PART A

Lecture Notes

Chapter 1. Computer Science: The Mechanization of Abstraction

The Big Picture

Before it gets obscured in the details, we wish to reinforce the theme that this book, and courses built around it, are about models, algorithms, and — in general — ideas. We don't want to ignore matters of programming style, but we will assume that good programming practice has already become part of the students' experience in an earlier course. We suggest that it is now time for students to appreciate that great programs are noted for their ideas, as well as their implementations.

Programming Is Mathematics Made Real

In keeping with our philosophy that students should bring some programming skill to a course based on this book, we encourage the instructor to look for opportunities to enhance the students' ability to understand mathematics and abstractions by calling upon their experience in programming. Some students may view mathematics as the "easy" thing, from which we can learn the "hard" thing — programming. Most students, however, will look at it the other way around: Understanding a mathematical idea, say an inductive proof, is hard because we have nothing but abstract symbols to stare at and try to understand. They may feel that recursive procedures are, by comparison, easy, because the program is tangible. They can run it, see what happens, change it, and in general experiment until they get the idea and make it work. The instructor should take advantage of the opportunities to use programming to make mathematical abstractions easier to comprehend.

We give many examples of the process of converting programming intuition into mathematical intuition in the book and these notes. However, here is one important example to firm up the idea. Many students may be confused when we observe that "if A then B" is a true statement when A is a false statement, regardless of what B is. We need this reasoning in almost all chapters, and we make it formal when we cover logic starting in Chapter 12. On the other hand, students should have had experience with a conditional statement. They know that a statement like

```
if X=Y then Z := Z+1
```

is executed without involving Z whenever the condition is false, that is, whenever $X \neq Y$.

Chapter 2. Iteration, Induction, and Recursion

What Is a Proof?

Many students will, in Chapter 2, face the need to develop proofs for the first time in their lives. Among the frustrations they may face is the lack of understanding regarding what constitutes a valid proof. We suggest that you remind them that proving theorems is a social process: A proof is good enough if it convinces your audience.

Until they gain the experience often called "mathematical sophistication," it is a good idea to lay down as many rules as you can regarding what constitutes a proof. Each instructor needs to think the matter through for himself or herself. But we suggest, for example, that an inductive proof be regarded as inadequate if it does not at some point say "and by the inductive hypothesis the following is true." Also we give, in the notes for this and other chapters, several templates for proofs of various kinds. You may wish to insist that the proof follow the template, or that it follow certain aspects thereof, for example that each inductive proof have a basis part and an inductive part.

Some Motivation

Students frequently have trouble believing that inductive proofs, or proofs of any kind, are illuminating. If they have been through a typical course on Pascal, they may be suspicious of recursive programming as well. We had some success convincing people of the utility of recursion when we taught ML as a second language, although languages like Lisp or Prolog can get the same point across, since these all use recursion as the primary way to repeat an operation.

It is useful to point out the way certain important structures, with which students are familiar, are naturally defined recursively. For example, we discussed directory structures, which we had to call "folders" because most of the students were familiar with the MAC environment and were just learning UNIX. The first file systems were not recursive; they were just directories of files. It was a great advance when people realized that it was feasible to let directories have subdirectories. Other examples of recursions include programming language structures like control-flow statements and expressions.

A good motivation for inductive proofs can center around a frank discussion of proving correctness of programs. After Sections 2.5 and 2.9, students should accept that inductive proofs are useful if they want to prove programs correct. We need to argue that such proofs can be performed in practice and that they can catch bugs missed by programmers. One can tell stories of missiles that went off course because of a misplaced comma and similar tales, and it is believable that such errors could have been caught by an automatic checker. Yet that is only one argument for why people should learn inductive proofs.

We believe that the real motivation for learning about inductive proofs of correctness is not so the student programmer will write formal proofs of code correctness. Rather it is to make the student aware that a proof lies behind correct code, and that he or she should always write a program with a correctness proof in mind — otherwise there is a much greater chance of writing faulty code.

One place where proofs of program correctness have been used is in the validation of protocols for communication.

◊ More Arithmetic Inductions

Formulas for sums make fodder for illustrative inductive proofs. We are probably all familiar with examples such as

1. $\sum_{i=1}^{n} i = n(n+1)/2$
2. $\sum_{i=1}^{n} i^2 = n(n+1)(2n+1)/6$
3. $\sum_{i=1}^{n} i^3 = n^2(n+1)^2/4$
4. $\sum_{i=1}^{n} i(i+1) = n(n+1)(n+2)/3$
5. $\sum_{i=1}^{n} 1/i(i+1) = n/(n+1)$

In fact, we can invent an almost unlimited number of similar formulas if we remember that the sum from $i = 1$ to n of any polynomial $p(i)$ is itself a polynomial $q(n)$ of degree one greater than the degree of p. In fact, it will be a polynomial with a constant term of 0, since its value must be 0 when $n = 0$. Thus, if p is of degree d, we know q has only d unknown coefficients. If we calculate the value of $\sum_{i=1}^{n} p(i)$ for $i = 1, 2, \ldots, d$, we can solve for the coefficients of q.

For example, suppose we want the formula for $\sum_{i=1}^{n} i(i+2)$, that is

$$1 \times 3 + 2 \times 4 + 3 \times 5 + 4 \times 6 + \cdots$$

We can write $q(n)$ in the form $an^3 + bn^2 + cn$ and observe that $q(n)$ is 3, 11, and 26 for $n = 1$, 2, and 3, respectively. We easily solve for the coefficients and find $a = 1/3$, $b = 3/2$, and $c = 7/6$. Thus, $q(n) = (2n^3 + 9n^2 + 7n)/6$, or $n(n+1)(2n+7)/6$. We have "invented" the inductive proof exercise to show that

$$\sum_{i=1}^{n} i(i+2) = n(n+1)(2n+7)/6$$

There is another way to create such formulas. Start with any polynomial $q(n)$ that has a 0 constant term. Let $p(i) = q(i) - q(i-1)$. Then $\sum_{i=1}^{n} p(i) = q(n)$. For example, if we pick $q(n) = n^3$, we get $p(i) = 3i^2 - 3i + 1$ and the inductive proof exercise "show that $\sum_{i=1}^{n} \big(3i(i-1)+1\big) = n^3$."

◊ Fibonacci Recursion

The Fibonacci recursion

$F(0) = 0$
$F(1) = 1$
$F(n) = F(n-1) + F(n-2)$ for $n \geq 2$

is a useful example that may be familiar to some students. It helps illustrate the point that the straightforward recursive way to compute some functions may not always be efficient, because the obvious recursive algorithm takes time $O\big((1.62)^n\big)$. Since we don't use big-oh notation in this chapter, you can state the point less formally, and/or you can demonstrate the program for different values of n and let students see that all of a sudden, as n gets larger, the program goes from "instantaneous" response to "taking forever."

Vaughan Pratt has pointed out how the problem also illustrates two important algorithmic techniques: divide-and-conquer and dynamic programming (the latter appears in Ch. 6). The dynamic programming algorithm, where we compute $F(n)$ by filling out an array `F[0..n]` in $O(n)$ time, should be obvious. Far less obvious is a divide-and-conquer technique, which we can sketch as follows.

We can relate the pair of values $\big(F(n), F(n-1)\big)$ to the pair of values

$\big(F(n-1), F(n-2)\big)$

by the matrix-vector multiplication

$$\begin{pmatrix} F(n) \\ F(n-1) \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F(n-1) \\ F(n-2) \end{pmatrix}$$

If we let Q be the matrix $\left(\begin{smallmatrix} 1 & 1 \\ 1 & 0 \end{smallmatrix}\right)$, then the pair $\big(F(n), F(n-1)\big)$ is related by Q^2 to $\big(F(n-2), F(n-3)\big)$, by Q^4 to $\big(F(n-4), F(n-5)\big)$, by Q^8 to $\big(F(n-8), F(n-9)\big)$, and so on.

Thus, by computing Q to the powers of 2, and then multiplying an appropriate subset of them, we can compute Q^{n-1} in $O(\log n)$ time. In justification, note that we can multiply any pair of 2 *times* 2 matrices in $O(1)$ time, at least we can if we assume we can multiply the large integers that get produced by this method in $O(1)$ time.

Now, Q^{n-1} relates $\big(F(n), F(n-1)\big)$ to $\big(F(1), F(0)\big)$. However, since $F(1) = 1$ and $F(0) = 0$, it follows that $F(n)$ is the upper left-hand element of Q. Thus, we have computed $F(n)$ in $O(\log n)$ time, much faster than the dynamic programming solution. Unfortunately, it is not realistic to demonstrate the difference in running times between these two solutions, because the numbers get too big too fast, so a discussion of these issues will either have to be very informal or wait until Chapter 3.

◇ Buy One, Get One Free

A useful example of an inductive proof that also uses the idea of sorted lists is the following. "Pizza Hovel" offers a deal in which if you buy one pizza, you can get a second pizza, of equal or less cost, free. Suppose n students each order a pizza. How should they pair up so that the total cost of the pizzas is minimum?

One way to state the solution is to sort the prices of the pizzas, largest first, and pay for the odd pizzas only. Note that if n is odd, you pay for the last pizza unpaired.

The inductive proof, worth discussing in class, is that this selection is the least possible cost. There are several nuances in the proof. First, there is a double basis, which you could take to be $n = 1$ and $n = 2$ (or $n = 0$ and $n = 1$ if you don't mind arguing that the notion of an "empty list of pizzas" makes sense). Second, the proof is actually a complete induction, because you must consider alternative solutions where the most expensive pizza is paired with another, which reduces to the case of $n - 1$ pizzas, as well as solutions where the most expensive pizza is unpaired; that reduces to the n-pizza case.

◇ n Choose m

Another instructive series of examples concerns the function $\binom{n}{m}$. The book has several exercises on the topic, but the function itself is something the students should learn about if they've never seen it. To begin, discuss the question of how many ways can we select, say, 5 cards out of 52, and generalize to get the rule that $\binom{n}{m} = n!/(n-m)!m!$.

Now we can introduce the recursive definition that says

BASIS. $\binom{n}{0} = \binom{n}{n} = 1$.

INDUCTION. $\binom{n}{m} = \binom{n-1}{m} + \binom{n-1}{m-1}$ if $0 < m < n$.

The explanation for the inductive step is that we can pick m things out of n either by picking the first element and then picking $m - 1$ out of the remaining $n - 1$ elements, or by rejecting the first element and then picking m things out of the remaining $n - 1$.

This recursive definition is slightly awkward because the basis case covers instances with arbitrary n's, and when we do an induction on n to prove something, we use the basis of the definition in both the basis and the inductive parts of the proof. In the notes for Chapter 5 we point out how this awkwardness can be avoided when we think of the recursion as defining a binary tree and do a structural induction on the tree.

However, an interesting inductive proof is that the recursive and nonrecursive definitions are equivalent. We expect them to be, because they each count the number of choices of m things out of n, in different ways. The proof is an induction on n, made slightly tricky by the factor mentioned above about the basis of the recursive definition appearing in both the basis and induction of the proof.

This example can be a running theme in your lectures. We shall return to it in the notes for Chapter 3, where it is a good problem in analyzing running time of a recursive program and in Chapter 5 where it is an example of structural induction.

◊ Proof by Example?

A student asked about the inductive proof on p. 37 whether it was not a "proof by example," and were such proofs acceptable. The problem was that Fig. 2.6 gives a picture of an example of the inductive step, and the student assumed that the figure was the proof. It is important to point out the distinction in this or other situations where you rely on an example to illustrate how an inductive proof works.

◊ A Template for Inductions

After covering some examples of simple and complete inductions, it is useful to give students an outline of the most general case. In general, there can be several basis cases, from i_0 through j_0, and the inductive step requires complete induction. An outline might look like this.

1. Specify the statement $S(n)$ to be proved. Say that you are going to prove $S(n)$ by induction on n, for $n \geq i_0$. Specify what i_0 is; often it is 0 or 1, but could be any integer. Explain intuitively what n represents, e.g., the length of a list.
2. State the basis case(s). These will be all the integers from i_0 up to some integer j_0. Often $j_0 = i_0$, but j_0 could be larger, and we then say we have "multiple basis cases."
3. Prove each of the basis cases $S(i_0), S(i_0+1), \ldots, S(j_0)$.
4. Set up the inductive step by stating that you are assuming

   $$S(i_0), S(i_0+1), \ldots, S(n)$$

   (the "inductive hypothesis") and that you want to prove $S(n+1)$. State that you are assuming $n \geq j_0$. Express $S(n+1)$ by substituting $n+1$ for n in the statement $S(n)$. In many cases, you need only assume $S(n)$ (that's a "simple," rather than "complete," induction).
5. Prove $S(n+1)$ under the assumptions mentioned in (4).
6. Conclude that $S(n)$ is true for all $n \geq i_0$ (but not necessarily for smaller n).

◊ Some Fallacious Inductive Arguments

There are some well known fallacies, most of which involve inductive arguments that seem to work but fail for one or more small values of the induction variable n. For example, many have heard of the proof that "all marbles are the same color." The basis is that it is true for one marble. For the "induction," suppose it true for $n \geq 1$ marbles and consider a group of $n+1$ marbles. If we eliminate one marble, say marble a, then by the inductive hypothesis the remaining n marbles are of the same color. Swap marble a with another marble and the new group, including marble a, are all of the same color. By "transitivity of same-color," all $n+1$ marbles are the same color. But the discerning student will instantly point out that this argument doesn't hold for $n+1 = 2$, by noting that our group may consist of one blue marble and one red marble.

A similar fallacious argument can be used to show that $a^n = 1$ for all $n \geq 0$. The basis, $n = 0$, is clearly true. For the (complete) induction, let $n \geq 0$, and consider a^{n+1}. A small amount of algebra tells us that $a^{n+1} = a^n \times a^n/a^{n-1}$. If we claim "by the inductive hypothesis" that a^n and a^{n-1} are known to equal 1, then $a^{n+1} = 1 \times 1/1 = 1$. Of course this argument fails when $n+1 = 1$, because then $a^{n-1} = a^{-1}$, which is not covered by the inductive hypothesis.

A similar error comes up in a buggy recursive algorithm for multiplying polynomials represented as lists. In the notes for Chapter 3 we'll discuss the functions $padd(P,Q)$ and $smult(s,P)$, which respectively add polynomials and multiply a polynomial P by the scalar (constant) s. Using these, we can define polynomial multiplication recursively. The induction is on the length of the list of coefficients representing the first polynomial. For the basis, an empty list represents the polynomial 0, so the product is 0.

For the induction, we can let polynomial P be represented by the list with first element a and remaining elements representing polynomial R. Note that $P = a + xR$ as polynomials. Thus, $pmult(P,Q)$ could be expressed as

$$padd\Big(smult(a,Q),\ pmult\big(X, pmult(R,Q)\big)\Big)$$

where X is the polynomial x represented by the list of coefficients $(0,1)$.

The reason this formula doesn't work is that when P is of degree 1, the recursion invokes another instance of $pmult$ with first argument, X, of degree 1. For P's of higher degree, both recursive uses of $pmult$ are of lower degree, and the recursion appears to work, until you try the program out. We shall discuss a better way to write these polynomial manipulation programs in the notes for Chapter 3.

◊ Geometric Sums

We have found that some students either do not remember, or were never taught, geometric and arithmetic sums. The general formula for a geometric sum is given in Exercise 2.3.11 on p. 39, and the formula for an arithmetic sum (average of the first and last terms times the number of terms) is easy to remember. It may be advisable to prove these formulas as additional examples of inductions and to be sure the students can use them. They are needed when we solve recurrence relations in Chapter 3.

It would also be useful to make sure students understand that a geometric sum with a ratio less than 1 has a limit, and that the limit is usually a good upper bound on the sum of a finite number of terms. That fact too is needed in the solution of certain recurrences.

◊ Confusing Statements and Sums

Some students make a "type error" when dealing with inductions like Example 2.4 (p. 33), where they treat $S(n)$, the statement they are trying to prove, as if it were a sum, such as $\sum_{i=0}^{n} 2^i$. It seems prudent to warn students more than once not to make this confusion, preferably as part of a more general campaign to get students to remember that type errors are as damaging to mathematics as to programming.

◇◇◇◇ Chapter 3. The Running Time of Programs

◇ Motivation for Big-Oh

Section 3.4 motivates the fact that we ignore constant factors when discussing running time; it therefore helps explain the constant c in the definition of big-oh. More mysterious to some is why we use a constant n_0 in the definition, below which the inequality need not hold. One simple motivation is that there are some common functions that behave strangely for some small values of n. For example, if $T(n)$ is the running time of a merge sort program, we would like to say that $T(n)$ is $O(n \log n)$. However, $n \log n$ is 0 for $n = 0$ and for $n = 1$, so it is not possible that $T(n) < cn \log n$ for $n < 2$, no matter what c is. If we couldn't select $n_0 = 2$, we couldn't claim that "merge sort is $O(n \log n)$."

When motivating the topic, it is also useful to point out that, while we need to understand how big-oh proofs are done, we do not make formal proofs every time we want to analyze a few lines of code. The analogy with epsilon-delta proofs in calculus may be fruitful. There too, the student has to understand the proof techniques lying under the surface but does not produce such a proof every time there is a need to take a derivative.

◇ Witnesses

The quantities n_0 and c that appear in big-oh proofs are often called *witnesses* (to the proof of a big-oh relationship). It may be helpful to encourage your students to use this term; having something to call them may help with their understanding.

◇ Template for Proofs of a Big-Oh Relationship

Many students are helped by being given outlines for proofs. Even the following two steps for proving a big-oh relationship can be a help. To prove $f(n)$ is $O\big(g(n)\big)$:

1. Give witnesses, n_0 and c. These witnesses must be specific constants, e.g., $n_0 = 47$ and $c = 12$. We must have n_0 be a nonnegative integer, and c must be a positive real number.
2. Prove that if $n \geq n_0$, then $f(n) \leq cg(n)$, for the particular constants n_0 and c you chose.

◇ Template for Proofs a Big-Oh Relationship Doesn't Hold

Proving the falsehood of a big-oh relationship is harder, and many students can be helped by the following outline. To prove $f(n)$ is *not* $O\big(g(n)\big)$:

1. Start by supposing that there were witnesses n_0 and c such that for all $n \geq n_0$, we have $f(n) \leq cg(n)$. Here, n_0 and c are symbols standing for unknown witnesses.

2. Define a particular integer a, expressed in terms of n_0 and c (for example, $a = \max(n_0, 1/c)$). This a will be the value of n for which we show $f(a) \leq cg(a)$ is false.

3. Show that for the chosen a we have $a \geq n_0$. This part can be very easy, since we may choose a to be at least n_0 in step (2).

4. Claim that because $a \geq n_0$, we have $f(a) \leq cg(a)$.

5. Derive a contradiction by showing that for the chosen a we have $f(a) > cg(a)$.

◇ Encouraging Thinking About Big-Oh

Many students when asked to come up with witnesses for a big-oh relationship will guess something like $n_0 = c = 10^{10}$, and they are usually right, if the relationship holds at all. Thus, on quizzes or homework, it may be useful to stipulate that the witnesses must be minimal; a reduction in either n_0 or c with the other fixed loses the desired property.

◇ Hard Examples of Computing Running Time

It may appear that all analysis of running time is simple, when in fact it is only the simplest examples of code where analysis is straightforward. A simple example of a challenge is the "$3n + 1$" problem, expressed as the while-loop:

```
readln(n);
while n > 1 do
    if odd(n) then n := 3*n+1
    else n := n DIV 2;
```

It is conjectured that starting with any positive integer n, we shall eventually set n to 1, but it is an open problem whether the conjecture is true, and if so, how many times we may have to go around the loop, as a function of n. For example, if we read $n = 13$, then we generate the sequence of values $n = 13, 40, 20, 10, 5, 16, 8, 4, 2, 1$ and terminate after executing the body nine times.

The following example of code in Fig. A.3.1 illustrates a more realistic situation. It represents an attempt to do a simple local optimization to solve a "knapsack" type problem as discussed in Laboratory 3. Here, we are given two arrays, A and B, of length n, and we wish to search for pairs of elements that when swapped make the sums of the elements in the two arrays more nearly equal. Incidentally, we can motivate this problem by talking about knapsacks, but perhaps a more realistic application would be balancing cargo containers on two sides of an airplane so the plane is not overweighted to either side. We should point out to the student that the code in Fig. A.3.1 is only a heuristic, and need not balance things exactly, even when it is possible to do so.

It is easy to analyze the for-loops in Fig. A.3.1; lines 7 through 13 are $O(n^2)$. It is only when we try to analyze the while-loop that we get stuck. The problem is that after making one pass, it is possible that a second pass through the arrays will make further improvements. We can only define r to be a parameter that, as a function of n, measures the maximum number of passes we can make over arrays of length n. Then we can say the big-oh upper bound on the whole program is $O(n^2 r)$.

```
(* first three lines set diff to the difference between
    the sum of A[1..n] and B[1..n] *)
diff := 0
for i:= 1 to n do
    diff := diff + A[i] - B[i];
change := TRUE; (* tells whether a round improved things *)
while change do begin
    change := FALSE;
    for i := 1 to n do
        for j := 1 to n do
            (* swap B[j] with A[i] if it improves diff *)
            if abs(diff+B[j]-A[i]) < abs(diff) then begin
                temp := A[i];
                A[i] := B[j];
                B[j] := temp;
                change := TRUE
            end
end
```

Fig. A.3.1. A heuristic for balancing sums of two arrays.

In one sense that begs the question: the program takes as long as it takes. In another sense, though, it does suggest that we can speed it up by replacing the $O(n^2)$ loops that compare all pairs with something faster and smarter. For example, we could sort the elements of both arrays together, consider only adjacent pairs on the list that come from different arrays, and swap them if they improve the difference of sums. That would replace the $O(n^2)$ loop by an $O(n \log n)$ loop, although it might increase r, especially if there is initially a large difference in the sums.

◇ Running Time of ML Programs

While the book discusses only the running time of Pascal programs, we can apply the same ideas to the analysis of ML programs. When a function is defined by a sequence of patterns, we can treat it as a sequence of if-then-else constructs. Putting a head and tail together to form a new list takes $O(1)$ time if we assume a normal linked-list implementation. Note that a "tail" is a pointer to a list; we do not have to copy the list itself. Likewise, given a list we can find its head and tail in $O(1)$ time if linked lists are used for representation. The only surprising part is the @ operator, where we might expect that concatenating two arbitrary lists could not be done in $O(1)$ time. However, since ML implementations can keep pointers to both the beginning and end of the list, it is in fact possible to perform this operation in $O(1)$ time.

◇ An Example With Polynomials

In CS109A at Stanford, we used some examples of recursive procedures by writing functions in ML and analyzing them. Each is proved correct by an induction that uses a different notion of "size" of the arguments. We represent polynomials as linked lists of their coefficients, starting with the constant term. That is, the list $(a_0, a_1, \ldots, a_k)$ represents the polynomial $\sum_{i=0}^{n} a_i x^i$. In ML, lists are primitives, so the students understood list notation. However, we expect that in most places where this book is used, the students will have been introduced to linked lists previously, so a Pascal equivalent would work as well.

To begin, we define the *smult* function to multiply scalar s by polynomial P. Its definition in ML is

```
fun smult(s:real,[]) = []
|   smult(s,a::Q) = (s*a)::smult(s,Q);
```

If you're not familiar with ML, this function just goes down a list, multiplying each element by s.

Remember that the list $a :: Q$ (head a and tail Q) represents the polynomial $a + xQ$. We can easily prove by induction on the length of list P (degree of the polynomial represented by P) that $smult(s, P)$ produces a list that represents the polynomial sP. Also, analyzing the running time of this function leads to the common recurrence

$$T(0) = O(1)$$
$$T(n) = O(1) + T(n-1) \text{ for } n > 0$$

with the expected solution that $T(n)$ is $O(n)$.

Next, define $padd(R, S)$ to produce the sum of polynomials R and S as follows.

```
fun padd(P,[]) = P
|   padd([],Q) = Q
|   padd(a::P,b::Q) = (a:real +b)::padd(P,Q);
```

Again, if you are not familiar with ML, the function adds an empty list to a list by taking the other list as the answer, and adds two nonempty lists by adding the first elements and recursing on the tails.

We can prove easily that $padd(R, S)$ is the list that represents the sum of the polynomials represented by lists R and S. Now, the induction is on the length of the shorter of R and S, and that parameter is the appropriate size measure for this function.

The recurrence describing the running time of *padd* is the same as for *smult*, and the solution is again $T(n) = O(n)$. However, note that here, n is the length of the shorter list.

Finally, we define function $pmult(R, S)$ to produce the list that represents the product of the polynomials represented by lists R and S.

```
fun pmult([],Q) = []
|   pmult(a::P,Q) = padd(smult(a,Q), 0::pmult(P,Q));
```

The meaning of this ML code should be apparent from the discussion below.

Now, the proof of correctness is an induction on the length of the first argument. To see why the function "works," suppose $R = a + xP$ as a polynomial. Then $RQ = aQ + xPQ$. We know already that $smult(a, Q)$ produces aQ, and we can invoke the inductive hypothesis to claim that $pmult(P, Q)$ produces PQ. Thus, $0 :: pmult(P, Q)$ represents xPQ. Since we already know *padd* works correctly, we see that $aQ + xPQ$ is produced as desired. Notice how this function avoids the problem discussed in the notes for Chapter 2, where we were unable to get the recursion to go through if we multiplied by polynomial x using *pmult*. Instead, we here push 0 onto the front of the list, which shifts the list one position and has the effect of multiplying by x.

To analyze the running time, we let n be the sum of the lengths of the arguments. The recursive call to *pmult* takes time $T(n-1)$, since the length of the first argument has decreased by 1 and the second argument stayed the same. Creating the list $0 :: pmult(P, Q)$ takes time $O(1)$ in addition to the time of the call. Computing $smult(a, Q)$ takes $O(n)$ time, since n is surely an upper bound on the length of Q, and *smult* is known to take time proportional to the length of the second argument. Finally, *padd* takes time proportional to the length of the shorter argument, which cannot be greater than n, so it too takes $O(n)$ time.

As a result, we get the recurrence

$$T(0) = O(1)$$
$$T(n) = O(n) + T(n-1)$$

whose solution is $T(n) = O(n^2)$. It is interesting to mention that there is a way to multiply polynomials in $O(n \log n)$ time (the fast Fourier transform), but we do not advise trying to use this method at this stage.

◊ n Choose m Again

We can do some instructive examples with the function $\binom{n}{m}$. First, we can do a simple analysis of the recursive function

```
function choose(n,m:integer): integer
begin
    if m=0 then choose := 1
    else if m=n then choose := 1
    else (* here 0<m<n if the input is proper *)
        choose := choose(n-1,m) + choose(n-1,m-1)
end;
```

We can let the value of the first argument, n, be the size measure for the function, and we easily get the recurrence

$$T(1) = O(1)$$
$$T(n) = O(1) + 2T(n-1)$$

whose solution is $O(2^n)$.

This analysis gives us an upper bound that is not exact, because even for large n, some values of m do not cause us to recurse, but to terminate with a value of 1. If one is familiar with the Gaussian distribution, we know that even for the worst case, when $m = n/2$, the recursive algorithm runs for only $O(2^n/\sqrt{n})$ time, but it does not seem reasonable to try to prove this fact.

We can prove something simpler, which is that when we start with a call to $choose(n, m)$, the number of calls we make, including the first, is exactly $2\binom{n}{m} - 1$. The proof is a simple induction on n. Since the procedure evidently takes $O(1)$ time outside of its recursive calls, that tells us the running time is $O\left(\binom{n}{m}\right)$. One can argue in a variety of ways that $\binom{n}{m}$ is less than 2^n.

◊ Patterns of Recursion and Their Growth Rates

It is useful to summarize by pointing out the difference in behavior among some common examples. We can classify many recursions as linear (one recursive call) or nonlinear (more than one recursive call). We can also classify them as *decrementing* the size by a constant (usually 1), or *dividing* the size by a constant (usually 2). For example:

- ◊ Most recursions, like *split*, *merge*, *padd*, *smult*, or *pmult*, are linear and decrementing. They have running time that is $O(n)$ times the time spent before and after the recursive call. For example, *pmult* spends $O(n)$ time outside the recursive call, so it is $O(n^2)$.
- ◊ Some recursions are nonlinear and decrementing. An example is *choose* discussed above, because it calls itself twice with a size one less than it starts with. Recursions in this class will have exponential running time and are rarely useful. In the case of the *choose* function there is a much better, $O(n)$-time algorithm to do the same job.
- ◊ Some recursions are nonlinear, but dividing. An example is *mergesort*. These recursions go on for only $\log n$ levels, and they lead to a running time that is $O(n \log n)$ if the procedure takes $O(n)$ time outside the recursive calls, as *mergesort* does. The analysis can be tricky for other amounts of time taken outside the recursive call.
- ◊ Some recursions are linear and dividing. An example is binary search. These take running time that is $O(\log n)$ times the amount of time the procedure takes outside the recursive call. For example, binary search takes $O(1)$ time outside the call and is an $O(\log n)$ algorithm.

◊ A Discussion Topic About Inflation

Every so often the Congress of the United States considers a scheme for taxing capital gains that would recognize that increases due to inflation are not really "gains." To implement such a scheme, the Internal Revenue Service (IRS) would have to publish tables giving, for each pair of years i and j, with $j > i$, the amount of inflation from year i to year j. If they must take into account inflation for n years, the complete table requires $O(n^2)$ space, but you can compute the inflation between years i and j by direct lookup in $O(1)$ time.

On the other hand, the IRS could only publish the inflation between each year i and the following year $i+1$. Then, $O(n)$ space is required, but it takes $O(n)$ time to compute the inflation between two years that are n apart, since we must multiply the yearly inflations. It might thus appear that the product of the space and the time is at least quadratic.

However, there is a "divide-and-conquer" scheme that records the inflation from any year to the next even year, the next year divisible by 4, by 8, and so on; we also record the inflation to any year from the previous even year, year divisible by 4, by 8, and so on. Then we can store information for n years in $O(n \log n)$ space, and we can compute the inflation between any two years in $O(\log n)$ time.

It would be instructive to let students develop some of these ideas independently. In addition to seeing questions of space as well as running time, they are challenged to deal with time/space tradeoffs.

◇ Are Constants Nonnegative?

When attempting to guess solutions to recurrences and check them, as on pp. 136-137, students may get confused about whether constants are nonnegative. You can allow them to assume that constants representing an amount of running time, such as the constants a and b in Example 3.26, are nonnegative. However, the constants c and d from the same example, which are unknown parameters of the expression that we guess is a big-oh upper bound, may not be assumed nonnegative, and in fact the coefficients of the lower-order terms often wind up negative.

To make matters more complex, we cannot assume that every recurrence relation defines a running time, although students may make this assumption. Therefore, technically students should not assume that the unknown constants in the recurrence are themselves nonnegative, unless it is clear that we are computing a running time.

Chapter 5. The Tree Data Model

◇ Motivation

It is useful to remind the student at the outset of some of the ways in which trees are used. They should have seen previously their use in directory structures (Section 1.3), arithmetic expressions, control-flow structures (Section 3.7), and as pictures of recursions such as the representation of the recursion for $\binom{n}{m}$, as discussed below.

◇ The Two Definitions of Trees

We give two definitions of trees, one nonrecursive and the other recursive (pp. 208–209). A useful example of induction is to prove the equivalence of the two. We can take the recursive definition to be "the" definition of *tree* and prove the parts of the nonrecursive definition by structural induction.

Additionally, we can show by complete induction on the number of nodes in the tree that every tree defined by the nonrecursive definition is also a tree by the recursive definition. The proof is fairly simple. We need to show that the children of the root define sets of nodes that are trees by the nonrecursive definition. As these trees must be smaller than the given tree, we can invoke the inductive hypothesis to claim that these sets of nodes are trees by the recursive definition.

◊ n Choose m by Structural Induction

In the notes for Chapter 3 we observed that when we evaluate $\binom{n}{m}$ by the recursive algorithm, we make exactly $2\binom{n}{m} - 1$ calls. The proof as an induction on n is a little messy.

A better approach is to think about a tree representing recursive calls, in which each node is labeled by a pair of integers $\binom{i}{j}$, and a node with this label has a left child labeled $\binom{i-1}{j}$ and a right child labeled $\binom{i-1}{j-1}$, provided $0 < j < i$. If $j = 0$ or $j = i$, then a node labeled $\binom{i}{j}$ is a leaf. Now, we can do a structural induction on this tree to show that the number of nodes in a subtree with root labeled $\binom{n}{m}$ is exactly $2\binom{n}{m} - 1$.

◊ Templates For Structural Induction

There is no denying that every structural induction can be viewed as an ordinary induction, either on the number of nodes in the tree or on the height of the tree. However, structural inductions are in general simpler than ordinary inductions, as would be expected from the fact that the former apply only to trees and are specialized to deal with trees. The following is a template for structural inductions on ordinary trees; it should be compared for simplicity with the template for general inductions in the notes for Chapter 2.

1. Specify the statement $S(T)$ to be proved, where T is a tree.

2. Prove the basis, that $S(T)$ is true whenever T is a tree with a single node.

3. Set up the inductive step by letting T be a tree with root r and $k \geq 1$ subtrees, $T_1, T_2, \ldots, T_k$. State that you assume the inductive hypothesis: that $S(T_i)$ is true for each of the subtrees T_i, $i = 1, 2, \ldots, k$.

4. Prove that $S(T)$ is true under the assumptions mentioned in (3).

The scheme above can also be used for binary trees. However, there is a related, and somewhat simpler scheme, in which we start the basis with the empty tree. Then, we can prove the induction assuming that the root has both a left and right subtree. The scheme is:

1. Specify the statement $S(T)$ to be proved, where T is a binary tree.

2. Prove the basis, that $S(T)$ is true where T is the empty tree.

3. Set up the inductive step by letting T be a tree with root r and subtrees T_L and T_R. State that you assume the inductive hypothesis: that $S(T_L)$ and $S(T_R)$ are true.

4. Prove that $S(T)$ is true under the assumptions mentioned in (3).

◇ Additional Motivation for Structural Induction

In addition to the relative simplicity of structural induction mentioned above, it is useful to train students to use structural induction because the style mirrors the way recursive programs on trees usually work, as in the recursion schemes of Figs. 5.13 and 5.30.

◇ A Fallacious Form of Tree Induction

Students often want to perform inductions on the number of nodes of the tree, especially proofs where they assume a statement for n-node trees and prove it for $(n+1)$-node trees. This proof may be fallacious.

When doing inductions on integers, we have suggested the proper methodology where we try to prove statement $S(n+1)$ by using $S(n)$; call this approach "leaning back." Sometimes students see this process as starting with $S(n)$ and proving $S(n+1)$; call this "pushing out." In the integer case, these are essentially the same idea. However, with trees, students will sometimes start by assuming the statement for an n-node tree, add a node somewhere, and claim that the result is proved for all $(n+1)$-node trees.

This argument is fallacious. For example, consider the claim $S(n)$: "all n-node trees have a path of length $n-1$." It is surely true for the basis, $n=1$. In a false "induction," we might hear: "assume an n-node tree T has a path of length $n-1$, say to node v. Add a child u of v. We now have an $(n+1)$-node tree with a path of length n, proving the inductive step."

This argument is, of course, fallacious because it does not prove the result for all $(n+1)$-node trees, just the selected few trees. A correct style of proof does not "push out" from n to $n+1$ nodes, but rather "leans back" by starting with an arbitrary $(n+1)$-node tree and carefully selecting a node to remove to get an n-node tree. Note that unlike integer inductions, where trees are concerned there is a significant difference between the two approaches.

◇ Variants of Trie Structures

As the student will note from calculations in the exercises, tries are very wasteful of space. A useful exercise is to consider a union type for nodes, where a node either has *bf* pointers to its children or has no pointers at all (*bf* is the branching factor). The latter would be used for the leaves. How much space does this variant save?

◇ Tries Versus Binary Trees

The difference between a binary tree and a trie with a branching factor of 2 is subtle and may need to be explained in more detail. First, the student should remember that the binary tree is an abstraction and the trie is a data structure. However, both share the ability to go, in one primitive operation, from a node to the ith child of that node, and they support the notion that a child can be "missing."

That notion, of slots for children and "missing" children, is not part of the (ordinary) tree data model. However, a data structure implementing a model is permitted to support additional "features." Thus, we can use the trie data structure and still treat the tree as "ordinary." For example, we can find the ith of the existing children by ignoring `NIL` pointers. The advantage of the trie is seen when we want to, say, find the child corresponding to the letter `x`. We can do so in one step, regardless of the number of children to the left and regardless of the number of `NIL` pointers among the slots for the children.

Chapter 6. The List Data Model

A Quick Once Over

If you have been coupling lectures with programming in a new language, such as ML, Lisp, or Prolog, that has lists as a primitive, then by the time you reach Chapter 6, the student will be fairly familiar with many of the concepts. You may thus wish to skim Chapter 6. We suggest you stress three important topics: the use of stacks for implementing recursion, the longest common subsequence problem, and implementation of strings.

Problems With Recursion

To motivate the use of dynamic programming, it is suggested that you analyze some "bad" recursions, including the recursion for the LCS problem. Another important example is the recursive definition of $\binom{n}{m}$, and it can be pointed out that the corresponding dynamic programming algorithm is the construction of Pascal's triangle.

Matching and LCS's

A good pair of inductive proofs is to show that there is a correspondence between common subsequences of two strings and matchings of those strings. These proofs strengthen the informal arguments in the bottom half of p. 308. Specifically, we show that for every common subsequence of length k for two strings w and x, there is a matching between w and x in which k positions for each string are matched. Also, for every matching involving k pairs of positions of these strings, there is a common subsequence of length k. As a result, the length of the LCS is also the maximum number of pairs of positions that can appear in one matching.

Chapter 7. The Set Data Model

Mutual Recursion

The mutual recursion used on p. 339 in which the `union` and `assemble` functions call each other, is illegal in most current implementations of Pascal. We think that is a pity and not justified by the state of compiler technology, but we may have to face the problem of simulating mutual recursion in Pascal or many other languages. There are two possible approaches, neither completely satisfactory.

1. We can expand one procedure in-line. For example, we could replace the call to `union` in line (3) of `assemble` by the body of `union` itself. Since the actual and formal parameters are the same (L and M), we do not even have to substitute for the parameters. In general we would have to make a suitable substitution and be very careful about variables declared in both procedures. This approach seems simple, but only because there is but a single call to `union` in `assemble`. If we substitute in `union` for each call to `assemble`, we wind up expanding the code greatly, as well as making a number of substitutions of formal for actual parameters. To make matters worse, substituting in `assemble` for the call to `union` is defective, because `union` is the procedure we want. We must now write a new `union(L,M)` that calls `assemble(x,L,M)` with some dummy x and then removes it from the answer.

2. Instead, we can introduce into one function, say `assemble`, a new argument, say F. F is a function, and `assemble` must call `F(L,M)` instead of `union(L,M)` on line (3). Now, `assemble` is no longer mutually recursive with `union`. However, it is legal to let `union` call `assemble` with actual parameter `union` for formal parameter F. In this way, the mutual recursion has been hidden, and the typical Pascal compiler will accept this code. However, it should be evident that the code has become far less transparent, and we do not like this solution any more than we like the first solution.

Equality of Sets

When discussing the equality of sets, we should note that two sets which look different can actually be the same sets. The point is discussed in the box at the top of p. 325. It may not be obvious the optional reordering extends to arbitrarily deep levels. For example, $\{\{1,2\},3\}$ and $\{3,\{2,1\}\}$ are the same set.

Another point worth reminding the student of is the distinction between equality of sets and equivalence of expressions. To emphasize the difference, we might ask "when is set R equal to set S?" The answer is "when R and S have the same members. However, if we think of R and S as expressions with one operand and no operators, we cannot sensibly ask "when is expression R equivalent to expression S?" The proper response is that these two expressions are not equivalent, period. We can easily suggest values for sets R and S that make the expressions R and S evaluate to different sets — just pick any two sets that are not equal.

◇ Outer Space in a Venn Diagram

It is worth explaining to students that the region marked 1 in both Figs. 7.1 and 7.2 (the points not in any of the sets) is not itself a set. If it were, it would be almost a universal set, and we could easily get caught in Russell's paradox. That is not a problem; it is still reasonable for this region to represent a typical element that is not in any of the sets.

◇ Proofs of Containment

The proofs of containment in Figs. 7.3 and 7.4 are reasonable models of the sorts of inferences we need to make. However, they fail to illustrate a common situation where you infer something like "x is in A or x is in B." You then need to divide the proof into two cases, one where x is assumed in A and the other where x is assumed in B, and you must draw the desired conclusion in both cases. The logic behind case analysis is discussed on p. 658, but the student should be given an intuitive grasp of the idea now. It can be likened to an if-then-else branch in a program.

◇ Static and Dynamic Relations

It is worth explaining that mathematics tends to view relations, for example $<$ on the integers, as static things — that is, things that never change. However, in computer science it is more normal to regard relations as dynamic; they change from time to time. There are many examples of this viewpoint in Chapter 8, but it may help to discuss in Chapter 7 a simple example such as the following. For each student in a class we keep a binary relation from assignment names to grades. Initially the relation is empty. As assignments are graded, new pairs are inserted. As the student may argue for revised grades, a pair may be deleted and replaced by one with a different grade.

The dynamic viewpoint also helps underscore the difference between the declared domain and range for a relation and the set of elements currently appearing in the first or second components, respectively, of the pairs in the relation. For example, we might declare the domain of an assignment-grade relation to consist of — say — the strings `hw1` through `hw10`, and the range to consist of the integers 0 through 100. That domain and range circumscribe the values that may ever appear in first or second components, respectively. However, at any given time, only a subset of the assignment names may actually appear, and it is unlikely that all the possible grades appear in one relation.

◇ Examples of Partial Orders

Two useful examples of partial orders are

1. Ancestor in a given tree T.
2. "To-the-left" in a given tree T.

◇ "Dictionary Operations" on Functions and Relations

We can summarize the assumptions made about the appropriate operations called *insert*, *delete*, and *lookup* in Sections 7.8 and 7.9 by the following table, which also includes the case where a relation is treated as a set whose elements are pairs and the ordinary dictionary operations are used. In this table we assume that a represents the domain element and b the range element. The elements listed are the arguments for the operation.

	As Set	Function	Relation
Insert	a, b	a, b	a, b
Delete	a, b	a	a, b
Lookup	a, b	a	a

It is also important to emphasize that these choices were motivated by our assumptions about how it made sense to use functions and relations stored as data. The student may indeed have some problem where he or she needs another set of operations on the data. Chapter 7 talks about the problem of accessing a relation or function "backwards," that is, given a range element find the associated domain elements, and the student should be reminded that Chapter 8 will explain data structures in which all combinations of operations such as these can be done as efficiently as possible.

◇ An Informal Definition of Infinite Sets

The student may not think that the formal definition of an infinite set (1-1 correspondence with a proper subset) is a very intuitive way to define the concept. It might be more appealing to say that an infinite set is one that, for each n, contains a subset of size n. We can prove from the formal definition that every infinite set has this property, and the proof is a nice induction.

STATEMENT $S(n)$: If I is an infinite set, then I has a subset with n elements.

BASIS. Let $n = 0$. Surely $\emptyset \subseteq I$.

INDUCTION. Assume $S(n)$ for some $n \geq 0$. We shall prove that I has a subset with $n+1$ elements. By the inductive hypothesis, I has a subset T with n elements. By the formal definition of an infinite set, there is a proper subset $J \subset I$ and a 1-1 correspondence f from I to J. Let a be an element in $I - J$; surely a exists because J is a proper subset.

Consider R, the image of T under f, that is, if $T = \{b_1, \ldots, b_n\}$, then $R = \{f(b_1), \ldots, f(b_n)\}$. Since f is 1-1, each of $f(b_1), \ldots, f(b_n)$ are different, so R is of size n. Since f is from I to J, each of the $f(b_k)$'s is in J; that is, $R \subseteq J$. Thus, a cannot be in R. It follows that $R \cup \{a\}$ is a subset of I with $n+1$ elements, proving $S(n+1)$.

◊ More About Real Numbers

The diagonalization of the reals over the integers beginning on p. 383 hides some details about the reals that are strictly speaking not needed but interesting to explain. It happens that not every two infinite sequences of digits represent different reals. For example, $.0999\cdots = .1000\cdots$, because the limits are the same and equal to $1/10$. In general, an infinite sequence of 9's starting in the ith place equals a 1 in the $(i-1)$st place followed by an infinite sequence of 0's. The situation in which one thing has more than one representation is not uncommon or inappropriate in any way. For example, among rationals, $1/4 = 2/8$.

As a result it is not sufficient to show that the constructed sequence of digits, r, is not identical as a sequence to every r_i; we must show that it differs by some positive amount from each r_i. Incidentally, the proof in the first printing is erroneous in that regard. Instead of building r from the digits 0 and 9, we should use 1 and 8. See the Errata section.

◊◊◊◊ Chapter 8. The Relational Data Model

◊ Motivation

Most of the chapters of the book have a clear relationship to subjects that form part of the traditional computer science curriculum. Chapter 8 introduces the student to a topic no less important — database systems. But this subject has not yet become integrated into the computer science curriculum as we believe it should, on a par with more traditional systems topics like compilers and operating systems. Thus, these remarks are provided not only to help the instructor motivate students to study the subject, but to motivate the instructor to give the subject suitable emphasis.

A generation ago, database system technology developed to handle some important, but rather restricted, large-scale problems such as bank data processing and airline reservation systems. There were a number of aspects to this technology. One was the development of data models, in the sense used in this book; of these, the relational model has become predominant and is the only one mentioned here. A second issue is the design of data structures for efficient access to very large amounts of data. This issue is covered in an introductory way, as we discuss indexes of various types. The most common data structure for indexes in database systems, the B-tree, is not covered because it requires discussion of a secondary-memory model of storage. However, it is similar in spirit to balanced binary trees.

Two other aspects of database systems are methods for sharing access to data and ways of assuring *resiliency*, the ability to preserve data in the face of failures. Neither of these topics are covered in the book.

In recent days, database technology has acquired increasing importance. For one, conventional database systems are now routinely used in many applications. Schools and businesses, whether large or small, use commercial database systems for record keeping. Personal filing systems also use the technology.

But more important is that database technology is found on the frontiers of our field. For example, many corporations find that they own information in many forms, and these forms are not exactly compatible. Database technology is needed to integrate information into a usable whole, and the problem of "heterogeneous databases" is an important topic for both research and commercial development.

For another example, satellites send down terabytes (10^{12} bytes) and soon petabytes (10^{15} bytes) of data per year. The data does not fit neatly in relations, but is raw signals. Biologists are trying to compile a database of information about the human genome, which will comprise many gigabytes of data. Scientists using these sorts of databases access them in ways that are far less predictable than the way airline reservation systems — say — are used. In general, coping with mammoth scientific databases is one of the important challenges computer science faces in the next decade. For these and other reasons, we encourage you to cover Chapter 8 in reasonable detail.

◇ An Airline Example

There are many other examples of relation schemes that could be used in a lecture. One is a simple airline reservation system with relation schemes

1. Flight-From-To-Dep-Arr. A typical tuple of this relation would be

   (123, SFO, JFK, 8:00, 16:30)

   meaning that flight number 123 departs SFO (San Francisco Intl. Airport) at 8AM and arrives at JFK (Kennedy Airport in New York) at 4:30PM (16.30, on the 24-hour clock).

2. Flight-Day-Name-Seat. A typical tuple of this relation would be

   (123, "May 15", "Pogo Possum", 12F)

   meaning that Mr. Pogo Possum has booked seat 12F on flight 123 for May 15.

There are several interesting aspects to this example. While the key for relation (1) is clearly Flight, the only reasonable keys for (2) are {Flight, Day, Name} and {Flight, Day, Seat}. Either is legal, but note that it is common to reserve a seat on a flight without getting a specific seat. Thus, we might expect to find tuples in which the Seat component is left blank, or *null*. The possibility of nulls, not discussed in the text, is a reason to avoid Seat in a key, since null values in the key complicate the primary index structure.

Several interesting queries to discuss are

a) Who is booked on flight 123 on May 15?

b) What flights are there from SFO to JFK? When do they leave?

c) Who is booked on a flight from SFO to JFK?

d) Construct the departure and arrival boards for San Francisco Intl. For example, the departure board has the flight number, the destination, and the departure time for all flights leaving SFO between two particular times (the present time and some time several hours in the future).

◊ A Movie Database

Consider relation schemes

1. Movie-Star. Typical tuples are

   (Casablanca, Bogart)
   (Casablanca, Bergman)

2. Movie-Year-Producer. A typical tuple is (Star Wars, 1977, Lucas).

Note that (1) has both attributes as the only possible key. Movie is a key for (2).

◊ Relations as Generalizations of Sets

Although the point is made in a sidebar on p. 412, we believe it is important to emphasize how Chapter 7 flows into Chapter 8. The relational model is a natural generalization of the dictionary, binary functions/relations, and so on.

◊ Proving Laws of Relational Algebra

The laws of relational algebra are more complex than those of the other algebras mentioned in the text, and we don't cover proofs of equivalence for these laws. However, the methodology is the same as that for sets developed on pp. 333–334, where we discuss showing containments in both directions. As an example, we might show that $\big(\sigma_C(R \bowtie S)\big) \equiv \big(\sigma_C(R) \bowtie S\big)$ whenever condition C involves only attributes of R. Here, we are assuming the natural join, but the proof doesn't make reference to the names of the attributes.

To begin, assume tuple t is in $\sigma_C(R \bowtie S)$.

◊ Then by the definition of selection, we know that t satisfies condition C, and t is in $R \bowtie S$.

◊ Since t is in $R \bowtie S$, there must be tuples r in R and s in S such that t agrees with r on the attributes of R, and t agrees with s on the attributes of S; the justification is the definition of the natural join.

◊ Since C involves only attributes of R, r agrees with t on those attributes, and t satisfies C, it follows that r satisfies C.

◊ Thus, r is in $\sigma_C(R)$.

◊ t is in $\big(\sigma_C(R)\big) \bowtie S$ because it agrees with r and s on their common attributes.

Of course we have to complete the proof by showing containment in the other direction. That is, if a tuple t is in $\big(\sigma_C(R)\big) \bowtie S$ then it is in $\sigma_C(R \bowtie S)$. The methodology is similar.

Chapter 9. The Graph Data Model

Motivation

A number of the algorithms discussed in this book solve significant problems. For example, finding connected components is an important step in "circuit extraction," the process whereby an integrated circuit design is analyzed. Such a design consists of many (perhaps millions) of rectangles and perhaps elements of other shapes. We can imagine a graph where each node is an element, typically a rectangle. Depending on the "layer" in which the element appears, it may be electrically connected to other elements that physically overlap. We can draw an edge between two elements that overlap and are on connected layers. The connected components of this graph are the circuit elements.

The minimum-weight spanning tree problem was considered by AT&T in the 1950's, because the federally mandated rule for pricing leased lines required it. That is, if a company wished to rent lines connecting a collection of cities, the charge was proportional to the weight of the minimum weight spanning tree in a graph formed from the cities as nodes, and all possible edges, weighted by the distance between the cities at the ends of the edge.

A curious property of this rule is that by adding a city, the purchaser could reduce the cost. For example, consider the four cities at the corners of a unit square, where the minimum-weight spanning tree has weight 3. If a city at the center of the square were added, the diagonals, with total weight $2\sqrt{2} = 2.828$, would be the MWST. Incidentally, this more general problem: given a graph in the plane and weights on edges equal to the physical distance between nodes, find the set of additional nodes that yields the minimum possible weight of all spanning trees, is called the *Steiner tree problem*. It is known to be NP-complete, so it is a good thing the government did not mandate solving that problem.

Dijkstra's algorithm is just what is needed for automatic route finding. There are automobiles now equipped with a screen that shows a map of your local area and accepts a starting point (the source) and finishing point. It will find and display the shortest route from the source to the destination.

Technically, Dijkstra's algorithm only gives the length of the shortest path, not the path itself. However, we can modify it simply so that whenever we update $dist(u)$ in line (11) of Fig. 9.47, we record the node v that is u's predecessor on the shortest path. Thus, we can trace back from any node w to the source and find the path from the source to w, in reverse.

Incidentally, it is worth mentioning that even though this problem only requires the shortest distance between two points, the best way known seems to be to search from one of the points (the source) and find the shortest distance to all points. However, if we only care about the distance to one particular point, we can stop when we settle that node.

◊ Equivalent Cycles

It is conventional to regard cycles as "the same" if they consist of the same nodes, with a different starting point. Exercises 9.2.7 and 9.2.8 deal with this notion, which should be explained in class. That is, the most convenient approach is to define an equivalence relation among cycles as in Exercise 9.2.8 and then ask the students to treat each equivalence class as one cycle when counting the number of cycles in a graph. If the graph is undirected, we may wish to define another equivalence relation that not only relates rotations of a cycle but relates a cycle with the same list of nodes in reverse order.

◊ The Accounting Trick in Depth-First Search Analysis

We believe it is important that students not miss the accounting trick on p. 473, which explains why depth-first search is an $O(m)$-time algorithm. Since the idea may be hard to grasp, a useful demonstration is to suggest a table in which the columns correspond to the calls to $dfs(v)$ for each of the nodes v. Imagine running the DFS algorithm, and as we examine each arc place a check mark in the column for the node at the tail of the arc. As time goes on, some of the columns get many checks, others few, depending on how many successors the node for that column has. However, each arc causes one check, and the total number of checks equals the number of arcs.

◊ NP-completeness

Although not mentioned in the book until p. 649, in connection with the tautology problem, there are lots of simple and interesting NP-complete graph problems, so this might be a good time to explore the topic. It is interesting to contrast the Euler circuit problem (see Exercise 6.5.5) where we are asked to find a cycle that touches each edge exactly once with the *Hamilton circuit problem*, in which we are asked to find a cycle that touches each node exactly once. The Euler problem has an $O(m)$ algorithm, while the Hamilton problem is NP-complete and therefore apparently has no better than exponential-time algorithm.

Similarly, one could discuss the *Traveling Salesman Problem* (find a minimum-weight Hamilton cycle, or *tour*, in a weighted, undirected graph). The greedy approach, which is successful for the minimum-weight spanning tree problem and yields an $O(m \log n)$ algorithm, does not always solve the TSP; the latter is NP-complete, of course.

Other interesting, NP-complete graph problems include *Node Cover* (find a smallest set of nodes such that each edge has at least one end in the selected set), the *Clique problem* (find a maximal clique in an undirected graph), and the *Feedback Arc problem* (find a smallest set of arcs whose removal makes a graph acyclic). Of course, each of these problems, and the TSP as well, must technically be stated in yes/no form, e.g., "is there a set of at most k arcs whose removal makes graph G acyclic?"

◇ Precise Definitions

There is a tendency for students to see definitions as more stringent than they are. As an example, the notion of "k-colorable" says that the graph can be colored with k or fewer colors. Yet many will say that K_3 is not 4-colorable, because clearly you cannot use more than 3 colors. It helps to contrast the notions of "k-colorable" and "chromatic number equals k."

The problem is not limited to computer science classes. Al Aho remembers visiting the elementary school classroom of one of his daughters and finding that the teacher firmly believed that a square is *not* a rectangle.

◇ Some Challenges to the Student

There is a family of problems that will test the deep understanding students have of many graph algorithms and concepts. Don't be surprised if they have trouble coming up with the answers. Ask what modifications occur to an algorithm when we add a single arc or edge to a graph G. For example, suppose we have constructed a depth-first spanning forest F for G, and we add the arc $u \to v$ between two of G's nodes. When must F change? When might it change? The answer in this case is that F is sure to change if $u \to v$ is a left-to-right cross arc (the kind that could not exist in F. It can never change if $u \to v$ is a backward arc or a cross arc (right-to-left). If $u \to v$ is a forward arc, then F need not change, provided v is last on u's list of successors, but F will surely change if v is first on that list. Notice how this analysis is relatively simple, but it requires the student to think carefully about what happens when you follow an arc $u \to v$ during a depth-first search.

A similar question can be asked about some other algorithms:

1. If we add an edge $\{u, v\}$ with weight w to an undirected graph G with minimum-weight spanning tree T, under what conditions will T be a MWST of the new graph?
2. If we add an arc $u \to v$ with a weight of 0 to a graph G, under what conditions will the order in which nodes are settled during Dijkstra's algorithm change?

◇ Examples of Topological Sort Usage

There are a number of interesting examples or exercises that use topological sort to advantage. For example, suppose we are given an acyclic directed graph, with weights on the arcs, and we want to know the length of the longest path from each node to anywhere. If we find a topological order, and then visit the nodes from the back, when we visit node u, we shall have the maximum path length from each of u's successors. We need only maximize over all successors v of u the sum of the weight of arc $u \to v$ plus the (stored) maximum path length emanating from v. This algorithm can be implemented to run in $O(m)$ time. As in the book, m is the larger of the number of nodes and arcs.

A similar application is single-source shortest path on an acyclic graph. Now, we visit the nodes u in the topological order, and minimize over the predecessors of u; the idea is close to that of the paragraph above. This algorithm takes $O(m)$ time, compared with $O(m \log n)$ for Dijkstra's algorithm, but of course it can only be used in acyclic graphs.

A third, related application is to find, for each node u in an acyclic graph, the longest path containing u. Now, we need to find the longest path emanating from u, and the longest path ending in u, and add them. Again, $O(m)$ time suffices if the graph is acyclic.

◊◊◊◊ Chapter 10. Patterns, Automata, and Regular Expressions

◊ The Two Principal Uses of Automata

More than any other point, students miss the distinction outlined in the box on p. 516, between what strings an automaton accepts and what strings are affected by that acceptance. The book has a number of examples where we accept when we see a specific string, like `man`, or one of a set of specific strings. They are concerned that when processing a string like `command`, it is `comman` that is accepted by the automaton, while `command` is formally rejected, and yet the system as a whole is said to accept `command`. We see no hope for correcting this misconception except to make sure it is covered in class, probably more than once.

◊ Understanding Nondeterministic Automata

Students are often puzzled in a second way by nondeterministic automata that accept the strings ending in a particular sequence like `man`. They look at the accepting state in Fig. 10.11, for example, and see that the automaton appears to "die" in the accepting state. They may believe that this automaton could not accept `manman`, for instance.

Of course it *does* accept `manman`, but it does so in a way that bothers some students. The path showing acceptance of `manman`, that is,

$$0 \to 0 \to 0 \to 0 \to 1 \to 2 \to 3$$

is not an extension of the path leading to acceptance of `man`. There is no basis for this expectation on the part of students, but we found a significant number of misconceptions of this type.

◊ Structural Inductions About Regular Expressions

There is an opportunity to do a number of constructions involving regular expressions by structural induction on the expression tree. Take the trouble to emphasize that we can draw expression trees for RE's. Then, for example, the construction on pages 554–555 can be presented as a structural induction on the expression tree, where Fig. 10.28 handles the case of a leaf, and Fig. 10.29 handles interior nodes.

◇ Important States

In the construction of p. 559, removing ϵ-transitions, it pays to define formally the term *important state*. A state is "important" if it is either the start state or a state with an entering transition on a non-ϵ symbol. The important states are the ones that will appear in the resulting automaton. This notion is used on p. 559, but not defined as a term, and it seems to help if students are given a name for the idea.

Chapter 11. Recursive Description of Patterns

◇ Limits of Languages

The inductive definition of the language for a syntactic category, given on p. 579, actually defines an infinite sequence of languages, $L_1, L_2, \ldots$, where L_i is the set of strings discovered on or before the ith round. The desired language L is related to the L_i's in a way that may confuse some students. Formally, L is the limit as $i \to \infty$ of L_i.

Remember that as defined, each L_i is contained in L_{i+1}, so the limit grows without bound as i grows. That is not what people are used to, because we expect that if a sequence approaches a limit, then the difference between the terms and the limit gets progressively smaller. In this case, no L_i is close to L (except in cases where the syntactic category really does not involve recursion).

Nevertheless, the limit does make sense; it is the set of strings that is in any of the L_i's. Put another way, it is the infinite union $L_1 \cup L_2 \cup \cdots$. The concept of an infinite union may also bother some students, since formally it is not an expression of set algebra (but we never said it was), and also, it has the strange, but essential property that if string x is in some L_i, then it is also in the infinite number of other terms, namely L_j, for $j = 1+1, i+2, \ldots$.

◇ A Useful Example

The following grammar defines (via syntactic category $<E>$) the set of all strings of 0's and 1's with an equal number of each.

$$<E> \to 0<P> \mid 1<M> \mid \epsilon$$
$$<P> \to 0<P><P> \mid 1$$
$$<M> \to 1<M><M> \mid 0$$

There are some instructive inductive proofs involved in showing this grammar works. First, $<P>$ stands for a "plus" string — a string with one more 1 than 0, such that each proper prefix does not have that property; i.e., all proper prefixes have at least as many 0's as 1's. The proof in one direction is a structural induction on parse trees that have $<P>$ at the root, and in the other direction it is a complete induction on the length of a string with the "plus" property. The ideas borrow from the discussion of balanced parenthesis strings in Section 2.6.

$<M>$ similarly represents a "minus" string — a string with one more 0 than 1, but with no proper prefix with that property. Finally, $<E>$'s strings are constructed from any number of pairs consisting of a 0 followed by a "plus" string (which gets us back to balance between 0's and 1's for the next time) or a 1 followed by a "minus" string (which does likewise). It is necessary to show that every balanced string can be broken up in this way.

◊ Bullseye Pictures

It helps when discussing the containment of the regular sets (languages defined by regular expressions) and the context-free languages (languages defined by what we here call "grammars") to draw a "bullseye" picture or Venn diagram. Here, the student must understand that the points in the plane represent languages, that is, sets of strings. The inner circle represents languages with regular expressions, and the next ring represents languages with grammars but without regular expressions; $\{0^n1^n \mid n \geq 1\}$ is a typical example. The outer area represents languages that do not have grammars. Appropriate examples are $\{0^n1^n2^n \mid n \geq 1\}$ or $\{0^p \mid p \text{ is prime}\}$.

The diagram also presents the opportunity to mention what else is "out there." For example, we can explain that there are languages that do not have algorithms to tell whether a string is in them, and the *recursive* languages, those that do, form a third ring, since every language with a grammar has an algorithm. Even that point is not obvious, and it is necessary to point out how the definition of the language "by rounds" suggested in Section 11.3 leads to an algorithm for the language of any given grammar.

◊ Proofs that a Language has no Automaton

The form of a proof that a language has no deterministic automaton is tricky, as are almost all proofs of negative results. The student may recall the difficulty of proving that a big-oh relationship fails to hold, and the form of such proofs can be compared with the similar form for "no-automaton" proofs. Here is a template to give the students.

1. Let L be the language we want to prove lacks an automaton. Suppose A were a deterministic automaton whose language is exactly L. Then A has some particular number of states, and let m stand for this number. It is important to remember that m is a fixed, finite number, even though we do not know exactly what it is.

2. As a function of m, select some word x known to be in L. For instance, on p. 614 we selected $x = 0^m1^m$.

3. Consider the sequence of states entered by A traveling from the start state to an accepting state along a path labeled x. Argue that there must be two identical states among some sequence of $m+1$ states (which means that x must be chosen to be length at least m).

4. "Splice out" the portion of x between the two occurrences of the same state. That step gives us some other word y that is also accepted by A.

5. If we have chosen wisely, we can claim that, no matter what y is, it cannot be in the language L.
6. We may then conclude that no possible deterministic automaton A can have L as its language. By the equivalences proved in Chapter 10, neither can any nondeterministic automaton or any regular expression have L as its language.

◇ FIRST and FOLLOW

For the class to understand the general rules whereby we fill out a parsing table, the instructor needs to define formally the functions FIRST and FOLLOW. FIRST($<A>$), for syntactic category $<A>$, is the set of terminal symbols that can be the first in a string in $L(<A>)$. As a special case, if ϵ is in $L(<A>)$, then `ENDM` is also in FIRST($<A>$).

We can calculate FIRST for the various syntactic categories by creating a graph with nodes corresponding to the syntactic categories and terminals. There is an arc from $<B>$ to X if there is some production with head $<B>$ and a body in which X is either the first symbol, or the only symbols appearing to the left of X in the body are syntactic categories that all have ϵ in their languages.

Now, we can compute FIRST by an application of transitive closure. We ask, for each syntactic category $<A>$ and terminal b, whether there is a path from the node for $<A>$ to the node for b. If so, b is in FIRST($<A>$), and if not, then not.

Incidentally, finding the syntactic categories that include ϵ in their language (these are called *nullating* syntactic categories) is an interesting example of an inductive definition. For the basis, $<A>$ is nullating if there is a production $<A> \rightarrow \epsilon$. If the induction, $<A>$ is nullating if there is a production whose head is $<A>$ and such that all symbols in the body are syntactic categories already known to be nullating.

The computation of FOLLOW($<A>$), for the syntactic categories $<A>$, is now easy. We look at each body. If $<A>$ appears immediately to the left of a terminal b, then b is in FOLLOW($<A>$). If $<A>$ appears immediately to the left of a syntactic category $<B>$ (which could be $<A>$), then every terminal in FIRST($<B>$) is in FOLLOW($<A>$). Additionally, if $<A>$ is the "start" syntactic category, the one whose language is being parsed, then `ENDM` is in FOLLOW($<A>$).

The above gives us an initial FOLLOW set for each syntactic category. Now, we need to create a graph with nodes for the syntactic categories. If there is a production with head $<A>$ and a body whose last symbol is the syntactic category $<B>$, then put an arc from $<A>$ to $<B>$. Now, FOLLOW($<A>$) is the union of the initial FOLLOW sets for all those nodes (including $<A>$, of course) that $<A>$ can reach along some path. Thus, FOLLOW also requires an application of transitive closure in its calculation.

◇ Top-Down Versus Bottom-Up Construction of Parse Trees

The definition of parse trees is "bottom-up," in the sense that we start with (one-node) trees for the leaves and proceed to build progressively higher trees. On the other hand, the parsers discussed in Sections 11.6 and 11.7 work "top-down"; they start with the root and fill in the nodes working downwards, using the input for clues. Although we felt the subject too complicated to discuss in this book, it should be noted that the parsers generated by the UNIX LR-parser generator YACC work bottom-up. They start with the leaves and piece them together into progressively larger trees, using the input for clues.

◇ When a String is Rejected by a Parser

There was much surprise when we assigned an exercise from the book where students had to simulate a parser on a string that is not in the language. They tended to interpret reaching a condition where the parse table is blank as a mistake on their part. Perhaps it is a mistake on our part that the examples show only the parser's action on strings that are in the language, but it is clear that the student needs to be warned that sometimes a parser fails to complete, and that this is a "feature rather than a bug."

Chapter 12. Propositional Logic

◇ An Induction Involving DeMorgan's Laws

DeMorgan's laws, applied in the "normal" direction, where the `NOT` moves down the tree, leads to an expression that is the `AND` and `OR` of literals, assuming the original expression has only the operators `AND`, `OR`, and `NOT`. A proof of this fact is a bit tricky and forms a hard exercise or an instructive class example. The trick is to do a complete induction on the sum over all nodes in the expression tree labeled with `NOT`, of 3^h, where h is the height of the node in question. Observe that applying DeMorgan's laws to push `NOT` down always reduces the sum, even though one `NOT` is replaced by two.

◇ Substitution Principles

Students may wonder why we did not state the substitution principle for other algebras, like the algebra of regular expressions. In those algebras, equalities among expressions [e.g., $(RS) \equiv (SR)$] are analogous to tautologies, since they are true for whatever expressions we substitute for the variables. However, we do not need a substitution principle explicitly in those algebras, because we write the expressions with variables that are understood to stand for any expression of the appropriate type. In contrast, the propositional variables are both placeholders for arbitrary expressions and the operands themselves.

◊ Applications of Algebraic Simplifications

The primary practical application of simplification of propositional logic expressions is surely the design of circuits, as discussed in Chapter 13. Another interesting application is the simplification of complex decision trees. In a sense, this is a larger version of the problem of simplifying a conditional with which we begin Chapter 12.

In a more complex decision problem we might be faced with dozens of decisions, each of which can be represented by a propositional variable. For instance, we might want to decide which of several suppliers to use for an order. The decision might be based on such variables as whether we are placing a large or small order, how quickly we need delivery, relative prices, relative exchange rates between dollars, yen, marks, and so on. An expression for when to order from company A might have thousands or millions of terms if written as a sum of products of literals. It is essential that a more succinct test for the same condition be found.

◊ Sorting Out Proof Methodologies

The two kinds of formal proofs discussed in this chapter are

1. Conversion of expressions into equivalent expressions by substitution of equals for equals and use of the substitution principle, and

2. Deduction.

(1) leads to the proof of tautologies of the form $E \equiv F$, while (2) proves tautologies of the form E_1 `AND` $\cdots$ `AND` $E_n \rightarrow F$. Another useful comparison is that (1) is like the proofs of trigonometric identities, and (2) is like the proofs in geometry, both of which your student should have seen in high school.

Techniques like "proof by contrapositive" are not another methodology in that sense. Rather, they are "subgoals" in a deductive proof, which can guide a few steps, where you prove `NOT` $E \rightarrow$ `NOT` F for some expressions E and F, and then apply the contrapositive tautology to deduce $F \rightarrow E$.

Also, it is important to make clear that entire "contrapositive" proofs can be formalized in a suitable logic, but we more commonly do proofs that are *not* formal deductions. Applying the contrapositive is still an important tool in an informal proof.

◊ Ground Rules for Deductive Proofs

Since the use of any tautology whatsoever makes deduction trivial, the instructor needs to develop a policy as to what tautologies may be used on homeworks and exams. The rule of limiting students to (substituted instances of) the laws enumerated in Sections 12.8 and 12.9 of the text is too stringent in some cases. We recommend that students be allowed to use these laws modified by the substitution of equals for equals in some cases.

Chapter 13. Using Logic to Design Components

Second-String Commandos

Here is an interesting exercise or classroom discussion. We wish to send out our commando team only when each of the n members of the team are ready, willing, and able. We could represent this problem by $3n$ inputs, $r_1, r_2, \ldots, r_n$, $w_1, w_2, \ldots, w_n$, and $a_1, a_2, \ldots, a_n$, where r_i indicates whether the ith commando is ready to go on the mission, w_i indicates whether he or she is willing to go on the mission, and a_i indicates whether he or she is able to complete the mission. The output says whether or not the team should be sent on the mission. This problem is trivial; the output is just the **AND** of the $3n$ inputs.

A less trivial problem involves the second-string commando team. They're not so good, so we're happy if each commando is *either* ready, willing, or able. Even that might be too much to expect, so we're satisfied if at most one commando is *neither* ready, willing, nor able. The problem is to design a circuit that tells whether the second-string team meets these conditions and should be sent.

The solution works by divide-and-conquer, but we must have as output two bits, which we might call z (zero commandos are neither ready, willing, nor able, and o (at most one commando is neither ready, willing, nor able. For the basis, o is always true, and $z = r_i + w_i + a_i$. For the induction, we use

$$z = z_L z_R$$
$$o = z_L o_R + o_L z_R$$

where z_L represents the z bit for the left side, and so on. By these rules we can construct the circuit for $2n$ commandos from two copies of the circuit for n commandos.

Definition of Carry-Generate

In the text, we define the carry-generate output g for the addition of two n-bit numbers to mean that there is a carry out of the leftmost place regardless of whether the carry-in at the right end is 0 or 1. An equivalent definition is that $g = 1$ if there is a carry-out at the left when the carry-in at the right is 0. It may be "obvious" that if we get a carry-out when the carry-in is 0, then we surely get a carry-out with a carry-in of 1. However, this statement forms a good inductive proof for the class to work out.

Adding to the Inductive Hypothesis

Chapter 13 has a number of examples where you need to do an induction, in this case design a circuit via divide-and-conquer, but in order to make the induction go through, you need to put into the inductive hypothesis more than you really need. The "second string commandos" mentioned above is an example; we really only need the output o (really its negation), but we have to compute z as well in order to have a hope of computing o recursively. Similarly, the addition circuit of Section

13.6 computes more than we really need, as does the majority circuit of Exercise 13.7.5.

There are many examples in mathematics where it is necessary to augment the inductive hypothesis with statements that are not part of the theorem but are essential to make the inductive proof go through. We have not seen many in this book, although the divide-and-conquer definition of the Fibonacci numbers mentioned in the notes for Chapter 2 is a good example. There, we must produce a matrix rather than a single number to compute the nth Fibonacci number in time $O(\log n)$.

Chapter 14. Predicate Logic

Predicates are Formal Symbols

It is important to get across the idea that a predicate is a formal symbol, devoid of inherent meaning. For example, $path(X, Y)$ may talk about graphs, but there is nothing in the use of "path" that tells us this is the case.

As in many issues of logic, the analogy between logic and programming may be helpful. In this case, the student's attention can be drawn to a function `path(X,Y)` that it is claimed returns `TRUE` if and only if there is a path from node X to node Y in a certain graph. However, you don't know that to be the case unless you carefully examine the code and determine that the function `path` works as intended. In the case of the predicate $path$, the analogy to the code is the set of expressions involving predicate $path$ that we assert to be true. These expressions may or may be not to imply the desired facts about $path$.

An Example Illustrating the Significance of Expressions

Here is another example of how asserted expressions can be used to infer facts from a given database of facts; it is analogous to Example 14.3 (p. 715). Suppose we assert the "facts"

$$parent(1, 2)$$
$$parent(2, 3)$$

and the rule

$$\big(parent(X, Y) \text{ AND } parent(Y, Z)\big) \rightarrow grandparent(X, Z)$$

We would expect to conclude $grandparent(1, 3)$.

Technically, we assert the above rule; it is not a tautology. Another way to look at the situation is that we prove

$$\Big(parent(1, 2) \text{ AND } parent(2, 3) \text{ AND } \big(parent(X, Y) \text{ AND } parent(Y, Z) \rightarrow grandparent(X, Z)\big)\Big) \rightarrow grandparent(1, 3)$$

is a tautology. In general, if we can conclude a fact like $grandparent(1, 3)$ from given facts and rules, then the expression that has the AND of the facts and rules implying the conclusion is a tautology.