$$\langle append([a],[b,c],L) \mid true\rangle$$
$$\Downarrow A2$$
$$\langle [a]=[F|R],[b,c]=Y,L=[F|Z],append(R,Y,Z) \mid true\rangle$$
$$\Downarrow$$
$$\langle [b,c]=Y,L=[F|Z],append(R,Y,Z) \mid F=a \wedge R=[]\rangle$$
$$\Downarrow$$
$$\langle L=[F|Z],append(R,Y,Z) \mid F=a \wedge R=[] \wedge Y=[b,c]\rangle$$
$$\Downarrow$$
$$\langle append(R,Y,Z) \mid F=a \wedge R=[] \wedge Y=[b,c] \wedge L=[a|Z]\rangle$$
$$\Downarrow A1$$
$$\langle R=[],Y=Y',Z=Y' \mid F=a \wedge R=[] \wedge Y=[b,c] \wedge L=[a|Z]\rangle$$
$$\Downarrow$$
$$\langle Y=Y',Z=Y' \mid F=a \wedge R=[] \wedge Y=[b,c] \wedge L=[a|Z]\rangle$$
$$\Downarrow$$
$$\langle Z=Y' \mid F=a \wedge R=[] \wedge Y=[b,c] \wedge L=[a|Z] \wedge Y'=[b,c]\rangle$$
$$\Downarrow$$
$$\langle \Box \mid F=a \wedge R=[] \wedge Y=[b,c] \wedge L=[a,b,c] \wedge Y'=[b,c] \wedge Z=[b,c]\rangle$$

Figure 6.4 Derivation for `append([a],[b,c],L)`

the goal

```
append(X,Y,[1,2]).
```

will return the answers

$$X=[] \wedge Y=[1,2],$$
$$X=[1] \wedge Y=[2], \text{ and}$$
$$X=[1,2] \wedge Y=[].$$

As another example of a simple list manipulation program, consider how we might model the `alldifferent` constraint introduced in Section 3.5.

Example 6.3

The user-defined constraint `alldifferent_neq`$([V_1,\ldots,V_n])$ is intended to hold if each of the elements in the list, V_1 to V_n, is different. We can define this in terms of the primitive constraint $\neq$ as follows:

```
alldifferent_neq([]).
alldifferent_neq([Y|Ys]) :- not_member(Y,Ys), alldifferent_neq(Ys).

not_member(_, []).
not_member(X, [Y|Ys]) :- X ≠ Y, not_member(X, Ys).
```

Like many list manipulation predicates, `alldifferent_neq` has two rules. The first is the base case for when the list is empty and the second is a recursive rule for a non-empty list. The base case is simple: every item in an empty list is (vacuously)

different. The recursive case is a little more complex: all items in the list $[Y|Ys]$ are different if Y does not equal any item in the list Ys and all items in Ys are different.

We use the auxiliary predicate `not_member` to check that Y does not equal any element of the list Ys. It also consists of two rules: a base case for the empty list and a recursive rule for the non-empty list. The base case states that every element is not a member of the empty list. The recursive case states that X is not a member of the list $[Y|Ys]$ if it does not equal Y and it is not a member of the list Ys.

Usually `alldifferent_neq` will be applied to a list of variables, to ensure that none of them can take the same value. The goal `alldifferent_neq([A,B,C])`, for example, has the single answer $A \neq B \wedge A \neq C \wedge B \neq C$.

By using lists we can build models composed of complex structured data. A data structure of interest in many mathematical and engineering applications is the matrix. For instance, the matrix is the standard way to represent rectangular grids used in finite modelling. One simple representation of a matrix is as a list of lists. We can now return to the motivating example given at the beginning of this chapter and give a program that models a plate using an arbitrary sized grid of temperatures.

Example 6.4

The following program ensures that every interior point of the grid has a value equal to the average of its four neighbours. It uses case based reasoning similar to `append` except that the two cases are whether the list has three or more elements or exactly two elements. The predicate `rows` iterates through the rows in the matrix, selecting each three adjacent rows in the matrix and passing these as arguments to `cols`. The predicate `cols` iterates through the points in these rows, constraining the middle point M of a square of nine points in the matrix to equal the average of its orthogonal neighbours.

```
rows([_, _]).                                                    (RW1)
rows([R1,R2,R3|Rs]) :- cols(R1, R2, R3), rows([R2,R3|Rs]).      (RW2)

cols([_, _], [_, _], [_, _]).                                    (CL1)
cols([TL,T,TR|Ts], [ML,M,MR|Ms], [BL,B,BR|Bs]) :-                (CL2)
    M = (T + ML + MR + B) / 4,
    cols([T,TR|Ts], [M,MR|Ms], [B,BR|Bs]).
```

The metal plate shown in Figure 6.1 can be represented by the following list of lists, which we abbreviate to *plate*:

```
[[0,  100,  100,  100,  100,  100,   0],
 [0,   _,    _,    _,    _,    _,    0],
 [0,   _,    _,    _,    _,    _,    0],
 [0,   _,    _,    _,    _,    _,    0],
 [0,   _,    _,    _,    _,    _,    0],
 [0,   0,    0,    0,    0,    0,   0]].
```

In our query we specify the temperatures of the points on the outside edges, but the temperature of each interior point is a distinct unknown variable which we indicate by using the underscore. Evaluation of the goal

```
P = plate, rows(P)
```

results in the answer

```
P = [[0.00, 100.00, 100.00, 100.00, 100.00, 100.00,   0.00]
     [0.00,  46.61,  62.48,  66.43,  62.48,  46.61,   0.00]
     [0.00,  23.97,  36.87,  40.76,  36.87,  23.97,   0.00]
     [0.00,  12.39,  20.27,  22.88,  20.27,  12.39,   0.00]
     [0.00,   5.34,   8.95,  10.19,   8.95,   5.34,   0.00]
     [0.00,   0.00,   0.00,   0.00,   0.00,   0.00,   0.00]].
```

Now let us examine the way in which the program works. On selection of the user-defined constraint `rows(P)`, evaluation of the first rule, $RW1$, fails because it constrains P to have exactly two rows. Execution of the rule $RW2$ proceeds by setting $R1$ to the first row in P, $R2$ to the second row, $R3$ to the third row and Rs to the remaining (4th to 6th) rows. Then `cols(R1, R2, R3)` is called. Initially the rule ($CL1$) is tried, but this fails since it constrains $R1$, $R2$, and $R3$ to be lists of only two elements. Next rule $CL2$ is tried. In effect, this rule sets the variables $TL, T, TR, ML, M, MR, BL, B, BR$ to be the nine element grid comprising the first three elements of each of the rows $R1$, $R2$ and $R3$.

$$\begin{array}{ll} R1 = [& \boxed{\begin{array}{ccc} TL & T & TR \end{array}} \quad |Ts] \\ R2 = [& \begin{array}{ccc} ML & M & MR \end{array} \quad |Ms] \\ R3 = [& \begin{array}{ccc} BL & B & BR \end{array} \quad |Bs] \end{array}$$

The variables Ts, Ms and Bs refer to the remaining elements in the top, middle and bottom of the three rows. Now the constraint $M = (T + ML + MR + B)/4$ is added to the constraint store, enforcing that the middle point is the average of its orthogonal neighbours. Next the recursive call to `cols` is passed the top, middle and bottom rows minus their first elements. When matching with the rule $CL2$ the grid is shifted one place to the right. When there are only two elements left in each of the three lists the original `cols(R1, R2, R3)` call finishes. Next `rows` is called with the first row of the plate removed, in effect moving the computation down one row. Eventually, when there are only two rows left, evaluation finishes.

Both of the last two programs illustrate an important constraint programming technique. Not only can we make use of data structures to store fixed values, we can also use them to store variables to represent data that is presently unknown.

6.3 Association Lists

Lists allow the constraint programmer to represent and manipulate collections of objects. We can store any kind of object in a list, ranging from simple objects such as numbers to more complex objects such as records. Lists of records are often useful since they provide a way of accessing information by associating it with a key.

Consider a simple phone record with two parts: a name and phone number. We can encode this record as the tree made from the binary constructor p whose first argument is the name of the person and whose second argument is the phone number. For example, the collection of phone numbers

peter	5551616
kim	5559282
nicole	5559282

can be represented by the list of records below, abbreviated by *phonelist*:

$$[p(peter, 5551616), p(kim, 5559282), p(nicole, 5559282)]$$

The phone list gives a simple example of an ***association list*** data structure. For each name there is an associated phone number, and each phone number is associated with one or more names. The most basic operation on an association list is to find the information, in this case the telephone number, corresponding to a key, in this case a name.

The `member(X, L)` predicate defined in the program below constrains X to be a member of the list L. It can be used to find information in an association list, for example to look up the phone number corresponding to a name.

```
member(X, [X | _]).                          (E1)
member(X, [_ | R])   :-  member(X, R).       (E2)
```

The first rule holds when X is the first element of the list L. The second rule holds when X is not the first element of L but is a member of the rest of the list.

The goal `member(p(kim, N),`*phonelist*`)` finds the phone number, N, of *kim*. The (partially) simplified derivation tree is shown in Figure 6.5. Notice how information can flow is in both directions. The term $p(kim, N)$ causes failure when equated to the term $p(peter, 5551616)$. Conversely when the term $p(kim, 5559282)$ is equated with $p(kim, N)$ then N is constrained to be 5559282.

We have already seen how to look up information in an association list. The predicate `lookup` uses `member` to find the correct entry in the list. This is captured