6 Using Data Structures

In the last chapter we investigated ways in which to model constraint problems in a constraint logic programming language. In this chapter we continue our investigation by examining how tree constraints can be used to provide data structures which allow the programmer to model more complex problems. Problems such as analysis and design of circuits or tree layout in which traversal of a (usually recursive) data structure is necessary to construct the constraints of the particular problem instance.

Example 6.1
Finite element modelling is a common technique used to approximate a continuous object by a grid of discrete points. As long as the points are sufficiently close together, they provide a good model for the original object. For instance, we can model the temperatures on a rectangular sheet of metal by using an $n \times m$ grid which details the temperature of the plate at each of the points on the grid. Such a finite element model allows us to approximate what happens in the actual sheet of metal. If the temperature of the sheet is in a steady-state, the temperature at every interior point on the sheet of metal is equal to the average of its four orthogonal neighbour points. Given information about the temperature of each point on the boundary of the plate, all of the (approximate) steady-state temperatures at the interior points can be determined.

We can model a particular instance of this problem quite simply using only arithmetic equations. For example, for a 3×4 grid with variables corresponding to the positions shown below:

```
T11  T12  T13  T14
T21  T22  T23  T24
T31  T32  T33  T34
```

The steady state temperature constraints are simply

$$T22 = (T12 + T21 + T23 + T32)/4 \land T23 = (T13 + T22 + T24 + T33)/4.$$

However, generating these arithmetic equations by hand for larger grids, such as the 6×7 grid shown in Figure 6.1, is tedious and error prone. The key question is how can we build a program which allows us to model a plate using a grid of arbitrary size? In order to do so, we must be able to represent an arbitrary sized grid of variables and build a program which iterates over this grid to set up the

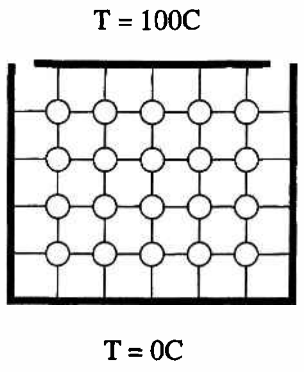

Figure 6.1 A 6 × 7 finite element description of a metal plate.

primitive constraints. In this chapter we shall investigate how to do this.

We will discover that tree constraints give us the ability to build and access data structures in a simple manner. Terms can be used to succinctly define almost all data structures that a user may require. Including, for example, records, lists, trees and stacks. In particular, seemingly complex recursive data structures are easy to define and program with. Because of the importance of data structures, almost all CLP languages provide tree constraints, and so the programming techniques discussed in this chapter are important tools in the constraint programmer's work box.

6.1 Records

The simplest type of data structure is the record. This simply packages together a fixed number of items of information, usually of a different type. We saw examples of how to use tree constraints to define record data structures in Section 1.4.

As an example, consider how we might represent complex numbers. A complex number $X + iY$ can be considered as an ordered pair of numbers, X and Y. This is easily represented using a term $c(X, Y)$ where c is a binary tree constructor. This is analogous to defining a record whose first field is the real component and whose second field is the imaginary component of the complex number.

Using this representation we can simply define predicates, `c_add` and `c_mult`, to perform addition and multiplication of complex numbers respectively.

```
c_add(c(R1,I1),c(R2,I2),c(R3,I3)) :-
     R3 = R1 + R2, I3 = I1 + I2.
c_mult(c(R1,I1),c(R2,I2),c(R3,I3)) :-
     R3 = R1*R2 - I1*I2, I3 = R1*I2 + R2*I1.
```

$$\langle C1 = c(1,3), C2 = c(2,Y), c\_add(C1,C2,C3) \mid true\rangle$$
$$\Downarrow$$
$$\langle C2 = c(2,Y), c\_add(C1,C2,C3) \mid C1 = c(1,3)\rangle$$
$$\Downarrow$$
$$\langle c\_add(C1,C2,C3) \mid C1 = c(1,3) \land C2 = c(2,Y)\rangle$$
$$\Downarrow$$
$$\langle C1 = c(R1,I1), C2 = c(R2,I2), C3 = c(R3,I3), R3 = R1 + R2, I3 = I1 + I2 \mid$$
$$C1 = c(1,3) \land C2 = c(2,Y)\rangle$$
$$\Downarrow$$
$$\langle C2 = c(R2,I2), C3 = c(R3,I3), R3 = R1 + R2, I3 = I1 + I2 \mid$$
$$C1 = c(1,3) \land C2 = c(2,Y) \land R1 = 1 \land I1 = 3\rangle$$
$$\Downarrow$$
$$\langle C3 = c(R3,I3), R3 = R1 + R2, I3 = I1 + I2 \mid$$
$$C1 = c(1,3) \land C2 = c(2,Y) \land R1 = 1 \land I1 = 3 \land R2 = 2 \land I2 = Y\rangle$$
$$\Downarrow$$
$$\langle R3 = R1 + R2, I3 = I1 + I2 \mid C1 = c(1,3) \land C2 = c(2,Y) \land R1 = 1 \land$$
$$I1 = 3 \land R2 = 2 \land I2 = Y \land C3 = c(R3,I3)\rangle$$
$$\Downarrow$$
$$\langle I3 = I1 + I2 \mid C1 = c(1,3) \land C2 = c(2,Y) \land R1 = 1 \land$$
$$I1 = 3 \land R2 = 2 \land I2 = Y \land C3 = c(3,I3) \land R3 = 3\rangle$$
$$\Downarrow$$
$$\langle \square \mid C1 = c(1,3) \land C2 = c(2,Y) \land R1 = 1 \land$$
$$I1 = 3 \land R2 = 2 \land I2 = Y \land C3 = c(3,3+Y) \land R3 = 3 \land I3 = 3 + Y\rangle$$

Figure 6.2 Derivation for a complex addition goal.

Let us examine how the predicate for addition works by using it to add the complex numbers $1 + 3i$ and $2 + Yi$ where Y is unknown. This is achieved by evaluating the goal

```
    C1 = c(1,3), C2 = c(2, Y), c_add(C1, C2, C3).
```

The derivation is shown in Figure 6.2, where the constraint store is shown in solved form.

Initally, the derivation adds $C1 = c(1,3)$ and $C2 = c(2,Y)$ to the constraint store, effectively constructing the parameters for the call to c_add. Now the call to c_add is evaluated.

The equation $C1 = c(R1,I1)$ has the effect of setting $R1$ to 1 and $I1$ to 3 since the equation $C1 = c(1,3)$ is in the constraint store. Similarly, the equation $C2 = c(R2,I2)$ sets $R2$ to 2 and forces $I2$ to take the same value as Y since the equation $C2 = c(2,Y)$ is in the store. Thus, these two equations which are implicit in the head of the rule are used to access the data in the record.

The third equation, $C3 = c(R3,I3)$, in effect builds the term $c(R3,I3)$. Evaluation of the equation $R3 = R1 + R2$ effectively sets $R3$ (or equivalently the first element of the record $C3$) to 3, and finally the equation $I3 = I1 + I2$ sets $I3$ to $3 + Y$. Thus, these equations are used to build a new record representing the result of the addition.

Simplifying the final constraint in terms of Y and $C3$ we see the resulting relationship clearly: $C3 = c(3, 3 + Y)$.

In general, programming with records requires us to use term equations both to access data in the record and to build a new record. If $T1$ and $T2$ are items, we can use the equation $R = c(T1, T2)$ to construct a new record R containing these items. Conversely, the same equation allows us to access the items in the record if R is fixed. Of course, as we have seen in the above example, not all information in a record needs to be fixed since records can contain variables as well as fixed data.

CLP languages also provide a feature which simplifies access to items in a record. An unnamed variable, written as an underscore "_", sometimes called an ***anonymous variable*** can be used as a place holder for items in the record which we are not interested in. Rather than having to give a name to a variable we do not wish to refer to again, CLP languages allow this special notation. The underscore is understood to be a variable with a different name from all other variables. It is important to realise that each occurrence of an underscore denotes a distinct new variable. For instance, we can use the constraint $R = c(T1, \_)$ to ensure that the variable $T1$ refers to the first item in R. Similarly, we can access the second item using the constraint $R = c(\_, T2)$.

6.2 Lists

Records allow us to group a fixed number of objects together. The standard data structure in CLP languages for collecting a variable number of similar objects is the list. Since lists are such an important data structure, CLP languages provide a special notation for dealing with them concisely. In the examples of Section 1.4 we used the tree constructors *cons* and *nil*. However, rather than use *cons* as the list constructor and *nil* to represent an empty list, CLP languages use "." as the list constructor and "[]" to represent an empty list. Note that "." is treated like any other binary tree constructor.

Special notation is also available to represent both lists and partially constructed lists. The expression $[X_1, X_2, \ldots, X_n]$ represents a fixed size list containing the n elements $X_1, \ldots, X_n$ while $[X_1, X_2, \ldots, X_n | Y]$ represents a list whose first n elements are $X_1, \ldots, X_n$ and whose remaining elements form the list Y. The various notations are summarized in Figure 6.3.

There are two standard equations for manipulating lists. The first equation, $L = [\,]$, constrains L to be an empty list. The second equation, $L = [F|R]$, constrains L to be a non-empty list with F as the first element and R the list of remaining elements. This equation can be used to break a list L into its first element F and remainder R, or to construct a list L from an element F and a list R. From the constraint viewpoint, these actions are equivalent.

Notation	Term with "."	Representation with *cons/nil*
[]	[]	*nil*
$[X\|Y]$	$X.Y$	$cons(X,Y)$
$[X_1,X_2,\ldots,X_n]$	$X_1.X_2.\cdots.X_n.[]$	$cons(X_1,cons(X_2,\cdots cons(X_n,nil)\cdots))$
$[X_1,X_2,\ldots,X_n\|Y]$	$X_1.X_2.\cdots.X_n.Y$	$cons(X_1,cons(X_2,\cdots cons(X_n,Y)\cdots))$

Figure 6.3 List notation

The key to programming with lists is to reason recursively about the list. Either the list is empty and the operation of interest should be straightforward, or it is non-empty and can be broken into a first element and a remaining shorter list which is dealt with recursively.

Example 6.2

Lists allow us to represent sequences. A rather important constraint on sequences that we may wish to model is that of concatenation. Suppose we wish to define a user-defined constraint append(L1, L2, L3) which holds if $L3$ is $L1$ concatenated with $L2$.

Let us consider the problem of concatenation. If $L1$ is an empty list then $L3$ is simply equal to $L2$. Otherwise $L1$ is of the form $[F|R]$ where F is the first element and R is the rest. Can we relate F and R to the problem at hand? Yes! Suppose we were to append $L2$ to the end of R, obtaining the list Z. Z is almost the answer but it is missing the element F which must be placed in front of Z to obtain the result. We can model this reasoning using the program

```
append([],Y,Y).                              (A1)
append([F|R], Y, [F|Z])  :-  append(R,Y,Z).  (A2)
```

Consider the execution of the goal append([a],[b,c],L). The (single) successful derivation is shown in Figure 6.4, where the constraint store is shown in solved form.

The first step in the successful derivation is to rewrite the initial goal using the second rule, $A2$, in the definition of append. Rewriting with this rule sets F to the first element and R to the remainder of the list $[a]$. That is, F is set to a and R to the empty list $[\,]$. Y is set to the second list $[b,c]$ and the final list L is constrained to be a followed by Z. Now the base case, $A1$, is used to process the remaining call to append. The first constraint, $R = [\,]$, checks that the end of the list has been reached. Subsequent constraints, $Y = Y'$ and $Z = Y'$, in effect equate Y and Z. This forces L to become equal to $[a,b,c]$, which is the answer.

One rather nice feature of the append program is its flexibility. Not only can it be used to concatenate lists together, it can also be use to test whether one list is the prefix of another, or even to enumerate all the lists which can be concatenated together to give a fixed list. For example, the goal

```
append([1,2],_,[1,2,3]).
```

can be used to confirm that $[1,2]$ is a prefix of the list $[1,2,3]$. On the other hand

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

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

```
      [[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]
P =    [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 $R3$.

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

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

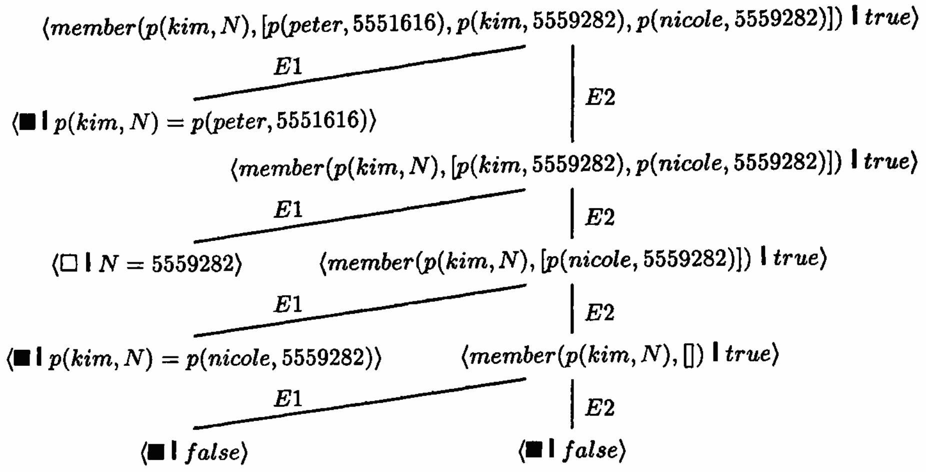
⟨member(p(kim, N), [p(peter, 5551616), p(kim, 5559282), p(nicole, 5559282)]) ▮ true⟩
E1
E2
⟨■ ▮ p(kim, N) = p(peter, 5551616)⟩
⟨member(p(kim, N), [p(kim, 5559282), p(nicole, 5559282)]) ▮ true⟩
E1
E2
⟨□ ▮ N = 5559282⟩
⟨member(p(kim, N), [p(nicole, 5559282)]) ▮ true⟩
E1
E2
⟨■ ▮ p(kim, N) = p(nicole, 5559282)⟩
⟨member(p(kim, N), []) ▮ true⟩
E1
E2
⟨■ ▮ false⟩
⟨■ ▮ false⟩

Figure 6.5 Simplified derivation tree for `member(p(kim, N),` *phonelist*`)`.

in the rule:

```
lookup(AList, Key, Info) :- member(p(Key, Info), AList).
```

Note that `lookup` can also be used to find all keys matching some information. For example, the goal `lookup(`*phonelist*`, Key, 5559282)` has the answers $Key = kim$ and $Key = nicole$.

Apart from looking up information in an association list, there are four other operations involving association lists that we may be interested in.

The first of these is determining if an association list is empty or building an empty association list. The same rule can be used for both tasks, since from a constraint viewpoint they are the same. It simply states that an empty association list is the empty list:

```
empty_alist([]).
```

The second operation is to add a new record, consisting of a key and associated information, to an association list:

```
addkey(AList0,Key,Info,AList) :- AList = [p(Key,Info)|AList0].
```

The predicate `addkey` simply adds another record to the start of the list *AList*0 to obtain the new association list *AList*. It assumes the key is not already in the list.

The third operation is to delete a record with a specific key from the association list:

```
delkey([], _, []).
delkey([p(Key,Info)|AList], Key, AList).
delkey([p(Key0,Info0)|AList0], Key, [p(Key0,Info0)|AList]) :-
     Key ≠ Key0, delkey(AList0, Key, AList).
```

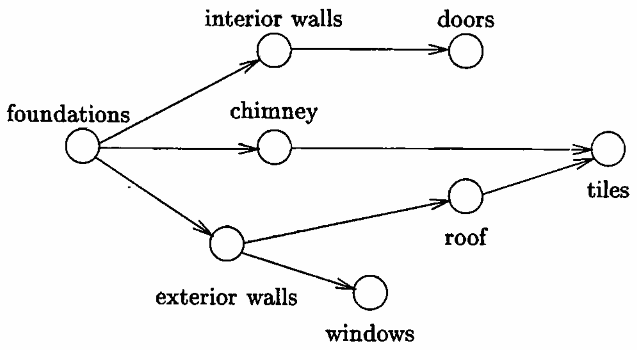

Figure 6.6 Job precedence for building a house.

The predicate `delkey` searches for the entry of the form $p(Key,\_)$ and deletes it from *AList*0 obtaining *AList*. If the key is not found in the association list the new association list is set to the original list.

The final operation is to update the information about a key:

```
modkey(AList0, Key, Info, AList) :-
    delkey(AList0, Key, AList1),
    addkey(AList1, Key, Info, AList).
```

The predicate `modkey` first deletes an element from the list and then inserts the new information.

Note that since each of the last three operations changes the association list, the predicates need two association lists as arguments—the list before the action, and the list after.

The association list data structure is useful for a variety of interesting tasks. As another example of its application, consider the project management problem of building a house from Chapter 1 and imagine that we wish to find all tasks which must be completed before a certain task can begin.

The immediate precedence relation is represented as a graph. For each task the graph contains arcs to those tasks that must be completed before this task can be started. For example, see the graph for house construction shown in Figure 6.6.

The first question is how to represent such a graph using the list and record data structures. The most obvious representation is to use a record to represent each arc in the graph. That is, to use a record of the form $arc(n_1, n_2)$ to represent an arc from node n_1 to n_2. A graph is then represented as list of arcs and, possibly, a list of nodes.

Unfortunately, this representation is not very suitable for our task since it requires multiple look ups to find all arcs from a node. A representation which is more suitable for this task is a variant of the *adjacency list*—the standard computer representation for sparse graphs. We can represent the graph as a list of records, each of which has two components. The first component is the name of a node, say n_1, in the graph and the second component is a list of those nodes who are at the start of an arc ending at n_1. That is, the second component is a list of those nodes n_2 for which there is an arc in the graph from n_2 to n_1.

Thus a node corresponding to a particular task t is represented by the pair $p(t, list\_of\_tasks)$ where *list_of_tasks* comprises those tasks for which there is an arc to t. The entire graph can be represented using a list of these pairs. It forms an association list between tasks and their list of immediate predecessor tasks.

For example, the graph for the house (shown in Figure 6.6) can be represented by the following data structure, which we will abbreviate by *house*:

```
    [p(foundations,[]), p(int_walls,[foundations]),
    p(chimney,[foundations]), p(ext_walls,[foundations]),
    p(roof,[ext_walls]), p(windows, [ext_walls]),
    p(tiles, [chimney, roof]), p(doors, [int_walls])].
```

Using this data structure it is relatively straightforward to find all tasks which must be completed before a given task can start.

Example 6.5

Suppose we wish to find all tasks that need to be completed before, for instance, the `tiles` task can begin. Clearly both `roof` and `chimney` need to be completed. However, before `roof` can be begun, its immediate predecessors must be completed, in this case `ext_walls`. Again `ext_walls` cannot begin before `foundations` is finished.

More generally, in order to find those tasks which must be completed before a given task T can start, we reason recursively as follows. If $T_1, \ldots T_n$ are the immediate predecessors of task T, and $L_1, \ldots, L_n$ are lists of all of the predecessors of $T_1, \ldots, T_n$ respectively, then the list of all predecessors of T is simply the concatenation of the lists L_1, …, L_n. This reasoning is succinctly captured in the following program:

```
predecessors(N, AList, Pre) :-
    lookup(AList, N, NImPre),
    list_predecessors(NImPre, AList, ListOfPre),
    list_append([NImPre | ListOfPre], Pre).

list_predecessors([], _, []).
list_predecessors([N|Ns], AList, [NPre|NsPres]) :-
    predecessors(N, AList, NPre),
    list_predecessors(Ns, AList, NsPres).

list_append([], []).
list_append([L|Ls], All) :-
    list_append(Ls, AppendedLs),
    append(L, AppendedLs, All).
```

The predicate predecessors works by first finding the immediate predecessors of *N* using the lookup predicate to find the record for *N* in the association list *AList*. Then list_predecessors recursively constructs a list, each of whose elements is a list of all predecessors of one of the immediate predecessors of *N*. Finally, all of these lists, together with the list of immediate predecessors of *N* are appended together using list_append.

For example, the goal

```
predecessors(tiles, house, Pre)
```

returns the answer

```
Pre = [chimney, roof, foundations, ext_walls, foundations].
```

Notice that the answer contains two copies of the foundations task because it can be reached on two paths (of precedence) from tiles. To avoid this duplication in the answer (which also required duplication of computation because two goals of the form predecessors(foundations, *house*, Pre) were executed) we need to keep track of which predecessors have already been discovered in the current computation. The above formulation does not allow us to access the information we have currently found when determining a new predecessor.

One approach to overcome this is to use an extra argument to predecessors to *accumulate* the predecessors we have already found. The meaning of the goal predecessors(N, AList, Initial, Final) is that if *Initial* is the list of tasks which we have already examined in the computation, then *Final* should be the initial list together with those tasks from which there is a path to task *N* not passing through a member of the list *Initial*

```
predecessors(N, AList, Pre0, Pre) :-
    lookup(AList, N, NImPre),
    cumul_predecessors(NImPre, AList, Pre0, Pre).

cumul_predecessors([], _, Pre, Pre).
cumul_predecessors([N|Ns], AList, Pre0, Pre) :-
    member(N, Pre0),
    cumul_predecessors(Ns, AList, Pre0, Pre).
cumul_predecessors([N|Ns], AList, Pre0, Pre) :-
    not_member(N, Pre0),
    predecessors(N, AList, [N|Pre0], Pre1),
    cumul_predecessors(Ns, AList, Pre1, Pre).
```

In cumul_predecessors if the first element of the list, N, is a member of the predecessors already visited, $Pre0$, then it can be ignored and the remainder of the tasks in the list Ns can be examined. Otherwise, if N is a not a member of the list, it is added to the list $[N|Pre0]$ and examined using predecessors. The resulting new list of predecessors $Pre1$ (which will include N) is then used when examining the remainder Ns of the tasks.

Notice that the program no longer needs to explicitly use append, since new predecessors are collected one by one in the *initial* argument as they are discovered. Using this program the goal

```
predecessors(tiles, house, [], Pre)
```

returns the answer

```
Pre = [ext_walls, roof, foundations, chimney]
```

This second program is preferable to the first. It is more succinct and it is also more efficient since the number of steps in the successful derivation is less.

6.4 Binary Trees

Lists are just one example of a recursive data structure. Another important example of a recursive data structure is binary trees.

We can define binary trees using tree constraints quite easily. We let the constant *null* represent the empty tree, and use a term of the form $node(t_1, i, t_2)$ to represent a non-null tree in which t_1 and t_2 are the left and right sub-trees, respectively, and i is the information stored at the root of the tree. For instance, the term

$$node(node(null, harald, null), kim, node(null, peter, null)),$$

which we shall abbreviate to *hkp*, is illustrated in Figure 1.5 (b) in Chapter 1.

Programs for dealing with tree data structures generally follow the recursive structure of the data structure itself. For example, as we have seen, list manipulation predicates often have a rule for the empty list and a rule for a non-empty list which processes the first element of the list and then calls the predicate recursively with the remaining elements in the list.

Most programs for manipulating binary trees also follow this pattern. They have a rule for the empty tree *null* and a recursive rule (or rules) for a non-empty tree $node(t_1, i, t_2)$. This structure will be apparent in almost all of our example programs which deal with binary trees.

As our first example of binary tree manipulation, consider how to write a program which performs an in-order traversal of a tree. The following predicate, traverse, does so. It returns a list containing the elements in the tree in the order in which they were encountered.

```
traverse(null, []).
traverse(node(T1,I,T2), L) :-
    traverse(T1, L1),
    traverse(T2, L2),
    append(L1, [I|L2], L).
```

The first rule is for the empty tree, *null*. Clearly, the list of items in an empty tree is empty. The second rule is for a non-empty tree of the form $node(t_1, i, t_2)$. The list of items visited in an in-order traversal of a tree of this form is the list obtained by traversing the left subtree, followed by i, followed by the list obtained by traversing the right subtree.

For instance, the goal `traverse(`*hkp*`, L)` gives the single answer

$$L = [harald, kim, peter].$$

Suppose we have a total order on our data items, expressed by a predicate `less_than(X, Y)` which succeeds if X is less than Y. A *binary search tree* is a particular type of binary tree in which for each sub-tree of the form $node(t_1, i, t_2)$ all of the items in tree t_1 are less than or equal to i, and all of the items in tree t_2 are greater than or equal to i. If `less_than` is defined by

```
less_than(harald, kim).
less_than(harald, peter).
less_than(kim, peter).
```

then the tree *hkp* is an example of a binary search tree.

Binary search trees are useful because they allow data items to be quickly accessed by their value. A goal of the form `find(T, E)` will determine if item E is in the binary search tree T using the following program:

```
find(node(_TL, I, _TR), E) :- E = I.                   (F1)
find(node(TL, I, _TR), E) :- less_than(E, I), find(TL, E). (F2)
find(node(_TL, I, TR), E) :- less_than(I, E), find(TR, E). (F3)
```

Note that some of the variable names begin with an underscore "_." By convention this indicates that these variables appear only once in the rule. Here the tests $E = I$, `less_than(E, I)`, and `less_than(I, E)` are used to determine if the desired item is equal to that stored in the root of the tree, less than it or greater than it. In the latter two cases this determines which subtree of the root should be examined for the item. The `find` predicate is analogous to the `member` predicate for lists.

The goal `find(`*hkp*`, peter)` has the following successful (simplified) derivation:

$$\langle find(node(node(null, harald, null), kim, node(null, peter, null)), peter) \mid true\rangle$$
$$\Downarrow F3$$
$$\langle less\_than(kim, peter), find(node(null, peter, null), peter) \mid true\rangle$$
$$\Downarrow$$
$$\langle find(node(null, peter, null), peter) \mid true\rangle$$
$$\Downarrow F1$$
$$\langle \square \mid true\rangle$$

It is quite easy to modify `find` so as to give a new predicate which takes a binary search tree and a data item and returns the new binary search tree which results from inserting the data item into the tree. If the item is already in the tree, the tree is left unchanged.

```
insert_bst(null, E, node(null,E,null)).
insert_bst(node(TL, I, TR), E, node(TL, I, TR)) :-
    E = I.
insert_bst(node(TL, I, TR), E, node(NTL, I, TR)) :-
    less_than(E, I), insert_bst(TL, E, NTL).
insert_bst(node(TL, I, TR), E, node(TL, I, NTR)) :-
    less_than(I, E), insert_bst(TR, E, NTR).
```

Using this predicate we can build a binary search tree, by inserting elements one by one into an initially empty tree.

Just as we can use lists to implement association lists, we can also use binary search trees. The advantage of the binary search tree implementation is that the time taken to find an item in a binary search tree is on average logarithmic in the size of the tree while the average time taken to find an element in the list representation of an association list is linear.

We will store the association list as a tree with each item in the tree being an association pair $p(key, info)$. Since we wish to access items in the tree by their key value, the ordering on the items must only depend on the key. Given the predicate key_less_than which compares two keys, the predicate to compare nodes in the tree is therefore

```
less_than(p(Key1, _), p(Key2, _)) :- key_less_than(Key1, Key2).
```

Given the above predicates for binary search tree manipulation, it is easy to implement the standard operations on association lists. The predicate to look up the information associated with a particular key simply calls find to search the tree for the item with that key value:

```
lookup(AList, Key, Info) :-
    find(AList, p(Key, Info)).
```

The empty association list is just the empty tree:

```
 empty_alist(null).
```

while the predicate insert_bst can be used to add a new association pair to the tree:

```
addkey(Key, Info, AList0, AList) :-
    insert_bst(AList0, p(Key, Info), AList).
```

To implement the delkey procedure for an association list defined using binary search trees we need a method for deleting an item from a binary search tree. Because of the nature of binary search trees this is somewhat complicated and is left as an exercise for the reader.

Example 6.6
Recall the association list for phone numbers. Consider how we can store this in a binary search tree. The first issue is how to order the keys. One possibility is to exhaustively detail the ordering between the names in the list using facts but this is quite tedious. A better technique is to map non-numeric keys into a unique number and then use ≤ to compare these numbers. For instance we can order the keys using

```
number(peter, 3).
number(kim, 1).
number(nicole, 2).

key_less_than(K1,K2) :-
    N1 ≤ N2,
    number(K1,N1),
    number(K2,N2).
```

As an example, the goal

```
    empty_alist(L0),
    addkey(L0,peter,5551616,L1),
    lookup(L1,peter,Number).
```

will build the binary search tree $L1$ with the single item $p(peter, 5551616)$ and then determine the phone number of *peter*.

Data structures add considerable power and flexibility to the CLP paradigm, allowing it to be used as a general purpose programming language as well as a powerful modelling language. In the remainder of this chapter, we describe two non-trivial modelling applications which make use of sophisticated data structures to represent the problem and which traverse these data structures to generate the primitive constraints that model the problem.

6.5 Hierarchical Modelling

Constraint logic programs allow us to model a problem's structure by directly using user-defined constraints for each important concept. Many engineering problems are based on the analysis of complex objects made from simpler sub-components. Thus they are naturally hierarchical in structure. Such problems can be readily modelled using a hierarchy of rules in which concepts in each level of the hierarchy are defined in terms of concepts lower in the hierarchy.

As an example, consider the analysis of steady-state RLC electrical circuits, that is to say, circuits consisting of resistors (R), inductors (L) and capacitors (C). At the lowest level of the problem, individual (sinusoidal) voltages and currents can be represented by a complex number which is understood as a vector whose amplitude is the size of the voltage or current, and whose direction gives the phase of the voltage or current. Individual circuit components form the next layer in the hierarchy. Their behaviour in terms of voltage, current and circuit values is easily modelled in terms of complex numbers. Finally, more complicated circuits can be modelled by describing them in terms of series and parallel connections of circuit components.

In order to use the hierarchical modelling we need to make decisions about how the various levels of the hierarchy are modelled. We have already seen on page 186 how to model complex numbers in terms of real numbers by using a record construct of the form $c(x, y)$ to model $x + iy$.

At the level of the individual circuit components, the important information is the type of the component and the components value. We could represent this as a pair, for example $p(resistor, 100)$, but since we will only be interested in the component's value once the type of component is known, a simpler representation is to use the component type as a tree constructor. Thus we use $resistor(R)$ to represent a resistor with resistance R, $inductor(L)$ to represent an inductor with inductance L and $capacitor(C)$ to represent a capacitor with capacitance C. The advantage of this representation is that case-by-case handling of the different components is easy.

At the highest level of the hierarchy, we need to represent complex circuits made from individual components connected in parallel or in series. Complex circuits can be modelled using a representation similar to that for individual circuit components. We let the tree $series(C_1, C_2)$ represent the circuit obtained by placing circuit C_1 in series with circuit C_2. Similarly, $parallel(C_1, C_2)$ represents a circuit obtained by placing circuit C_1 in parallel with circuit C_2. Note that the data structure for representing circuits is recursive since we can connect arbitrary circuits or components in parallel or in series to create a more complex circuit.

The following program analyses the steady state behaviour of RLC circuits with sinusoidal frequency W.

```
resistor(R, V, I, _W) :- c_mult(I, c(R,0), V).
inductor(L, V, I, W) :- c_mult(c(0,W*L), I, V).
capacitor(C, V, I, W) :- c_mult(c(0,W*C), V, I).

circuit(resistor(R), V, I, W) :- resistor(R, V, I, W).
circuit(inductor(L), V, I, W) :- inductor(L, V, I, W).
circuit(capacitor(C), V, I, W) :- capacitor(C, V, I, W).
circuit(series(C1, C2), V, I, W) :-
     circuit(C1, V1, I, W),
     circuit(C2, V2, I, W),
     c_add(V1, V2, V).
circuit(parallel(C1, C2), V, I, W) :-
     circuit(C1, V, I1, W),
     circuit(C2, V, I2, W),
     c_add(I1, I2, I).
```

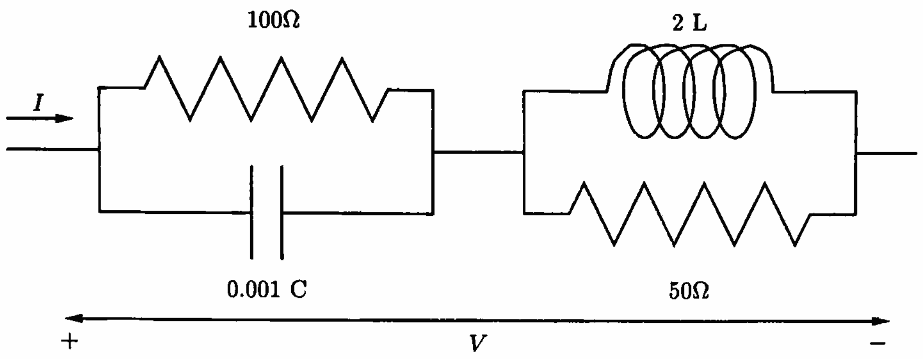

Figure 6.7 An example RLC circuit.

The behaviour of individual components is modelled using the user-defined constraints for complex numbers which specify the relationship between voltage and current for that component. The behaviour of circuits uses the component definitions for simple circuits of only one component, as well as simple recursive rules for more complex circuits. A series circuit, $series(C_1, C_2)$, implies the current through the two circuits is the same, and the voltage of the series circuit is the sum of the voltages over the individual circuits. A parallel circuit, $parallel(C_1, C_2)$, has the dual relation between voltage and current.

We can use this deceptively simple program to answer many goals. For example, we can analyze the behaviour of the circuit shown in Figure 6.7 when operating at 60 Hz by executing the goal

```
circuit(series(parallel(resistor(100),capacitor(0.001)),
                   parallel(inductor(2),resistor(50))),
          V, I, 60).
```

The result is

```
I = c(_t23, _t24)
V = c(-1.53526*_t24 + 45.3063*_t23, 45.3063*_t24 + 1.53526*_t23)
```

where the relationship between voltage V and current I is shown in terms of the real and imaginary parts of the current ($\_t23, \_t24$).

6.6 Tree Layout

Trees are used in a wide variety of applications and are best understood if they are displayed pictorially. However, as anyone who has ever tried to write LaTeX code to display a tree knows, this is not that easy.

Our next example application is to write a program which computes a good layout for a tree. The program must take an arbitrary binary tree and compute (x, y) coordinates for each node which is where that node will be drawn. The layout must be aesthetically pleasing and should be no wider than necessary. As an example of what constitutes good layout consider the following term representing a binary tree:

```
node(node(node(node(null,kangaroo,null),
               marsupial,
               node(null,koala,null)
          ),
          mammal,
          node(null,
               monotreme,
               node(null,platypus,null)
          )
     ),
     animal,
     node(node(node(null,cockatoo,null),
               parrot,
               node(null,lorikeet,null)
          ),
          bird,
          node(null,
               raptor,
               node(null,eagle,null)
          )
     )
).
```

How would we like to display this tree? Reasonable requirements are that:

- Nodes of the tree on the same level are aligned horizontally.
- Different levels are spaced at equal intervals.
- There is a minimum gap between adjacent nodes on the same level.
- A parent node lies above and midway between its children.
- The width of the tree is minimized.

The basic idea behind the program is straightforward. The main predicate `layout_constraints` defines the constraints on the layout. It first builds an association list which associates each node in the input tree with a record $c(X,Y)$. The record will store the position of that node in the layout. Initially the coordinates are variables. Next it traverses the tree and adds layout constraints between nodes in which one is the parent of the other. Then it traverses the tree and adds layout constraints between nodes which are on the same level. Then the root is placed at the desired location. The overall goal is to build a layout which minimizes the width. The high level program is therefore:

```
layout_constraints(T,AL,W) :-
    traverse(T, L),
    build_assoc_list(L, AL),
    parent_constraints(T, AL),
    height(T, D),
    sideways_constraints(T, 1, D, AL),
    place_root(T, AL),
    width_exp(T, AL, W).

tree_layout(T, AL) :- minimize(layout_constraints(T,AL,W), W).
```

The predicate traverse was defined earlier in Section 6.4. It returns a list of the elements in a tree. The procedure to build the association list containing the node's coordinates is a straightforward list traversal. It associates a structure containing two new variables to each node of the binary tree.

```
build_assoc_list([], []).
build_assoc_list([N|Ns], [p(N,c(_,_))|AL]) :- build_assoc_list(Ns, AL).
```

The predicate to set up the constraints arising between the coordinates of each internal node and its children is relatively simple. It traverses the original tree and for each node N with children it looks up the coordinates of N and those of the children. It then adds the constraints that the y-coordinate of each child is 10 below that of its parent and that the parent's x-coordinate is midway between that of the children. The only complication is that several cases must be considered because of the different possible tree structures.

```
parent_constraints(null, _).
parent_constraints(node(null,_,null), _).
parent_constraints(node(null, N, node(T1,N2,T2)), AL) :-
    lookup(AL, N, c(XN,YN)),
    lookup(AL, N2, c(XN2,YN2)),
    XN2 = XN, YN2 = YN - 10,
    parent_constraints(node(T1,N2,T2), AL).
parent_constraints(node(node(T1,N2,T2), N,null ), AL) :-
    lookup(AL, N, c(XN,YN)),
    lookup(AL, N2, c(XN2,YN2)),
    XN2 = XN, YN2 = YN - 10,
    parent_constraints(node(T1,N2,T2), AL).
parent_constraints(node(node(T1,N1,T2), N, node(T3,N2,T4)), AL) :-
    lookup(AL, N, c(XN,YN)),
    lookup(AL, N1, c(XN1,YN1)),
    lookup(AL, N2, c(XN2,YN2)),
    XN1 + XN2 = 2 * XN, YN1 = YN - 10, YN2 = YN - 10,
    parent_constraints(node(T1,N1,T2), AL),
    parent_constraints(node(T3,N2,T4), AL).
```

There is a rule for each of the possible tree structures: an empty tree, a tree with no children, a tree with only a right child, only a left child, and a tree with two children. Note how the children are handled recursively.

Next we need to add the constraint that adjacent nodes are at least a certain distance apart. We do this by first computing the height of the tree using `height` (whose definition we leave as an exercise) and then calling the predicate `sideways_constraints`. This iterates through each level of the tree using `level` and by calling `separate_list` for each level to ensure adjacent nodes on this level are separated. The predicate `level(T,N,L)` returns a list L of items in tree T at level N. The predicate `separate_list` works by finding the x coordinate of the first element in the list and then calling `sep_list` with this and the remainder of the list. Predicate `sep_list` constrains the x coordinate of the first element FX in the list to be 10 more than the previous x coordinate PX. It then processes the remainder of the list with the previous x coordinate now set to FX.

```
sideways_constraints(_T, N, D, _AL) :-
    N > D.
sideways_constraints(T, N, D, AL) :-
    N ≤ D,
    level(T, N, L),
    separate_list(L, AL),
    sideways_constraints(T, N+1, D, AL).

level(null, _, []).
level(node(_, N, _), D, [N]) :- D = 1.
level(node(T1, _N, T2), D, L) :-
    D ≥ 1,
    level(T1, D-1, L1),
    level(T2, D-1, L2),
    append(L1, L2, L).

separate_list([F|R], AL) :-
    lookup(AL, F, c(FX,_FY)),
    sep_list(R, AL, FX).

sep_list([], _, _).
sep_list([N|Ns], AL, PX) :-
    lookup(AL, N, c(XN,_YN)),
    XN ≥ PX + 10,
    sep_list(Ns, AL, XN).
```

The predicate `place_root` fixes the position of the root of the tree. Its definition is left as an exercise.

Now we have to minimize the width of the tree. In fact we need to do more than this because this does not sufficiently restrict the positioning of nodes. Instead what

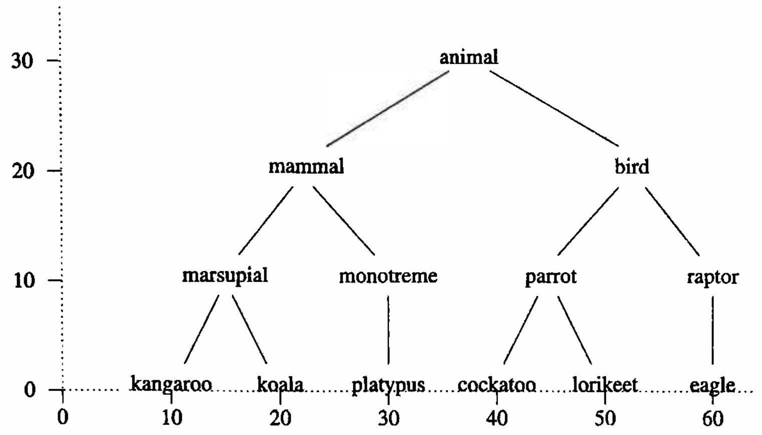

Figure 6.8 Example tree layout.

we can do is to minimize the sum of the distances between all pairs of children. To do this we recurse through the tree once more, summing the width of each subtree and returning the expression. The code is rather similar to `parent_constraints` and could in fact be combined with this phase albeit making it a little more complex.

```
width_exp(null, _, 0).
width_exp(node(null,_,null), _, 0).
width_exp(node(null, _N, node(T1,N2,T2)), AL, W) :-
     width_exp(node(T1,N2,T2), AL, W).
width_exp(node(node(T1,N2,T2), _N, null), AL, W) :-
     width_exp(node(T1,N2,T2), AL, W).
width_exp(node(node(T1,N1,T2), _N, node(T3,N2,T4)), AL, W) :-
     lookup(AL, N1, c(XN1,_YN1)),
     lookup(AL, N2, c(XN2,_YN2)),
     W = XN2 - XN1 + W1 + W2,
     width_exp(node(T1,N1,T2), AL, W1),
     width_exp(node(T3,N2,T4), AL, W2).
```

If we run this program with the tree given above and constrain the root to be at (37.5, 30) we obtain the layout shown in Figure 6.8.

This program is relatively simple yet is remarkably flexible. With little effort it can be modified to take into account additional constraints on the node positions such that some nodes must be placed in a particular region or that nodes must be aligned vertically.

6.7 Summary

One of the most interesting features of CLP languages is that the same mechanism, constraints, provides both control and data structures. In this chapter we have seen that the tree constraint domain provides most of the usual data structures found in traditional languages—records, lists and trees—and that tree constraints provide a single uniform mechanism for building and accessing these data structures.

When programming with recursive data structures it is common for the program to reflect the structure of the data structure with recursive rules to process the recursive component of the data structure. One useful programming technique is the use of accumulators.

Association lists are another useful programming technique. They allow information to be accessed via a key. We have seen how association lists can be implemented as a list of records. An even better implementation of association lists makes use of binary search trees.

We have seen two larger examples illustrating the power of data structures for modelling. The first is for the analysis of electric circuits and uses a hierarchical representation of the circuit. The second computes a nice layout for binary trees. It makes extensive use of data structures to construct the layout constraints and the expression to minimize.

6.8 Practical Exercises

P6.1. We can also use multiple facts to represent association lists. For instance, the association list for phone numbers could be represented by the facts

```
phone_number(peter, 5551616).
phone_number(kim, 5559282).
phone_number(nicole, 5559282).
```

What are two advantages of using a list or tree to represent the association list? [Hint: consider why a multiple fact representation would not work for the tree layout example.]

P6.2. Use the circuit program from Section 6.5 to evaluate the circuit shown in Figure 2.1.

P6.3. Consider a metal plate modelled using a 9×9 grid. Suppose we know that the temperature at the centre of the plate is 50°C and at the point midway between the centre and the north sides, as well as the point midway between the centre and the east sides is 90°C. Suppose we also know that exterior points on the same side have the same temperature. Use the program of Example 6.4 to find values for N, S, E, and W where these are the temperature of the exterior points on the north, south, east and west sides respectively. Note that the temperature of the corner points is irrelevant.

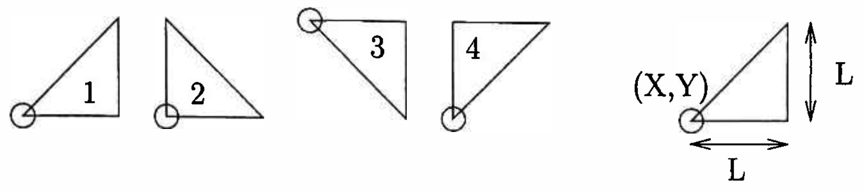

Figure 6.9 Four forms of right triangles.

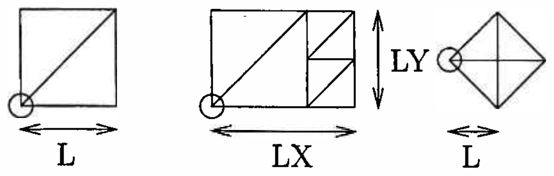

Figure 6.10 Shapes made from right triangles.

P6.4. Consider a domain of intervals where each element $[r_1, r_2]$ represents the set of points $\{X \mid r_1 \leq X \leq r_2\}$ on the real line. Addition of intervals I_1, I_2 is defined to be the smallest interval I containing the points $\{X_1 + X_2 | X_1 \in I_1, X_2 \in I_2\}$. Similarly for multiplication. Choose a representation of intervals. Write definitions for user-defined constraints to model interval addition `i_add(I1, I2, I3)` and multiplication `i_mult(I1, I2, I3)` using your representation.

P6.5. "Quarters" are right-angled triangles with both horizontal and vertical edges of equal length. Thus they take one of the four forms shown in Figure 6.9.
They can be represented using a record of the form `quarter(X,Y,L,T)` where (X, Y) are the coordinates of the leftmost point, and in the cases where there are multiple leftmost points (types 2, 4) the lowest such point, L is the length of the horizontal and vertical edges, and T is either 1, 2, 3 or 4 indicating the type of the quarter.
Write rules to define the predicate `intersect(Q1,Q2)` which succeeds if the two quarters $Q1$ and $Q2$ share a point in common and fails otherwise.

P6.6. Various shapes can be represented by lists of quarters, ***quarter descriptions***, for example those shown in Figure 6.10.
Write rules that define three predicates which construct a quarter description for a square, rectangle and diamond respectively. For example, the predicate `square(X,Y,L,R)` should succeed if R is a quarter description of a square whose leftmost bottom point is (X, Y) and whose side length is L. For instance, evaluation of `square(0,0,1,R)` could give the answer

$$R = [quarter(0,0,1,1), quarter(0,0,1,4)].$$

Note that your predicate only has to produce one description, not all possible descriptions. Similarly define the predicates `rectangle(X,Y,LX,LY,R)` and `diamond(X,Y,L,R)`.
Extend the predicate `intersect(R1, R2)` to take two quarter descriptions. That is, it should succeed if $R1$ and $R2$ are quarter descriptions that intersect.

Write a definition for the predicate `area(R, A)` which calculates the area A of a quarter description R.

P6.7. In an options trading problem, we can represent an action to buy or sell a call option by the record *call*(*cost*, *exercise_price*, *buy_or_sell*), where *cost* is the cost of the option, *exercise_price* is its exercise price, and *buy_or_sell* is 1 if buying and -1 for selling. A put option can be represented similarly as *put*(*cost*, *exercise_price*, *buy_or_sell*). An option strategy is a list of options. Write a program for predicate value(OS, S, P) which calculates the pay off P for option strategy OS given the share price S.

P6.8. Write a predicate `build_bst(L, T)` which builds a binary search tree T from a list L of items. Your definition should make use of an accumulator by means of an auxiliary predicate `cumul_insert(L, T0, T)` which inserts each item in the list L into tree $T0$ obtaining tree T. The second argument of `cumul_insert_bst` is the tree built so far, while the third argument holds the final tree. Your program should work by iterating down the list of elements, using the second argument to accumulate the result of inserting each element into an initially empty tree until the list has been traversed.

P6.9. Write a predicate `height(T,D)` which computes the height of a binary tree. Note that the empty tree has height of 0.

P6.10. (*) Write a predicate `delete_bst(T0,I,T)` which deletes item I from the binary search tree $T0$ to give the new binary search tree T.

P6.11. The tree layout program is rather inefficient because every time we look up the coordinates of the node it takes time proportional to the size of the association list. A better approach is to build an augmented tree which is essentially the original tree but which directly stores the coordinates of each node at the node in the tree. The predicate `add_coords_to_nodes` does this:

```
add_coords_to_nodes( null, null).
add_coords_to_nodes( node(T1,N,T2), node(T3,p(N,c(X,Y)),T4)) :-
     add_coords_to_nodes( T1, T3),
     add_coords_to_nodes( T2, T4).
```

Modify the tree layout to work with the augmented tree rather than with an association list.

P6.12. Write predicates `put(Q0,I,Q)` and `get(Q0,I,Q)` which implement a queue. The call `put(Q0,I,Q)` should place item I at the tail of queue $Q0$ to give Q and the call `get(Q0,I,Q)` should return the item I at the head of queue $Q0$ and the remainder of the queue in Q. You should represent a queue simply as a list of elements.

P6.13. (*) Define put and get from the above exercise using the following data structure, called a *difference list*. We use the tree *queue*($[a_1, \ldots, a_n|T], T$) to represent a queue with elements $[a_1, \ldots, a_n]$. The variable T provides fast access to the tail of the queue regardless of how long the queue is, so both `put(Q0,I,Q)` and `get(Q0,I,Q)` can take constant time.

P6.14. (*) Using the difference list $dl([a_1, \ldots, a_n|T], T)$ to represent the sequence $[a_1, \ldots, a_n]$ write a non-recursive predicate append_dl which returns the concatenation of two sequences. For instance, the goal

```
append_dl( dl([a,b|T1],T1), dl([c,d|T2],T2), dl(L,T3))
```

should have the answer

$$L = [a, b, c, d|T3] \wedge T2 = T3 \wedge T1 = [c, d|T3].$$

6.9 Notes

The ability of syntactic trees to represent common data structures such as records, lists and trees and the use of term constraints to build and manipulate these data structures is the basis of the logic programming language Prolog. The programming techniques and representations described in this chapter for records, lists, association lists and binary trees originate from the earliest days of Prolog programming and are more fully described in Prolog programming texts such as that of Sterling and Shapiro [125].

The hierarchical modelling of circuits is taken from Heintze *et al* [62]. Tree layout is a well-studied special case of graph layout. The idea of couching binary tree layout as a linear programming optimization problem appears in Supowit and Reingold [131]. For more about tree and graph layout the interested reader is referred to the survey of Battisa *et al* [8].