y\} \) is \( \{y_1\} \), where 
\[ y_1 \text{ is } \text{MARK} \]

3) The set of program variables \( \{x\} \) is \( \{x_1, x_2, \ldots, x_9\} \), where
\[ x_1 \text{ is } \text{KOUNT}(1) \]
\[ x_2 \text{ is } \text{KOUNT}(2) \]
\[ x_3 \text{ is } \text{KOUNT}(3) \]
\[ x_4 \text{ is } \text{KOUNT}(4) \]
\[ x_5 \text{ is } \text{KOUNT}(5) \]
\[ x_6 \text{ is } \text{KOUNT}(6) \]
\[ x_7 \text{ is } \text{I} \]
\[ x_8 \text{ is } \text{MOST} \]
\[ x_9 \text{ is } \text{L} \]

4) The set of output variables \( \{z\} \) is \( \{z_1, z_2\} \), where

5) The set of individual constants is \( \{a_1, a_2, \ldots, a_6\} \), where
\[ a_1 \text{ is } 0 \]
\[ a_2 \text{ is } 1 \]
\[ a_3 \text{ is } 2 \]
\[ a_4 \text{ is } 5 \]
\[ a_5 \text{ is } 6 \]
\[ a_6 \text{ is } 90 \]

6) The following assignments are made to the predicate variables:
\[ p_1(x) : x < 6 \]
\[ p_2(x) : x < 90 \]
\[ p_3(f_4(x_1, x_2)) : x_1 - x_2 = 0 \]
\[ p_4(f_4(x_1, x_2)) : x_1 - x_2 < 0 \]
\[ p_5(f_4(x_1, x_2)) : x_1 - x_2 > 0 \]

7) The following assignments are made to the function variables:
\[ f_1(x) : x + 1 \]
\[ f_2(x) : x/5 + 1 \]
\[ f_3(x) : x \times 5 - 5 \]
\[ f_4(x_1, x_2) : x_1 - x_2 \]
\[ g_1(x) \text{ is the input assignment function} \]

8) The augmented set of program variables is:
\[ \{y_1, x_1, x_2, \ldots, x_9, z_1, z_2\} \]
9) The assignment function for arc \( \alpha \) is:

\[ \tau_{\alpha} = (t_1^{(\alpha)}, t_2^{(\alpha)}, \ldots, t_{12}^{(\alpha)}) \]

where:
- \( t_1^{(\alpha)} \) is the assignment function for \( y_1 \)
- \( t_2^{(\alpha)} \) is the assignment function for \( x_1 \)
- \( \ldots \)
- \( t_{10}^{(\alpha)} \) is the assignment function for \( x_9 \)
- \( t_{11}^{(\alpha)} \) is the assignment function for \( z_1 \)
- \( t_{12}^{(\alpha)} \) is the assignment function for \( z_2 \)

41.6 The wff of the abstract program: The wff of the abstract program constructed according to Manna's algorithm. In the following:

1) \( \xi \) represents the current value of the augmented set of program variables at the initial vertex of the arc, and

2) \( \xi' \) represents the value of the augmented set of program variables at the terminal vertex of the arc, after replacing \( \xi \) by \( t_{\alpha}^{(\alpha)} \). The assignment function \( t_1^{(\alpha)} \) is given explicitly only for those variables that are changed in value along the arc. For all other variables along that arc, the assignment function is the identity function, \( x_i + x_i \). The superscript \( (\alpha) \) is omitted from the assignment functions to improve readability.

The wff of the abstract program is:

\[ (x)(y)(z)(\xi_{i=1}^{20} W_i) \]

where \( W_i \) is the wff of arc \( \alpha_i \). The wffs of the arcs are given in Figure 7.
\[ W_1: \ T \quad \& \quad T \]
\[ W_2: \ q_1(\xi) \quad \& \quad T \]
\[ W_3: \ q_2(\xi) \quad \& \quad p_1(x_7) \]
\[ W_4: \ q_2(\xi) \quad \& \quad \neg p_1(x_7) \]
\[ W_5: \ q_3(\xi) \quad \& \quad T \]
\[ W_6: \ q_4(\xi) \quad \& \quad T \]
\[ W_7: \ q_5(\xi) \quad \& \quad T \]
\[ W_8: \ q_6(\xi) \quad \& \quad p_2(x_7) \]
\[ W_9: \ q_6(\xi) \quad \& \quad \neg p_2(x_7) \]
\[ W_{10}: \ q_7(\xi) \quad \& \quad p_3(f_4(x_8, U_{x_7}(x_1, \ldots, x_6))) = q_8(\xi') \quad \text{where} \quad \tau_9 \text{ is} \quad x_8 \quad + \quad x_1 \]
\[ W_{11}: \ q_7(\xi) \quad \& \quad p_4(f_4(x_8, U_{x_7}(x_1, \ldots, x_6))) = q_9(\xi') \]
\[ W_{12}: \ q_7(\xi) \quad \& \quad p_5(f_4(x_8, U_{x_7}(x_1, \ldots, x_6))) = q_9(\xi') \]
\[ W_{13}: \ q_8(\xi) \quad \& \quad p_1(x_7) \]
\[ W_{14}: \ q_8(\xi) \quad \& \quad \neg p_1(x_7) \]
\[ W_{15}: \ q_8(\xi) \quad \& \quad p_3(f_4(x_8, U_{x_7}(x_1, \ldots, x_6))) = q_{10}(\xi') \quad \text{where} \quad \tau_{10} \text{ is} \quad x_9 \quad + \quad f_3(x_7) \]
\[ W_{16}: \ q_{10}(\xi) \quad \& \quad T \]
\[ W_{17}: \ q_9(\xi) \quad \& \quad p_4(f_4(x_8, U_{x_7}(x_1, \ldots, x_6))) = q_{11}(\xi') \]
\[ W_{18}: \ q_9(\xi) \quad \& \quad p_5(f_4(x_8, U_{x_7}(x_1, \ldots, x_6))) = q_{11}(\xi') \]
\[ W_{19}: \ q_{11}(\xi) \quad \& \quad p_1(x_7) \]
\[ W_{20}: \ q_{11}(\xi) \quad \& \quad \neg p_1(x_7) \]
\[ = \Gamma \]

Figure 7. The wffs of the arcs.
42 Construction rules.

42.1 In this section, the rules for constructing the graph of the program from a Fortran source program are outlined. Obviously one need consider only executable statements as the others are present only to provide information to the compiler. The relations between the graph and the source statements are explained, and examples of the various constructions are given. In 42.7, it is shown that this construction satisfies Manna's definition of a program (AP,1).

42.2 There are 9 kinds of executable statements to be considered here, each of which has a particular graphical representation. They are:

1) The arithmetic If is a vertex with 3 test predicates and 3 arcs leaving the vertex.

\[
\text{IF (exp)} \stackrel{s_1, s_2, s_3}{\text{exp<0}} \quad \text{exp>0} \quad \text{exp=0}
\]

2) The logical If is a vertex with 2 test predicates and 2 arcs leaving the vertex.

\[
\text{IF (lexp) stmt} \quad \text{lexp} = \text{.TRUE.} \quad \text{lexp} = \text{.FALSE.}
\]
3) The computed GO TO is a vertex with \( n+1 \) test predicates and \( n+1 \) arcs leaving the vertex, where \( n \) is the number of statement numbers which occur in the statement. The \((n+1)^{st}\) test predicate is "none of these", and that arc leads to the halt vertex (in principle).

\[
\text{GO TO } (1,2,3), 1
\]

4) The construction for a DO-loop includes one vertex for the DO statement and one vertex for the terminal statement of the loop. The latter vertex has 2 test predicates and 2 arcs leaving the vertex; the former has one arc with a test predicate of \( T \).

\[
\text{DO } 1 \ I = m_1, m_2, m_3
\]

\[
\{\text{body}\}
\]

\[
1 \text{ CONTINUE}
\]

The test predicates at the vertex following the terminal statement are constructed according to the parameters of the DO.
The syntax of Fortran is followed in that the value of the index of a DO is unspecified upon normal termination of the DO. The body of the DO is constructed according to the individual statements contained therein.

5) A sequence of arithmetic assignment statements requires a continuation vertex according to the following:

Let \( x_1 = b_1, x_2 = b_2, \ldots, x_n = b_n \) be a sequence of arithmetic assignment statements as they appear in the source program. Each \( b_i, 1 \leq i \leq n \), is a Fortran arithmetic expression consisting of program variables, which may include \( x_i \), joined by operation symbols. The sequence requires a continuation vertex if the following conditions are NOT met (i.e., these conditions must hold for all assignments included along a single arc):

(a) the set \( \{x_1, x_2, \ldots, x_n\} \) is distinct, for no \( i, j \) such does \( x_i = x_j \), and

(b) No \( x_i \) occurs in an expression \( b_j \), where \( j > i \). \( x_i \) may occur in an expression \( b_j \) if \( j < i \).

For example, the sequence of Fortran statements

\[
\begin{align*}
N &= 4 \\
J &= N + K
\end{align*}
\]

requires a continuation vertex, and the sequence

\[
\begin{align*}
J &= N + K \\
N &= 4
\end{align*}
\]

does not. The continuation vertex has a single arc leaving it and the test predicate of that arc is \( T \). There is an explicit assignment function for that arc.

6) The GO TO statement does not require a vertex. It represents an unconditional transfer of control and in constructing the
graph the path of control is followed until a vertex is specified. It simply redirects the sequence in which the statements are processed.

7) READ, WRITE, and DATA statements are considered assignment statements. The READ statement represents the transmittal of data from an input device; in our interpretation these are the assignment functions for the program variables which were added to represent the input variables [c.f., 35.2]. The WRITE statement represents the transmittal of data to an output device; in our interpretation these are the assignment functions for the program variables which were added to represent the output variables [c.f., 35.2]. At present, implied DO-loops within an I/O statement are considered simple assignment functions. The DATA statement represents the assignment of initial values to the program variables prior to execution of the program. In our interpretation, they are initial assignment functions for program variables and they are located immediately following the start vertex.

8) A single start vertex, S, is inserted corresponding to the first statement of the source program. If a program has multiple entry points, this is represented by a single start vertex with a set of arcs joining it to the entry points.

9) Multiple STOP or other varieties of termination statements are represented by a single halt vertex, H. Assignment functions are not permitted along an arc leading to the halt vertex. If necessary, a continuation vertex is inserted. This vertex has a single arc joining it to H, and the test predicate of this arc is T.
42.3 These are the only statements and conditions that cause vertices to appear in the graph. All test predicates and assignment functions which appear in the graph do so by virtue of being specified by the source program in one of these 9 ways. The sequence of vertices follows the sequence of execution of the statements as defined by the Fortran language specifications. All executable statements in the source program must be represented in the graph.

42.4 But the language specifications do not require that every statement be executed nor that every numbered statement be referenced. By following the flow of control through every possible sequence of executable statement, the graph will represent all possible paths leading from the start vertex. This leaves the possible existence of program fragments to which control is never passed. If, after constructing the graph (i.e., all paths under construction have reached the halt vertex), there remain Fortran statements which have no representation in the graph, and therefore there is no path to them from the start vertex, eliminate the program from further consideration. It fails to meet Manna's definition of an abstract program in that there exists a vertex which is not on a path that joins S and H. Also, most Fortran compilers will reject such programs with a low severity error message.

42.5 After assuring that every source statement has a representation in the graph, there still exist programs that cannot be dealt with by the proposed method. This includes programs which contain built-in infinite loops and therefore do not terminate, and programs which consist of a sequence of trivial "do-nothing" statements but do obviously terminate. Programs such as these are not the carefully written programs under consideration here; they are quite properly to
be excluded. These cases can be syntactically detected by the
application of a flow-checking procedure to the constructed graph.
That is, there is an effective procedure for rejecting such programs.

42.6 By construction, every executable Fortran statement is
represented in the graph; there is a single start vertex and a single
halt vertex. The number of vertices and the number of arcs are finite.
To check the flow, begin at the start vertex and enumerate all paths,
recording for each arc the initial vertex, the terminal vertex, the
test predicate, and the number of assignment functions. The following
cases are to be eliminated from further consideration:

1) Any program which contains a path of the form

\[ S \rightarrow T \rightarrow I \rightarrow T \rightarrow H \]

where the test predicate for each arc is \( T \) and there are no
assignment functions other than the identity function. A
program containing a path of this type obviously terminates.
An example of such a program is the "null job":

```
STOP
END
```

2) Any program which contains a vertex which is not on a path
beginning at \( S \) and ending at \( H \). This effectively eliminates
programs which contain built-in infinite loops and/or isolated
program fragments. An example of a built-in infinite loop is:

```
20 GO TO 10
...
...
10 GO TO 20
```
This case violates Manna's definition of an abstract program in that there exists a vertex which is not on a path that joins $S$ and $H$, and is therefore rejected.

42.7 Now we must address the question: Does this graph satisfy Manna's definition of a program (AP,I), c.f., 33.1, 33.2? The answer is in the affirmative.

42.8 The construction assures a single start vertex and a single halt vertex. The application of the flow-checking procedure assures that each vertex is on some path that joins $S$ and $H$. Any programs not meeting this requirement have already been eliminated. Thus, the construction provides a finite directed graph as required by definition 31.1.

42.9 The set of individual input variables is exactly the set of distinct variables which occur in READ statements. For each such variable, the compiler assigns a unique storage location, hence the set is distinct and finite. The set of individual program variables is exactly the set composed of distinct variables which occur:

1) in DATA statements,

2) to the left of the equality symbol in an assignment statement,

3) as the index of a DO-loop.

as well as:

4) the $m$ distinct program variables added to represent the $m$ input variables [c.f., 33.2], and

5) the $k$ distinct program variables added to represent $k$ output variables. The set of output variables consists of exactly those distinct variables which occur in WRITE statements.
The Fortran language specifications require that the variables 1-d are distinct. This corresponds to the contents of the symbol table; variables defined in such ways are assigned to unique memory locations and are thus distinct. The program variables added by 5 are distinct by construction [c.f., 33.2]. Thus, there is a finite set of distinct input variables and a finite set of distinct program variables as required by definition 31.1

42.10 The test predicate of an arc \( \alpha = (v, v') \) is:

1) \( T \) when the vertex is constructed from the DO statement,

2) \( T \) when the vertex is a continuation vertex,

3) formed from the expression in an arithmetic or logical IF statement,

4) formed from the terminal conditions of a DO-loop, or

5) is a representation of the form \( x_i = g, \) for \( 1 \leq i \leq m, \) or \( x_i > m, \) where \( m \) is the number of statement numbers which occur in a computed GO TO statement and \( x_i \) is the variable named in the statement.

These are the only vertices described by the construction. Thus, with each arc is associated a test predicate, as required by definition 33.1.

42.11 The syntax of Fortran does not permit the presence of quantifiers or predicate variables in the expressions in IF statements, nor as terminal conditions of DO-loops. The only variables permitted in the expressions are those that have been previously defined; these are thus contained in the set of program variables. Hence, the test predicates do not contain any predicate variables nor any individual variables other than input or program variables as required by definition 35.1.
42.12 The assignment functions are derived from the assignment statements. The variable occurring to the left of the equality symbol names the program variable to which the assignment function applies. Any variable occurring to the right must have been previously defined and is therefore included in the set of program variables. The rule for the insertion of a continuation vertex assures that there is, along a single arc, at most one explicit assignment function for each program variable. For all program variables for which an assignment function is not explicitly given, the identity function is inserted. Thus, for each arc there is an n-tuple of terms called the assignment function of the arc as required by definition 31.1.

42.13 The set of test predicates at any vertex is complete and mutually exclusive; there are 4 cases:

1) Continuation vertex or DO statement. There is one arc and the test predicate is T; this set is obviously complete and mutually exclusive.

2) Arithmetic IF. There are 3 arcs and the 3 test predicates are \((\text{exp})<0\), \((\text{exp})=0\), and \((\text{exp})>0\), where \text{exp} is a Fortran arithmetic expression. By the law of trichotomy, this set is complete and mutually exclusive.

3) Logical IF. There are 2 arcs and the 2 test predicates are \((\text{exp})\) and \((\neg\text{exp})\), where \text{exp} is a Fortran logical expression whose value is one of the logical constants, .TRUE. or .FALSE.. This set is obviously complete and mutually exclusive.

4) Computed GO TO. There are \(n+1\) arcs and the \(n+1\) test predicates are \(x_1=1\), \(x_1=2\), \(\ldots\), \(x_1=n\), and \(x_1>n\), where \(n\) is the number of statement numbers which occur in the statement and \(x_1\) is the variable which occurs in the statement. The addition of the
(n+1)st arc and the corresponding test predicate assures that this set is complete and mutually exclusive (and is, in fact, provided by most Fortran compilers).

42.14 The domain of the interpretation, $D_I$, is the set of positive integers.

42.15 The assignment function assigned to each function variable is specified by Fortran expressions, the initialization of the index of a DO-loop, the incrementation of the index of a DO-loop, or is the identity function. The initialization function is a replacement function of the form $x_i = g$; the incrementation function is of the form $x_i = x_i + g$. The Fortran expressions are sequences of one or more variables joined by operator symbols. The functions are limited to those that can be shown to be primitive recursive because the finiteness of the machine word demands a bounded least number operator; an overflow condition will result if the search exceeds the word size.

42.16 The assignment of predicates to the predicate variables is specified by the expressions in IF statements or the number of statement numbers which occur in a computed GO TO statement or the terminating conditions of a DO-loop. As in 42.15, they are restricted to those that can be shown to be primitive recursive.

42.17 Therefore, the construction satisfies Manna's definition of a program $(AP, I)$.

43 **The tabular representation of a program $(AP, I)$**.

43.1 The goal of this study is to count, for the WFF of each program, the number of matching complementary literals occurring in the set of distinct literals. The number of matching complementary literals determines the number of steps of resolution required to
This is the table from which the set of distinct literals will be counted; it is described in 43.8.

2) two auxiliary tables of distinct Fortran expressions and assignment statements. These tables, kept only during the construction, are:

a) distinct Fortran expressions which specify a test predicate, and

b) distinct Fortran assignment statements which specify assignment functions.

43.5 Construction. The construction of the auxiliary tables is described first:

43.6 The auxiliary table of distinct "test predicates". Whenever a Fortran expression specifying a test predicate in encountered (or generated, in the case of DO-loops), an entry is made in the table only if the expression is distinct. The entries are of the form (i,exp), where i is the entry number and exp is the Fortran expression. The abbreviation \( p_i \) refers to the test predicate formed from the \( i^{th} \) Fortran expression in this table.

43.7 The auxiliary table of distinct "assignment functions". Whenever a Fortran assignment statement is encountered (or generated, in the case of DO-loops), an entry is made in the table only if the statement is distinct. The entries are of the form (i,\( \text{stmt} \)), where i is the entry number and \( \text{stmt} \) is the Fortran assignment statement. The abbreviation \( f_i \) refers to the assignment function formed from the \( i^{th} \) statement in this table. This specifies both the program variable and the assignment function.

43.8 The main table of vertices and arcs. Each entry in this table is of the form:
where $v_i$ is a sequential number assigned to the vertex (the start vertex is S; the halt vertex, H), and the other entries represent the set of
arcs leaving this vertex. For each arc $a_k$, 1 ≤ k ≤ n, the following
entries are made:

1) $v_i$ is the initial vertex of this arc,
2) $v_j$ if the terminal vertex of this arc,
3) $a_k$ represents the test predicate of this arc, and is either $T$
or an abbreviated test predicate of the form $p_i$, $~p_i$, $p_i < 0$, $p_i > 0$,
$p_i = 0$, $p_i = 2$, or $p_i > 2$. where $i$ is the entry number of the
Fortran expression specifying the predicate. The different
forms represent the different statements from which the test
predicates are formed.
4) $t_a$ represents the explicit assignment functions along this
arc and is a list of abbreviated assignments. If $t_a = 0$, there
are no assignment functions other than the identity function
along this arc. Otherwise the entry is a list of items, each
of the form $f_i$, where $i$ is the entry number of the Fortran
statement which specifies the program variable and the
assignment function.

43.9 After these tables have been constructed, the auxiliary
tables are no longer needed. (If desired, they could be used to construct
the wff of the program.) The function of these tables here is to allow
the assignment of distinct predicates to each distinct expression and
statement. As Fortran expressions can be lengthy, the use of
abbreviations keeps the main table concise and of manageable size;
however, distinctness of literals would be assured if this is not done.

43.10 The main table of vertices and arcs can also be used to
enumerate all paths [c.f., 42.3] as well as to obtain the set of
distinct literals which will appear in the wff of the program. From this set, the number of matching complementary literals is counted.

43.11 Example. The tables constructed for the program of 41.3 are given in Figure 8.

43.12 Examination of this table quickly provides the data needed for obtaining the set of distinct literals. This set includes all distinct test predicates and all distinct state predicates. Whenever a test predicate, \( t_{a_1} \), or a state predicate, \( q_i(\xi) \) or \( q_j(\xi') \), is referred to as a literal, the intended meaning is the Gödel form of the wff. The abbreviations \( p_i \) and \( f_i \) refer to Fortran expressions and statements by which the wffs \( q_i(\xi) \) or \( q_j(\xi') \) are specified. If two Fortran expressions \( p_i \) and \( p_j \) are identical, the corresponding wffs \( \phi_{a_1} \) and \( \phi_{a_j} \) will be identical; and so will the Gödel form, the literals. So, each distinct \( p_i \) represents the presence of a distinct literal in the wff of the program. Similarly, each distinct \( f_i \) represents the presence of a distinct assignment function in the state predicate \( q_j(\xi') \).

The wff of an arc \( a = (v_i, \xi, v_j) \) is \( q_i(\xi) \wedge \phi_\alpha \supset q_j(\xi') \); the clause formed from this wff is

\[ \neg q_i(\xi), \neg \phi_\alpha, q_j(\xi') \]

where each component is a literal. Thus, the state predicate at the initial vertex and the test predicate appear negated in the clause formed from the disjunctive normal form of the wff of each arc.

43.13 To obtain the set of distinct literals formed from the test predicates, enumerate the test predicates, negate each, and eliminate duplicates.

43.14 The state predicate of each vertex, \( v_i \), will occur in the clauses in two ways; when \( v_i \) is the initial vertex of an arc and when \( v_i \) is the terminal vertex of an arc. To obtain the set of distinct
The table of vertices and arcs

5 \((8,1,T,(f_1))\)
1 \((1,2,T,(f_2))\)
2 \((2,1,P_1,(f_3)) (2,3,\neg P_1,(f_4))\)
3 \((3,4,T,(f_5))\)
4 \((4,5,T,(f_6))\)
5 \((5,6,T,(f_7))\)
6 \((6,5,P_2,(f_8)) (6,7,\neg P_2,(f_7,f_8))\)
7 \((7,8,P_3=0,(f_9)) (7,8,P_3=0,0) (7,8,P_3>0,0)\)
8 \((8,7,P_1,(f_9)) (8,9,\neg P_1,(f_1))\)
9 \((9,11,P_3<0,0) (9,10,P_3=0,(f_{10})) (9,11,P_3>0,0)\)
10 \((10,11,T,(f_{11},f_{12}))\)
11 \((11,9,P_1,(f_3)) (11,1,H,\neg P_1,0)\)

Zero-assignment arcs sorted

8 \((7,8,P_3=0,0) (7,8,P_3=0,0)\)
11 \((9,11,P_3<0,0) (9,11,P_3>0,0)\)

THE AUXILIARY TABLES

<table>
<thead>
<tr>
<th>Test predicates</th>
<th>Assignments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (I&lt;6)</td>
<td>1 (I=1)</td>
</tr>
<tr>
<td>2 (I&lt;90)</td>
<td>2 (\text{KOUNT}(I)=0)</td>
</tr>
<tr>
<td>3 (\text{MOST} = \text{KOUNT}(I))</td>
<td>3 (I=I+1)</td>
</tr>
<tr>
<td>4 (\text{MARK} = g_I(x))</td>
<td>4 (\text{MARK} = \text{MARK}/5+1)</td>
</tr>
<tr>
<td>5 (\text{KOUNT(MARK)} = \text{KOUNT(MARK)} + 1)</td>
<td>5 (\text{KOUNT(MARK)} = \text{KOUNT(MARK)} + 1)</td>
</tr>
<tr>
<td>7 (\text{MOST} = \text{KOUNT}(I))</td>
<td>7 (\text{MOST} = \text{KOUNT}(I))</td>
</tr>
<tr>
<td>8 (I=2)</td>
<td>8 (I=2)</td>
</tr>
<tr>
<td>9 (\text{MOST} = \text{KOUNT}(I))</td>
<td>9 (\text{MOST} = \text{KOUNT}(I))</td>
</tr>
<tr>
<td>10 (z_1 \leftarrow \text{MOST})</td>
<td>10 (z_1 \leftarrow \text{MOST})</td>
</tr>
<tr>
<td>11 (z_2 \leftarrow L)</td>
<td>11 (z_2 \leftarrow L)</td>
</tr>
</tbody>
</table>

Figure 8. The tables for the program of 41.3.
literals formed from the state predicates, it is necessary to consider two cases:

1) \( v_i \) is the initial vertex of an arc. The literal appears in the clause in the form \( \neg q_i(\xi) \). It has been assigned to the start vertex; \( F \) to the halt vertex. A distinct state predicate has been assigned to each vertex, therefore, there are \( n \) distinct literals from the state predicates of the form \( \neg q_i(\xi) \); they are:

\[
\neg q_1(\xi), \neg q_2(\xi), \ldots, \neg q_n(\xi)
\]

where \( n \) is the number of vertices excluding \( S \) and \( H \).

2) \( v_i \) is the terminal vertex of an arc. The literal appears in the clause in the form \( q_i(\xi') \), where \( \xi' \) is \( \xi \) with 0 or more variables replaced by explicit assignment functions. When 0 variables have been replaced, (i.e., the assignment function for each variable is the identity function), \( \xi = \xi' \), and \( q_i(\xi') \) is the same as \( q_i(\xi) \). If one or more variables have been replaced by assignment functions, \( \xi = \xi' \), and \( q_i(\xi') \) is distinct from \( q_i(\xi) \). If two or more arcs have the same terminal vertex and identical assignment functions, the state predicates will be the same, i.e., \( q_i(\xi') \) will be distinct from \( q_i(\xi) \), but \( q_i(\xi') \) will occur in two or more clauses by virtue of \( v_i \) being the terminal vertex of two or more arcs with explicit assignment functions. (This condition will occur when the arithmetic IF statement occurs in the Fortran program in the form IF (exp) \( n_1, n_2, n_3 \), where for some \( i, j, 1 \leq i, j \leq 3 \), \( i \neq j \), or when two or more transfers are made to the same point in the program.) The distinct literals from state predicates of the form \( q_i(\xi) \) are obtained
by enumerating the abbreviated state predicates at those vertices which are terminal vertices of arcs which have 0 assignment functions and eliminating duplicates. The distinct literals from state predicates of the form $q_i(\xi')$ are obtained by enumerating the abbreviated state predicates at those vertices which are terminal vertices of arcs which have explicit assignment functions and eliminating duplicates. To locate the duplicates of this type easily, sort the list of arcs on terminal vertex. If for any vertex, two or more have identical assignment functions, $q_i(\xi')$ will appear in two or more clauses.

The set of distinct literals formed from the state predicates consists of those literals enumerated in 1 and 2.

43.15 The set of distinct literals in the wff of the program is the union of the set of distinct literals formed from the test predicates and the set of distinct literals formed from the state predicates. The number of matching complementary literals is counted from this set. The number of steps of resolution required to obtain the first resolution is $M/2$, where $M$ is the number of matching complementary literals in the set of distinct literals.

43.16 Example. An example of this procedure using the table in Figure 8 is given.

1) the set of distinct literals formed from the test predicates is:
   \{p_1, \neg p_1, p_2, \neg p_2, p_5<0, p_5'=0, p_5>0\}

2) the set of distinct literals formed from the state predicates is:
   \{\neg q_1(\xi), \neg q_2(\xi), \ldots, \neg q_{11}(\xi), q_1(\xi'), q_2(\xi'), q_3(\xi'), q_4(\xi'), q_5(\xi'), q_6(\xi'), q_7(\xi'), q_8(\xi'), q_9(\xi'), q_{10}(\xi'), q_{11}(\xi'), q_{12}(\xi), q_{13}(\xi)\}
3) the set of distinct literals in the wff of the program is the union of the two above sets.

4) the matching complementary literals are:
   \( P_1, \neg P_1, P_2, \neg P_2, q_8(\xi), \neg q_8(\xi), q_{11}(\xi), \neg q_{11}(\xi) \)

5) The number of steps of resolution required to obtain the first resolution is 4. If the first resolution contains the empty clause, a solution has been reached.

43.17 The above example was worked out in full detail to illustrate the full enumeration of the set of distinct literals in the wff of the program. To count the number of matching complementary literals, this full enumeration is not necessary. Only the set of literals from the test predicates need be enumerated. The literals from the state predicates which are of the form \( q_i(\xi') \) cannot have a matching complement. Such a literal would have to be of the form \( \neg q_i(\xi') \), and the only negated literals are of the form \( \neg q_i(\xi) \), i.e., \( v_1 \) is the initial vertex of the arc. Thus, any literal formed from the state predicate of a vertex \( v_i \) because \( v_i \) is the terminal vertex of an arc with explicit assignments will not have a matching complement in the wff of the program, and therefore need not be enumerated. There is a literal of the form \( \neg q_i(\xi) \) for each vertex \( v_i \), \( 1 \leq i \leq n \), where \( n \) is the number of vertices excluding \( S \) and \( H \). There will be a literal of the form \( q_i(\xi) \) for any vertex \( v_i \) appearing as the terminal vertex of an arc with 0 explicit assignments (i.e., all assignments are the identity function). Therefore, the number of matching complementary literals in the wff of the program is the sum of:

1) the number of matching complementary literals in the enumerated set of literals from the test predicates,

2) the number of literals of the form \( q_i(\xi) \). This is the
number of arcs with 0 explicit assignments and distinct terminal vertices.

3) the number of literals of the form \(-q_i(\xi)\) which match those of the form \(q_j(\xi)\). (This number is exactly the same as 2.)

44 An algorithm to count matching complementary literals.

44.1 Algorithm. An algorithm to count the number of matching complementary literals appearing in the wff of the program is:

1) Construct a table of vertices and arcs according to the method given in section 43. Sort the list of arcs on terminal vertex.

2) Enumerate the set of literals from the test predicates.

3) Count the matching complementary literals in this set, \(N_1\).

4) Count the number, \(N_2\), of arcs with 0 explicit assignments and distinct terminal vertices excluding the halt vertex. (This counting is easier if the list of arcs is sorted on terminal vertex.)

5) The number of matching complementary literals is given by:

\[ 2N_2 + N_1 \]

44.2 It is the case that this algorithm counts all matching complementary literals appearing in the wff of a program because by definition 32.2, the wff of the program is the universal quantification over the program variables of the conjunction of the wffs of the arcs.

The wff of an arc \(a = (v_i, e, v_j)\) is:

\[ q_i(\xi) \land \phi_a = q_j(\xi') \]

To use the resolution procedure, each component of the wff of an arc is replaced by its G"odel form and each wff of an arc is converted to disjunctive normal form. Each conjunct in the wff of the program is
a clause for the resolution procedure and the clause is considered simply the set of its disjuncts. For each arc \( a = (v_i, i, v_j) \), the clause is:

\[ \neg q_i(\xi), \neg \phi_a, q_j(\xi') \]

There are no other components to the wffs of the arcs, therefore any literal appearing in the set of clauses (in the wff of the program) does so by virtue of being present in the clause formed from the wff of an arc. There are four cases to consider:

1) The state predicate of a vertex \( v_i \) appears in the clause in the form \( \neg q_i(\xi) \) when \( v_i \) is the initial vertex of an arc. By construction a distinct predicate was assigned to each vertex, therefore each \( \neg q_i(\xi) \) is distinct. The vertices have been sequentially numbered \( S, 1, 2, \ldots, n, H \). To \( S \) was assigned the predicate \( T \) and to \( H \), the predicate \( F \). Therefore, there are \( n \) distinct state predicates of the form \( \neg q_i(\xi) \), \( 1 \leq i \leq n \), where \( n \) is the number of vertices excluding \( S \) and \( H \).

2) The test predicate of an arc appears in the clause formed from the wff of the arc in the form \( \neg \phi_a \). Because the program is finite, the set of test predicates is finite. It is enumerated and the duplicates eliminated, yielding the set of distinct test predicates. Count the matching complements.

3) The state predicate of a vertex \( v_i \) appears in the clause formed from the wff of an arc in the form \( q_i(\xi') \) when \( v_i \) is the terminal vertex of the arc. \( \xi' \) is the result of substituting the assignment function of the arc for the program variables in \( \xi \). If the assignment function for all variables is the identity function, then \( q_i(\xi') \) is the same as \( q_i(\xi) \). If there are one or more variables for which the assignment function is
not the identity function, then $q_1(\xi')$ is distinct from $q_1(\xi)$ in that one or more variables have been replaced by the corresponding assignment function. To illustrate this, let $\xi$ be the set $\{x_1, x_2, x_3\}$ and consider the following structure:

![Diagram](image)

The clauses for the wffs of the arcs $a_1$ and $a_2$ are:

- $C_{a_1}: \neg q_1(\xi), \phi_{a_1}, q_k(\xi')$
- $C_{a_2}: \neg q_k(\xi), \phi_{a_2}, q_1(\xi')$

Let the assignment functions along $a_1$ be $x_1^1 x_1, x_2^2 x_2, x_3^3 x_3$, i.e., all the identity function. The clauses are:

- $C_{a_1}: \neg q_1(x_1, x_2, x_3), \phi_{a_1}, q_k(x_1, x_2, x_3)$
- $C_{a_2}: \neg q_k(x_1, x_2, x_3), \phi_{a_2}, q_1(\xi')$

Clearly, $q_k(\xi')$ is the same as $q_k(\xi)$, and $q_1(\xi)$ and $\neg q_k(\xi)$ are matching complementary literals. Now, let the assignment functions along $a_1$ be $x_1^1 f(x_1), x_2^2 x_2, x_3^3 f(a)$. The clauses are:

- $C_{a_1}: \neg q_1(x_1, x_2, x_3), \phi_{a_1}, q_k(f(x_1), x_2, f(a))$
- $C_{a_2}: \neg q_k(x_1, x_2, x_3), \phi_{a_2}, q_1(\xi')$

Clearly, $q_k(\xi')$ is distinct from $q_k(\xi)$ and there are no matching complements. Therefore, for each arc $a = (v_1, \xi, v_j)$ with explicit assignment functions, the literal formed from the state predicate at vertex $v_j$ will appear in the clause in the form $q_j(\xi')$, and $q_j(\xi')$ is distinct from $q_j(\xi)$. There will not be a matching complement $\neg q_j(\xi')$ since the only negated literals from the state predicates are of the form $\neg q_1(\xi)$ where $v_1$ is the initial vertex of an arc. For each arc $a = (v_1, \xi, v_j)$ with no explicit assignment functions (i.e., all are the identity function), the literal formed from the
state predicate at vertex $v_j$ will appear in the clause in the form $q_j(\xi)$ and there will be a matching complement $\neg q_j(\xi)$ which is present because $v_j$ is the initial vertex of an arc. To arrive at the number of matching complementary literals of this kind, count all arcs with distinct terminal vertices (excluding the halt vertex) which have zero explicit assignments and multiply by 2.

4) It is not the case that all literals of the form $q_j(\xi')$ are distinct. If there are $m$ arcs entering vertex $v_j$, there will be $m$ clauses containing $q_j(\xi')$. If the assignment functions along any two or more of these $m$ arcs, the literals from the state predicates formed because $v_j$ is the terminal vertex of these arcs will be identical. To illustrate, let $\xi$ be the set $\{x_1, x_2, x_3\}$ and consider the following structure:

Let $\alpha_1, \alpha_2, \ldots, \alpha_m$ be the set of arcs entering vertex $v_j$.
Let $q_j(\xi'_1), q_j(\xi'_2), \ldots, q_j(\xi'_m)$ be the corresponding state predicates at vertex $v_j$. Let the assignment functions along arcs $\alpha_1$ and $\alpha_2$ be $x_1 = f(a), x_2 = f(x_2)$, and $x_3 = x_3$. The clauses are:
\[ C_{a_1} : \neg q_{i_1}(x_1, x_2, x_3), \neg f_{j_1}, q_j(f(a), f(x_2), x_3) \]

\[ C_{a_2} : \neg q_{i_2}(x_1, x_2, x_3), \neg f_{j_2}, q_j(f(a), f(x_2), x_3) \]

\[ \vdots \]

\[ \vdots \]

\[ C_{a_m} : \neg q_{i_m}(x_1, x_2, x_3), \neg f_{j_m}, q_j(\xi') \]

Clearly, \( q_j(\xi') \) is the same as \( q_j(\xi') \). However, for the reasons stated above in 3, there will not be a matching complement of the form \( \neg q_j(\xi') \). Therefore, in counting the number of matching complementary literals, it is not necessary to include this case.

Since we have considered all components of the clauses, the number of matching complementary literals in the conjunctive normal form of the wff of the program (the set of clauses) is exactly those counted in 2 and 3 above.

44.3 The number of steps of resolution required to obtain the first resolution is \( M/2 \), where \( M \) is the number of matching complementary literals occurring in the set of clauses; the application of this procedure to a group of Fortran programs is presented in the next chapter.
V. EVALUATION OF MANNA'S METHOD.

50 Introduction.

Using these procedures, it is now possible to estimate the amount of work required to verify the termination and correctness of a program by Manna's method. To evaluate the method, a group of "tested" Fortran programs were chosen; all programs had terminated and given correct results for at least one set of input data. For each program the graph \((AP,I)\) and the table of vertices and arcs were constructed; the matching complementary literals were counted. From this the number of steps of resolution required to obtain the first resolution is determined. If this contains the empty clause, a solution has been reached. The results are summarized in Figure 9.

Using Knuth's data [6], an estimated table of vertices and arcs was constructed. Knuth's study included the examination of a large number of programs in terms of the distribution of statement types so as to describe a "typical" program. Using these results, the amount of work required to verify the program was estimated; the results are given in Figure 10.

Based on these results and timing data about current resolution programs, the conclusion is reached that this procedure for verifying programs is not unreasonable. The effort involved in writing a program to construct the table of vertices and arcs and to apply flow-checking procedures appears to be reasonable. Since this appears to be a reasonable amount of work, the next step is to consider the possibility of obtaining a solution in the first resolution. It is
shown that except for trivial cases which can be detected, a solution in the first resolution is not possible. An experiment to study this problem further is described.

51 **Student Fortran programs.**

A group of 12 "tested" Fortran programs were chosen; 6 from an undergraduate text [3], and 6 from a set of problems assigned in a graduate course in numerical methods [4]. All programs had terminated and given correct results for at least one set of input data. The undergraduate programs were typical of those that would be assigned during a one semester course in programming. The problems included:

1) Given a set of 50 numbers, determine how many of these numbers are within a specified range.

2) Given a set of numbers, determine the largest number, the smallest number, and the largest number in absolute value.

3) Given a set of test scores, determine the mode.

4) Sort a set of numbers into ascending sequence using the exchange method of sorting.

5) The input data consists of questionnaire responses. Prepare a frequency table showing the responses to each question.

6) Write a program that includes routines for matrix addition, matrix subtraction, matrix multiplication, and developing the transpose of a matrix. The input data consists of two matrices and an operations code to indicate which operation(s) are desired.

These problems are quite simple and the programs are short; the longest program is 110 statements (the matrix program). None of these programs
included complicated Fortran expressions or sophisticated programming techniques; rather they appear to be designed to illustrate various standard elementary techniques. The programs for the graduate problems do not display complicated structures either; these problems are designed to illustrate methods of solution, not necessarily programming expertise. This set of problems included:

1) Integral evaluation by trapezoids and Romberg integration.
2) Euler's method of solving first order differential equations.
3) Solution of a second order differential equation by Gaussian elimination and the back solution.
4) Solution of simultaneous equations by a factorization method.
5) Solution of a system of equations using a fully-pivoted method.
6) Solution of a system of equations by two methods: cursive matrix factorization and a pivoting method.

For each program, the graph and tables were constructed; the number of steps of resolution required to obtain the first resolution was determined. The results are summarized in Figure 9.

52 An estimate from a "typical" program.

In a recent publication by D.E. Knuth [6], some statistics about the composition of Fortran programs is given. Knuth's study included examining a large number of programs in terms of the distribution of statement types in a typical program. His purpose was to discover quantitatively "what programmers really do", how Fortran is being used and how this affects compiler design. Among the surprising results are:

1) Arithmetic expressions have an average length of two operands.
2) The general style of programming showed little evidence of "sophistication".
<table>
<thead>
<tr>
<th>Program No.</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. statements</td>
<td>14</td>
<td>23</td>
<td>21</td>
<td>26</td>
<td>23</td>
<td>110</td>
<td>34</td>
<td>26</td>
<td>47</td>
<td>41</td>
<td>25</td>
<td>84</td>
</tr>
<tr>
<td>No. vertices</td>
<td>8</td>
<td>18</td>
<td>13</td>
<td>10</td>
<td>12</td>
<td>50</td>
<td>17</td>
<td>9</td>
<td>18</td>
<td>16</td>
<td>16</td>
<td>44</td>
</tr>
<tr>
<td>No. arcs</td>
<td>13</td>
<td>15</td>
<td>20</td>
<td>18</td>
<td>19</td>
<td>94</td>
<td>22</td>
<td>11</td>
<td>26</td>
<td>22</td>
<td>21</td>
<td>68</td>
</tr>
<tr>
<td>$N_1$</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>10</td>
<td>12</td>
<td>4</td>
<td>18</td>
<td>14</td>
<td>10</td>
<td>36</td>
</tr>
<tr>
<td>$N_2$</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>16</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>$2N_2$</td>
<td>8</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td>2</td>
<td>32</td>
<td>6</td>
<td>6</td>
<td>2</td>
<td>6</td>
<td>4</td>
<td>24</td>
</tr>
<tr>
<td>$2N_2 + N_1$</td>
<td>12</td>
<td>4</td>
<td>8</td>
<td>8</td>
<td>6</td>
<td>42</td>
<td>18</td>
<td>10</td>
<td>20</td>
<td>20</td>
<td>14</td>
<td>60</td>
</tr>
<tr>
<td>No. steps for 1st resolution</td>
<td>6</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>21</td>
<td>9</td>
<td>5</td>
<td>10</td>
<td>10</td>
<td>7</td>
<td>30</td>
</tr>
</tbody>
</table>

The number of statements is the count of all statements in the program excluding comment statements.

$N_1$ is the number of matching complementary literals in the set of distinct test predicates.

$N_2$ is the number of arcs with zero assignment functions and distinct terminal vertices, excluding H.

Figure 9. The summary of student Fortran programs.
3) About 95% of the DO statements used the default increment of 1; most DO-loops were quite short, involving only one or two statements.

4) Complex assignment statements are rare; 68% of those analyzed were trivial replacements of the form \( A = B \) where no arithmetic operators were present.

Knuth's conclusion is that compilers spend most of their time doing surprisingly simple things. While the data from this study does not provide sufficient information to count the exact number of matching complementary literals, it does permit a rough estimate based on the following:

1) Each logical IF statement specifies a vertex with 1 entering arc and 2 exiting arcs. The literals formed from the test predicates of these 2 arcs will appear in the set of clauses as matching complementary literals. Assume each logical IF statement contains a distinct Fortran expression.

2) Each arithmetic IF statement specifies a vertex with 1 entering arc and 3 exiting arcs. The literals formed from the test predicates of these 3 arcs will appear in the set of clauses as distinct literals and will have no matching complements.

3) Each DO-loop specifies two vertices. The first is from the DO statement and has 1 entering arc containing the assignment function for the initialization of the loop and 1 exiting arc whose test predicate is \( T \). The second vertex follows the terminal statement of the DO-loop and has 1 entering arc and 2 exiting arcs. The literals formed from the test predicates
of these 2 arcs will appear in the set of clauses as matching complementary literals. Assume each such pair is distinct.

3) There is no way to estimate the number of arcs with zero assignment functions.

The estimated number of matching complementary literals for a "typical" Fortran program according to Knuth's study is summarized in Figure 10.

53 Evaluation of the Procedure.

53.1 There are a number of resolution programs in existence; experiments have been made to judge the effectiveness of various strategies [1, 8, 18]. Some results have been published in terms of the number of clauses generated and retained, the length of the literals, and depth or level number. Level number is related to the resolution number \( R^n(S) \), where \( n \) indicates the \( n \)th resolution; these programs embody techniques whereby the number of clauses generated and retained for each resolution is controlled, the full set is not generated. It is difficult to relate and apply these results to this particular problem. The examples given have a small number of clauses in the initial set, and there is a known proof for these examples. In one group of 10 examples [1], the number of clauses in the initial set ranged from 4 to 15; clauses generated from 16 to 136; clauses retained from 0 to 17; level of proof from 4 to 10; time to obtain a proof from .6 to 24.5 seconds. In another group of 4 examples, the initial set was not shown [18]. This experiment measured time and memory requirements for each example using different parameters, in all 19 cases were illustrated. This experiment was designed to show the effectiveness of a particular strategy; four cases failed to produce
<table>
<thead>
<tr>
<th>Statement type</th>
<th>No.</th>
<th>Vertices</th>
<th>Arcs</th>
<th>Assign.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assignment</td>
<td>178</td>
<td></td>
<td></td>
<td>178</td>
</tr>
<tr>
<td>READ</td>
<td>2</td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>WRITE</td>
<td>18</td>
<td></td>
<td></td>
<td>18</td>
</tr>
<tr>
<td>Logical IF</td>
<td>44</td>
<td>44</td>
<td>132</td>
<td></td>
</tr>
<tr>
<td>Arith. IF</td>
<td>19</td>
<td>19</td>
<td>76</td>
<td></td>
</tr>
<tr>
<td>DO-loops</td>
<td>17</td>
<td>34</td>
<td>34</td>
<td>34</td>
</tr>
</tbody>
</table>

Total number of arcs specified - 242

Number of matching complementary literals (from logical IF statements and DO-loops) - 122

Number of steps of resolution estimated - 61

Figure 10. Estimate from a "typical" program.
a proof within the time and memory constraints. In those cases where a proof was obtained, the number of clauses generated ranged from 337 to 40,094; clauses retained from 73 to 1,443; level of proof from 4 to 10; time to obtain a proof from .5 to 411 seconds.

53.2 While these are interesting results, they are not directly applicable to the present problem. To obtain the first resolution, it is necessary to generate all possible resolvents for each pair of matching complementary literals. Current resolution programs allow the user to direct the process in different ways. For example, pick two clauses and generate all possible resolvents; or pick two matching complementary literals and generate all possible resolvents among the entire set. Current programs running on reasonably fast computers can do 1-2 hundred of the first type per second; the second type is slower by a factor of about 10 [14]. For the examples in Figures 9 and 10, the number of clauses in the original set ranges from 11 to 242; the number of matching complementary literals ranges from 4 to 122. Based on all these factors, it appears that the first resolution can be generated within a reasonable amount of time.

53.3 The next question to be considered is: What is the effort required to write a program to construct the wff of the program? This is required to obtain the set of clauses for the resolution procedure. A rough estimate is that it is approximately equivalent to that of writing a compiler because each statement must be parsed and the Gödel form of each test predicate and assignment function must be constructed. Knuth's conclusion that compilers spend most of their time doing very simple things suggests that a relatively simple unsophisticated parsing algorithm will suffice; the construction of the Gödel forms would replace
complicated compiler methods, rather than add a further complexity. A more accurate estimate is beyond the scope of this paper.

33.4 On the basis of constructing the formula and using an existing resolution program to obtain the first resolution, Manna's method to verify the termination and correctness of programs appears to be quite reasonable. But, it is at all reasonable to expect a solution in the first resolution? Except for exactly one trivial program which can be detected, a solution will not appear in the first resolution.

33.5 **Theorem** Provided there is no path from S to H along which all test predicates \( \phi_{a_i} = T \), 1 ≤ i ≤ n, where n is the number of arcs along this path, there are no programs \( (AP, I) \) with formula \( (W_{AP}', I) \) for which \( \Box \in R^I(W_{AP}', I) \), i.e., a proof of the termination or correctness of the program \( (AP, I) \) will not appear in the first resolution.

**Proof**

By definition 32.2, the wff of the program is the universal quantification over the program variables of the conjunction of the wffs of the arcs.

The wff of an arc \( \alpha = (v_i, \xi, v_j) \) is:

\[
q_1(\xi) \land \phi_{\alpha} = q_j(\xi')
\]

To use the resolution procedure, each component of the wff of an arc is replaced by its Gödel form and each wff of an arc is converted to disjunctive normal form. Each conjunct in the wff of the program is a clause for the resolution procedure and the clause is considered simply the set of its disjuncts. For each arc \( \alpha = (v_i, \xi, v_j) \), the clause is:

\[
\neg q_1(\xi), \neg \phi_{\alpha}, q_j(\xi')
\]

There are no other components to the wffs of the arcs, therefore any literal appearing in the set of clauses (in the wff of the program) does so by virtue of being present in the clause formed from the wff
of an arc. To obtain the empty clause in the first resolution, there must exist two clauses whose resolvent is the empty clause. The four factors influencing the number and type of literals in the clause are:

1) the initial vertex may be either the start vertex or some other vertex, \( v_i \);

2) the terminal vertex may be either the halt vertex or some other vertex, \( v_j \);

3) the test predicate of the arc may be either \( T \) or some predicate \( \phi \); or

4) the number of assignment functions along the arc is either 0 or some number \( n \), \( 1 \leq n \leq m \), where \( m \) is the number of program variables.

There are 16 possible combinations of these four factors. As the construction does not permit assignment functions along an arc leading to the halt vertex, four of these are not applicable. The flow-checking procedure has detected and eliminated those programs containing a path from \( S \) to \( H \) along which all test predicates are \( T \), and also those programs which do not satisfy Manna's definition. There remain 8 possible clause structures; they are:

<table>
<thead>
<tr>
<th>Case</th>
<th>Init.</th>
<th>Term.</th>
<th>Pred.</th>
<th>A.F.</th>
<th>Clause from the wff of the arc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( S )</td>
<td>( v_j )</td>
<td>( \phi )</td>
<td>0</td>
<td>( \neg \phi, q_j(\xi) )</td>
</tr>
<tr>
<td>2</td>
<td>( S )</td>
<td>( v_j )</td>
<td>( \phi )</td>
<td>( n )</td>
<td>( \neg \phi, q_j(\xi') )</td>
</tr>
<tr>
<td>3</td>
<td>( S )</td>
<td>( v_j )</td>
<td>( T )</td>
<td>( n )</td>
<td>( q_j(\xi') )</td>
</tr>
<tr>
<td>4</td>
<td>( S )</td>
<td>( H )</td>
<td>( \phi )</td>
<td>0</td>
<td>( \neg \phi )</td>
</tr>
<tr>
<td>5</td>
<td>( v_i )</td>
<td>( v_j )</td>
<td>( \phi )</td>
<td>0</td>
<td>( \neg q_i(\xi), \neg \phi, q_j(\xi) )</td>
</tr>
<tr>
<td>6</td>
<td>( v_i )</td>
<td>( v_j )</td>
<td>( \phi )</td>
<td>( n )</td>
<td>( \neg q_i(\xi), \neg \phi, q_j(\xi') )</td>
</tr>
<tr>
<td>7</td>
<td>( v_i )</td>
<td>( v_j )</td>
<td>( T )</td>
<td>( n )</td>
<td>( \neg q_i(\xi), q_j(\xi') )</td>
</tr>
<tr>
<td>8</td>
<td>( v_i )</td>
<td>( H )</td>
<td>( \phi )</td>
<td>0</td>
<td>( \neg q_i(\xi), \neg \phi )</td>
</tr>
</tbody>
</table>
For each case, construct a clause such that the resolvent is the empty clause and examine the arc from which such a clause would be formed.

**Case 1.** The original clause was formed from an arc of which the initial vertex is $S$, the test predicate is $\phi_\alpha$, the terminal vertex is $v_j$, and no assignments are made along the arc. To resolve to the empty clause requires a clause of the form $\neg q_j(\xi)', \phi_\alpha$. Such a clause would be formed from an arc of which $v_j$ is the initial vertex, the test predicate is $\neg \phi_\alpha$, and the terminal vertex is $H$, and no assignments are made along the arc. For these two clauses to be present in the original set, both $\phi_\alpha$ and $\neg \phi_\alpha$ are true, a contradiction.

**Case 2.** The original clause was formed from an arc of which the initial vertex is $S$, the test predicate is $\phi_\alpha$, the terminal vertex is $v_j$, and assignments are made along the arc. To resolve to the empty clause requires a clause of the form $\neg q_j(\xi)', \phi_\alpha$. The only way that the state predicate of a vertex appears in a clause as a negation is when the vertex $v_j$ is the initial vertex of the arc, and then it appears in the form $\neg q_j(\xi)$. The only time $q_j(\xi')$ is the same as $q_j(\xi)$ is when there are no assignments along that arc, which contradicts the construction of the original clause. Also, for $\phi_\alpha$ to resolve with $\neg \phi_\alpha$ implies both are true, a contradiction.

**Case 3.** The original clause was formed from an arc of which the initial vertex is $S$, the test predicate is $T$, the terminal vertex is $v_j$, and assignments are made along the arc. To resolve to the empty clause requires a clause of the form
Such a clause would be formed from an arc of which the initial vertex is $v_j$, the test predicate is $I$, the terminal vertex is $H$, and no assignments are made along the arc. The clause from this arc contains the state predicate in the form $\neg q_j(\xi)$. If $q_j(\xi')$ is the same as $q_j(\xi)$, there were no assignments along the arc of which $v_j$ is the terminal vertex, which contradicts the construction of the original clause.

**Case 4.** The original clause was formed from an arc of which the initial vertex is $S$, the test predicate is $\phi_a$, the terminal vertex is $H$, and no assignments are made along the arc. To resolve to the empty clause requires a clause of the form $\phi_a$. Such a clause would be formed from an arc of which the initial vertex is $S$, the test predicate is $\neg \phi_a$, the terminal vertex is $H$, and no assignments are made along the arc. This means that the set of test predicates at $S$ includes $\phi_a$ and $\neg \phi_a$; the set of test predicates at any vertex is required to be complete and mutually exclusive. Therefore, if both clauses are present in the original set, the program does not meet Manna's definition. We have already eliminated all programs which do not meet the definition, thus, a contradiction.

**Case 5.** The original clause was formed from an arc of which the initial vertex is $v_i$, the test predicate is $\phi_a$, the terminal vertex is $v_j$, and no assignments are made along the arc. To resolve to the empty clause requires a clause of the form $\neg q_j(\xi)$, $\phi_a$, $q_i(\xi)$. Such a clause would be
formed from an arc of which the initial vertex is \( v_j \),
the test predicate is \( \neg \phi_a \), the terminal vertex is \( v_i \), and
no assignments are made along the arc. As no assignments
are made, both \( \phi_a \) and \( \neg \phi_a \) are true, a contradiction.

Case 6. The original clause was formed from an arc of which the
initial vertex is \( v_i \), the test predicate is \( \phi_a \), the
terminal vertex is \( v_j \), and there are assignments along
the arc. To resolve to the empty clause requires a
clause of the form \( \neg q_j(\xi') \), \( \phi_a \), \( q_i(\xi) \). Such a clause
would be formed from an arc of which the initial vertex
is \( v_j \), the test predicate is \( \neg \phi_a \), the terminal vertex
is \( v_i \), and no assignments are made along the arc. But
the state predicate appearing in this clause because
\( v_i \) is the initial vertex is in the form \( q_j(\xi) \). If \( q_j(\xi') \)
is the same as \( q_j(\xi) \), there were no assignments along
the arc of which \( v_j \) is a terminal vertex, which contradicts
the construction of the original clause.

Case 7. The original clause was formed from an arc of which the
initial vertex is \( v_i \), the test predicate is \( T \), the
terminal vertex is \( v_j \), and there are assignments along
the arc. To resolve to the empty clause requires a
clause of the form \( \neg q_j(\xi) \), \( q_i(\xi) \), where \( q_j(\xi') \) is the
same as \( q_j(\xi) \). For this to be the case, it requires
that no assignments are made along the arc for which
\( v_j \) is the terminal vertex, which contradicts the
construction of the original clause.

Case 8. The original clause was formed from an arc of which the
initial vertex is \( v_i \), the test predicate is \( \phi_a \), the
terminal vertex is N, and there are no assignments along the arc. To resolve to the empty clause requires a clause of the form $\phi_a \land q_i(\xi)$. Such a clause would be formed from an arc of which the initial vertex is S, the test predicate is $\sim \phi_a$, the terminal vertex is $v_i$, and no assignments are made along the arc. As no assignments are made, both $\phi_a$ and $\sim \phi_a$ are true, a contradiction.

Since all possible clause structures have been examined and in none of the 8 cases can there exist a complementary clause which will resolve to the empty clause in the first resolution, a proof of the termination or correctness of the program (AP,I) will not appear in the first resolution.

53.6 As a proof of the termination or correctness of a program (AP,I) will not appear in the first resolution, it becomes necessary to investigate resolution techniques further. Is it possible to predict if or when a solution might be reached? In the general case this is a very difficult problem. The problems arising in the use of automatic theorem proving techniques include the following:

1) the generation of a large number of irrelevant clauses;
2) the generation of a discouragingly large number of apparently relevant but redundant clauses;
3) the need for determination of a direction in which to proceed—one in which the expectation of finding a proof within reasonable time and memory requirements is high;
4) the reduction both of number (ideally to zero) and sensitivity to choice of parameters governing the theorem proving procedure; and
5) the retention of clauses germane but dispensable to the theorem under consideration [18].
A number of strategies have been developed to deal with these very serious problems; among them are the following:

1) purity, subsumption, and replacement principles [11]. The application of these principles to a set $S$ of clauses produces a set $S'$ which either has fewer clauses than $S$ or has the same number of clauses as $S$ but with one or more shorter clauses.

2) the set of support strategy [18]. The application of this strategy can reduce the number of clauses generated and retained.

3) unit preference strategy [17].

4) model elimination [7, 8].

5) demodulation [19].

In general, a theorem proving program will embody several of these strategies. All of them rely on some degree of heuristic problem solving and deal with the general problem. The formulas constructed by Manna's algorithm have a well defined and rigid structure. The number of distinct literals can be counted. The number of clauses in the original set is the number of arcs in the program $(AP,I)$. The length of the literals is a function of the number of variables in the program. The maximum number of literals per clause is 3. Since a great deal is known about the form and composition of these formulas, it may be possible to predict the behavior of one of the above techniques. Additional study into general resolution techniques is beyond the scope of this paper, but an experiment to investigate this problem is clearly indicated.
The experiment.

Select a group of carefully "tested" Fortran programs; student programs provide a good starting point. Construct the wffs of the programs by hand. Construct the Gödel forms of test predicates and state predicates; assemble the set of clauses. Select an existing resolution program and proceed to attempt to obtain the empty clause, a proof of the program. The data to collect from this experiment should include, for each run:

1) the number of clauses in the original set,
2) the number of matching complementary literals in the original set,
3) the number of individual variables,
4) the number of vertices,
5) the length of the literals,
6) the number of clauses generated,
7) the number of clauses retained,
8) the number of steps of resolution required,
9) the proof level,
10) the number of substitutions for individual variables, and
11) the time required to reach a solution, or the time limitation (in the case where a proof is not obtained).

In addition to these, many of which are specific to these particular formulas, also collect any data from the resolution program such as parameter settings and any heuristics used. Of course, vary as many of these items as possible, in order to observe the effect on the procedure. For example, a program to perform matrix operations might be tried using a maximum matrix size of $5 \times 5$; and again with a maximum of $10 \times 10$, to observe the effect of changing only the size of the
literals. For those programs for which a solution is reached, what is the size of the Fortran source program? For any heuristics applied, keep a list of the particular paths chosen. Through such experiments with heuristic modifications, one may gain insight into the problem of tuning resolution programs and may provide guidance to the proper approach in proving theorems with specific bounds.

There is considerable work yet to be done to investigate Manna's method of verifying the termination and correctness of programs; the work done here indicates the next direction of study and points out some of the observations to be made. These things yet to be done are beyond the scope of this present paper.
REFERENCES.


[4] Fender, F.G. *A set of problems assigned in a graduate course "Numerical Analysis" at Rutgers University, Spring 1971.*


