150
Notre Dame Journal of Formal Logic
Volume X, Number 2, April 1969

DOING LOGIC BY COMPUTER

RICHARD L. PURTILL

Teachers of logic often tell their classes that a great number of tasks in logic can be performed mechanically. Surprisingly enough, most logicians have failed to make full use of existing mechanical devices which could demonstrate this point, and by so doing, increase the interest of students in the theory and application of logic.

In this paper[1], I will first describe some simple ways of using digital computers to do certain tasks connected with logic, for example, draw up truth tables, decide whether statements are tautologies, contingent or contradictory and whether arguments are valid or invalid. I will then go on to discuss some extensions of these techniques. The advantage of the techniques I am about to describe is that they can be used on most machines which can be programmed by means of FORTRAN, which is a "programming language" in which instructions can be written in combinations of English and algebraic statements. Many such machines are in use including some small computers used mainly for bookkeeping operations. There are a few specially designed logical computers in existence, and some rather rare and expensive computers have certain logical capabilities in addition to their mathematical ones. But such computers exist in fairly small numbers, and are fairly difficult of access, whereas machines which can be programmed in FORTRAN are fairly common, and more likely to be available.

Let us begin with a very simple problem in logic. A and B are true statements; X and Y are false statements. Our problem is to discover the truth or falsity of certain compound statements made up of A's, B's, X's and Y's joined by the truth functional connectives "." ("and") "v," ("or"). Now the average computer has no such special symbols as, "." and "v" and it is designed to do mathematical rather than logical calculations. What

1. Part of this research was carried out with the aid of several grants made by the Research Office of Western Washington State College. I wish to thank the following former students who contributed ideas and help: David Ault, Larry Lingbloom and Richard Thompson.

Received April 10, 1967

we must do is to translate our logical problem, which the machine cannot solve, into a mathematical problem, which it can. One simple way to go about this (which we will eventually discard for a better technique) is to replace all "v" ("or") symbols with "+" and all "." ("and") symbols by "*" (the FORTRAN symbol for "times," or multiplication). We then give A and B the numerical value "1" and X and Y the numerical value "0." In this way we change e.g. $((A \vee X) . ((X \vee Y) . B)) . (A \vee (X . Y))$ which means nothing to the machine into ((1 + 0) * ((0 + 0) *1)*(1 +(0 * 0)), a simple calculation which the machine can do in a few microseconds. It is easy to see that any true compound statement will result in a positive number, while any false compound statement will result in zero, when subjected to this process. What, if anything, have we gained by doing the problem in this way? In the first place, the machine once told that A and B are equal to one, and X and Y are equal to zero, will make the substitutions for us: we can write $(A + X) * ((X + A) * B)$ etc. in our program and the machine will calculate it as (1 + 0) * ((0 + 0) * 1) etc. without further instructions. Furthermore, the machine will calculate the value of any expression, no matter what its complexity without mistakes and in a very short time.

Besides its ability to calculate, the standard computer has the ability to test the value of an expression and do one thing if its value is a negative number, another if its value is zero, and yet another if its value is a positive number. If we like, we can use this capability and instruct the computer to type out the word "FALSE" if the value of an expression is zero, the word "TRUE" if the value of an expression is a positive number, thus saving us the trouble of retranslating from numbers to truth values.

Without some substitute for "$\sim$" ("not") we cannot express all possible compound statements, and it would be convenient to have substitute for "$\supset$" ("if-then") and "$\equiv$" ("if and only if"). No mathematical operators are as conveniently analogous to these symbols as "+" and "*" are to "v" and ".". Fortunately, almost all computers allow us to define special operators, technically called "functions," to perform non-standard calculations. Thus, we can define a function named "**X**" such that, if the machine reads the expression "**X**(A)" it will ascertain the value of A, if it is zero change it to one, and if it is any positive number change it to zero. Furthermore, the complexity of the expression operated on by "**X**" does not matter. If the machine reads "**X**$((A + X) * ((X + A) * B))$" it will calculate the value of "$(A + X) * ((X + A) * B)$" and if it is zero change it to one, or if it is positive change it to zero. Also eg. "**X**(**X**(**X**(**X**(A)))))" is a perfectly legitimate expression and will be correctly calculated by the machine.

So far we have substitutes for "." ("and"), "v" ("or"), and "$\sim$" ("not"): which will enable us to express any statement in the propositional calculus. It is possible to define a function, named for example, "**FIF**"[2] which will take the place of "$\supset$" ("if-then"). One way to do this is to give

2. For technical reasons these functions may not begin with the letters I, J, K, L, M, or N.

the machine the following instructions: "When you read the expression **FIF**(A,B), subtract B from A. If the result is zero or negative, substitute a value of 1 for the expression **FIF**(A,B), if the result of the subtraction is positive substitute a value of 0 for **FIF**(A,B)." This instruction gives us an analogue for "$\supset$" ("if-then"). "$p \supset q$" ("if p then q") is true if p and q are both true, both false, or if p is false and q true. It is false if p is true and q false. Now substitute one and zero for true and false and it can be seen that we want a value of one (=true) for the combinations one-one, zero-zero, and zero-one, and a value of zero (=false) for one-zero. The instructions described have this effect. One minus one is zero, and the machine substitutes one. One minus zero is one, and the machine substitutes one. One minus zero is one, and the machine substitutes zero. Zero minus one is minus one, and the machine substitutes one, and finally zero minus zero is zero, and the machine substitutes one. If we use this simple way of defining "if-then" functions, however, we must now define a special "v" ("or") function to replace "+." For otherwise, if we had $(A \vee B) \supset C$, where A,B and C were all true this would be calculated as (1 + 1) - 1 and the machine would mistakenly substitute zero, that is call the expression false. But it is easy to define a function, named for example, "**OR**" which will leave a zero unchanged but substitute a 1 for any positive number that results from addition. That is, the machine is told "if you read **OR**(A,B), add A and B. If the result is zero, let it stand, but if the result is any positive number, substitute 1." A function for "$\equiv$" ("if and only if") can also be defined, and called for example "**EQ.**" We could retain "*" for "." ("and") but for consistency it is as well to define a separate "**AND**" function. We then have a quasi-Polish notation which, however, needs punctuation.

So far so good. But calculating truth values of compound statements composed of statements with fixed truth values is not a very advanced part of propositional calculus which is not a very advanced part of logic. Can we determine whether statements expressed in terms of variables are tautologous, contingent or contradictory, and whether arguments are valid or invalid? We can, but let us first consider a simpler problem; that of writing truth tables by machine.

This involves another capability of computers: the ability to repeat a given task many times while keeping a "tally" of the number of repetitions as it goes along each repetition adding to the tally number. In fact, the task itself is generally written as a function of the tally number, or index, e.g. If we wish to square the digits from 1-100 we merely instruct the machine to square the current value of the tally and store the separate results. Such statements might read:

```
DO 1 I = 1,100
1 SQ (I) = I **2
```

The machine at the start, looks at the current value of the index, which is 1, squares it and stores it in a location it has reserved called SQ (1). The second time around, the tally is 2, so the machine squares 2 and stores it in a location called SQ (2). This it would continue, through the Jth

repetition, squaring J and storing it in SQ (J) that is, the Jth location reserved for squares, till it got to 100, which it sees is the upper limit on the number of repetitions, so after squaring and storing in SQ (100) it would go on to the next instruction.

Now in order to write a truth table we put a number of the repetition instructions, called "DO statements" or "DO loops" together in series. Suppose that we want to write a truth table for three variables, first using 1's and 0's instead of T's and F's. We put three "DO statements" together, each one of which is instructed to set its tally number (or actually for technical reasons a number computed from the tally number)[3] first at zero and then at one. (Call the tally number of the first "DO statement" I, of the second J, and of the third K.)

When the machine reads the first DO statement it sets the tally, I, to zero, and goes on. It then encounters the next DO statement, sets J to zero and goes on. It then encounters the third DO statement, sets K to zero and goes on to the next instruction. Suppose that this is to print out the values of I, J, and K. This would give us the line:

0 0 0

Now the crucial thing is what the machine does next. Since the third DO statement instructs the machine to do two things, the machine will keep repeating till it has done them. It thus goes back and changes the value of K to one. Since the machine has not yet changed the value of I and J they remain the same, (i.e. zero) and the machine goes on again to the next statement which is, of course, (since the machine has gone back to the third DO statement) the instruction to print I, J, and K. This time we get the line:

0 0 1

Now the machine has done all the repetitions required by the third DO statement, and it goes back to the second DO statement and changes J from zero to one. But since it has gone back, it comes again to the third DO statement which must begin with the value 0 first time around and so sets K at zero again. It then comes to the PRINT, I, J, K, instruction again and prints the line:

0 1 0

As before, it repeats with K as one, giving the line

0 1 1

Now as the second DO statement has been "satisfied," that is, its instructions have been carried out, so the machine goes back to the first DO statement and changes I to one. But what comes next--the second DO statement! So everything described so far is repeated again with I at zero,

---

3. The tally number cannot be zero, so we set it first at 1 and then at 2, then subtract 1 from this tally number so that the number we use is first zero, then 1.

that is J is set first at 0, then at 1 while K is first 0, then 1 each time resulting in the lines:

1 0 0
1 0 1
1 1 0
1 1 1

We have had then, eight lines:

0 0 0
0 0 1
0 1 0
0 1 1
1 0 0
1 0 1
1 1 0
1 1 1

which is a truth table for three variables using 1's and 0's for **T**'s and **F**'s. (The table is upside down, but this can be remedied.)[4] The machine now has finally obeyed all its repetition instructions and stops or goes on to other tasks as instructed. By printing out the contents of certain memory locations labeled by use of I, J, and K instead of the values of I, J, and K, we can get the machine to print **T**'s and **F**'s instead of 1's and 0's. We can easily have the machine label the rows P, Q, R or whatever variables we are using. And there is no limit to the number of variables for which we can draw up a truth table. For example, I have one for eight variables, which has 256 lines and covers about three sheets of paper. In itself, it is merely a curiosity: since the techniques I am about to describe make it unnecessary for human beings to actually look at truth tables.

Suppose that our problem is to determine whether some statement in the propositional calculus is a tautology, contingent, or a contradiction. Take, for example, $(P \vee Q) \equiv (P \vee (Q. \sim P))$ (which happens to be a tautology, but is a particularly counter-intuitive one). First, we need to rewrite this, as we did with our fixed truth-value statements. It becomes **EQ** (**OR** (P,Q), **OR** (P, **AND**(Q,**X**(P)))). We then need to test this by finding its value for each line of the truth table. We can do this simply by setting up two DO loops similar to those described above whose tally numbers, are labeled P and Q. Thus, the first time through, both P and Q will be zero, the second time P will be zero and Q one, the third P will be one and Q zero and the fourth, both will be one. By simply putting the statement to be tested in the position of the PRINT, I, J, K, statement in our previous example, its value will be computed for each set of values of P and Q. (For any given set of values, the machine precedes just as in the first case we discussed.) We can then make the next statement one of the type that test the value of an expression and take different paths depending on whether the

4. By subtracting the tally from 2 instead of subtracting 1 from the tally.

value is a zero or a positive number. For any line which gives the value zero to the expression to be tested, we can make some sort of tally, and if we like, also print out that line of the truth table. If all lines of the truth table give the statement being tested a value of zero, (that is, false) it is of course contradictory; if no line gives it a value of zero, it will of course be a tautology. If some lines but not others give the expression a value of zero, then it is contingent, and as I have said the lines for which the expression is false can be printed out. An argument can be tested for validity by testing whether it becomes a tautology when written as a statement in which the premises are joined by "ands" and are asserted to jointly imply the conclusion.

The programs described so far actually exist, and have worked correctly exercises from a number of standard logic texts, for example those in chapter two of Copi's *Symbolic Logic*.

Our success so far has been achieved by reducing logical manipulations to arithmetical ones, and by using the special reiterative features of the DO statement to generate truth tables. We can also perform other tasks which are normally performed by symbolic manipulation, by working through truth tables and building on the programs described so far. One such task is finding the Boolean expansion of a statement. Corresponding to the lines of the truth table for n variables are the disjuncts of the Disjunctive Boolean Normal Form for n variables of a tautology.

p	q	r	
T	T	T	$(p.q.r)$
T	T	F	$(p.q.\sim r)$
T	F	T	$(p.\sim q.r)$
T	F	F	$(p.\sim q.\sim r)$
F	T	T	$(\sim p.q.r)$
F	T	F	$(\sim p.q.\sim r)$
F	F	T	$(\sim p.\sim q.r)$
F	F	F	$(\sim p.\sim q.\sim r)$

The Disjunctive Boolean Normal Form of any expression consists of the disjunction of those disjuncts corresponding to the lines for which it is true. Thus the expression "$(p.q)$" is true for the first and second lines of the truth table and its Disjunctive Boolean Normal Form is "$((p.q.r) \vee (p.q.\sim r))$."

So by a slight adjustment of our previous program we can print out the Disjunctive Boolean Normal Form of an expression instead of printing the lines of a truth table for which it is false. Since a contradiction has no Disjunctive Boolean Normal Form we can utilize the Conjunctive Boolean Normal Form of contradictions.

A somewhat more complex program will serve to simplify expressions with a limited number of variables. It can be shown that for n variables there are 2^{2^n} distinct non-equivalent expressions which can be formed from these variables. These correspond to the 2^{2^n} possible arrangements of **T**'s and **F**'s under an expression for the 2^n lines of a truth table for n variables.

Of course any expression with a given pattern of **T**'s and **F**'s for some standard form of truth table is equivalent to any other expression with that pattern for that standard form of truth table. Thus, with the standard truth table we have been using "$((\sim p \vee \sim q) \cdot (p \vee q))$" and "$\sim (p \equiv q)$" have the same pattern;

p q	$\sim (p \equiv q)$	$((\sim p \vee \sim q) \cdot (p \vee q))$
T T	**F**	**F**
T F	**T**	**T**
F T	**T**	**T**
F F	**F**	**F**

and are thus equivalent.

Now, of a set of equivalent expressions, some must be simpler than others. By a certain amount of conventionalization with regard to what will count as simplicity a unique simplest expression can be found for each pattern of **T**'s and **F**'s. We can record this list of simplest expressions in the computer's memory, and for any expression find its equivalent simplest expression, by finding its pattern of **T**'s and **F**'s for a standard truth table, and locating in the machine memory the simplest expression with that pattern.

In practice one convenient way to do this is to replace the **T**'s and **F**'s by ones and zeroes and read the pattern from top to bottom as a binary number. Thus $p \vee (\sim p \supset q \vee (\sim q \supset r)))$ has the following pattern:

$$p \vee (\sim p \supset (q \vee (\sim q \supset r)))$$

p q r		or numerically
T T T	**T**	1
T T F	**T**	1
T F T	**T**	1
T F F	**T**	1
F T T	**T**	1
F T F	**T**	1
F F T	**T**	1
F F F	**F**	0

Read as a binary number this is 11111110, which in decimal notations is 254. We then look in a memory location labelled by use of the number 254 and find the simplest form corresponding to $p \vee (\sim p \supset (q \vee (\sim q \supset r)))$, which is $p \vee (q \vee r)$. The computation of the decimal number, (which I call the *Characteristic Number* of an expression) which corresponds to the binary number which correspond to the pattern of **T**'s and **F**'s, can be performed by a fairly minor modification of the programs described so far.

This method of simplification would be ideal except for one fatal limitation: while there are a manageable 256 simplest expressions for 3 variables, there are 65,536 simplest forms for 4 variables and a fantastic 2^{32} simplest forms for 5 variables. Thus, though I have a working program which will solve, in a few seconds, the most complex simplification problems given in logic books (none that I have seen go beyond three variables) the method is in a way a dead end.

A more general solution to the simplification problem is to first use the Disjunctive Boolean Normal Form writing program then apply the relatively cut and dried techniques available to simplify the resulting Boolean Form. Mechanization of this second stage is a current research problem. The Disjunctive Boolean Normal Form writer has no inherent limitations, though in fact the working program is only for three variables.

Thus, the existing programs cover all the essential manipulations of propositional logic. Proofs, of course, are never really necessary in propositional logic, since truth table techniques can always show whether a given inference is valid. The techniques described could be modified to check proofs, although there are special problems in distinguishing a genuine proof from a string of disconnected lines, each of which follows from the premises. The creation of proofs is a different sort of problem although there are techniques available for propositional logic which could probably be computerized.

Beyond propositional logic, some possibilities exist for mechanization of logical calculations. I have a rather cumbersome program which will take syllogisms written in what approximates to ordinary English and check their validity. I now have programs for propositional modal logic, using techniques related to those which I have described for propositional logic.

DO loops which go from 1 to 4 are used to create the four-valued tables by which modal propositions can be tested, and new functions called **REQ** (for necessity) and **POS** (for possibility) are used, along with redefinitions of the other connectives for the four-valued tables. These functions can be adjusted to create a modal S3, S4 or S5 system, as desired. Some very complex expressions can be tested, and some rather interesting results are beginning to come out of this research, which I hope to report on in due course.

The mechanization of logical calculations involving quantified statements is a difficult problem for which I see at present no very satisfactory solutions.

The advantage of the programs described in this paper is that logical expressions can be translated into relatively clear and simple FORTRAN statements, and tested, simplified etc. by insertion into relatively short and easily understandable programs written in FORTRAN. If Polish notation is used the translation process becomes even easier, although for technical reasons not all of the usual letters of Polish notation can be used as names of functions.

What is the point of being able to perform logical calculations by computer? So far at least there is no special demand for complex or large scale logical calculation, although the development of simple and workable programs for such calculations may help to encourage a demand for logical calculations which would be tedious and unprofitable if they had to be done "by hand."

However, as I suggested at the beginning of this paper, the heuristic value of such techniques is considerable. The analogies demonstrated between logical and mathematical calculations, the relation between mechanical methods in logic and the performance of such tasks by actual machines

all offer valuable insights. If computer time can be made available for a logic class, the mechanical performance of necessary calculations can enable problems to be assigned which go beyond the usual trivialities.

But the greatest rewards are reserved for those who actually invent new programs for performing logical tasks. Programming in FORTRAN enables the interested non-specialist to use the resources of modern computer technology without becoming involved in technicalities or details, and the insight into the structure of a problem or method gained by translating it into computer terms is a valuable by-product of such research. And if enough interested logicians turn their attention to the problems of doing logic by computer, the crude beginnings here described may lead to results of great value.

APPENDIX

Introduction

This list was produced by computer. The fact that it appears to be in standard logical notation may be puzzling. Actually in my programs the input must be in the form described, but there is greater freedom with regard to output By using the letter "V" for "or," a period for "and," an equal sign for "if and only if" and a minus sign for "not" the output can be made to look approximately like standard logical notation. There is, of course, no horseshoe for "if then," and also the equal sign rather than a triple bar must be used for "if and only if" and the straight minus sign rather than the "squiggle" or tilde for "not."

The leftmost number is the Characteristic Number described in the paper. The number to the right of this is the Characteristic Sum. The number of lines of an eight-line truth table for which the statement is true. This list can be recorded in the computer and in the simplification problem described in the paper, an input of the form **OR** (*P*, **FIF** (**OR** (*P*, **FIF** (**X** (*Q*), *R*)))) (in the appropriate program) will reduce the output:

$$(PVQVR).$$

which is the simplest form of the expression written in machine notation above.

```
 0   0                          (  P  .  -P)
 1   1                 -P  .(  -Q  .  -R)
 2   1                 -P  .(  -Q  .   R)
 3   2                        -P  .  -Q
 4   1                 -P  .(   Q  .  -R)
 5   2                        -P  .  -R
 6   2                 -P  .(   Q  =  -R)
 7   3                 -P  .(  -Q  V  -R)
 8   1                 -P  .(   Q  .   R)
 9   2                 -P  .(   Q  =   R)
10   2                        -P  .   R
11   3                 -P  .(  -Q  V   R)
12   2                        -P  .   Q
13   3                 -P  .(   Q  V  -R)
14   3                 -P  .(   Q  V   R)
15   4                                -P
16   1                  P  .(  -Q  .  -R)
17   2                        -Q  .  -R
18   2                 -Q  .(   P  =  -R)
19   3                 -Q  .(  -P  V  -R)
```

```
20  2                                                  -R   .(   P   =  -Q)
21  3                                                  -R   .(  -P   V  -Q)
22  3          (  -P   .(   Q   =  -R))   V(  -R   .(  -Q   .   P))
23  4          (  -P   .(  -Q   V  -R))   V(  -R   .(  -Q   .   P))
24  2                        (   P   =  -Q)   .(   P   =  -R)
25  3                        (  -P   V  -Q)   .(   Q   =   R)
26  3                        (  -P   V  -Q)   .(   P   =  -R)
27  4                        (  -P   .   R)   V(  -Q   .  -R)
28  3                        (  -P   V  -R)   .(   P   =  -Q)
29  4                        (  -P   .   Q)   V(  -Q   .  -R)
30  4                                    P   =(  -Q   .   -R)
31  5                                   -P   V(  -Q   .  -R)
32  1                                    P   .(  -Q   .   R)
33  2                                   -Q   .(   P   =   R)
34  2                                                 -Q   .   R
35  3                                   -Q   .(  -P   V   R)
36  2                        (   P   =  -Q)   .(   P   =   R)
37  3                        (  -P   V  -Q)   .(   P   =   R)
38  3                        (  -P   V  -Q)   .(   Q   =  -R)
39  4                        (  -P   .  -R)   V(  -Q   .   R)
40  2                                    R   .(   P   =  -Q)
41  3          (  -P   .(   Q   =   R))   V(   R   .(  -Q   .   P))
42  3                                    R   .(  -P   V  -Q)
43  4          (  -P   .(  -Q   V   R))   V(   R   .(  -Q   .   P))
44  3                        (  -P   V   R)   .(   P   =  -Q)
45  4                                    P   =(  -Q   .   R)
46  4                        (  -P   .   Q)   V(  -Q   .   R)
47  5                                   -P   V(  -Q   .   R)
48  2                                                P   .  -Q
49  3                                   -Q   .(   P   V  -R)
50  3                                   -Q   .(   P   V   R)
51  4                                                     -Q
52  3                        (   P   V  -R)   .(   P   =  -Q)
53  4                        (   P   .  -Q)   V(  -P   .  -R)
54  4                                    Q   =(  -P   .  -R)
55  5                                   -Q   V(  -P   .  -R)
56  3                        (   P   V   R)   .(   P   =  -Q)
57  4                                    Q   =(  -P   .   R)
58  4                        (   P   .  -Q)   V(  -P   .   R)
59  5                                   -Q   V(  -P   .   R)
60  4                                             P   =  -Q
61  5                        (  -P   .  -R)   V(   P   =  -Q)
62  5                        (  -P   .   R)   V(   P   =  -Q)
63  6                                             -P   V  -Q
64  1                                    P   .(   Q   .  -R)
65  2                                   -R   .(   P   =   Q)
66  2                        (   P   =   Q)   .(   P   =  -R)
67  3                        (  -P   V  -R)   .(   P   =   Q)
68  2                                             Q   .  -R
69  3                                   -R   .(  -P   V   Q)
70  3                        (  -P   V   Q)   .(   Q   =  -R)
71  4                        (  -P   .  -Q)   V(   Q   .  -R)
72  2                                    Q   .(   P   =  -R)
73  3          (  -P   .(   Q   =   R))   V(  -R   .(   Q   .   P))
74  3                        (  -P   V   Q)   .(   P   =  -R)
75  4                                    P   =(   Q   .  -R)
76  3                                    Q   .(  -P   V  -R)
77  4          (  -P   .(   Q   V  -R))   V(  -R   .(   Q   .   P))
78  4                        (  -P   .   R)   V(   Q   .  -R)
79  5                                   -P   V(   Q   .  -R)
80  2                                             P   .  -R
81  3                                   -R   .(   P   V  -Q)
82  3                        (   P   V  -Q)   .(   P   =  -R)
83  4                        (   P   .  -R)   V(  -P   .  -Q)
84  3                                   -R   .(   P   V   Q)
85  4                                                     -R
86  4                                    R   =(  -P   .  -Q)
87  5                                   -R   V(  -P   .  -Q)
88  3                        (   P   V   Q)   .(   P   =  -R)
```

```
 89  4                                                R   =(  -P   .   Q)
 90  4                                                    P   =  -R
 91  5                          (  -P   .  -Q)   V(   P   =  -R)
 92  4                          (   P   .  -R)   V(  -P   .   Q)
 93  5                                          -R   V(  -P   .   Q)
 94  5                          (  -P   .   Q)   V(   P   =  -R)
 95  6                                              -P   V  -R
 96  2                                                P   .(   Q   =  -R)
 97  3          (   P   .(   Q   =  -R))   V(  -R   .(  -Q   .  -P))
 98  3                          (   P   V  -Q)   .(   Q   =  -R)
 99  4                                            Q   =(   P   .  -R)
100  3                          (   P   V   Q)   .(   Q   =  -R)
101  4                                            R   =(   P   .  -Q)
102  4                                                Q   =  -R
103  5                          (  -P   .  -Q)   V(   Q   =  -R)
104  3          (   P   .(   Q   =  -R))   V(   R   .(   Q   .  -P))
105  4                                            P   =(   Q   =  -R)
106  4                                            R   =(  -P   V  -Q)
107  5          (  -P   V(   Q   =  -R))   .(   R   V(  -Q   V   P))
108  4                                            Q   =(  -P   V  -R)
109  5          (  -P   V(   Q   =  -R))   .(  -R   V(   Q   V   P))
110  5                          (  -P   .   Q)   V(   Q   =  -R)
111  6                                          -P   V(   Q   =  -R)
112  3                                            P   .(  -Q   V  -R)
113  4          (   P   .(  -Q   V  -R))   V(  -R   .(  -Q   .  -P))
114  4                          (   P   .  -R)   V(  -Q   .   R)
115  5                                          -Q   V(   P   .  -R)
116  4                          (   P   .  -Q)   V(   Q   .  -R)
117  5                                          -R   V(   P   .  -Q)
118  5                          (   P   .  -Q)   V(   Q   =  -R)
119  6                                              -Q   V  -R
120  4                                            P   =(  -Q   V  -R)
121  5          (   P   V(   Q   =   R))   .(  -R   V(  -Q   V  -P))
122  5                          (   P   .  -Q)   V(   P   =  -R)
123  6                                          -Q   V(   P   =  -R)
124  5                          (   P   .  -R)   V(   P   =  -Q)
125  6                                          -R   V(   P   =  -Q)
126  6                          (   P   =  -Q)   V(   P   =  -R)
127  7                                          -P   V(  -Q   V  -R)
128  1                                            P   .(   Q   .   R)
129  2                          (   P   =   Q)   .(   P   =   R)
130  2                                            R   .(   P   =   Q)
131  3                          (  -P   V   R)   .(   P   =   Q)
132  2                                            Q   .(   P   =   R)
133  3                          (.  -P   V   Q)   .(   P   =   R)
134  3          (  -P   .(   Q   =  -R))   V(   R   .(   Q   .   P))
135  4                                            P   =(   Q   .   R)
136  2                                                Q   .   R
137  3                          (  -P   V   Q)   .(   Q   =   R)
138  3                                            R   .(  -P   V   Q)
139  4                          (  -P   .  -Q)   V(   Q   .   R)
140  3                                            Q   .(  -P   V   R)
141  4                          (  -P   .  -R)   V(   Q   .   R)
142  4          (  -P   .(   Q   V   R))   V(   R   .(   Q   .   P))
143  5                                          -P   V(   Q   .   R)
144  2                                            P   .(   Q   =   R)
145  3                          (   P   V  -Q)   .(   Q   =   R)
146  3          (   P   .(   Q   =   R))   V(   R   .(  -Q   .  -P))
147  4                                            Q   =(   P   .   R)
148  3          (   P   .(   Q   =   R))   V(  -R   .(   Q   .  -P))
149  4                                            R   =(   P   .   Q)
150  4                                            P   =(   Q   =   R)
151  5          (  -P   V(   Q   =   R))   .(  -R   V(  -Q   V   P))
152  3                          (   P   V   Q)   .(   Q   =   R)
153  4                                                Q   =   R
154  4                                            R   =(  -P   V   Q)
155  5                          (  -P   .  -Q)   V(   Q   =   R)
156  4                                            Q   =(  -P   V   R)
157  5                          (  -P   .   Q)   V(   Q   =   R)
```

```
158  5          (  -P   V(   Q   =   R))   .(   R   V(   Q   V   P))
159  6                                    -P   V(   Q   =   R)
160  2                                         P   .   R
161  3                        (   P   V  -Q)   .(   P   =   R)
162  3                                    R   .(   P   V  -Q)
163  4                        (   P   .   R)   V(  -P   .  -Q)
164  3                        (   P   V   Q)   .(   P   =   R)
165  4                                         P   =   R
166  4                                    R   =(   P   V  -Q)
167  5                        (  -P   .  -Q)   V(   P   =   R)
168  3                                    R   .(   P   V   Q)
169  4                                    R   =(   P   V   Q)
170  4                                             R
171  5                                    R   V(  -P   .  -Q)
172  4                        (   P   .   R)   V(  -P   .   Q)
173  5                        (  -P   .   Q)   V(   P   =   R)
174  5                                    R   V(  -P   .   Q)
175  6                                        -P   V   R
176  3                                    P   .(  -Q   V   R)
177  4                        (   P   .   R)   V(  -Q   .  -R)
178  4          (   P   .(  -Q   V   R))   V(   R   .(  -Q   .  -P))
179  5                                   -Q   V(   P   .   R)
180  4                                    P   =(  -Q   V   R)
181  5                        (   P   .  -Q)   V(   P   =   R)
182  5          (   P   V(   Q   =  -R))   .(   R   V(  -Q   V  -P))
183  6                                   -Q   V(   P   =   R)
184  4                        (   P   .  -Q)   V(   Q   .   R)
185  5                        (   P   .  -Q)   V(   Q   =   R)
186  5                                    R   V(   P   .  -Q)
187  6                                        -Q   V   R
188  5                        (   P   .   R)   V(   P   =  -Q)
189  6                        (   P   =  -Q)   V(   P   =   R)
190  6                                    R   V(   P   =  -Q)
191  7                                   -P   V(  -Q   V   R)
192  2                                         P   .   Q
193  3                        (   P   V  -R)   .(   P   =   Q)
194  3                        (   P   V   R)   .(   P   =   Q)
195  4                                         P   =   Q
196  3                                    Q   .(   P   V  -R)
197  4                        (   P   .   Q)   V(  -P   .  -R)
198  4                                    Q   =(   P   V  -R)
199  5                        (  -P   .  -R)   V(   P   =   Q)
200  3                                    Q   .(   P   V   R)
201  4                                    Q   =(   P   V   R)
202  4                        (   P   .   Q)   V(  -P   .   R)
203  5                        (  -P   .   R)   V(   P   =   Q)
204  4                                                  Q
205  5                                    Q   V(  -P   .  -R)
206  5                                    Q   V(  -P   .   R)
207  6                                        -P   V   Q
208  3                                    P   .(   Q   V  -R)
209  4                        (   P   .   Q)   V(  -Q   .  -R)
210  4                                    P   =(   Q   V  -R)
211  5                        (   P   .  -R)   V(   P   =   Q)
212  4          (   P   .(   Q   V  -R))   V(  -R   .(   Q   .  -P))
213  5                                   -R   V(   P   .   Q)
214  5          (   P   V(   Q   =  -R))   .(  -R   V(   Q   V  -P))
215  6                                   -R   V(   P   =   Q)
216  4                        (   P   .  -R)   V(   Q   .   R)
217  5                        (   P   .   Q)   V(   Q   =   R)
218  5                        (   P   .   Q)   V(   P   =  -R)
219  6                        (   P   =   Q)   V(   P   =  -R)
220  5                                    Q   V(   P   .  -R)
221  6                                             Q   V  -R
222  6                                    Q   V(   P   =  -R)
223  7                                   -P   V(   Q   V  -R)
224  3                                    P   .(   Q   V   R)
225  4                                    P   =(   Q   V   R)
226  4                        (   P   .   Q)   V(  -Q   .   R)
```

```
227  5                                  (  P  •  R)  V(  P  =  Q)
228  4                                  (  P  •  R)  V(  Q  •  -R)
229  5                                  (  P  •  Q)  V(  P  =  R)
230  5                                  (  P  •  Q)  V(  Q  =  -R)
231  6                                  (  P  =  Q)  V(  P  =  R)
232  4        (  P  •(  Q  V  R))  V(  R  •(  Q  •  -P))
233  5        (  P  V(  Q  =  R))  •(  R  V(  Q  V  -P))
234  5                                  R  V(  P  •  Q)
235  6                                  R  V(  P  =  Q)
236  5                                  Q  V(  P  •  R)
237  6                                  Q  V(  P  =  R)
238  6                                        Q  V  R
239  7                                  -P  V(  Q  V  R)
240  4                                              P
241  5                                  P  V(  -Q  •  -R)
242  5                                  P  V(  -Q  •  R)
243  6                                        P  V  -Q
244  5                                  P  V(  Q  •  -R)
245  6                                        P  V  -R
246  6                                  P  V(  Q  =  -R)
247  7                                  P  V(  -Q  V  -R)
248  5                                  P  V(  Q  •  R)
249  6                                  P  V(  Q  =  R)
250  6                                        P  V  R
251  7                                  P  V(  -Q  V  R)
252  6                                        P  V  Q
253  7                                  P  V(  Q  V  -R)
254  7                                  P  V(  Q  V  R)
255  8                                        P  V  -P
```

Western Washington State College
Bellingham, Washington