10. Predicate Logic

10.1 Introduction

Predicate logic builds heavily upon the ideas of proposition logic to provide a more powerful system for expression and reasoning. As we have already mentioned, a **predicate** is just a function with a range of two values, say `false` and `true`. We already use predicates routinely in programming, e.g. in conditional statements of the form

```
        if( p(...args ...) )
```

Here we are using the two possibilities for the return value of p, (`true` or `false`). We also use the propositional operators to combine predicates, such as in:

```
        if( p(....) && ( !q(....) || r(....) ) )
```

Predicate logic deals with the combination of predicates using the propositional operators we have already studied. It also adds one more interesting element, the "quantifiers".

The meaning of predicate logic expressions is suggested by the following:

Expression + Interpretation + Assignment = Truth Value

Now we explain this equation.

An **interpretation** for a predicate logic expression consists of:

a domain for each variable in the expression

a predicate for each predicate symbol in the expression

a function for each function symbol in the expression

Note that the propositional operators are not counted as function symbols in the case of predicate logic, even though they represent functions. The reason for this is that we do not wish to subject them to interpretations other than the usual propositional interpretation. Also, we have already said that predicates are a type of function. However, we distinguish them in predicate logic so as to separate predicates, which have truth values used by propositional operators, from functions that operate on arbitrary domains. Furthermore, as with proposition logic, the **stand-alone convention** applies with predicates: We do not usually explicitly indicate == 1 when a predicate expression is true; rather we just write the predicate along with its arguments, standing alone.

An **assignment** for a predicate logic expression consists of:

a value for each variable in the expression

Given an assignment, a truth value is obtained for the entire expression in the natural way.

Example

Consider the expression:

```
x < y || ( y < z  && z < x)
  ^          ^         ^         predicate symbols
```

Here || and && are propositional operators and < is a predicate symbol (in infix notation). An assignment is a particular predicate, say the *less_than* predicate on natural numbers, and values for x, y, and z, say 3, 1, and 2. With respect to this assignment then, the value is that of

```
3 < 1 || ( 1 < 2  && 2 < 3)
```

which is

```
false || ( true && true)
```

i.e.

```
true.
```

With respect to the same assignment for <, but 3, 2, 1 for x, y, z, the value would be that of

```
3 < 2 || ( 2 < 1  && 1 < 3)
```

which would be `false`. As long as we have assigned meanings to all variables and predicates in the expression, we can derive a `false` or `true` value. Now we give an example where function symbols, as well as predicate symbols, are present.

```
( (u + v) < y ) || ( (y < (v + w)) && v < x)
     ^                        ^                  function symbols
```

would be an example of an expression with both function and predicate symbols. If we assign + and < their usual meanings and u, v, w, x, y the values 1, 2, 3, 4, 5 respectively, this would evaluate to the value of

```
( (1 + 2) < 4 ) || ( (4 < (2 + 3)) && 2 < 4 )
```

which is, of course, `true`.

Validity

It is common to be concerned with a fixed interpretation (of domains, predicates, and functions) and allow the assignment to vary over individuals in a domain. If a formula evaluates to true for all assignments, it is called **valid with respect to** the interpretation. If a formula is valid with respect to every interpretation, it is called **valid**. A special case of validity is where sub-expressions are substituted for proposition symbols in a tautology. These are also called *tautologies*. However, not every valid formula is a tautology, as is easily seen when we introduce quantifiers later on.

10.2 A Database Application

An important use of predicate logic is found in computer databases and the more general notion of "knowledge base", defined to be a database plus various computation rules. In this application, it is common to use predicate expressions containing variables as above as "queries". The predicates themselves represent the underlying stored data, computable predicates, or combinations thereof. A query asks the system to find all individuals corresponding to the variables in the expression such that the expression is **satisfied** (evaluates to 1). Next we demonstrate the idea of querying a database using the Prolog language as an example. Prolog is not the most widely-used database query language; a language known as SQL (Structured Query Logic) probably has that distinction. But Prolog is one of the more natural to use in that it is an integrated query language and programming language.

Prolog Database Example

There are many ways to represent the predicates in a database, such as by structured files representing tables, spreadsheet subsections, etc. In the language **Prolog**, one of the ways to represent a predicate is just by **enumerating** all combinations of values for which the predicate is true. Let us define the predicates *mother* and *father* in this fashion. These predicates provide a way of modeling the family "tree" on the right.

```
mother(alice, tom).
mother(alice, carol).
mother(carol, george).
mother(carol, heather).
mother(susan, hank).

father(john, tom).
father(john, carol).
father(fred, george).
father(fred, heather).
father(george, hank).
```

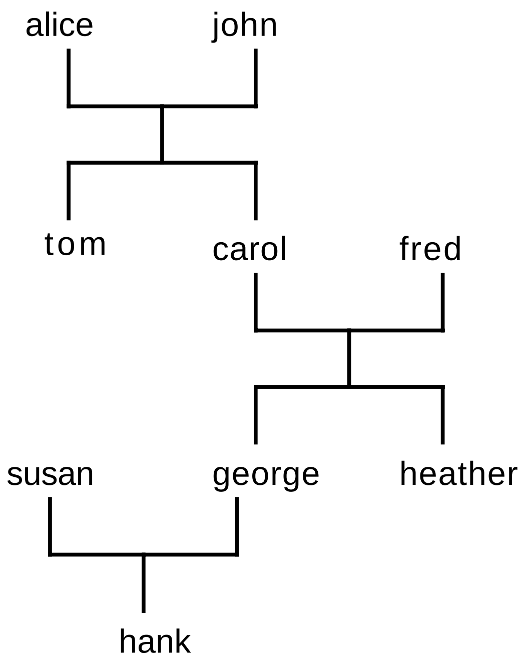

Figure 163: A family "tree" modeled as two predicates, `mother` and `father`.

It is possible for a query to contain no variables, in which case we would expect an answer of 1 or 0. For example,

```
mother(susan, hank)      ⇒ true

mother(susan, tom)       ⇒ false
```

More interestingly, when we put variables in the queries, we expect to get values for those variables that satisfy the predicate:

mother(alice, X) ⇒ X = tom; X = carol (two *alternatives* for X)

father(tom, X) ⇒ false (no such X exists)

mother(X, Y) ⇒ (several *alternative combinations* for X, Y)

```
      X = alice, Y = tom;
      X = alice, Y = carol;
      X = carol, Y = george;
      X = carol, Y = heather;
      X = susan, Y = hank
```

Note that the `X` and `Y` values must be in correspondence. It would not do to simply provide the set of `X` and the set of `Y` separately.

Defining *grandmother* using Prolog

The Prolog language allows us to present queries and have them answered automatically in a style similar to the above. Moreover, Prolog allows us to define new predicates using logic rather than enumeration.

Such a predicate is defined by the following logical expression:

```
grandmother(X, Y) :- mother(X, Z), parent(Z, Y).
```

Here :- is read as "if" and the comma separating *mother* and *parent* is read as "and". This says, in effect, "X is the grandmother of Y if X is the mother of (some) Z and Z is the parent of Y". We have yet to define *parent*, but let's do this now:

```
parent(X, Y) :- mother(X, Y).
parent(X, Y) :- father(X, Y).
```

Here we have two separate logical assertions, one saying that "X is the parent of Y if X is the mother of Y", and the other saying a similar thing for father. These assertions are not contradictory, for the connective :- is "if", not "if and only if". However, the collection of all assertions with a given predicate symbol on the *lhs* exhausts the possibilities for that predicate. Thus, the two rules above together could be taken as equivalent to:

`parent(X, Y)` iff (`mother(X, Y)` or `father(X, Y)`)

Given these definitions in addition to the database, we now have a "knowledge base" since we have rules as well as enumerations. We can query the defined predicates in the same way we queried the enumerated ones. For example:

```
grandmother(alice, Y)   ⇒   Y = george; Y = heather

grandmother(X, Y)       ⇒   X = alice, Y = george;
                            X = alice, Y = heather
                            X = carol, Y = hank

grandmother(susan, Y)   ⇒   false
```

Quantifiers

Quantifiers are used in predicate logic to represent statements that range over **sets** or "domains" of individuals. They can be thought of as providing a very concise notation for what might otherwise be large conjunctive ($\wedge$) expressions and disjunctive ($\vee$) expressions.

Universal Quantifier $\forall$ ("for all", "for every")
$(\forall x) P(x)$ means **for every x** in the domain of discourse P(x) is true.

Existential Quantifier $\exists$ ("for some", "there exists")
$(\exists x) P(x)$ means **for some x** in the domain of discourse P(x) is true.

If the domain can be enumerated $\{d_0, d_1, d_2, ...\}$ (and this isn't always possible) then the following are suggestive

$$(\forall x)\, P(x) \equiv (P(d_0) \wedge P(d_1) \wedge P(d_2) \wedge ...)$$

$$(\exists x)\, P(x) \equiv (P(d_0) \vee P(d_1) \vee P(d_2) \vee ...)$$

This allows us to reason about formulas such as the of **DeMorgan's laws for Quantifiers**:

$$\neg (\forall x)\, P(x) \equiv (\exists x)\, \neg P(x)$$
$$\neg (\exists x)\, P(x) \equiv (\forall x)\, \neg P(x)$$

The definition of validity with respect to an interpretation, and thus general validity, is easily extended to formulas with quantifiers. For example, in the natural number interpretation, where the domain is {0, 1, 2, …} and > has its usual meaning, we have the following:

Formula	**Meaning**	**Validity**
$(\exists x)\ x > 0$	There is an element larger than 0.	valid
$(\forall x)\ x > x$	Every element is larger than itself.	invalid
$(\forall x)(\exists y)\ x > y$	Every element is larger than some element.	invalid
$(\forall x)(\exists y)\ y > x$	Every element has a larger element.	valid
$(\exists x)(\forall y)\ (y \mathrel{!=} x) \rightarrow x > y$	There is an element larger than every other.	invalid

Exercises

With respect to the interpretation in which:
The domain is the natural numbers
== is equality
> is greater_than
- is proper subtraction

which of the following are valid:

1 •• $x < y \rightarrow (x - z) < (y - z)$

2 •• $x < y \lor y < x$

3 •• $(x < y \lor x == y) \land (y < x \lor x == y) \rightarrow (x == y)$

Assuming that `distinct` are predicates such that `distinct(X, Y)` is true when the arguments are different, express rules with respect to the preceding Prolog database that define:

4 •• `sibling(X, Y)` means `X` and `Y` are siblings (different people having the same parents)

5 •• `cousin(X, Y)` means that `X` and `Y` are children of siblings

6 ••• `uncle(X, Y)` means that `X` is the sibling of a `Z` such that `Z` is the parent of `Y`, or that `X` is the spouse of such a `Z`.

7 ••• `brother_in_law(X, Y)` means that `X` is the brother of the spouse of `Y`, or that `X` is the husband of a sibling of `Y`, or that `X` is the husband of a sibling of the spouse of `Y`.

Which of the following are valid for the natural numbers interpretation?

8 • $(\exists x)\ (x \,!\!= x)$

9 • $(\forall y)(\exists x)\ (x \,!\!= y)$

10 • $(\forall y)(\exists x)\ (x = y)$

11 •• $(\forall y)(\forall x)\ (x = y) \lor (x > y) \lor (x < y)$

12 •• $(\forall y)\ [\ (y = 0) \lor (\exists x)\ (x < y)]$

13 •• $(\forall y)\ [\ (y = 0) \lor (\exists x)\ (x > y)]$

Bounded Quantifiers

Two variants on quantifiers are often used because they conform to conversational usage. It is common to find statements such as

"For every x such that ...., P(x)."

For example,

"For every even x > 2, not_prime(x)."

Here the .... represents a condition on x. The added condition is an example of a "bounded quantifier", for it restricts the x values being considered to those for which .... is true. However, we can put .... into the form of a predicate and reduce the bounded quantifier case to an ordinary quantifier. Let Q(x) be the condition "packaged" as a predicate. Then

"For every x such that Q(x), P(x)."

is equivalent to

$(\forall x)\ [Q(x) \rightarrow P(x)]$

Similarly, existential quantifiers can also be bounded.

"For some x such that Q(x), P(x)."

is equivalent to

$(\exists x)\ [Q(x) \wedge P(x)]$

Note that the bounded existential quantifier translates to an "and", whereas the bounded universal quantifier translates to an "implies".

Quantifiers and Prolog

Prolog does not allow us to deal with quantifiers in a fully general way, and quantifiers are never explicit in prolog. Variables that appear on the lefthand side of a Prolog rule (i.e. to the left of :-) are implicitly quantified with $\forall$. Variables that appear only on the righthand side of a rule are quantified as around the righthand side itself. For example, above we gave the definition of grandmother:

```
grandmother(X, Y) :- mother(X, Z), parent(Z, Y).
```

With explicit quantification, this would appear as:

$(\forall x)(\forall y)$ [grandmother(X, Y) **if** $(\exists Z)$ mother(X, Z) and parent(Z, Y)]

The reason that this interpretation is used is that it is fairly natural to conceptualize and that it corresponds to the *procedural interpretation* of Prolog rules.

Logic vs. Procedures

Although Prolog mimics a subset of predicate logic, the real semantics of Prolog have a **procedural** basis. That is, it is possible to interpret the logical assertions in Prolog as if they were a kind of generalized procedure call. This duality means that Prolog can be used as both a procedural language (based on actions) and as a declarative language (based on declarations or assertions). Here we briefly state how this works, and in the process will introduce an important notion, that of *backtracking*.

To a first approximation, a Prolog rule is like an ordinary procedure: The lefthand side is like the header of a procedure and the righthand side like the body. Consider, then, the rule

```
grandmother(X, Y) :- mother(X, Z), parent(Z, Y).
```

Suppose we make the "call" (query)

```
grandmother(alice, Y)
```

Satisfying this predicate becomes the initial "goal". In this case, the call matches the *lhs* of the rule. The body is detached and becomes

```
mother(alice, Z), parent(Z, Y)
```

This goal is read: "Find a `Z` such that `mother(alice, Z)`. If successful, using that value of `Z`, find a `Y` such that `parent(Z, Y)`. If that is successful, `Y` is the result of the original query."

We can indeed find a `Z` such that `mother(alice, Z)`. The first possibility in the definition of *mother* is that `Z = tom`. So our new goal becomes `parent(tom, Y)`. We then aim to solve this goal. There are two rules making up the "procedure" for `parent`:

```
parent(X, Y) :- mother(X, Y).
parent(X, Y) :- father(X, Y).
```

Each rule is tried in turn. The first rule gives a body of `mother(tom, Y)`. This goal will fail, since `mother` is an enumerated predicate and there is no `Y` of this form. The second rule gives a body of `father(tom, Y)`. This goal also fails for the same reason. There being no other rules for parent, the goal `parent(tom, Y)` fails, and that causes the body

```
mother(alice, Z), parent(Z, Y)
```

to *fail for the case* `Z = tom`. Fortunately there are other possibilities for `Z`. The next rule for mother indicates that `Z = carol` also satisfies `mother(alice, Z)`. So then we set off to solve

```
parent(carol, Y).
```

Again, there are two rules for parent. The first rule gives us a new goal of

```
mother(carol, Y)
```

This time, however, there is a `Y` that works, namely `Y = george`. Now the original goal has been solved and the solution `Y = george` is returned.

10.3 Backtracking Principle

The trying of alternative rules when one rule fails is called **backtracking**. Backtracking also works to find multiple solutions if we desire. We need only pretend that the solution previously found was not a solution and backtracking will pick up where it left off in the search process. Had we continued in this way, Y = heather would also have produced as a solution. The arrows below suggest the path of backtracking in the procedural interpretation of Prolog. One can note that the backtracking paradigm is strongly related to *recursive descent* and *depth-first search*, which we will have further occasion to discuss.

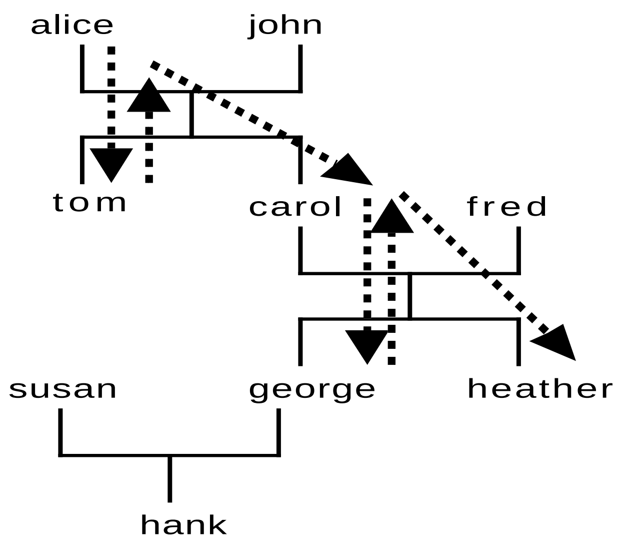

Figure 164: Backtracking in Prolog procedures

Recursive Logic

We close this section by illustrating a further powerful aspect of Prolog: rules can be recursive. This means that we can combine the notion of backtracking with recursion to achieve a resulting language that is strictly more expressive than a recursive functional language. At the same time, recursive rules retain a natural reading in the same way that recursive functions do.

Earlier we gave a rule for `grandmother`. Suppose we want to give a rule for `ancestor`, where we agree to count a parent as an ancestor. A primitive attempt of what we want to accomplish is illustrated by the following set of clauses:

```
ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(X, Z1), parent(Z1, Y).
ancestor(X, Y) :- parent(X, Z1), parent(Z1, Z2), parent(Z2, Y).
...
```

The only problem is that this set of rules is infinite. If we are going to make a program, we had better stick to a finite set. This can be accomplished if we can use `ancestor` recursively:

```
ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
```

This pair of rules provides two ways for `X` to be an ancestor of `Y`. But since one of them is recursive, an arbitrary chain of parents can be represented. In the preceding knowledge, all of the following are true:

```
parent(alice, carol), parent(carol, george), parent(george, hank)
```

It follows logically that

```
ancestor(george, hank)
ancestor(carol, hank)
ancestor(alice, hank)
```

so that a query of the form `ancestor(alice, Y)` would have `Y = hank` as one of its answers.

Using Backtracking to Solve Problems

In chapter Compute by the Rules, we gave an example of a program that "solved" a puzzle, the Towers of Hanoi. Actually, it might be more correct to say that the *programmer* solved the puzzle, since the program was totally deterministic, simply playing out a pre-planned solution strategy. We can use backtracking for problems that are not so simple to solve, and relieve the programmer of some of the solution effort. Although it is perhaps still correct to say that the programmer is providing a strategy, it is not as clear that the strategy will work, or how many steps will be required.

Consider the water jugs puzzle presented earlier. Let us use Prolog to give some logical rules for the legal moves in the puzzle, then embed those rules into a solution mechanism that relies on backtracking. For simplicity, we will adhere to the version of the puzzle with jug capacities 8, 5, and 3 liters. The eight liter jug begins full, the others empty. The objective is to end up with one of the jugs containing 4 liters.

When we pour from one jug to another, accuracy is ensured only if we pour all of the liquid in one jug into the other *that will fit*. This means that there two limitations on the amount of liquid transferred:

(a) The amount of liquid in the source jug

(b) The amount of space in the destination jug

Thus the amount of liquid transferred is the minimum of those two quantities.

Let us use terms of the form

```
jugs(N8, N5, N3)
```

to represent the state of the system, with Ni liters of liquid in the jug with capacity i. We will define a predicate => representing the possible state transitions. The first rule relates to pouring from the 8 liter jug to the 5 liter jug. The rule can be stated thus:

```
jugs(N8, N5, N3) => jugs(M8, M5, N3) :-
    N8 > 0,
    S5 is 5 - N5,
    min(N8, S5, T),
    T > 0,
    M8 is N8 - T,
    M5 is N5 + T.
```

The conjunct `N8 > 0` says that the rule only applies if something is in the 8 liter jug. S5 is computed as the space available in the 5 liter jug. Then T is computed as the amount to be transferred. However, `T > 0` prevents the transfer of nothing from being considered a move. Finally, M8 is the new amount in the 8 liter jug and M5 is the new amount in the 5 liter jug. The 3 liter jug is unchanged, so N3 is used in both the "before" and "after" states. The predicate *min* yields as the third argument the minimum of the first two arguments. Its definition could be written:

```
min(A, B, Min) :-
    A =< B,
    Min is A.
min(A, B, Min) :-
    A > B,
    Min is B.
```

In a similar fashion, we could go on to define rules for the other possible moves. We will give one more, for pouring from the 3 liter to the 5 liter jug, then leave the remaining four to the reader.

```
jugs(N8, N5, N3) => jugs(N8, M5, M3) :-
    N3 > 0,
    S5 is 5 - N5,
    min(N3, S5, T),
    T > 0,
    M3 is N3 - T,
    M5 is N5 + T.
```

The rules we have stated are simply the constraints on pouring. They do not solve any problem. In order to do this, we need to express the following recursive rule:

> A solution from a final state consists of the empty sequence of moves.
>
> A solution from a non-final state consists of a move from the state to another state, and a solution from that state.

In order to avoid re-trying the same state more than once, we need a way to keep track of the fact that we have tried a state before. We will take a short-cut here and use Prolog's device of dynamically asserting new logical facts. In effect, we are building the definition of a predicate on the fly. Facts of the form `marked(State)` will indicate that *State* has already been tried. The conjunct `\+marked(State1)` says that we have not tried the state before. So as soon as we determine that we have not tried a state, we indicate that we are now trying it. Then we use predicate move as constructed above, to tell us the new state and recursively call solve on it. If successful, we form the list of moves by combining the move used in this rule with the list of subsequent moves.

```
solve(State1, []) :-
    final(State1).            % final state reached, success

solve(State1, Moves) :-
    \+marked(State1),                % continue if state not tried
    assert(marked(State1)),          % mark state as tried
    (State1 => State2),              % use transition relation
    solve(State2, More),                         % recurse
    Moves = [move(State1, State2) | More].       % record sequence
```

The following rules tell what states are considered final and initial:

```
initial(jugs(8, 0, 0)).

final(jugs(4, _N5, _N3)).

final(jugs(_N8, 4, _N3)).
```

When we call, in Prolog, the goal

```
    initial(State),
    solve(State, Moves).
```

two distinct move sequences are revealed, one shorter than the other, as shown be the following solution tree.

$$\begin{tikzcd}
& 800 \arrow[d] & \\
& 350 \arrow[dl] \arrow[dr] & \\
053 \arrow[d] & & 323 \arrow[d] \\
503 \arrow[d] & & 620 \arrow[d] \\
530 \arrow[d] & & 602 \arrow[d] \\
233 \arrow[d] & & 152 \arrow[d] \\
251 \arrow[d] & & 143 \\
701 \arrow[d] & & \\
710 \arrow[d] & & \\
413 & &
\end{tikzcd}$$

Figure 165: A tree of solutions for a water jug puzzle

We'll discuss further aspects of solving problems in this way in the chapter Computing with Graphs. Earlier, we stated the recursion manifesto, which suggests using recursion to minimize work in solving problems. The actual problem solving code in this example, exclusive of the rules for defining legal transitions, is quite minimal. This is due both to recursion and backtracking. So the **Prolog programmers' manifesto** takes things a step further:

Let recursion and backtracking do the work for you.

Prolog programmers' manifesto

Backtracking in "Ordinary" Languages

This is somewhat of a digression from logic, but what if we don't have Prolog available to implement backtracking? We can still program backtracking, but it will require some "engineering" of appropriate control. The basic feature provided by backtracking is to be able to try alternatives and if we reach failure, have the alternatives available so that we may try others. This can be accomplished with just recursion and an extra data structure that keeps track of the remaining untried alternatives in some form. In many cases, we don't have to keep a list of the alternatives explicitly; if the alternatives are sufficiently well-structured, it may suffice to be able to generate the next alternative from the current one.

A case in point is the classical *N-queens problem*. The problem is to place N queens on an N-by-N chessboard so that no two queens attack one another. Here two queens attack each other if they are in the same row, same column, or on a common diagonal. Below we show a solution for N = 8, which was the output from the program we are about to present.

Q							
				Q			
							Q
					Q		
		Q					
						Q	
	Q						
			Q				

Figure 166: A solution to the 8-queens problem

To solve this problem using backtracking, we take the following approach: Clearly the queens must all be in different rows. We call these the "first" queen, "second" queen, etc. according to the row dominated by that queen. So it suffices to identify the columns for the queens in each row. Thus we can proceed as follows:

Place the first queen in the first unoccupied row.
Place the second queen in the next unoccupied row so that it doesn't attack the first queen.
Place the third queen in the next unoccupied row so that it doesn't attack the first two queens.
....

Continuing in this way, one of two things will eventually happen: We will reach a solution, or we will be unable to place a queen according to the non-attacking constraint. In the latter case, we *backup* to the most recent discretionary placement and try the *next* alternative column, and proceed forward from there. The current problem is well structured: the next alternative column is just the current alternative + 1. So we can accomplish pretty much all we need by the mechanism of recursion.

Here is the program:

```
import java.io.*;
import Poly.Tokenizer;

/* N Queens puzzle solver: The program accepts an integer N and produces a
 *  solution to the N Queens problem for that N.
 */

class Queens
  {
  int N;                // number of rows and columns

  int board[];          // board[i] == j means row i has a queen on column j

  static int EMPTY = -1;        // value used to indicate an empty row


  Queens(int N)                         // construct puzzle
    {
    this.N = N;
    board = new int[N];                 // create board
    for( int i = 0; i < N; i++ )
      board[i] = EMPTY;                 // initialize board
    }


  public static void main(String arg[]) // test program
    {
    Poly.Tokenizer in = new Poly.Tokenizer(System.in);
    int token_type;
    while( prompt() && (token_type = in.nextToken()) != Poly.Tokenizer.TT_EOF )
      {
      if( token_type != Poly.Tokenizer.TT_LONG || in.lval <= 0 )
        {
        System.out.println("Input must be a positive integer");
        continue;
        }

      Queens Puzzle = new Queens((int)in.lval);

      if( Puzzle.Solve() )      // solve the puzzle
        {
        Puzzle.Show();
        }
      else
        {
        System.out.println("No solutions for this size problem");
        }
      }
    System.out.println();
    }


  static boolean prompt()               // prompt for input
    {
    System.out.print("Enter number of queens: ");
    System.out.flush();
    return true;
    }


 boolean Solve()                        // try to solve the puzzle
    {
```

```
    return Solve(0);
    }

  //
  // Solve(row) tries to solve by placing a queen in row, given
  // successful placement in rows < row. If successful placement is not
  // possible, return false.
  //

  boolean Solve(int row)
    {
    if( row >= N )
      return true;                        // queens placed in all rows, success
    for( int col = 0; col < N; col++ )  // Try each column in turn
      {
      if( !Attack(row, col) )           // Can we place in row, col?
        {
        board[row] = col;               // Place queen in row, col.
        if( Solve(row+1) )              // See if this works for following rows
          return true;                  // success
        else
          board[row] = EMPTY;           // undo placement, didn't work
        }
      }
    return false;                       // no solution found for any column
    }


  // see if placing in row, col results in an attack given the board so far.

  boolean Attack(int row, int col)
    {
    for( int j = 0; j < row; j++ )
      {
      if( board[j] == col || Math.abs(board[j]-col) == Math.abs(j-row) )
        return true;
      }
    return false;
    }


  // show the board

  void Show()
    {
    int col;

    for( col = 0; col < N; col++ )
      System.out.print(" _");
    System.out.println();

    for( int row = 0; row < N; row++ )
      {
      for( col = 0; col < board[row]; col++ )
        System.out.print("|_");
      System.out.print("|Q");
      for( col++; col < N; col++ )
        System.out.print("|_");
      System.out.println("|");
      }
    System.out.println();
    }
  }  // Queens
```

Functional Programming is a form of Logic Programming

Prolog includes other features beyond what we present here. For example, there are predicates for evaluating arithmetic expressions and predicates for forming and decomposing lists. The syntax used for lists in rex is that used in Prolog. We have said before that functions are special cases of predicates. However, functional programming does not use functions the way Prolog uses predicates; most functional languages cannot "invert" (solve for) the arguments to a function given the result. In another sense, and this might sound contradictory, functions are a special case of predicates: An n-ary function, of the form $D^n \rightarrow R$, can be viewed as an (n+1)-ary predicate. If *f* is the name of the function and *p* is the name of the corresponding predicate, then

$$f(x_1, x_2, ..., x_n) == y \text{ iff } p(x_1, x_2, ..., x_n, y)$$

In this sense, we can represent many functions as Prolog predicates. This is the technique we use for transforming rex rules into Prolog rules. A rex rule:

$f(x_1, x_2, ..., x_n)$ => *rhs*.

effectively becomes a Prolog rule:

$p(x_1, x_2, ..., x_n, y)$:-
... expression determining y from *rhs* and $x_1, x_2, ..., x_n$, !.

The ! is a special symbol in Prolog known as "cut". Its purpose is to prevent backtracking. Recall that in rex, once we commit to a rule, subsequent rules are not tried. This is the function of cut.

Append in Prolog

In rex, the `append` function on lists was expressed as:

```
append([ ], Y) => Y;
append([A | X], Y) => [A | append(X, Y)];
```

In Prolog, the counterpart would be an `append` predicate:

```
append([ ], Y, Y) :- !.

append([A | X], Y, [A | Z]) :- append(X, Y, Z).
```

In Prolog, we would usually *not* include the cut (!), i.e. we *would* allow backtracking. This permits `append` to solve for the lists being appended for a given result list. For example, if we gave Prolog the goal append(X, Y, [1, 2, 3]), backtracking would produce four solutions for X, Y:

```
X = [ ], Y = [1, 2, 3];
X = [1], Y = [2, 3];
X = [1, 2], Y = [3];
X = [1, 2, 3], Y = [ ]
```

10.4 Using Logic to Specify and Reason about Program Properties

One of the important uses of predicate logic in computer science is specifying what programs are supposed to do, and convincing oneself and others that they do it. These problems can be approached with varying levels of formality. Even if one never intends to use logic to prove a program, the techniques can be useful in thinking and reasoning about programs. A second important reason for understanding the principles involved is that easy-to-prove programs are usually also easy to understand and "maintain"[†] . Thinking, during program construction, about what one has to do to prove that a program meets its specification can help guide the structuring of a program.

Program Specification by Predicates

A standard means of specifying properties of a program is to provide two **predicates** over variables that represent input and output of the program:

Input Predicate: States what is assumed to be true at the start of the program.

Output Predicate: States what is desired to be true when the program terminates.

For completeness, we might also add a third predicate:

Exceptions Predicate: State what happens if the input predicate is not satisfied by the actual input.

For now, we will set aside exceptions and focus on input/output. Let us agree to name the predicates **In** and **Out**.

Factorial Specification Example

int n, f; (This declares the types the variables used below.)

[†] The word "maintenance" is used in a funny way when applied to programs. Since programs are not mechanical objects with frictional parts, etc., they do not break or wear out on their own accord. However, they are sometimes unknowingly released with bugs in them and those bugs are hopefully fixed retroactively. Also, programs tend not to be used as is for all time, but rather evolve into better or more comprehensive programs. These ideas: debugging and evolution, are lumped into what is loosely called program "maintenance".

In(n): n >= 0 (States that n >= 0 is assumed to be true at start.)

Out(n, f): f == n! (States that f == n! is desired at end.)

Programs purportedly satisfying the above specification:

```
/* Program 1: bottom-up factorial*/

f = 1;
k = 1;
while( k <= n )
  {
  f = f * k;
  k = k + 1;
  }
```

The program itself is almost independent of the specification, except for the variables common to both. If we had encapsulated the program as a function, we could avoid even this relationship.

```
/* Program 2: top-down factorial*/

f = 1;
k = n;
while( k > 1 )
  {
  f = f * k;
  k = k - 1;
  }
```

```
/* Program 3: recursive factorial*/

f = fac(n);

where

long fac(long n)
{
if( n > 1 )
  return n*fac(n-1);
else
  return 1;
}
```

Each of the above programs computes factorial in a slightly different way. While the second and third are superficially similar, notice that the third is not tail recursive. Its multiplications occur in a different order than in the second, so that in some ways it is closer to the first program.

Proving Programs by Structural Induction

"Structural induction" is induction along the lines of an inductive data definition. It is attractive for functional programs. Considering program 3 above, for example, a structural induction proof would go as follows:

> Basis: Prove that *fac* is correct for n == 1 and n == 0.
>
> Induction: Assuming that *fac* is correct for argument value n-1, show that it is correct for argument value n.

For program 3, this seems like belaboring the obvious: Obviously fac gives the right answer (1) for arguments 0 and 1. It was designed that way. Also, if it works for n-1, then it works for n, because the value for n is just n times the value for n-1.

The fact that functional programs essentially are definitions is one of their most attractive aspects. Many structural induction proofs degenerate to observations.

In order to prove programs 1 and 2 by structural induction, it is perhaps easiest to recast them to recursive programs using McCarthy's transformation. Let's do this for Program 2:

```
fac(n) = fac(n, 1);

fac(k, f) => k > 1 ? fac(k-1, f*k) : f;
```

Again, for `n == 0` or `n == 1`, the answer is `1` by direct evaluation.

Now we apply structural induction to the 2-argument function. We have to be a little more careful in structuring our claim this time. It is that:

> $(\forall$`f`$)$ `fac(k, f)` $\Rightarrow$ `f * k!`.

We arrived at this claim by repeated substitution from the rule for fac:

> fac(k, f) ⇒
> fac(k-1, f*k) ⇒
> fac(k-2, f*k*(k-1)) ⇒
> fac(k-3, f*k*(k-1)*(k-2)) ⇒...

Why we need the quantification of for all values of f is explained below. When called with `k == 0` or `k == 1` initially, the result f is given immediately. But `k! == 1` in this case, so `f == f*k!`.

Now suppose that `k > 1`, we have the inductive hypothesis

> $(\forall$`f`$)$ `fac(k-1, f)` $\Rightarrow$ `f * (k-1)!`.

and we want to show

(∀`f`) `fac(k, f)` ⇒ `f * k!`.

For any value of f, the program returns the result of calling `fac(k-1, f*k)`. By the inductive hypothesis, the result of this call is `(f * k) *(k-1)!`. But this is equal to `f * (k * (k-1)!)`, which is equal to `f * k!`, what we wanted to show.

The quantification (∀`f`) was necessary so that we could substitute `f*k` for `f` in the induction hypothesis. The proof would not be valid for a fixed `f` because the necessary value of `f` is different in the inductive conclusion.

Now let's look at an example not so closely related to traditional mathematical induction. Suppose we have the function definition in rex:

```
shunt([ ], M) => M;

shunt([A | L], M) => shunt(L, [A | M]);
```

This definition is a 2-argument auxiliary for the reverse function:

```
reverse(L) = shunt(L, []);
```

We wish to show that `shunt` as intended, namely:

The result of `shunt(L, M)` is that of appending `M` to the reverse of `L`.

In symbols:

(∀`L`) (∀`M`) `shunt(L, M)` ⇒ `reverse(L) ^ M`

where ⇒ means *evaluates to* and `^` means *append*.

To show this, we structurally induct on one of the two arguments. The choice of which argument is usually pretty important; with the wrong choice the proof simply might not work. Often, the correct choice is the one in which the list dichotomy is used in the definition of the function, in this case the first argument `L`. So, proceeding with structural induction, we have

Basis `L == [ ]`: (∀`M`) `shunt([ ], M)` ⇒ reverse([ ]) `^ M`)

The basis follows immediately from the first rule of the function definition; `shunt([ ], M)` will immediately rewrite to `M`. and M == reverse([ ]) `^ M`.

Induction step `L == [A | N]`: The inductive hypothesis is:

(∀`M`) `shunt(N, M)` ⇒ `reverse(N) ^ M`

and what is to be shown is:

(∀M) `shunt([A | N], M) ⇒ reverse([A | N]) ^ M`

From the second definition rule, we see that the `shunt([A | N], M)` rewrites to

```
shunt(N, [A | M])
```

From our inductive hypothesis, we have

```
shunt(N, [A | M]) ⇒ reverse(N) ^ [A | M]
```

because of quantification over the argument M. Now make use of an equality

```
reverse(N) ^ [A | M] == reverse([A | N]) ^ M
```

which gives us what is to be shown

To be thorough, the equality used would itself need to be established. This can be done by appealing to our inductive hypothesis: Notice that the *rhs* of the equality is equivalent to `shunt([A | N], [ ]) ^ M`, by the equation that defines `reverse`. According to the second definition rule, this rewrites to `shunt(N, [A]) ^ M`. But by our inductive hypothesis, this evaluates to (`reverse(N) ^ [A]) ^ M`, which is equivalent to *lhs* of the equality using associativity of `^` and the equality `[A] ^ M == [A | M]`. If desired, both of these properties of `^` could be established by secondary structural induction arguments on the definition of `^`.

Proving Programs by Transition Induction

Transition induction takes a somewhat different approach from structural induction. Instead of an inductive argument on the data of a functional program, the induction proceeds along the lines of how many transitions have been undertaken from the start of the program to the end, and in fact, to points intermediate as well.

A common variation on the transition induction theme is the method of "loop invariants". A loop invariant is a logical assertion about the state of the program at a key point in the loop, which is supposed to be true whenever we get to that point. For a while loop or a for loop, this point is just before the test, i.e. where the comment is in the following program:

```
initialization

while( /* invariant */  test )

    body
```

For example, in factorial program 2 repeated below with the invariant introduced in the comment, the loop invariant can be shown to be

```
      k > 0 && f == n! / k!
```

```
/* Program 2: top-down factorial*/

f = 1;
k = n;
while( /* assert:  k > 0 && f == n! / k! */ k > 1 )
  {
  f = f * k;
  k = k - 1;
  }
```

There are two main issues here:

1. Why the loop invariant is actually invariant.

2. Why the loop invariant's truth implies that the program gives the correct answer.

Let us deal with the second issue first, since it is the main reason loop invariants are of interest. The loop will terminate only when the test condition, k > 1 in this case, is false. But since k is assumed to have an integer value and we have the assertion k > 0, this means that `k == 1` when the loop terminates. But we also have the assertion `f == n! / k!`. Substituting `k == 1` into this, we have `f == n!`, exactly what we want to be true at termination.

Now the first issue. Assume for the moment that `n > 0` when the program is started. (If `n == 0`, then the loop terminates immediately with the correct answer.) Essentially we are doing induction on the number of times the assertion point is reached. Consider the first time as a basis: At this time we know `f ==1` and `k == n`. But `n > 0`, so the `k > 0` part of the assertion holds. Moreover, `n! / k! == 1`, and f == 1 because we initialized it that way. So `f == n! / k!` and the full assertion holds the first time.

Inductively, suppose the assertion holds now and we want to show that it holds the next time we get to the key point, assuming there will be a next time. For there to be a next time, `k > 1`, since this is the loop condition. Let `f'` be the value of `f` and `k'` be the value of k the next time. We see that `f' == f*k` and `k' == k-1`. Thus `k' > 0` since `k > 1`. Moreover, `f' == f*k == (n! / k!)*k == n! / (k-1)! == n! / k'!`, so the second part of the invariant holds.

This completes the proof of program 2 by transition induction. Note one distinction, however. Whereas structural induction proved the program terminated and gave the correct answer, transition induction did not prove that the program terminated. It only proved that *if* the program terminates, the answer will be correct. We have to go back and give a second proof of the termination of program 2, using guess what? Essentially structural induction! However, the proof is easier this time: it only needs to show termination, rather than some more involved logical assertion. We essentially show:

> The loop terminates for `k <= 1`. This is obvious.
>
> If the loop terminates for `k-1`, it terminates for `k`. This is true because `k` is replaced by `k-1` in the loop body.

Further Reflection on Program Specification

Note that the input specification for factorial above is `n >= 0`. Although we could run the programs with values of n not satisfying this condition, no claims are made about what they will do. A given program could, for example, do any of the following in case `n < 0`:

a) Give a "neutral" value, such as 1, which is the value of `f(0)` as well.

b) Give a "garbage" value, something that is based on the computation that takes place, but is relatively useless.

c) Fail to terminate.

The problem with actually specifying what happens in these non-standard cases is that it commits the programmer to satisfying elements of a specification that are possibly arbitrary. It may well be preferable to "filter" these cases from consideration by an appropriate input specification, which is what we have done.

Another point of concern is that the output specification `f == n!` alludes to there being some *definition* of `n!`. For example, we could give a definition by a set of rex rules. But if we can give the rex rules, we might not need this program, since rex rules are executable. This concern can be answered in two ways: (i) Having more than one specification of the solution to a problem such that the solutions check with each other increases our confidence in the solutions. (ii) In some cases, the output specification will not specify the result in a functional way but instead will only specify properties of the result that could be satisfied by a number of different functions. Put another way, we are sometimes interested in a program that is just *consistent* with or *satisfies* an input-output *relation*, rather than computing a specific function.

Finally, note that specifying the factorial program in the above fashion is a sound idea only if we can be assured that `n` is a **read-only variable**, i.e. the program cannot change it. Were this not the case, then it would be easy for the program to satisfy the specification without really computing factorial. Specifically, the program could instead just consist of:

```
n = 1;
f = 1;
```

Certainly `f = n!` would then be satisfied at end, but this defeats our intention for this program. If we declared in our specification

```
read_only: n
```

then the above program would not be legitimate, since it sets n to a value.

Another way to provide a constraining specification by introducing an **anchor variable** that does not appear in the program. Such variables are read-only by definition, so we might declare them that way. For factorial, the specification would become, where n_0 is the initial value of n and does not occur in the program proper:

```
int n, $n_0$, f;

read_only: $n_0$

In(n0):  n == $n_0$  ∧ $n_0$ >= 0

Out(n0, f):  f = $n_0$!
```

This doesn't look all that much different from the original specification, but now the "short-circuit" program

```
n = 1;
f = 1;
```

does *not* satisfy the specification generally. It only does so in the special case where n_0 `== 0` or n_0 `== 1`, since only then is n_0`! == 1`. We can remind ourselves that anchor variables are read-only

Array Summation Specification Example

A specification for a program summing the elements of an array `a`, from `a[0]` through `a[n-1]`.

```
float a[];
float max;
int n;

read_only: a, n

In(n, a): n >= 0

Out(n, a, max): max == sum(i = 0 to n-1, a[i])
```

Here we have introduced a notation *sum* to indicate the result of summing an array. As with some of the other examples, this specification would probably be enough to serve as a second solution to the problem if sum were a valid programming construct. An example of a program purporting to satisfy the specification is:

```
s = 0;
k = 0;
while( k < n )
  {
  s = s + a[k];
  k++;
  }
```

Were it not for the `read_only` specification, we could satisfy the output predicate by merely setting `n` to `0` or by setting all elements of `a` to `0`, and setting `s` to `0`.

Using Quantifiers over Array Indices

An array is typically an arbitrarily-large collection of data values. As such, we cannot refer to each value by name in a specification; we must resort to quantifiers to talk about all of the elements of an array.

Array Maximum Example

As an example, consider the specification of a program for computing the maximum of an array in a variable `max`. Here two things are important for `max`:

The value of `max` should be >= each array element.

The value of `max` should be == some array element.

So the output assertions will be:

```
(∀i) max >= a[i]

(∃i) max == a[i]
```

where the array bounds are understood, or to make the bounds explicit:

```
(∀i) (i >= 0 && i < n) → max >= a[i]

(∃i) (i >= 0 && i < n) && max == a[i]
```

The complete specification would then be:

```
float a[];
float s;
int n;

read_only: a, n

In(n, a): n >= 0
```

```
Out(n, a, s): ( (∀i) (i >= 0 && i < n) → max >= a[i] )
              && ( (∃ i) (i >= 0 && i < n) && max == a[i] )
```

Array Sorting Example

The following is an example wherein the specification would not readily translate into a solution of the problem (e.g. using rex). Also, since we intend to rearrange the values of an array in place, we cannot use the read_only annotation for the array itself. We must instead introduce a new read-only variable that represents the original array contents. We will use equal(a, b, n) to designate that *a* and *b* have the same values, element-by-element, from 0 through n.

Array Sorting specification:

```
float a[], a0[];

int n;

read_only a0;

In(n, a, a0): n >= 0 && equal(a, a0, n)

Out(n, a0, a):  permutation(a, a0, n) && sorted(a, n)
```

For the sorting specification, we used two auxiliary predicates to express **Out**. By `permutation(a, a0, n)` we mean that the elements of a are the same as those of a_0, except possibly in a different order (their contents are the same when they are viewed as "bags"). By `sorted(a, n)` we mean that the elements of a are in non-decreasing order. We can express `sorted` in a logical notation as:

```
sorted(a, n) is (∀i) ( ( 0 <= i  ∧ i < n-1 ) → ( a[i] <= a[i+1] ) )
```

Expressing permutation is messier, due to the need to handle possibly duplicated elements. If we introduce a notation for counting the number of a given element, say #(e, a, n) meaning the **number of occurrences** of e in a, we could define:

```
permutation(a, a0, n) is

(∀i) (∀e)
     ( 0 <= i  ∧ i < n) → ( e == a[i] → #(e, a, n) == #(e, a0, n) )
```

We could give an appropriate rex-like rules for #(e, a, n):

```
#(e, a, -1) => 0;

#(e, a, i) => ( e == a[i] )? 1 + #(e, a, i-1);
```

```
    #(e, a, i) => #(e, a, i-1);
```

The above rules would read: The number of times `e` occurs in `a[0]....a[n-1]` is `0`. For `i>= 0`, the number of times `e` occurs in `a[0]....a[i]` is 1 more than the number of times it occurs in `a[0]....a[n-1]` if `e == a[i]`. Otherwise it is the same as the number of times it occurs in `a[0]....a[n-1]`.

The fact that we need such recursive expressions, which are effectively programs themselves in an appropriate language, dampens our hope that specifications and programs can be totally distinct domains of endeavor. Indeed, writing a specification has much in common with writing a program. In the former, however, we hope for greater succinctness through the use of an appropriate specification language, such as the predicate calculus.

Correctness Defined

Given an input/output specification and a program intended to satisfy that specification, we now focus on what it *means* to satisfy a specification. Some terminology is helpful.

> **Partial Correctness**: A program P is said to be *partially correct* with respect to a specification (a pair of predicates **In**, **Out**) in case that:
>
> > If the program is started with variables satisfying **In** (and ip (instruction pointer) at the initial position), then *when and if the program terminates*, the variables will satisfy **Out**.

Notice the "when and if" disclaimer. Nothing is being claimed for cases where the program does not terminate,

> **Termination**: A program P is said to *terminate* with respect to a specification if
>
> > If the program is started with variables satisfying **In** (and ip at the initial position), then the program will terminate (i.e. will reach a state where its ip is at the final position).
> >
> > (Termination does not use the **Out** part of the specification.)
>
> **Total Correctness**: A program P is *totally correct* with respect to a specification if
>
> > The program is both partially correct and terminates with respect to that specification.

There are reasons why we separate partial correctness and termination in this way:

(i) Some programs cannot be guaranteed to terminate, so are partially correct at best.

(ii) Sometimes it is easier to prove partial correctness and termination separately.

Partial Correctness

The Floyd Assertion Principle

This principle is perhaps the easiest-to-understand way to prove partial correctness. (A special case of this method is the "**loop invariant**" idea introduced earlier.) We demonstrate it using the flowchart model for programs. Each of the nodes in the program flowchart is annotated with an **assertion**. The intent of the assertion is to **represent information about the state of the program at that particular node**, when and if the ip reaches that node.

Partial correctness is established by proving a set of **verification conditions** (**VC**s) associated with the invariants, the enabling conditions on the arcs, and the assignments on the arcs.

The beauty of this method is that, *if* the assertions are valid, the VCs can be proved individually in isolation without referring back to the original program. Here is how a VC relates to an arc in a program: Suppose that the following is a piece of the flowchart, where A_i and A_j are assertions, E is an enabling condition, and F represents the assignment being done.

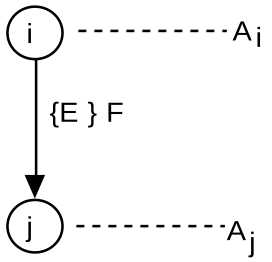

Figure 167: Flowchart fragment where nodes i and j have been annotated with assertions. E represents the enabling condition that must be satisfied for the ip (instruction pointer) to move from i to j, while F represents the change of state variables that will occur when the ip moves from i to j.

Specifically, express F as an equation between primed and unprimed versions of the program variables, representing the values before and after the statement is executed, respectively. Let A'_j be assertion A_j with all variables primed. Then the prototype **verification condition** for this arc is:

$$(A_i \wedge E \wedge F) \rightarrow A'_j$$

which is interpreted as follows: If the program's ip is at i with variables satisfying assertion A_i and the enabling condition E is satisfied, and if F represents the relation between variables before and after assignment, then A'_j holds for the new values. The names given to A_i and A'_j are **pre-condition** and **post-condition**, respectively.

Floyd Assertion Example

Consider the following fragment of a flowchart, which has been annotated with assertions at places i and j.

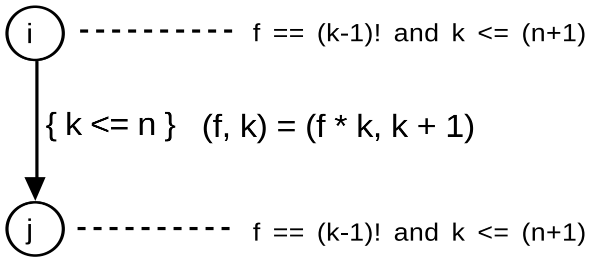
i
f == (k-1)! and k <= (n+1)
{ k <= n } (f, k) = (f * k, k + 1)
j
f == (k-1)! and k <= (n+1)

Figure 168: Fragment of a flowchart program, showing enabling condition, action, and possible assertions at nodes i and *j*

The verification condition $(A_i \land E \land F) \rightarrow A'_j$ in this case has the following parts:

A_i:	f == (k-1)! and k <= (n+1)
E:	k <= n
F:	(f', k') == (f * k, k + 1)
A'_j:	f' == (k'-1)! and k' <= (n+1)

Notes:

F represents a parallel assignment to f and k. The primed values indicate the values after the assignment, while the unprimed ones indicate the values before.

n is a read-only variable, so no primed value is shown for it.

A_i and A'_j are the same assertion, except that A'_j has its *k* and *f* variables primed, to denote their values after the assignment rather than before.

Spelled out more fully, if ξ represents the vector of all program variables, then the general enabling condition and assignment will take the form

$$\{ E(\xi) \}\ \xi = F(\xi)$$

while the verification condition for the arc is:

$$(A_i(\xi) \wedge E(\xi) \wedge \xi' == F(\xi)) \rightarrow A_j(\xi')$$

In summary, in the verification condition, we use an equality between primed variables and a function of unprimed variables to represent the effect of an assignment to one and the same set of program variables. The reason for choosing this approach is that we don't have any other way of relating assignment to a statement in predicate logic.

We continue the example by providing the verification conditions for all of the arcs of the compact factorial program, repeated here for convenience. The VC that was indicated above will be recognized as that for the arc going from node 1 to node 1.

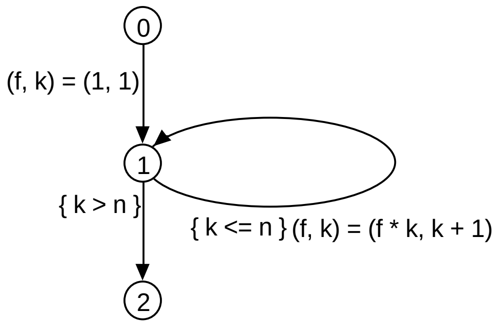

Figure 169: Compact flowchart for the factorial program

Using A_0, A_1, A_2 to represent the assertions at each node, the verification conditions, one per arc are:

arc 0 → 1: $(A_0 \wedge \textit{true} \wedge (f', k') == (1, 1)) \rightarrow A'_1$

arc 1 → 1: $(A_1 \wedge k <= n \wedge (f', k') == (f*k, k+1)) \rightarrow A'_1$

arc 1 → 2: $(A_1 \wedge k > n \wedge (f', k') == (f, k)) \rightarrow A'_2$

To complete the proof, we also need to insure that:

$\text{In} \rightarrow A_0$ and $A_{exit} \rightarrow \text{Out}$

where A_{exit} is the assertion at the exit point. In this and most cases, this is implied by just *equating* A_0 to **In** and A_{exit} to **Out**.

Before we can actually conduct the proof, we must choose the remaining assertions A_i. The guidelines for doing this are as follows:

> A_i should be always true for the program state whenever the instruction pointer points to i (*i.e.* each A_i should be "invariant").

Let us try to choose appropriate assertions for the factorial example. If we equate A_0 to In and A_2 to Out, i.e.

A_0: n >= 0

A_2: f = n!

we only have to determine an appropriate A_1. Let us try as the value of A_1 the assertion

$f == (k-1)! \wedge k <= (n+1)$

By looking at the state-transition diagram, we can get support for the idea that this condition is invariant. The VCs can now be filled in:

VC$_{01}$ arc 0 → 1:

$$(\underbrace{A_0}_{\text{assertion at place 0}} \wedge \underbrace{\mathit{true}}_{\text{arc enabling condition}} \wedge \underbrace{(f', k') == (1, 1)}_{\text{arc assignment}}) \rightarrow \underbrace{A'_1}_{\text{assertion at place 1 (primed variables)}}$$

i.e.

$$(n >= 0 \wedge \mathit{true} \wedge (f', k') == (1, 1)) \rightarrow (f' == (k'-1)! \wedge k' <= (n+1))$$

[*n* does not get primed, as it is a read_only variable.]

[VC$_{11}$ was discussed earlier.]

VC$_{11}$ arc 1 → 1:

$$(\underbrace{A_1}_{\text{assertion at place 1}} \wedge \underbrace{k <= n}_{\text{arc enabling condition}} \wedge \underbrace{(f', k') == (f*k, k+1)}_{\text{arc assignment}}) \rightarrow \underbrace{A'_1}_{\text{assertion at place 1 (primed variables)}}$$

i.e.

$$(\underbrace{f == (k-1)! \wedge k <= (n+1)}_{\text{assertion at place 1}} \wedge k <= n \wedge (f', k') == (f*k, k+1))$$

$$\rightarrow \underbrace{(f' == (k'-1)! \wedge k' <= (n+1))}_{\text{assertion at place 1 (primed variables)}}$$

VC$_{12}$ arc 1 → 2: $(A_1 \wedge k > n \wedge (f', k') == (f, k)) \rightarrow A'_2$

i.e.

$$(\underbrace{f == (k-1)! \wedge k <= (n+1)}_{\text{assertion at place 1}} \wedge k > n \wedge (f', k') == (f, k))$$

$$\rightarrow \underbrace{f' = n!}_{\text{assertion at place 2 (primed variables)}}$$

Example Proofs of Verification Conditions

We have now translated the partial correctness of the program into three logical statements, the VCs. The proof of the three VCs is straightforward and we can take them in any order. Since each is of the form H → C (hypothesis implies conclusion), we shall assume the hypothesis and show the conclusion.

VC$_{01}$ assume: (n >= 0 ∧ *true* ∧(f', k') == (1, 1))
show: (f' == (k'-1)! ∧ k' <= (n+1))

By the rightmost equation in the assumption (f' == 1, k' == 1), what we are to show follows from a simpler equivalent:

1 == (1 - 1)! ∧ 1 <= (n + 1)

The left conjunct simplifies to 1 == 0! and is true since 0! == 1 by definition of factorial. The right conjunct 1 <= (n + 1) follows from n >= 0.

VC$_{11}$ assume: (f == (k-1)! ∧ k <= (n+1)∧k <= n ∧(f', k') == (f*k, k+1))
show: (f' == (k'-1)! ∧ k' <= (n+1))

By the rightmost equation in the assumption, what we are to show follows from a simpler equivalent:

f*k == ((k+1)-1)! ∧ (k+1) <= (n+1)

i.e.

f*k == k! ∧ (k+1) <= (n+1)

The left conjunct follows from the assumption that f == (k-1)! and the definition of factorial. The right conjunct follows from the assumption k <= n. [Note that the assumption k <= (n+1) was subsumed by k <= n in this VC, and therefore was not of any particular use.]

VC$_{12}$ assume: (f == (k-1)! ∧ k <= (n+1) ∧k > n ∧(f', k') == (f, k))
show: f' = n!

What we are to show is equivalent, using the equation f' == f, to f = n!. This will follow from the assumption that f == (k-1)! if we could establish that k = n+1. But we have k <= (n+1) and k > n in our assumption. Since we are working in the domain of *integers*, this implies k = n+1.

Having proved these VCs, we have established the partial correctness of our factorial program.

A Note on Choice of Assertions

Although A_i does not have to completely characterize the state whenever the instruction pointer points to i, it must characterize it sufficiently well that all of the VCs can be proved. The possible pitfalls are:

> If A_i is chosen to be too weak (i.e. too near to universally true), then some successor post-condition might not be provable.
>
> If A_i is chosen to be too strong (i.e. too near to false), then it might not be possible to prove it from some predecessor pre-condition.

Termination

Termination proofs proceed by an additional sort of reasoning from partial correctness proofs. One method, which we call the **energy function method** involves constructing an expression E in terms of the program variables that:

> E never has a value less than 0
>
> On each iteration of any loop, E decreases.

For the second factorial program,

```
f = 1;
k = n;
while( k > 1 )
  {
  f = f * k;
  k = k - 1;
  }
```

it is very clear that the expression `k-1` by itself decreases on each iteration and that 0 is its minimum, since any attempt to make `k-1` less than 0 (i.e. `k` less than `1`) will cause the loop to terminate.

For the first factorial program,

```
f = 1;
k = 1;
while( k <= n )
  {
  f = f * k;
  k = k + 1;
  }
```

the energy function we want is `n - k + 1`. The value of this expression decreases on each iteration, since k increase. Moreover, if n - k is small enough, the condition k <= n, which is the same as `n - k + 1 > 0`, is no longer true, so the loop will terminate.

Perspective

For most individuals, the real value of understanding the principles of program correctness is not mostly for proving programs. More importantly, constructing a program as if correctness had to be proved will give us better-structured programs. Poorly structured programs are not only hard to prove; they are hard to understand and hard to build upon.

10.5 Use of Assertions as a Programming Device

Some languages provide for the inclusion of assertions in the program text. The idea is that the program asserts a certain predicate should be true at the point where the assertion is placed. At execution time, if the assertion is found to be false, the program terminates with an error message. This can be useful for debugging. Various C and C++ libraries include assert.h, which provides such an assertion facility. The idea is that

```
assert( expression );
```

is an executable statement. The expression is evaluated. If it is not true (non-zero), then the program exits, displaying the line number containing the assert statement. This is a useful facility for debugging, but it is limited by the fact that the assertion must be expressed in the programming language. The kinds of assertions needed to make this generally useful require substantial functions that mimic predicate logic with quantifiers. The only way to achieve these is to write code for them, which sometimes amounts to solving part of the problem a second time.

10.6 Program Proving vs. Program Testing

There is no question that programs must be thoroughly tested before during development. However, it is well worth keeping mind a famous statement by E.W. Dijkstra:

Testing can demonstrate the presence of bugs, but it can never prove their absence.

The only situation in which this is not true is when a program can be exhaustively tested, for every possible input. But even for inputs restricted to a finite set, the number of possibilities is impractically large. Consider the number of combinations for a 32-bit multiplier, for example.

10.7 Using Assertional Techniques for Reasoning

Understanding the basis for assertion-based verification can help with reasoning about programs. The form of reasoning we have in mind includes

> "if an assertion is known to be true at one place in the program, what can we say is true at some other place?"
>
> "if an assertion must be true at one place in the program, what must be true at some other place in order to insure that the first assertion is true?"

Although we will use textual programs to illustrate the principles, it might be helpful to think in terms of the corresponding graph model.

Conditional statements

```
if( P )
  statement-1
```

Assuming that P as a procedure call has no side-effects, we know that P as an assertion is true before statement-1. More generally, if assertion A is also true before the *if* statement (and P has no side-effects), we know that $A \wedge P$ is true before statement-1.

```
if( P )
  statement-1
else
  statement-0
```

Under the same assumptions, we know that $A \wedge \neg P$ before statement-0.

While Statements

```
while( P )
  statement-1
```

Assuming that P has no side-effects, P will be true before each execution of statement-1. Also, $\neg P$ will be true *after* the overall *while* statement. (Even if A is true before the *while* statement, we cannot be sure that A is true before statement–1 the next time around, unless P implies A.)

In general, if B is true before the *while* statement, and statement-1 reestablishes B, then B will also be true on exit.

We can summarize this reasoning as follows:

If we can prove a verification condition

$(B \wedge P \wedge \text{statement-1}) \rightarrow B$

then we can infer the verification condition

$B \wedge (\text{while } P \text{ statement-1}) \rightarrow (B \wedge \neg P)$

Here again, B is called a "loop invariant".

Example – While statement

A typical pattern occurs in the factorial program. The loop condition is k <= n. The invariant B includes, k+1 <= n. On exit, we have the negation of the loop condition, thus k > n. Together $(B \wedge \neg P)$ give k == n+1.

Reasoning by Working Backward

A somewhat systematic way to derive assertions internal to a program is to work backward from the exit assertion. In general, suppose we know that A is a post-condition that we want to be true after traversing an arc. We can derive a corresponding pre-condition that gives the minimal information that must be true in order for A to be true. This pre-condition will depend on the enabling predicate E and the assignment F for the corresponding arc, as well as on A. Since it depends on these three things, and entails the vector of program variables ξ, it is noted as wlp(A, E, F)(ξ), where wlp stands for **"weakest liberal precondition"**. A little thought will show that wlp(A, E, F)(ξ) can be derived as:

$$\text{wlp}(A, E, F)(\xi) \equiv (E(\xi) \rightarrow A(F(\xi)))$$

In other words, wlp(A, E, F)(ξ) is true just in case that whenever the enabling condition E(ξ) is satisfied, we must have A satisfied for the resulting state after assignment.

Notice that in relation to A, wlp(A, E, F) will always satisfy the verification condition for the corresponding arc, that is, we can substitute A for A_j and wlp(A, E, F) for A_i in the prototypical verification condition:

$(A_i \wedge E \wedge F) \rightarrow A'_j$

and end up with a true logical statement. Let's try it. Substituting the formula claimed for wlp in place of A, we have:

$$((E(\xi) \rightarrow A(F(\xi))) \wedge E(\xi) \wedge \xi' = F(\xi)) \rightarrow A(\xi')$$

Suppose the overall hypothesis is true. Then from $E(\xi) \rightarrow A(F(\xi))$ and $E(\xi)$, we get $A(F(\xi))$. But from the equation $\xi' = F(\xi)$, we then have $A(\xi')$.

Weakest Liberal Precondition Examples

(given) statement	**(given) post-condition**	**wlp**
x = y + 5;	x > 0	true → y + 5 > 0, *i.e.* y > -5
x = x + 5;	x == 0	true → x + 5 = 0 *i.e.* x == -5
x = x + y;	x == 0	true → x + y = 0 *i.e.* x + y = 0
[x > y] x++;	x > 0	x > y → (x + 1) > 0
[x > y] x = x - y;	x > 0	x > y → (x -y) > 0 *i.e.* true
[x > y] x++;	y > x	x > y → y > (x + 1) *i.e.* false
[x > y] y++;	x > y	x > y → x > (y + 1) *i.e.* x > (y + 1)

A wlp of *false* says that the given post-condition cannot be achieved for that particular statement. A wlp of *true* says that the given post-condition can always be achieved, independent of the variable state before the statement.

Exercises

1 •• Consider the following program that computes the square of a number without using multiplication. Devise a specification and show that the program meets the specification by deriving an appropriate loop invariant.

```
static long square(long N)
{
long i, sum1, sum2;
sum1 = 0;
sum2 = 1;

for( i = 0; i < N; i++ )
  {
  sum1 += sum2;
  sum2 += 2;
  }
return sum1;
}
```

The technique shown in this and the next problem, generalizes to computing any polynomial using only addition. This is called "finite differences" and is the basis of Babbage's *difference engine*, an early computer design. It works based on the observation that an integer squared is always the sum of a contiguous sequence of odd numbers. For example,

25 == 1 + 3 + 5 + 7 + 9 (sum of the first 5 odd numbers)

This fact can be discovered by looking at the "first differences" of the sequence of squares: they are successive odd numbers. Furthermore, the first differences of those numbers (the "second differences" of the squares") are uniformly 2's. For any n-th degree polynomial, if we compute the n-th differences, we will get a constant. By initializing the "counter" variables differently, we can compute the value of the polynomial for an arbitrary argument by initializing these constants appropriately.

2 •• Consider the following program, which computes the cube of a number without using multiplication. Devise a specification and show that the program meets the specification by deriving an appropriate loop invariant.

```
static long cube(long N)
{
long i, sum1, sum2, sum3;
sum1 = 0;
sum2 = 1;
sum3 = 6;

for( i = 0; i < N; i++ )
  {
  sum1 = sum1 + sum2;
  sum2 = sum2 + sum3;
  sum3 = sum3 + 6;
  }
return sum1;
}
```

3 ••• Consider the following Java code:

```
// assert X == X0

polylist L = X;
polylist R = NIL;
while( /* */ !null(L) )
  {
  R = cons(first(L), R);
  L = rest(L);
  }

// assert R == reverse(X0)
```

Here *reverse* denotes the usual list reversal function. Note that we can apply reverse to both sides of the equality in the final assertion to get `R == reverse(X0)`, since for any list `R` , `reverse(reverse(R)) == R`. In other words, we are asserting that this code reverses the original list. What loop invariant would you assert at /* */ in order to establish that the final assertion follows from the initial assertion? (You may make use of the functions such as *reverse* and *append* in your loop invariant, as well as "obvious" identities for these functions.) Give an argument that shows that the final assertion follows from the loop invariant, and that the proposed invariant really is invariant.

4 ••• For any properties of functions such as *reverse* and *append* you used in the preceding problem, prove those properties by structural induction on appropriate functional programs for those functions. An example of such a property is:

(∀X) `reverse(reverse(X)) == X`

where it is assumed that the domain of `X` is that of lists.

5 ••• Devise a square-root finding program based on the squaring program above. Provide a specification and show the correctness of the program.

6 •• Show that the array summation program is totally correct with respect to its specification.

7 ••• Show that the array maximum program is totally correct with respect to its specification.

8 •••• Show that the sorting program is totally correct with respect to its specification.

10.8 Chapter Review

Define the following terms:

assert library
assignment
backtracking
DeMorgan's laws for quantifiers
energy function
existential quantifier
Floyd assertion principle
interpretation
N-queens problem
partial correctness
post-condition
pre-condition

predicate
quantifier
structural induction
termination
total correctness
transition induction
universal quantifier
valid
verification condition

10.9 Further Reading

Babbage, H.P. (ed.) *Babbage's Calculating Engines*. London, 1889.

Jon Barwise and John Etchemendy, *The Language of First-Order Logic*, Center for the Study of Language and Information, Stanford, California, 1991. [Introduction to proposition and predicate logic, including a Macintosh™ program "Tarski's World. Easy to moderate.]

W.F. Clocksin and C. S. Mellish, *Programming in Prolog*, Third edition, Springer-Verlag, Berlin, 1987. [A readable introduction to Prolog. Easy.]

R.W. Floyd, *Assigning meanings to programs*, Proc. Symp. Appl. Math., 19, in J.T. Schwartz (ed.), Mathematical Aspects of Computer Science, 19-32, American Mathematical Society, Providence, R.I., 1967. [Introduction of the Floyd assertion principle.]

Cordell Green, *The Application of Theorem Proving to Question Answering Systems*, Ph.D. Thesis, Stanford University Computer Science Department, Stanford, California, 1969. [Seminal work on the connection between logic and program proving.]

R.M. Keller. *Formal verification of parallel programs*. Communications of the ACM, **19**, 7, 371-384 (July 1976). [Applies assertions to general transition-systems. Moderate.]

Zohar Manna, *Mathematical Theory of Computation*, McGraw-Hill, New York, 1974. [Examples using Floyd's Assertion Principle. Moderate.]