LOGIC: A COMPUTER APPROACH

Morton L. Schagrin
State University of New York at Fredonia

William J. Rapaport
State University of New York at Buffalo

Randall R. Dipert
State University of New York at Fredonia

McGraw-Hill Book Company
New York St. Louis San Francisco Auckland Bogotá Hamburg Johannesburg London Madrid Mexico Montreal New Delhi Panama Paris São Paulo Singapore Sydney Tokyo Toronto

© 1985

CHAPTER 11

SENTENTIAL LOGIC: A Method for Producing Proofs

We now take up the second of our two questions from Chapter 10: how to *construct* a proof of a valid argument given the premises and the conclusion. We could give an algorithm for constructing a proof, in the logic of sentences, for any premises and any conclusion validly following from them. But the resulting algorithm would be either very long or "unnatural" (in failing to follow the "natural" inferential rules we gave in Chapters 8 and 9). Consequently, we shall not aim in this chapter to give the full algorithm for constructing a derivation. We shall instead content ourselves with describing a method for constructing a proof that will work *most* of the time. That is, in some cases, some creative intervention by a human user might be required. We shall call this method PROOF-GIVER.

CHAPTER 11

General Strategies for Constructing Proofs

The task we face in constructing a derivation of a conclusion from given premises seems, at first, like nothing we have ever done before. Consequently, the beginner might feel somewhat lost when it comes to constructing a proof. But, in fact, the construction of a proof in logic is a great deal like problems we regularly solve without much difficulty. The construction of a proof also resembles tasks for which computers (and hence algorithms) are regularly employed.

Analogous Problems

One task analogous to constructing proofs is the problem of finding our way through a maze.

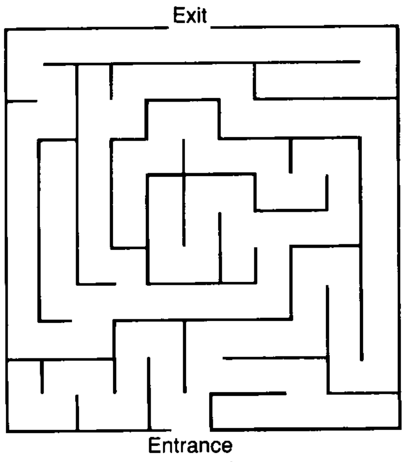

Figure 11-1 A maze.

We all quickly recognize what the goal is: to find a path through the maze, beginning at the entrance and coming out at the exit, without crossing a line (a "wall"). Finding our way through a maze might not seem at first much like constructing a proof. But the two problems have some interesting similarities. In a maze, we are always told where to begin (at the "entrance"). In constructing a proof in logic, we are also told where to begin: with our *premises.* The premises are our "entrance" to a logic puzzle. In a maze, we are also told where we are supposed to end up (at the "exit"). In constructing a proof in logic, our goal also lies clearly before us: the conclusion, which we must somehow reach.

In finding our way through a maze, certain methods of proceeding are allowed, and others are forbidden to us. We may turn left or right or go straight ahead, but we are not allowed to "jump over" or "go through" a wall. In a proof in logic, there are certain maneuvers that are allowed: these are the rules of our natural deduction system: &ELIM, →INTRO, and so on. But certain maneuvers are denied to us: we are not allowed to infer any old sentence we would like.

As it will turn out, the *strategies* we use in finding a way through a maze and in constructing a proof are quite similar. In finding our way through a maze on paper, we might visually start at the entrance and see where we can go from there. Or we might glance ahead to our goal, the exit, and see how we might get there. That is, we might glance ahead to see what routes *lead to* the exit. In constructing a proof in logic, there are also two basic strategies. We might look at the premises and see what follows from them by the rules that are permitted to us. Or we might look ahead and see how the conclusion might come about through using the known rules.

Another analogous task is a more practical problem that a computer—in the hands of an able travel agent—is frequently used to solve. Suppose that you must fly from Chicago to New Orleans but that, unfortunately, there are no direct flights at the time, you want to make your trip. Suppose that the relevant flights have the following pattern:

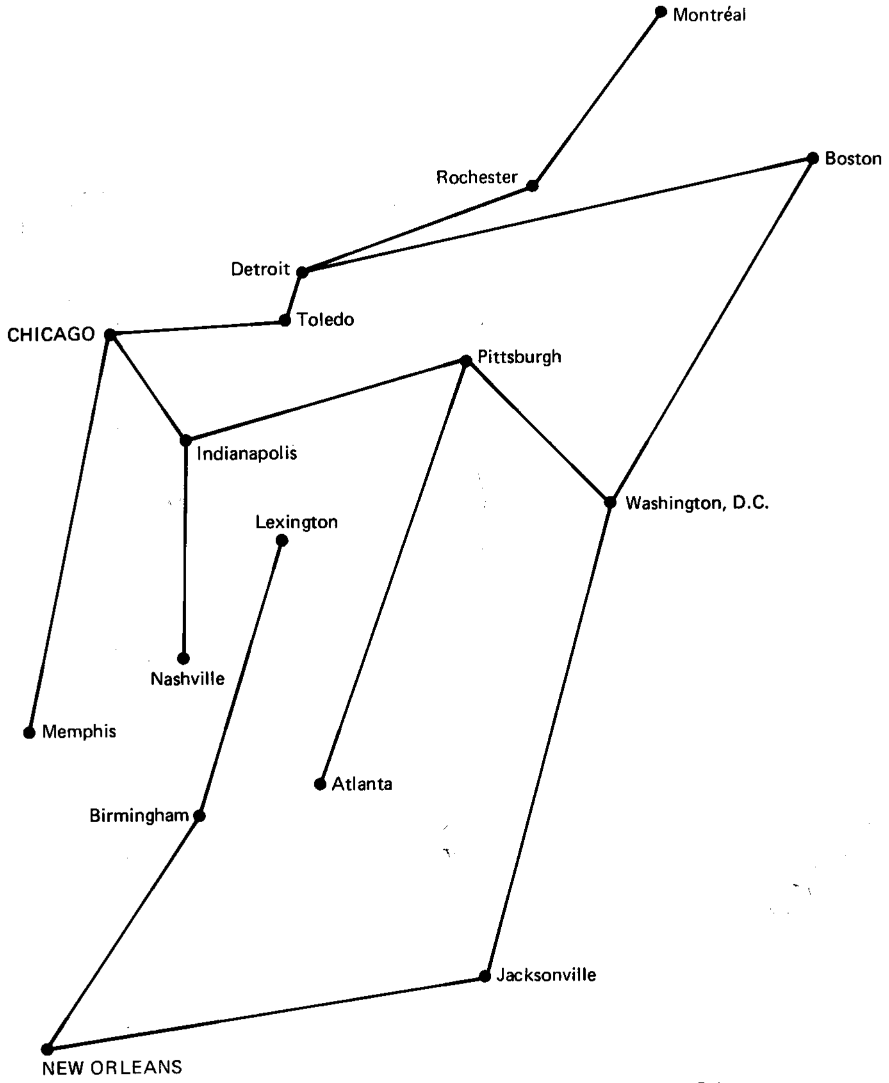

Figure 11-2 Flights with connections to Chicago and New Orleans.

Just as with the maze and with constructing a proof in logic, there are "dead ends" that will not lead to our destination, and there are sometimes *several* ways to reach our destination—some that are shorter and some that are longer. The strategies we might use to solve this flight problem are like the strategies we might use to find our way through a maze. For example, we might use a finger to trace a path beginning with Chicago, or we might use a finger to trace a path backward from New Orleans. We might even combine the two strategies to see where they meet.

The flight problem has some features that make it more like a logic problem than the maze. In order to fly from Chicago to New Orleans, we might first have to fly to other cities—Atlanta, for example. In the jargon of a travel agent, these are our "connections." We might even have to go considerably out of our way to get to New Orleans: we might have to fly to Seattle, for example, if there are no direct flights to New Orleans.

In constructing a proof in logic, there might also be no "direct flights." We might first have to make "trips" to intermediate "destinations." A "direct flight" in the proof of an argument would be one in which the conclusion follows from the premises through the application of only one rule. For example, the argument

$(A \rightarrow B)$
A
$\therefore B$

has a proof that corresponds to a direct flight:

1.	$(A \rightarrow B)$	:PREMISE
2.	A	:PREMISE
3.	B	:$\rightarrow$ELIM,1,2

In a proof, however, we do not often have the luxury of a "direct flight"; we must make "connections" by deducing intermediate sentences that eventually allow us to reach our destination: the conclusion.

Forward-Looking and Backward-Looking Strategies

The strategy used in constructing a proof will be so remarkably similar to the strategies we would ordinarily use to solve the maze and the flight problems that it bears repeating.

> We can begin with our starting point and work forward, or we can glance ahead to our destination and work backward.

In constructing a proof, these two methods would correspond, respectively, to

1. Considering the premises and wondering what follows from them by our rules.
2. Considering the conclusion and wondering how it could arise by our rules.

For several reasons, method 2—looking ahead to the destination (our conclusion) and working backward—will turn out to be especially fruitful in logic.

There is one big difference between the logical problem of constructing a proof and the maze and flight problems. This difference will make approaching the construction of a proof with a strategy all the more important. The difference is that the *ways* we get from our point of departure to our destination in the maze and flight problems are few in number. In the maze problem, we move in any direction we wish, so long as we don't cross a line; in the flight problem, we move along connecting lines. But in the logical problem of "departing from" the premises and "arriving at" the conclusion, there are at any point *many* different rules we could use. Still worse, there are in the construction of a proof many more "intermediate destinations" than in the case of the flight from Chicago to New Orleans. There are in fact an infinite number (so we could not even chart the options, as we did in the Chicago–New Orleans case). And most of these intermediate destinations are blind alleys: they do not get us any closer to our destination—the conclusion. We must then plan our proof very carefully. And to plan our proof, we must have a *method* for creating the plan—a plan for making plans, if you wish. This method for creating plans for a proof is the purpose of PROOF-GIVER.

Consider the following argument:

$$\begin{array}{ll} & A \\ & (A \rightarrow (B \rightarrow C)) \\ \therefore & (B \rightarrow C) \end{array}$$

The argument is valid. But how do we produce a proof? We first begin by examining the premises, the conclusion, and their connection. We see that the conclusion is a conditional, '(B → C)'. Furthermore, we should observe that this very same conditional is part of one of the premises: (A → (B → C)), the second premise. Reflecting briefly on this premise and its main connective, →, we see that if we could somehow *eliminate* this connective (and the antecedent 'A') from the premise, we would be left with the desired result: (B → C). But, of course, to eliminate →, we just apply the →ELIM rule. To apply it, we need the antecedent 'A', which, conveniently, is the first premise.

The proof resulting from these observations is:

```
1. A                  :PREMISE
2. (A → (B → C))      :PREMISE
3. (B → C)            :→ELIM,2,1
```

This proof resulted from our observation that the desired conclusion is a *subformula* of one of the premises. We then reasoned how this subformula could be derived by itself on a line. Obtaining a subformula by itself will typically involve the use of an ELIM rule.

But consider the following argument:

$$\begin{array}{ll} & B \\ \therefore & ((A \rightarrow C) \rightarrow B) \end{array}$$

Here, the conclusion is not a subformula of a premise. The lengthy conclusion is not embedded anywhere in the simple premise. What information *do* we have to go on in determining our strategy for the proof? Even though we cannot see the relationship of the conclusion to the premises (as we did in the previous example), we can examine the structure of the conclusion itself. It is, again, a conditional. How could a conditional arise in a proof? It might well come from an application of →INTRO. (In fact, it is difficult in this example to see how a conditional could be arrived at other than through → INTRO.)

So we might guess that one step in the proof uses →INTRO. But what previous lines could produce '((A → C) → B)' by →INTRO? The answer is this. If '(A → C)' were an ASSUMPTION and 'B' were a later line in the same subproof, then '((A → C) → B)' could be inferred by →INTRO. A good guess, then, is that the proof proceeds as follows:

```
 1. B                  :PREMISE
*2.  (A → C)           :ASSUMPTION
      .                 .
      .                 .
      .                 .
*?.  B                 :?
*?.  ((A → C) → B)     :→INTRO,?,?
 ?. ((A → C) → B)      :RETURN,?
```

The dots and question marks indicate parts of the proof yet to be filled in. The only mystery is how to derive 'B'. That, however, is easy to guess in this example: it was "sent" into the subproof from the premise, 'B'. The resulting proof, with comment lines inserted, is:

```
 1. B                      :PREMISE
     /BEGIN: →INTRO to derive ((A → C) → B)/
*2.  (A → C)               :ASSUMPTION
*3.  B                     :SEND,1
*4.  ((A → C) → B)         :→INTRO,2,3
     /END:   →INTRO to derive ((A → C) → B)/
 5. ((A → C) → B)          :RETURN,4
```

These two examples give us the basis of a *strategy* for creating proofs. An overall view of the method we have just seen applied in devising a strategy is:

1. If the conclusion is a subformula in a previous line, then use the appropriate ELIM rule to isolate this subformula on its own line.

2. If the conclusion is not a subformula of a previous line, then examine the structure of this conclusion, and use the appropriate INTRO rule to reconstruct the conclusion.

These two suggestions are the basic elements in PROOF-GIVER.

In building up a derivation, we shall need two items. The first is simply the ongoing derivation. It will be composed of the steps of the proof, as far as we have gotten. The second item is what we shall call the *task list.* The task list will be our list of the steps yet needed to complete the proof. The task list is best thought of as our notes to ourselves on how to complete the proof. In this respect, the task list is little more than what we have been calling "comments."

This task list can be written immediately following any portion ot the proof we have completed (or it can be kept in a separate place) and constitutes instructions on our "plan of attack" for how best to complete the proof. When we are first given the premises and the conclusion that validly follows from them, the first instruction is to *derive* that conclusion. We would write:

```
PROOF
  1. A                  :PREMISE
  2. (A → (B → C))      :PREMISE

TASK LIST
  Derive (B → C)
```

The last line, 'Derive (B → C)', is our first task and so is the sole item in our task list at the beginning of our attempt to create a proof.

The rest of the method consists of replacing 'Derive <the conclusion>' with more helpful advice; accordingly, the task list grows. Furthermore, as some of these tasks become precise enough to be turned into lines of a proof, the proof itself also grows.

Tree-Searching Strategies

One of the main areas of research in artificial intelligence concerns the topic of a "search": how to program computers to find solutions or reach goals by guided, or "intelligent," methods (including trial and error), rather than by "exhaustively" searching all possibilities. The possession of such methods seems to be essential for anything claiming to employ "intelligence."

We have already drawn some parallels between the problem of constructing a proof and other problems that require a search. Let us now display with more precision what our search looks like when we are attempting to construct a proof.

Argument A
(A→B)
∴ B

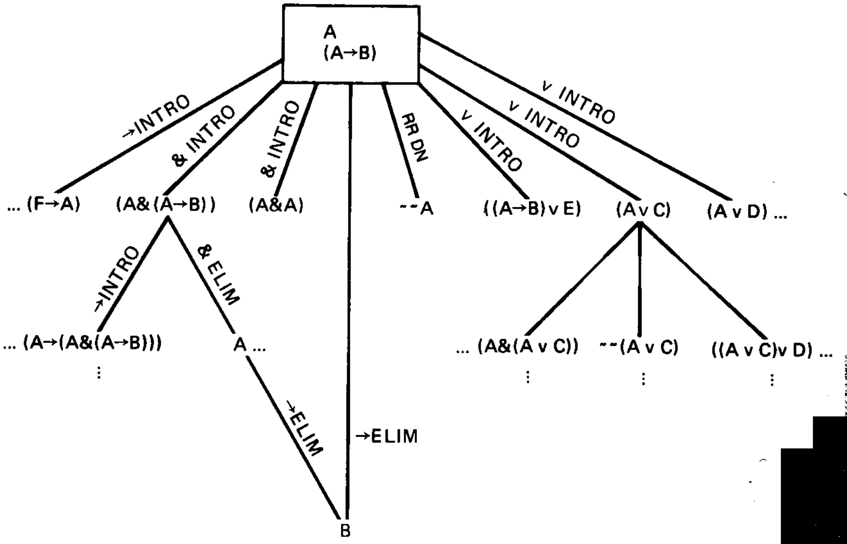

Figure 11-3 Search tree for the argument A, (A → B), ∴ B.

Such diagrams are called "trees." Our starting point is the set of our premises. Our goal is the conclusion. After correctly applying a rule, we call the point that we have reached a *node* of the tree. At this node, we write the new sentence that the rule allowed us to derive. We also have available at this node any of the previous sentences—the sentences we have derived *above* this node. We can apply any of the rules to these available sentences. (Another way of thinking about the proof-search tree is that at each node there is a *set* of "available" sentences. This set "grows" each time a rule is applied, and we can stop when the conclusion is included in the new set.) Such a display is called a *search tree,* and what it displays is called the *search space* of the problem.

Acceptable means of "traveling" from the premises to the conclusion are restricted by the available rules. Let us now introduce some terminology. We can measure the *distance* of a sentence from the premises (or the conclusion) by the number of rules used on the shortest path between the premises and the sentence. For example, all the sentences on the first row below the premises have a distance of one unit from the premises.

We can now make some observations about the above display of a rather typical proof problem.

1. The number of sentences in the tree whose distance is one unit from the premises is *infinite*—no matter what the premises are. The rules vINTRO and →INTRO can be used to add indefinitely many new nodes.

2. If there is one path through the tree from the premises to the conclusion, then there are an infinite number of such paths.
3. If there is a path of length *n* from the premises to the conclusion, then there is also a path of still greater length.
4. There are an infinite number of sentences whose distances are one unit from the conclusion.
5. The tree branches endlessly into directions that are not especially close to the premises or conclusion.

The job of PROOF-GIVER is to find a reasonably short path from the premises to the conclusion. Two strategies for searching a tree for a particular node are the "breadth-first" search and the "depth-first" search, illustrated in Figure 11-4.

A *breadth-first search* considers, first, all the nodes that are one unit distant from the starting node. Then it considers all the nodes that are one unit distant from *those* nodes, and so on until the goal is reached. It is a search which seeks its goal by first checking out all the nearest nodes from the starting node, then the next-nearest nodes, and so on.

Consider this analogy. In conducting a breadth-first search for a lost dollar, I might begin by visiting *all* the places where I might have lost it and only then turn to more "distant" options—for example, that I dropped it on the sidewalk, but it then blew away.

A breadth-first search of our proof tree would not be a reasonable way of discovering a proof. The number of sentences one unit distant from our premises is infinite (as we observed above). So we would first have to consider *all* of these sentences before going further with the search. But this first step would itself take forever. In short, a breadth-first search is not reasonable when our search tree has "unlimited branching," which rules such as vINTRO and →INTRO permit.

Although an "all-out" breadth-first search would not be feasible in our case, we might consider a limited breadth-first search. For example, we might ignore certain of the branches. (Carrying on the "tree" metaphor, researchers in artificial intelligence speak of this technique as "pruning" the tree.) The branches we might ignore would be:

1. Branches that use vINTRO or →INTRO and result in a sentence containing an atomic sentence that is contained in neither the premises nor the conclusion
2. Branches that use &INTRO and result in a sentence that is not contained as a subformula in any of the premises or in the conclusion
3. Branches that use &ELIM and result in a sentence that is neither a subformula of any premise or conclusion nor are the premises or conclusion a subformula of the sentence.

The result of this limited breadth-first search would be a much-pruned search tree that would usually reach the conclusion. But the tree would still often be quite large.

The second major search strategy is a *depth-first search.* In depth-first search, we make a single, deep plunge into the search tree, hoping to catch our quarry, the goal, in one quick thrust. This strategy contrasts with the many, equally shallow plunges (or "guesses") of breadth-first search. The depth-first strategy might also be described as pursuing our "best guesses" as far as we can take them.

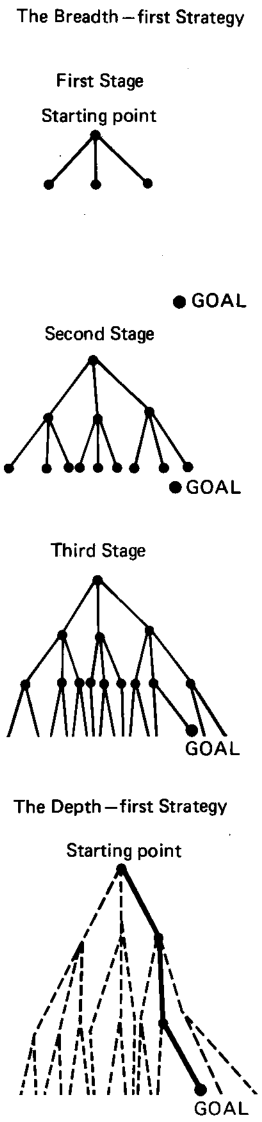

Figure 11-4 Breadth-first and depth-first search strategies.

The Method: PROOF-GIVER

Several concepts will be useful in our presentation of PROOF-GIVER. First, recall the notion of subproof depth from Chapter 10. A subproof of any depth may be said to *contain* itself. A subproof S of depth *n* may be said to *contain* a subproof S′ of a depth greater than *n* iff there is no line with fewer than *n* stars between the last line of S and the first line of S′. In particular, the main proof (which is a subproof of depth 0) contains itself and all subproofs of any depth.

Intuitively, proofs can be pictured as "nested boxes," as in Figure 11-5.

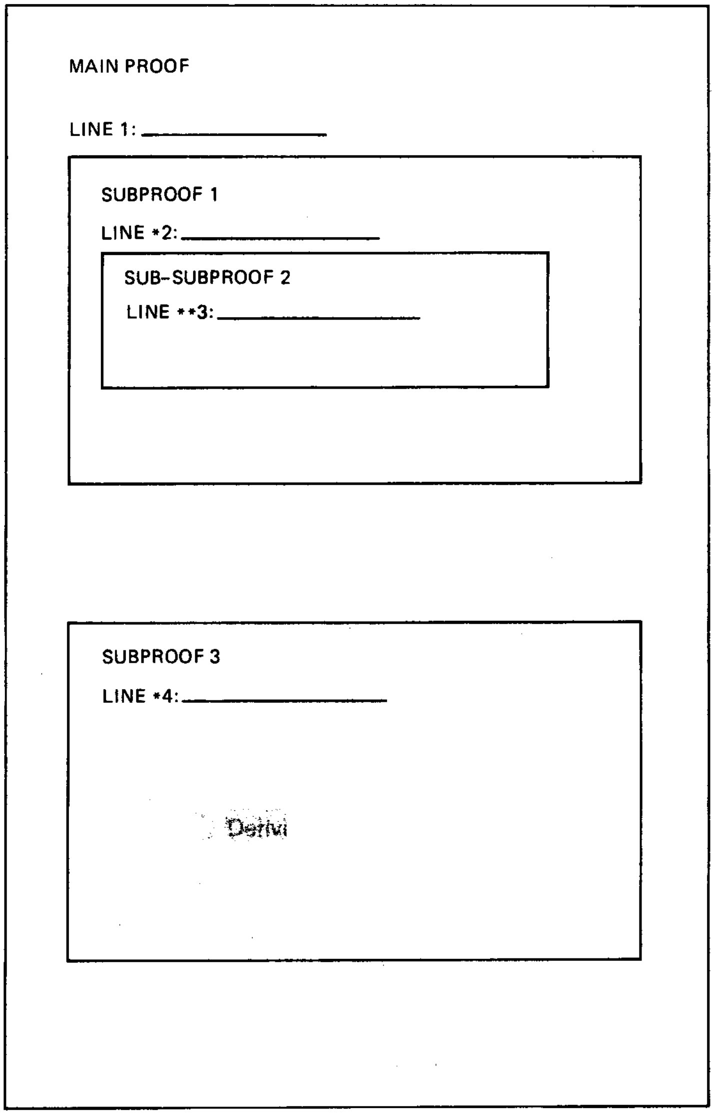

Figure 11-5 Diagram of a proof.

In Figure 11-5, the main proof contains itself and all three subproofs. The first subproof contains only itself and subproof 2. The second and third subproofs contain only themselves.

We can now say that a line L is *accessible* to a later line L′ iff the subproof that L is in contains the subproof that L′ is in. For instance, in Figure 11-5, line 1 is accessible to lines in all subproofs, line 2 is only accessible to lines in subproofs 1 and 2, line 3 is only accessible to lines in subproof 2, and line 4 is only accessible to lines in subproof 3. We shall sometimes simply say that "the line is accessible." (In the terminology of computer science, a sentence in a particular subproof is said to be "local" to that subproof and "global" to all subproofs contained within that subproof. Thus line L is accessible to line L′ iff the sentence on line L is global to the subproof that L′ is in or L and L′ are in the same subproof, in which case the sentence is local.)

Second, an asterisk before a task will indicate that a subproof must be begun. Third, a sentence in brackets, such as [**P**], will indicate that the first accessible line number containing sentence **P** should be cited. Fourth, we shall always call the sentence in the first 'Derive' statement in the task list the "desired result," sometimes representing it as '**DR**'.

Finally, suppose that a sentence, **Q**, is on an accessible line L and that we wish to use **Q** in our derivation. If the line we wish to use it on is in the same subproof as L, then we need do nothing special. But if that line is in a sub-subproof, we need to SEND **Q** into that sub-subproof. In addition, there will be some "bookkeeping" to take care of. To make our presentation of PROOF-GIVER simpler, we shall use a procedure called 'OBTAIN **Q**':

```
PROCEDURE OBTAIN Q
    1. IF the line containing Q has fewer asterisks than the desired result (that is,
       Q is in a containing proof)
         THEN

         (a) Replace the first 'Derive' statement in the task list with: SEND,[Q].
         (b) Let all subsequent references to line [DR] in the current subproof be
             replaced by the line number of this SEND line. (This is the "book-
             keeping.")

    2. IF the line containing Q has the same number of asterisks as the desired
       result (that is, Q is in the same subproof)
         THEN

         (a) Delete the first 'Derive' statement from the task list
         (b) Let all subsequent references to line [DR] in the current subproof be
             replaced with the line number of this previous line (more "bookkeep-
             ing").
```

We now have enough background to present the method.

```
METHOD PROOF-GIVER:
    1. INPUT the premises, with the justification 'PREMISE', in the standard format
       for proofs.
    2. Add 'Derive ⟨conclusion⟩' to the task list.
```

```
3. WHILE the task list is not empty and there is a 'Derive' statement in the task
   list, keep repeating (a) to (c):

   (a) IF **DR** = a subformula of a sentence, **Q,** on an accessible line
         THEN

         (i) OBTAIN **Q.**
         (ii) IF **DR** = **Q**
                THEN GO TO step 3(c).
         (iii) IF **DR** is a proper subformula of **Q**
                THEN

                (1) IF **Q** is a conditional
                    and the desired result is a subformula of **Q**'s consequent
                      THEN replace the first 'Derive' statement in the task list
                        with:
                        Derive ⟨antecedent of **Q**⟩
                        Apply $\rightarrow$ELIM,[**Q**],[⟨antecedent of **Q**⟩]
                        Derive **DR** (if **DR** $\neq$ ⟨consequent of **Q**⟩)
                    and GO TO step 3(c).

                (2) IF **Q** is a conjunction
                    and the desired result is a subformula of one of its conjuncts
                      THEN replace the first 'Derive' statement in the task list
                        with:
                        Apply &ELIM,[**Q**] (to obtain conjunct containing **DR**)
                        Derive **DR** (if **DR** $\neq$ the inferred conjunct)
                    and GO TO step 3(c).

                (3) IF **Q** is a disjunction, (**P** v **R**),
                    and the desired result is a subformula of one of its disjuncts,
                    say **R**
                      THEN replace the first 'Derive' statement in the task list
                        with:
                        Derive ~**P**
                        Apply vELIM, [**Q**],[~**P**]
                        Derive **DR** (if **DR** $\neq$ **R**)
                    and GO TO step 3(c).

                (4) IF **Q** is a negation, say ~**P**, and **DR** is a subformula of **P**, say
                    **R**
                      THEN replace the first 'Derive' statement in the task list
                        with:
                        *Assume ~**R**
                        *OBTAIN **Q**
                        *Derive **P**
                        *Apply ~ELIM,[~**R**],[**P**],[**Q**]
                        RETURN,[**R**]
                    and GO TO step 3(c).
```

```
(b) IF DR is not a subformula of a sentence on an accessible line meeting
    the above conditions
     THEN

     (i)  IF the desired result has the form $(P \rightarrow Q)$
             THEN replace the first 'Derive $(P \rightarrow Q)$' in the task list with:
               *Assume $P$
               *Derive $Q$
               *Apply $\rightarrow$INTRO,[$P$],[$Q$]
               Apply RETURN,[$P \rightarrow Q$]
             and GO TO step 3(c).

     (ii) IF the desired result has the form $(P \& Q)$
             THEN replace the first 'Derive $(P \& Q)$' in the task list with:
               Derive $P$
               Derive $Q$
               Apply &INTRO,[$P$],[$Q$]
             and GO TO step 3(c).

     (iii) IF the desired result has the form $(P \vee Q)$
             THEN replace the first 'Derive' statement in the task list either
               with:
             (1) *Assume $\sim(P \vee Q)$
                 *Derive $R$
                 *Derive $\sim R$
                 *Apply $\sim$ELIM,[$\sim(P \vee Q)$],[$R$],[$\sim R$]
                 RETURN,[$(P \vee Q)$]
          or with:
             (2) Derive $P$
                 Apply vINTRO,[$P$] to obtain $(P \vee Q)$
          or with:
             (3) Derive $Q$
                 Apply vINTRO,[$Q$] to obtain $(P \vee Q)$
          and GO TO step 3(c).

     (iv) IF the desired result has the form $\sim P$
             THEN replace the first 'Derive $\sim P$' in the task list with:
               *Assume $P$
               *Derive $R$
               *Derive $\sim R$
               *Apply $\sim$INTRO,[$P$],[$R$],[$\sim R$]
               RETURN,[$\sim P$]
             and GO TO step 3(c).

     (v)  IF none of the previous steps have been applied
               THEN replace the first 'Derive $P$' in the task list with:
               *Assume $\sim P$
               *Derive $R$
```

```
        *Derive ~**R**
        *Apply ~ELIM,[~**P**], [**R**], [~**R**]
        RETURN,[**P**]
      and GO TO step 3(c).
```

(c) Convert all statements in the task list that precede the first 'Derive' statement into lines of proof, removing them from the task list.

4. STOP.

Applications of PROOF-GIVER

Let us now apply the method to an argument:

A
B
D
∴ ((A & B) & (C → D))

Following steps 1 and 2 of PROOF-GIVER, we obtain:

```
PROOF
  1. A    :PREMISE
  2. B    :PREMISE
  3. D    :PREMISE

TASK LIST
  Derive ((A & B) & (C → D))
```

The desired result is '((A & B) & (C → D))'. At step 3(a) we can see that it does *not* appear in the previous lines (the premises), so we pass on to step 3(b).

Since the desired result is a *conjunction,* step 3(b)(ii) applies, and we replace the original 'Derive ((A & B) & (C → D))' in our task list with:

```
Derive (A & B)
Derive (C → D)
Apply &INTRO,[(A & B)],[(C → D)]
```

obtaining:

```
PROOF
  1. A    :PREMISE
  2. B    :PREMISE
  3. D    :PREMISE
```

```
TASK LIST
  Derive (A & B)
  Derive (C → D)
  Apply &INTRO,[(A & B)],[(C → D)]
```

We now go to step 3(c), which returns us to step 3(a).

At this point, '(A & B)' becomes our "desired result." It is a conjunction, so again step 3(b)(ii) is applied, resulting in:

```
PROOF
  1. A    :PREMISE
  2. B    :PREMISE
  3. D    :PREMISE

TASK LIST
  Derive A
  Derive B
  Apply &INTRO,[A],[B]
  Derive (C → D)
  Apply &INTRO,[(A & B)],[(C → D)]
```

After step 3(c), we again return to step 3(a). Now, however, the desired result is simply 'A', and it *is* contained in a previous line—namely, it is identical to the first premise. So, step 3(a)(i) requires us to *delete* the first 'Derive' statement and replace references to [A] with 1. Looking at the task list only, we see that the result is:

```
Derive B
Apply &INTRO,1,[B]
Derive (C → D)
Apply &INTRO,[(A & B)],[(C → D)]
```

Since there is a 'Derive' statement at the top of the task list, step 3(c) again returns us to step 3(a).

Now the desired result is 'B', which is identical to the second premise. After following the directions in the appropriate clause of step 3(a), we find that our task list looks like this:

```
Apply &INTRO,1,2
Derive (C → D)
Apply &INTRO,[(A & B)],[(C → D)]
```

We again drop down to step 3(c), but this time we do not simply return to step 3(a). It tells us to convert the first line in the task list, 'Apply &INTRO,1,2', into a line of proof. The result is:

```
PROOF
  1. A          :PREMISE
  2. B          :PREMISE
  3. D          :PREMISE
  4. (A & B)    :&INTRO,1,2

TASK LIST
  Derive (C → D)
  Apply &INTRO,4,[(C → D)]
```

Returning to step 3(a), we see that our desired result is now, (C → D)'. It is *not* contained in a previous line, including our newly acquired line 4, so we proceed to step 3(b). The desired result is a *conditional,* so, applying 3(b) (i), we find that our task list becomes:

```
*Assume C
*Derive D
*Apply →INTRO,[C],[D]
Apply RETURN,[(C → D)]
Apply &INTRO,4,[(C → D)]
```

Dropping down to step 3(c), we must convert every statement that occurs in the task files before the first 'Derive' statement into a new line of proof. We obtain:

```
PROOF
  1. A          :PREMISE
  2. B          :PREMISE
  3. D          :PREMISE
  4. (A & B)    :&INTRO,1,2
 *5.  C         :ASSUMPTION

TASK LIST
 *Derive D
 *Apply →INTRO,5,[D]
  Apply RETURN,[(C → D)]
  Apply &INTRO,4,[(C → D)]
```

Returning to step 3(a), we see that 'D' is now the desired result. It is part of a previous sentence, so step 3(a)(i) applies. Following those directions, we have in the task list:

```
*SEND,3
*Apply →INTRO,5,[D]
Apply RETURN,[(C → D)]
Apply &INTRO,4,[(C → D)]
```

Dropping down to step 3(c), we now see that there are *no* 'Derive' statements left in the task list. This means that all instructions in the task list must be converted into lines of proof. Applying the instructions in the task list line by line, we obtain the following result:

```
 1. A                        :PREMISE
 2. B                        :PREMISE
 3. D                        :PREMISE
 4. (A & B)                  :&INTRO,1,2
*5.   C                      :ASSUMPTION
*6.   D                      :SEND,3
*7.   (C → D)                :→INTRO,5,6
 8. (C → D)                  :RETURN,7
 9. ((A & B) & (C → D))      :&INTRO,4,8
```

Limitations of PROOF-GIVER

PROOF-GIVER, as we have described it, falls short of being a true algorithm. The construction of proofs for *some* arguments requires imagination or inspiration. The shortcomings of PROOF-GIVER fall into three major areas.

First, the application of several rules requires the user of the procedure to make additional "guesses." Chief among these are arguments that take us to steps 3(b)(iii), 3(b)(iv), and 3(b)(v). Although many steps in PROOF-GIVER could be straightforwardly translated into algorithms, these steps are not among them. Step 3(b)(iii) requires us to *choose* among three ways of introducing a disjunction. Steps 3(b)(iv) and 3(b)(v) require us to derive a contradiction: some sentence, **R,** on one line and its negation, ~**R,** on another. But *which* sentence and its negation should be derived? This is left to our own invention.

The second and third parts of step 3(b)(iii) apply the rule vINTRO. There is no ambiguity concerning what to derive. But this technique will not always work; where it does work, it works easily. So we find ourselves in a dilemma: the first technique, of indirect proof, always works (and so is listed first) but requires "creativity" to find a contradiction. The second method does not always work, but when it does, it requires no creativity.

The kinds of cases where the two different techniques of 3(b)(iii) are appropriate can be illustrated by two examples.

Consider:

```
   (A & B)
∴ (A v C)
```

Using the first technique, we start with:

```
PROOF
 1. (A & B)     :PREMISE
```

```
TASK LIST
  Derive (A v C)
```

Then we have (by step 3(b)(iii)(2)):

```
PROOF
  1. (A & B)     :PREMISE

TASK LIST
  Derive A
  Apply vINTRO,[A] to obtain (A v C)
```

Going back to step 3(a), we arrive at step 3(a)(iii)(2), which results in:

```
PROOF
  1. (A & B)     :PREMISE

TASK LIST
  Apply &ELIM,1 to obtain A
  Apply vINTRO,[A] to obtain (A v C)
```

And finally, after step 3(c) is performed, we have:

```
PROOF
1. (A & B)     :PREMISE
2. A           :&ELIM,1
3. (A v C)     :vINTRO,2
```

Step 3(b)(iii)(1) could also be applied—since it *always* works. Its application might result in:

```
PROOF
1. (A & B)          :PREMISE
*2.   ~(A v C)      :ASSUMPTION
*3.   (A & B)       :SEND,1
*4.   A             :&ELIM,3
*5.   (A v C)       :vINTRO,4
*6.   (A v C)       :~ELIM,2,5,2
7. (A v C)          :RETURN,6
```

In this proof, '(A v C)' functions as the **R**. This choice of **R** results in citing line 2 *twice:* once as the assumption of the ~ELIM, and again as part of the contradiction. The unusual appearance of line 6 of this proof is in fact a symptom that there is an easier way to construct a proof of '(A v C)'—namely, using the vINTRO strategy.

Sometimes, however, the vINTRO option in 3(b)(iii) cannot be applied successfully at all. In these cases, we *must* resort to the longer, indirect method of proof. This unfortunate circumstance will arise when neither disjunct of the desired disjunction is derivable *by itself.* Consider, for example, the following argument:

```
   (A v B)
   (A → C)
   (B → D)
∴ (C v D)
```

The "natural" strategy might be to derive '(C v D)' by first deriving 'C' or 'D' and then applying vINTRO. But in this example, the sad fact is that 'C' alone cannot be derived, and neither can 'D'. So we would find ourselves blocked if we tried to use step 3(b)(iii)(2) on this argument:

```
PROOF
  1. (A v B)     :PREMISE
  2. (A → C)     :PREMISE
  3. (B → D)     :PREMISE

TASK LIST
  Derive C   [You should note that 'Derive D' is just as bad.]
  Apply vINTRO,[C] to obtain (C v D)
```

We could *never* derive C from these premises—a fact that could be shown by using truth tables—and so the task list would never be emptied. Hence, we could never complete the proof with these instructions.

A successful proof, using step 3(b)(iii)(1), would be:

```
 1. (A v B)              :PREMISE
 2. (A → C)              :PREMISE
 3. (B → D)              :PREMISE
*4.   ~(C v D)           :ASSUMPTION
*5.   (~C & ~D)          :RR DM,4
*6.   (A → C)            :SEND,2
*7.   (B → D)            :SEND,3
*8.   (A v B)            :SEND,1
*9.   ~C                 :&ELIM,5
*10. ~A                  :MT,9,6
*11. B                   :vELIM,8,10
*12. ~D                  :&ELIM,5
*13. ~B                  :MT,7,12
*14. (C v D)             :~ELIM,4,11,13
 15. (C v D)             :RETURN,14
```

In this proof, the contradiction derived involved the sentences 'B' and '~B'. This was, however, a matter of choice (and discovery): contradictions *could* be derived involving the sentences 'A' and '~A', or 'C' and '~C', or even '(A v B)' and '~(A v B)'. PROOF-GIVER can help get us to line 4; after that, we're on our own.

Step 3(b)(iv) also cannot easily be transformed into a mechanical procedure, for it requires us to derive contradictory sentences **R** and ~**R**, but we are not told which sentences this might involve.

A second difficulty with PROOF-GIVER is that it builds a task list based on whether a sentence is *identical* to a previous sentence or subformula in the derivation. Sometimes, however, a sentence might not be exactly identical to a previous sentence but might be *logically equivalent* to it. Consider this argument:

(~A & ~B)
(~(A v B) → C)
∴ C

Our task list would at first contain only:

Derive C

Since sentence 'C' is contained in the second premise, PROOF-GIVER would then direct us to step 3(a)(iii)(1), at which point the task list would become:

Derive ~(A v B)
Apply →ELIM,2,[~(A v B)]

But how do we derive '~(A v B)'? Glancing at our list of logical equivalences, we might see that '~(A v B)' is logically equivalent to '(~A & ~B)' and so can be derived in one step using the rule RR DM. But PROOF-GIVER does not see this and notices only that the two are not *identical*. PROOF-GIVER directs us to step 3(b)(iv).

PROOF-GIVER could be corrected to allow it to "see" logical equivalences as identities and then use RR. In other words, every time PROOF-GIVER refers to "identical" sentences, we could replace this with "identical or logically equivalent" sentences. But then, unfortunately, *testing* to see whether a sentence might be logically equivalent to a previous sentence or subformula would consume almost all the time used in applying PROOF-GIVER. Consequently, the astute user of PROOF-GIVER should keep a sharp eye out for when a rule of replacement might be used. But such equivalences will not be built into PROOF-GIVER.

A still worse problem is that we can occasionally be "hung up" at step 3(a), when we should go to step 3(b). Consider the following argument:

(A → (B & C))
B
C
∴ (B & C)

After performing steps 1 and 2 of PROOF-GIVER, we have:

```
PROOF
  1. (A → (B & C))     :PREMISE
  2. B                 :PREMISE
  3. C                 :PREMISE
```

TASK LIST
  Derive (B & C)

We now go to step 3(a), because the desired result, '(B & C)', is contained in an earlier line (the first premise). After step 3(a)(iii)(1) we have:

```
PROOF
  1. (A → (B & C))     :PREMISE
  2. B                 :PREMISE
  3. C                 :PREMISE

TASK LIST
  Derive A
  Apply →ELIM,1,[A]
```

But it is easy to see—and could be shown by a truth table—that 'A' does not validly follow from the premises. Consequently, we would never be able to derive 'A' on a line by itself. (More precisely, we would never be able to eliminate all the 'Derive' statements from the task list for this argument, once it is begun in this way.)

The problem lies with step 3(a). Whenever the desired result is a subformula of an accessible line, step 3(a) applies. But *sometimes* the desired result can *only* be derived by using parts of step 3(b). As an example, if we went to step 3(b)(ii) instead of step 3(a), we would have in the task list

```
Derive B
Derive C
Apply &INTRO,[B],[C]
```

which would eventually result in this proof:

```
1. (A → (B & C))     :PREMISE
2. B                 :PREMISE
3. C                 :PREMISE
4. (B & C)           :&INTRO,2,3
```

The problem is not easy to correct. About the only symptom that we are hung up is if we *feel* that our proof is not going anywhere. Another symptom is that our task list grows and grows, with no end in sight. In these cases, we should retrace our steps to find where our task list seems to have gone wrong. That will be a step where PROOF-GIVER placed us at step 3(a) when it would have been more fruitful to be at step 3(b)(i). Once we have found where the difficulty seems to lie, we should rebuild the task list, this time going to step 3(b)(i) instead of step 3(a).

In our discussion of the method PROOF-GIVER, we may have become too immersed in the details of constructing proofs. Let us rise above the sometimes dreary details for a moment and review the general significance of the steps in PROOF-GIVER.

As we mentioned earlier, certain features of the construction of proofs—notably the infinitely many possible connections between the premises and the conclusion using our rules—require that we "work backward" from the conclusion. We must first somehow determine how the conclusion *might* have arisen.

There are essentially two ways that a conclusion can be derived: It can be "part of" an earlier line of the proof (such as a premise), or it can be "reconstituted" from information contained somewhere in the premises.

It is the purpose of steps 3(a) and 3(b) to deal with these possibilities. If the conclusion is contained in some part of a previous line, we are at step 3(a), which then tells us how to extract the conclusion from the previous line. If the conclusion is *not* contained in a previous line, we are at step 3(b). There, there are numerous recipes for building up the conclusion from other bits of information that might somehow be contained in the premises.

PROOF-GIVER, as we have described it, is primarily a depth-first search strategy: It guides us on a single path through the search tree. The main flaw with PROOF-GIVER, as with other depth-first strategies, is that if we are wrong—that is, if we do not reach our goal easily—we must back up and reconsider one of the branches we earlier ignored. In other words, depth-first strategies will often require us to "backtrack."

Summary

In this chapter, we presented a method PROOF-GIVER, for constructing proofs of arguments known to be valid. PROOF-GIVER is not an algorithm, since it will not always work and, at certain points, requires human intervention. Nevertheless, it can be useful and illustrates some important techniques. PROOF-GIVER uses a *depth-first search strategy* to search a *tree* of possible lines of a proof; that is, it follows a "best guess" as to how the proof should proceed rather than trying all possibilities at once. You should find it helpful in constructing proofs of arguments.

Exercises

A. 1. Using PROOF-GIVER, determine what the next change of the proof or task list should be.

a. PROOF

1. $(A \rightarrow B)$	:PREMISE
2. $(C \& A)$	:PREMISE

TASK LIST

Derive $(B \& C)$

b. PROOF

1. $((A \rightarrow {\sim}B) \rightarrow D)$	:PREMISE
2. ${\sim}B$	:PREMISE
3. $(D \rightarrow (E \vee B))$	:PREMISE

TASK LIST

Derive D

Apply $\rightarrow$ELIM,3,[D]

c. PROOF

1. $(A \& (B \vee D))$	:PREMISE
2. $((B \vee D) \rightarrow (A \rightarrow \sim E))$	:PREMISE
3. A	:&ELIM,1

TASK LIST
- Apply &ELIM,1
- Apply →ELIM, $[(B \vee D)]$,2
- Apply →ELIM, $[(A \rightarrow \sim E)]$,3

2. From an inspection of the proof and task list of (c), what is the desired conclusion of the argument?

B. Using PROOF-GIVER, construct proofs of the following arguments:

1. $(A \& (A \rightarrow (B \rightarrow C)))$
   $\therefore (B \rightarrow C)$
2. $((A \rightarrow B) \& (B \rightarrow C))$
   $(C \rightarrow D)$
   $\therefore (A \rightarrow D)$
3. $(A \& (\sim B \& C))$
   $\therefore \sim B$
4. $(A \rightarrow B)$
   $(A \rightarrow \sim E)$
   $\therefore \sim A$
5. $(A \rightarrow B)$
   $(A \& D)$
   $\therefore B$
6. $\sim A$
   $(A \vee \sim B)$
   $(B \rightarrow \sim C)$
   $\therefore (\sim B \& (B \rightarrow \sim C))$
7. $(A \rightarrow \sim C)$
   $(B \rightarrow C)$
   $\therefore (\sim A \vee \sim B)$
8. $(\sim B \leftrightarrow (A \& D))$
   $(\sim B \& (A \leftrightarrow E))$
   $\therefore E$
9. $(A \rightarrow (B \rightarrow (\sim C \rightarrow D)))$
   $(A \& B)$
   $\therefore (\sim C \rightarrow D)$

C. Add a step to PROOF-GIVER that will enable it to handle 'Derive $(P \leftrightarrow Q)$' in a task list.

D. For programmers:
1. Why doesn't "Apply &INTRO,*n*,*m*" in a task list *require* a goal sentence? That is, why is the goal sentence optional?
2. Assume that premises and the conclusion contain only atomic-sentence letters and the symbol &. Write a program to input such an argument and construct a proof of it.
3. Assume that premises and the conclusion contain only atomic-sentence letters, parentheses, and the symbol →. Write a program to input such an argument and construct a proof of it.

Suggestions for Computer Implementation

The full implementation of PROOF-GIVER should be supremely gratifying to any ambitious "hacker." As in PROOF-CHECKER, an appropriate data structure (such as an array) is necessary for storing the ongoing proof. Initially, of course, only the premises would be stored; the conclusion—with its justifying rule left blank—might be stored temporarily in some arbitrarily high line number of the final proof, sure to exceed the other lines of the proof (say, 100).

A separate data structure is also needed to store the distinct elements of the task list. Furthermore, since the contents of the task list are constantly changing, we frequently need to "sort" these elements into their order of priority.

We have not been as rigid in the text with the format of each record in the task list as we would have to be if we wished to implement PROOF-GIVER. We can impose the required rigidity here. The 'Derive' statement should have four fields:

1. A RANK: a number to indicate the priority of a task in the task list
2. A TASK: the task to be done—"DERIVE" (or "APPLY")
3. A sentence to be derived (our "goal")
4. The subproof-depth of the task

For example, consider the following argument:

```
   A
   B
∴ (A & B)
```

Our task list might at first contain

```
RANK(1) = 1
TASK(1) = DERIVE
TSEN(1) = (A & B)
TSUB(1) = 0
```

indicating, respectively, that the first element in the task list is *first* in order of priority, that the task is to *derive* a sentence, that the goal is to derive the sentence '(A & B)', and that it is not in a subproof. The names TSEN and TSUB indicate *task-list* sentences and subproof depths.

The other kind of statement in the task list is the 'APPLY' statement, which could have the following fields:

1. A RANK
2. The TASK: Here, 'APPLY'
3. The goal sentence (optional, except in the case of rules, ASSUMPTION, &ELIM, vINTRO)
4. The TRULE to be applied
5. The line(s) to which the rule is to be applied—expressed either as a line number or as a sentence
6. The subproof depth of the apply line

Thus we might have

```
RANK(2)  = 3
TASK(2)  = APPLY
TSEN(2)  = (A v B)
TRULE(2) = vINTRO
TCIT1(2) = 1
```

```
TCIT2(2)  = 0
TCIT3(2)  = 0
TSUB(2)   = 1
```

indicating that the priority of task-list line 2 is 3, that the task is to APPLY, that the goal sentence is '(A v B)', that the rule is to be applied to line 1, and that the APPLY is in a subproof of depth 1.

Note that the task can be stored in numerical fashion, since there are only two possible tasks. For example, 1 = DERIVE, and 2 = APPLY. Similarly, TRULE can also be stored numerically: 1 for &INTRO, 2 for &ELIM, 3 for vELIM, and so on. Using numbers where possible might spare us some nasty string operations. A sentence, however, such as '(A & (B v C))' cannot (easily!) be converted into numerical information.

Once the stage is set in this fashion, it is relatively straightforward to convert METHOD PROOF-GIVER into a computer program. Several features of the problem might, however, threaten this conversion.

1. Sometimes a line in a task list might be replaced by *two* (or more) lines. These new lines are always inserted at the beginning, and so they disturb the order of items in the task list. The clue to their priority in the task list is their RANK. If a task of rank 1 is to be replaced by two tasks, we might let these two tasks have ranks 1.1 and 1.2. We would first delete the original task, and we would know that the task with rank 1.1 is to be performed before the task with rank 1.2 and that both are to be performed before a task of rank 2.

   Similarly, a task of rank 2.1 might be replaced by tasks of ranks 2.11, 2.12, and 2.13. Including the RANK of a task performs an automatic sort or ordering of the elements in our task list.
2. As we have noted, sometimes PROOF-GIVER is misled or tricked, and the task list grows prodigiously—never getting closer to the conclusion. Since computers work so quickly, it might be a matter of mere microseconds before the task list is full of hundreds of tasks which will never allow PROOF-GIVER to reach the conclusion.

   There are two solutions to this problem—one a "quick fix," the other more elegant.

   We could write our program so that any time the task list grows to a certain size (say, twenty or thirty lines), then the program stops and alerts the user to the problem. (This is a so-called "disaster cutoff.")

   Another solution is to display to the user each revision of the task list and ask the user if he or she wishes to:
   a. Go on.
   b. Stop.
   c. Back up to an earlier stage of the proof and task list.
   d. Override step 3(a).
   e. Intervene and make a "human" suggestion on what to do: what contradiction to aim for, what step to follow, or what goal sentence to have.
3. The mention of "backing up" alerts us to the fact that we also need to store information about *past* proofs and task lists, as well as about the current, ongoing ones. We thus need to have the proof and task list of every previous step available

to us if we are going to be able to "back up." The easiest way to implement this suggestion is to store the proofs and task lists in some data structure (such as two-dimensional arrays). For example, TRULE(3,2) might be the TRULE in the second element of a task-list array on the third step of applying PROOF-GIVER.

4. It is possible to program other "hunches" and strategies into PROOF-GIVER beyond the ones given in this chapter. Strategies for choosing the contradictions to be aimed for with ~ELIM and ~INTRO could be given, as well as tips for keeping proofs shorter. If the arguments PROOF-GIVER is attempting to prove are all being created by a single person, we might also program strategies to deal with what seem to be this person's habits. The user might, for example, use vINTRO frequently or →INTRO rarely.