5 Simple Modelling

In this chapter we investigate how to model simple constraint problems using a constraint logic programming language. Modelling is at the heart of constraint programming since it is by this process that the problem is specified in terms of constraints that can be handled by the underlying solver.

Novice CLP programmers often find their greatest difficulty is adjusting to the high level nature of CLP languages: the programmer can simply state the constraints and leave the constraint solving to the underlying system. This contrasts with traditional programming languages in which the programmer is responsible for solving the constraints and explicitly assigning values to all variables. The primary role of the constraint programmer is simply to model the problem, by translating the high level constraints of the problem to the lower level primitive constraints supported by the solver.

Clearly, before starting to model the problem, the programmer needs to know what are the primitive constraints available to them, since ultimately the problem must be described in terms of these primitives. In this and the following chapter we will employ an idealised CLP language, $CLP(\mathcal{R})$, which provides tree constraints as well as arithmetic constraints over the reals.

We shall now look at how to use $CLP(\mathcal{R})$ for modelling a variety of problems and explain the generic modelling techniques we have used. In the next chapter we will explore more complex modelling in which recursive data structures are used to create the constraints modelling the problem.

5.1 Simple Modelling

We have already seen examples of simple modelling in Chapters 1 and 3 in which a problem can be straightforwardly described by a conjunction of primitive constraints. Let us now discuss how such models can be developed.

The first step in modelling is to choose the variables that will be used to represent the parameters of the problem. In many cases this step is straightforward. In other cases there may be a number of different ways to define the problem and the choice may be crucial to the final performance of the model. We defer a discussion about different problem modelling until Section 8.4.

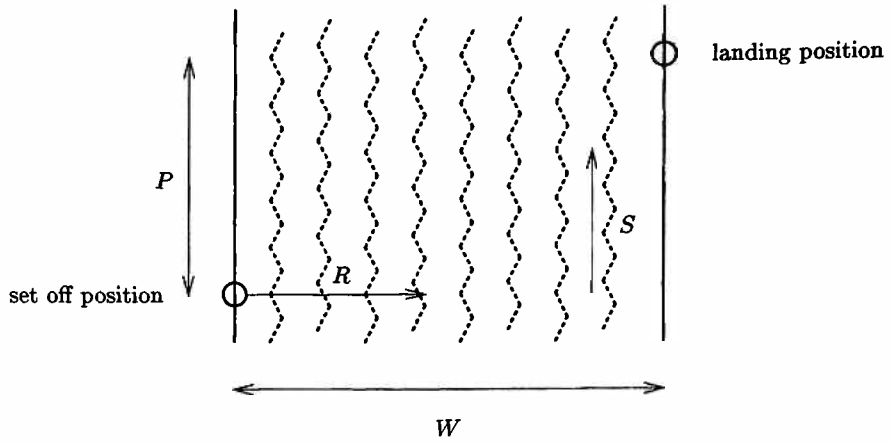

Figure 5.1 Modelling a river crossing.

For many problems, however, determining the variables of the problem is straightforward, and modelling is simply a matter of writing the constraints defining the problem in the right form.

Consider a traveller somewhere in the tropical rain forest of northern Australia. She wishes to paddle across a fast flowing river in an inflatable boat. It is important to cross the river as quickly as possible since it is full of crocodiles. There is a single small clearing on the other side of the river at which she must land. Where should she set off from her side of the river in order to reach the clearing in the least time?

We can refine this problem by observing that the quickest way to cross a river is to row directly across it, allowing the boat to drift downstream. The problem therefore becomes: how far upstream of this clearing should she set off from her side of the river?

What are the variables of the problem? Clearly the width, W, of the river is important as well as the speed, S, at which it flows. The rowing speed, R, of the traveller is also important and finally we need a variable, P, to represent the distance upstream—the desired answer. Let all measurements be in metres or metres/second. The problem with its variables is illustrated in Figure 5.1

When defining the constraints for the problem, we reason that in the time the traveller rows the width of the river, the boat floats downstream a distance given by the speed of flow by the time. We have discovered an auxiliary variable, T, the time to cross. We may not be interested in this variable but we need it to fully model the problem. Now we can write the constraints modelling the problem as a user-defined constraint `river(W, S, R, P)`. Note that, since we are not interested in the auxiliary variable T, it is not an argument of the user-defined constraint.

```
river(W, S, R, P) :- T = W/R, P = S*T.
```

We can use this model to answer the traveller's question. Suppose the traveller can row at 1.5 m/s and the river flows at speed 1 m/s and is 24 m wide. The goal:

```
S = 1, W = 24, R = 1.5, river(W, S, R, P).
```

gives the answer $P = 16 \wedge S = 1 \wedge W = 24 \wedge R = 1.5$ from which we can see the traveller needs to set off from 16 m upstream.

A feature of constraint programming is its flexibility. We can often use the same model of the problem to answer many different questions. Suppose the traveller is unsure of her precise rowing speed but knows it is somewhere between 1 and 1.3 m/s and that thick undergrowth on her side of the river prevents her setting off more than 20 m upstream. Can she make it? The goal

```
S = 1, W = 24, 1 ≤ R, R ≤ 1.3, P ≤ 20, river(W, S, R, P).
```

models this question. The reader is encouraged to determine if the intrepid traveller will survive the river crossing (see Exercise P5.1).

5.2 Modelling Choice

The underlying solver in a CLP language can only handle conjunctions of primitive constraints. However, problems often involve relationships which are not directly expressible as a conjunction of primitive constraints. Multiple rules in the definition of a predicate allow us to model these more complex relationships.

One form of information that cannot be expressed simply as a conjunction of primitive constraints are *relations*, that is to say tables of data. But by using multiple facts it is simple to represent a relation in a constraint logic program.

Consider a genealogical database describing a small family. The `father(X, Y)` relation holds when X is the father of Y. Similarly, the `mother(X, Y)` relation holds when X is the mother of Y. In addition, the relation `age(X,Y)` details the age Y of person X. The following program defines these relations for a not so fictional family (although the ages are fictional):

```
father(jim,edward).
father(jim,maggy).
father(edward,peter).
father(edward,helen).
father(edward,kitty).
father(bill,fi).

mother(maggy,fi).
mother(fi,lillian).

age(maggy,63).
age(helen,37).
age(kitty,35).
age(fi,43).
age(lillian,22).
age(jim,85).
age(edward,60).
age(peter,33).
age(bill,65).
```

The goal `father(edward,X)` can be used to find the children of Edward. It gives the answers $X = peter$, $X = helen$ and $X = kitty$. Conversely, the goal `mother(X,fi)` can be used to find the mother of Fi. It has the answer $X = maggy$.

We can extend the simple program above by defining the predicates `parent`, `sibling`, `cousin` and `older` in terms of `father`, `mother` and age. The relation `parent(X,Y)` holds if X is a parent of Y, `sibling(X,Y)` holds if X and Y are siblings, `cousin(X,Y)` holds if X and Y are cousins and `older(X,Y)` holds if X is older than Y. They may be defined by:

```
parent(X,Y)   :- father(X,Y).
parent(X,Y)   :- mother(X,Y).
sibling(X,Y)  :- parent(Z,X), parent(Z,Y), X ≠ Y.
cousin(X,Y)   :- parent(Z,X), sibling(Z,T), parent(T,Y).
older(X,Y)    :- AX ≥ AY, age(X,AX), age(Y,AY).
```

Notice how the `parent` predicate is defined with two rules. The first for the case the parent is the mother, the second for when the parent is the father.

For example the goal, `cousin(peter, X)`, has the single answer $X = fi$. We can also determine if Fi has a cousin who is older than she is using the goal

```
cousin(fi,Y), older(Y,fi).
```

This finitely fails, indicating that there are no older cousins.

Indeed, this example illustrates how constraint logic programs provide a powerful and high-level query language for databases. We shall return to this theme in Chapter 11.

Another more involved example is the interesting real life problem of modelling options trading. Options are contracts whose value depends upon the value of some underlying commodity such as company shares or frozen concentrated orange juice. We will concentrate on the two most common types of option: "call" and "put" options on company shares.

A *call* option gives the holder the right to buy a fixed number of shares for a fixed price, known as the *exercise* price. A *put* option gives the holder the right to sell a fixed number of shares for a fixed exercise price. Both call and put options have an expiry date, after which they cannot be used to buy or sell shares. Usually option contracts are for lots of 100 shares and like (almost) any other commodity may be bought or sold.

As an example, you might have had the good fortune to have bought for $200 a call option which gives you the right to buy 100 shares in a new software company, CLP Inc., at an exercise price of $300 with an expiry date of January 1st 2000. In order to understand the worth of this option consider the *pay off*, that is the profit or loss, associated with it as a function of the actual price of CLP Inc. shares on December 31st 1999. If share prices of CLP Inc. are only worth $2, then it is pointless to use the option, since you can buy 100 shares of CLP Inc. more cheaply without using it. Thus the pay off is a loss equal to the cost of purchasing the

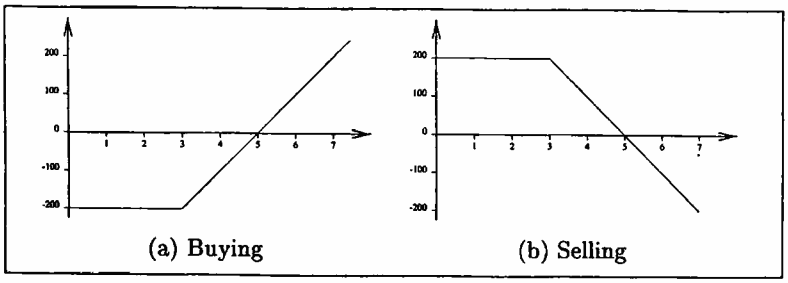

Figure 5.2 Payoff for a *call* option.

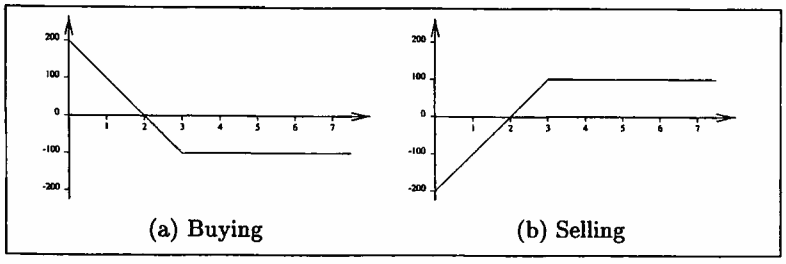

Figure 5.3 Payoff for a *put* option.

option, in other words −$200. However, if the company has done well and share prices have soared to $7, then you can use the option to buy 100 shares for $300 and immediately sell them for $700. After subtracting the cost of the option, the pay off is therefore $200 (being $700 − $300 − $200). Figure 5.2 (a) shows the graph of the pay off function for buying this call option.

We can also look at the cost of *selling* a call option. This is simply the negation of the cost of buying the call option. For instance, if Peter sold you the above call option and share prices were $2, then his pay off would be $200 since that is what you paid him for the option, and you have chosen not to use it. On the other hand, if share prices are $7, then he would make a loss of $200 since he made $200 from the original sale, but he must buy 100 shares for $700 and sell them to you for $300, giving a pay off of $200 + $300 − $700. Figure 5.2 (b) shows the pay off function for selling this call option.

We can also consider the pay off associated with a put option. Imagine that you have also bought a put option for $100 which gives you the right to sell 100 shares in CLP Inc. for $500. If the share price is $2, the pay off is $500 - $200 - $100 = $200. However, if share prices have risen to $7, the pay off is -$100, the initial purchase cost, since it is not worth using the option. The pay off for selling a put option is, of course, just the negation of the pay off for buying the option. Figure 5.3 shows the pay off functions for buying and selling this put option.

Options trading involves buying and selling complex combinations of options together with shares and bonds. Combining options allows traders to tailor their risk and return in almost arbitrary ways. For example, imagine that you believe

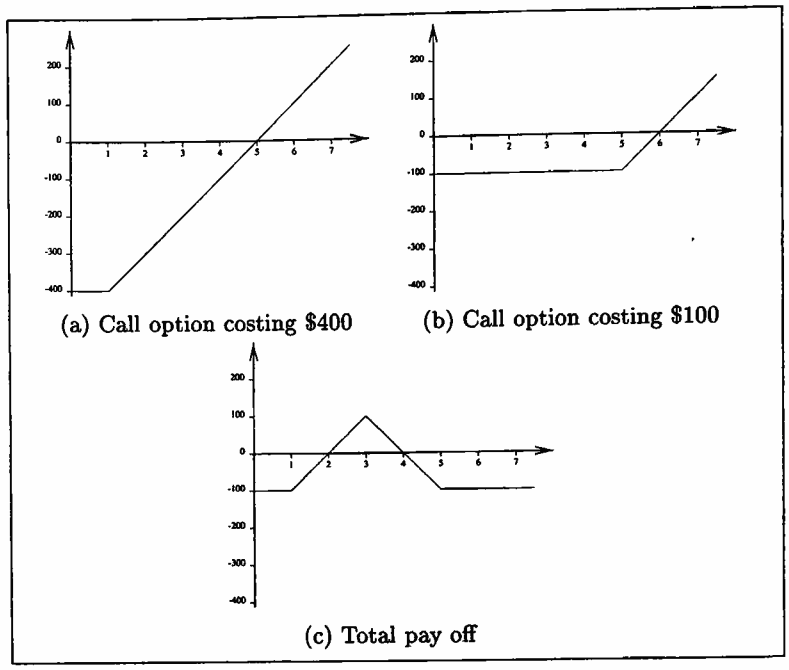

Figure 5.4 Butterfly combination.

share prices for CLP Inc. will remain around $3. However, being a prudent options trader, you would like to ensure that if share prices for CLP Inc. do go dramatically up or down you do not lose too much money. This can be done by buying a call at a exercise cost greater than $300, say at $500, buying another call at an exercise cost less than $300, say at $100, and selling two calls for an exercise cost of $300. To understand this combination, called a *butterfly*, imagine that you bought the first call for $100, the second for $400 and sold each call for $200. The pay off for each of the calls bought is shown in Figure 5.4 (a) and (b), the pay off for each of the two calls sold is shown in Figure 5.2 (b), and the combination of all four options (obtained by summing them) is shown in Figure 5.4 (c). We can see that you will make money if share prices are between $2 and $4 and you can never lose more than $100.

In order to model such problems we need to model the pay off behaviour of an option. This is reasonably straightforward. The variables of interest are clear: C is the cost of the option, E is the exercise price, S is the share price, and P is the pay off of the option. We have chosen not to include the expiry date of the option since it does not influence the pay off until it is reached.

Each pay off function is a piecewise linear function. For instance, the function associated with buying the call option shown in Figure 5.2 (a) (where $C = 200$ and

$E = 300$) is

$$PayOff(S) = \begin{cases} -200, & \text{if } 0 \leq S \leq 3 \\ 100S - 500, & \text{if } S \geq 3. \end{cases}$$

More generally, the pay off for an arbitrary call option is given by the function which maps stock price S, cost C and exercise price E to pay off as follows:

$$PayOff(S, C, E) = \begin{cases} -C, & \text{if } 0 \leq S \leq E/100 \\ 100S - E - C, & \text{if } S \geq E/100. \end{cases}$$

This function is modelled by the user-defined constraint `buy_call_payoff(C,E,S,P)` defined below, where P is the pay off and S, C, E are defined as above.

```
buy_call_payoff(S, C, E, P) :- 0 ≤ S, S ≤ E / 100, P = -C.
buy_call_payoff(S, C, E, P) :- S ≥ E / 100, P = 100 * S - E - C.
```

Notice how each of the two cases in the function definition is modelled by a different rule. Together they completely define the behaviour of the pay off function. This exemplifies a standard modelling technique for representing a function by a constraint logic program. The idea is to model a function f taking n arguments by a predicate with $n+1$ arguments in which the last argument to the predicate is the value of the function.

If a call option is sold, the pay off function is simply $-PayOff(S, C, E)$. Because the relationship between buying and selling an option is so simple, we can model both possibilities simply through a parameter, B, which multiplies the pay off for buying the option by either 1 or -1.

The user-defined constraint, `call_option(B,C,E,S,P)`, models the profit for buying or selling a call option, where S, C, E, and P are as above and B is a parameter set to 1 if the call option is bought, and set to -1 if it is sold.

```
call_option(B,S,C,E,P) :- 0 ≤ S, S ≤ E / 100, P = -C * B.
call_option(B,S,C,E,P) :- S ≥ E / 100, P = (100 * S - E - C) * B.
```

Similarly, we can model a put option by means of the user-defined constraint, `put_option(B,S,C,E,P)` defined below. Again S is the current share price, C is the cost of the put option, E the exercise price, P is the pay off and B is a flag set to 1 if the put option is bought, and sent to -1 if it is sold.

```
put_option(B,S,C,E,P) :- 0 ≤ S, S ≤ E/100, P = (E - 100 * S - C) * B.
put_option(B,S,C,E,P) :- S ≥ E / 100, P = - C * B.
```

For instance, if we want to know the value of the call option shown in Figure 5.2 (a) when share prices are \$7, we simply evaluate the goal,

```
call_option(1, 7, 200, 300, P).
```

which gives the answer $P = \$200$.

We can also ask more interesting questions. For example, the following program models the collection of options in the "butterfly" shown earlier in Figure 5.4.

```
butterfly(S, P1 + 2*P2 + P3) :-
    Buy = 1, Sell = -1,
    call_option(Buy, S, 100, 500, P1),
    call_option(Sell, S, 200, 300, P2),
    call_option(Buy, S, 400, 100, P3).
```

A natural question to ask is: "For what values of the share price will we make a profit?" This is simply couched as the goal

```
P ≥ 0, butterfly(S,P).
```

The first answer found is $P = 100S - 200 \wedge 2 \leq S \wedge S \leq 3$, indicating that if S is between 2 and 3 we make a profit. If we ask the system to find another answer, we obtain $P = -100S + 400 \wedge 3 \leq S \wedge S \leq 4$, indicating that we also make a profit if S is between 3 and 4. If we ask for another answer, the system says no, indicating these are the only values of S which give a profit.

At this point the reader might want to think about how they would write a program in a traditional language to answer this query. It would not be very easy, and would certainly require more than a few lines of code. Nor could we perform this type of reasoning using a spreadsheet, since it requires a symbolic reasoning.

As can be seen even from this simple example, options trading is a rather complex matter and sophisticated mathematical modelling is required to make money out of it. CLP languages are ideal for this sort of modelling. This is because complex trading strategies, usually formulated as rules, require a combination of symbolic and numerical calculation. There is a need to search through the different ways of combining options, together with the use of well developed mathematical models for describing option behaviour. It is also important to have a flexible powerful language for such modelling so as to allow the investigation of hypothetical or "what-if" scenarios.

5.3 Iteration

Sometimes the most natural way of modelling a constraint problem is to use iteration over some parameter of the problem in order to guide the number and form of the primitive constraints in the model. For example, if we have N warehouses in a warehouse allocation problem, we may want to iterate from 1 to N and, at each stage, add the primitive constraints modelling the i^{th} warehouse. The parameter i is used to control an iterative loop for collecting the constraints.

CLP languages do not directly provide iteration constructs, such as **for** or **while** loops, provided in more traditional languages. Instead iteration is programmed using recursive rules. We have already used such programming in the factorial program introduced in Section 4.2, in which we used a recursive rule to iterate over

the number we wish to find the factorial of.

A program to compute mortgage repayments provides a more interesting example of the use of recursive rules. We wish to model the `mortgage` constraint that holds between the following parameters: P the principal or amount owed, T the number of periods in the mortgage, I the interest rate of the mortgage, R the repayment due each period of the mortgage and B the balance owing at the end.

For a single period mortgage the relationship is easy to write,

$$B = P + P \times I - R.$$

That is, the balance is given by the principal plus the interest on the principal minus the repayment. For a loan of three periods the relationship is also easy to write,

$$\begin{aligned} P_1 &= P + P \times I - R \wedge \\ P_2 &= P_1 + P_1 \times I - R \wedge \\ P_3 &= P_2 + P_2 \times I - R \wedge \\ B &= P_3. \end{aligned}$$

The relationships between the principal after i periods, P_i, and the principal after $i + 1$ periods, P_{i+1}, is straightforward. But how can we write a program that will, in effect, build these constraints for an arbitrary value of the parameter T?

The direct approach is to reason about the `mortgage` constraint recursively. Let us consider the relationship between the parameters P, T, I, R, and B. When the number of periods in the loan is 0 the relationship is easy. The balance B is just the principal P. Now consider mortgages of one or more periods. The key step is to see that a mortgage of $T \geq 1$ periods can be viewed in the following terms. In the first period the new amount owing is obtained by adding the interest (principal by interest rate) and then subtracting the repayment from the amount currently owing. The remaining periods simply form another mortgage for the new amount owing but for one less period.

This leads us to the program below where there is one rule for the case where $T = 0$ and another rule for where $T \geq 1$. The program is simple and concise:

```
mortgage(P,T,I,R,B) :-                    (M1)
     T = 0,
     B = P.
mortgage(P,T,I,R,B) :-                    (M2)
     T ≥ 1,
     NP = P + P * I - R,
     NT = T - 1,
     mortgage(NP,NT,I,R,B).
```

The (partially simplified) successful derivation for the goal `mortgage(P,3,I,R,B)` is:

$$\langle mortgage(P,3,I,R,B) \mid true\rangle$$
$$\Downarrow M2$$
$$\langle mortgage(P_1,2,I,R,B) \mid P_1 = P + P \times I - R\rangle$$
$$\Downarrow M2$$
$$\langle mortgage(P_2,1,I,R,B) \mid P_1 = P + P \times I - R \wedge P_2 = P_1 + P_1 \times I - R\rangle$$
$$\Downarrow M2$$
$$\langle mortgage(P_3,0,I,R,B) \mid \begin{array}{l} P_1 = P + P \times I - R \wedge P_2 = P_1 + P_1 \times I - R \wedge \\ P_3 = P_2 + P_2 \times I - R \end{array}\rangle$$
$$\Downarrow M1$$
$$\langle \Box \mid \begin{array}{l} P_1 = P + P \times I - R \wedge P_2 = P_1 + P_1 \times I - R \wedge \\ P_3 = P_2 + P_2 \times I - R \wedge B = P_3 \end{array}\rangle$$

Clearly it collects the constraints we desire. No other successful derivation is possible because of the constraints on T. In effect T is a variable controlling the number of times a copy of the constraint for a single period is added to the constraint store.

Describing a mortgage as a recursive relationship may be difficult for many readers. A trick which is sometimes useful for novice CLP programmers is to think about writing a procedure in a traditional language to compute one parameter given all of the other parameters. Given this procedural definition, we first remove all direct iteration constructs and replace them by recursive calls. Finally, we "translate" the tests and assignments into constraints, and obtain a constraint logic program which models the problem.

For instance, in this case imagine we compute the balance in terms of the other parameters. Given we know everything but the balance we can reason as follows. For each time period in the mortgage we recompute the amount owed by adding to the current amount owed the interest which is interest rate multiplied by the current amount owed and subtracting the repayment. After the last time period the amount owed is the balance. A pseudo-C function which returns the balance looks something like

```
float mortgage1(float P, int T, float I, float R)
{
     while ( T >= 1) {
          P = P + P * I - R;
          T = T - 1;
     }
     return P;
}
```

We first remove the **while** loop from this program to give the following recursive pseudo-C procedure:

```
float mortgage2(float P, int T, float I, float R)
{
     if (T >= 1) {
          P = P + P * I - R;
          T = T - 1;
          return mortgage2(P, T, I, R); }
     else
          return P;
}
```

To move from this procedural specification to a constraint program we need to be aware that *variables* in procedural languages can take multiple values over the course of computation. This is not true of constraint variables. For instance, the constraint $T = T - 1$ is simply equivalent to $false$. So a C statement such as

```
P = P + P * I - R
```

is actually relating two variables (as we have seen before), the principal before a time period, and the principal after. Therefore, in a constraint program we need to represent this using two distinct variables. We thus modify our program to

```
float mortgage3(float P, int T, float I, float R)
{
     if (T >= 1) {
          NP = P + P * I - R;
          NT = T - 1;
          return mortgage3(NP, NT, I, R); }
     else
          return P;
}
```

Finally, predicates as used in constraint logic programs do not return numbers. Instead of returning the answer using a function, we must modify the program so that the balance is passed by reference, and the procedure sets the balance rather than returning its value. We thus modify our program to

```
mortgage3(float P, int T, float I, float R, float *B)
{
     if (T >= 1) {
          NP = P + P * I - R;
          NT = T - 1;
          mortgage3(NP, NT, I, R, B); }
     else
          *B = P;
}
```

We can now translate this to CLP giving the `mortgage` program

```
mortgage(P,T,I,R,B)  :-  T ≥ 1,
                         NP = P + P * I - R,
                         NT = T - 1,
                         mortgage(NP,NT,I,R,B).
mortgage(P,T,I,R,B)  :-  T = 0, B = P.
```

Note that we have translated the **if** (C) S_1 **else** S_2 statement into two rules—one for the **if** branch and one for the **else** branch. When doing so we must be careful to add the branching condition C to the rule representing the **if** branch and the negation of the branching condition to the rule representing the **else** branch. Note that in this case the negation of the branching condition is actually $T < 1$. However, given our implicit assumption that T is a non-negative integer, this is equivalent to $T = 0$.

The program we arrived at has the rules in the opposite order to the original CLP program. This will modify the order of branches in the search tree. We discuss why the program should be written in the original way in Section 7.3.

At this point it is worthwhile comparing the CLP and the pseudo-C programs. All are simple and concise and all compute the remaining balance from the other parameters with roughly the same efficiency. However, the CLP program for computing mortgage repayments is considerably more versatile than the versions written in pseudo-C. This is because the CLP program states constraints between the arithmetic variables, rather than specifying assignments to the variables. Thus it can be used to compute a number of different functions.

For instance, given the interest rate is 10% and we can repay \$150 per year we may wish to find out how much can be borrowed in a 3 year loan so that we owe nothing at the end. This question is expressed by the goal

```
mortgage(P, 3, 10/100, 150, 0).
```

The simplified successful derivation is

$$\langle mortgage(P, 3, 10/100, 150, 0) \mid true \rangle$$
$$\Downarrow M2$$
$$\langle mortgage(P_1, 2, 10/100, 150, 0) \mid P_1 = 1.1 * P - 150 \rangle$$
$$\Downarrow M2$$
$$\langle mortgage(P_2, 1, 10/100, 150, 0) \mid P_2 = 1.21 * P - 315 \rangle$$
$$\Downarrow M2$$
$$\langle mortgage(P_3, 0, 10/100, 150, 0) \mid P_3 = 1.331 * P - 496.5 \rangle$$
$$\Downarrow \cdot M1$$
$$\langle \square \mid P = 373.028 \rangle$$

This has answer $P = 373.028$.

A more complex query is to ask for the relationship between the initial principal, repayment and balance in a 10 month loan at 10%. This is done with the goal:

```
mortgage(P, 10, 10/100, R, B).
```

The answer is a constraint relating the variables P, R and B. It is

$$P = 0.385543 * B + 6.14457 * R.$$

Writing a C program to answer this query would not be straightforward.

5.4 Optimization

All of the examples we have seen so far model satisfaction problems. That is, they model problems in which we are interested in finding if there is a solution to the problem and, if so, returning one. However, in many problems we are interested in finding the "best" solution to a problem. This can be couched as an optimization problem. Suppose we are given a function which evaluates the merit of a particular solution by returning a real value. By convention, the lower the value the better the solution is. In an optimization problem we wish to find the answer to a goal which minimizes the value of this function.

Most CLP systems provide the capacity for finding the minimal or maximal solutions to a goal. Unfortunately, there is, as yet, no clear acceptance of the precise syntax and behaviour of these built-ins. In this book we will use the notation $minimize(G, E)$.

Definition 5.1
A *minimization literal* is of the form $minimize(G, E)$ where G is a goal and E is an arithmetic expression. The answers to the literal $minimize(G, E)$ are the answers to goal G which minimize the expression E.

As an example of its use, consider the program

```
p(X,Y)  :-  X=1.
p(X,Y)  :-  Y=1.
```

and the goal

```
X ≥ 0, Y ≥ 0, minimize(p(X,Y), X+Y).
```

Evaluation will return the answers $X = 1 \wedge Y = 0$ and $X = 0 \wedge Y = 1$.

A minimization literal need not return an assignment to all variables in the minimization expression. For instance, the goal

```
X ≥ 0, X ≥ Y, minimize(true, X-Y)
```

has the answer $X \geq 0 \wedge X = Y$.

As a slightly more realistic example, consider the options trading example from

Section 5.2. The following goal can be used to find the maximum profit possible for the butterfly combination:

```
minimize( butterfly(S,P), -P).
```

This will return the answer $S = 3 \wedge P = 100$ indicating that the maximum profit is \$100.

A precise definition of the operational behaviour of optimization built-ins is quite complex. The definition for $minimize(G, E)$ is, unfortunately, no exception.

Definition 5.2

A valuation θ is a *solution* of a state or goal if it is a solution of some answer to the state or goal.

We extend the definition of a derivation step from $\langle G_1 \mid C_1 \rangle$ to $\langle G_2 \mid C_2 \rangle$ by allowing a *minimization derivation step* in which a minimization subgoal is rewritten. Let G_1 be of the form

$$L_1, L_2, \ldots, L_m$$

where L_1 is $minimize(G, E)$. There are two cases.

1. There is at least one solution ϕ of $\langle G \mid C_1 \rangle$ with $\phi(E) = m$, and for all other solutions θ of $\langle G \mid C_1 \rangle$, $m \leq \theta(E)$. Then C_2 is $C_1 \wedge E = m$ and G_2 is $G, L_2, \ldots, L_m$.

2. If evaluation can detect that $\langle G \mid C_1 \rangle$ has no solutions or has an unbounded minimum, then C_2 is $false$ and G_2 is the empty goal.

This definition captures the following intuition. Suppose we encounter a minimization goal $minimize(G, E)$ when the current constraint is C_1. We first find the minimum value m that E can take in any answer to the state $\langle G \mid C_1 \rangle$. Now the answers to $minimize(G, E)$ are simply the answers to the state $\langle G \mid C_1 \wedge E = m \rangle$, that is to say, the answers to G in which E takes the minimum value m.

If $\langle G \mid C_1 \rangle$ has a finite deri ıtion tree and has no minimal solutions, either because $\langle G \mid C_1 \rangle$ finitely fails or beca. ɜe it has an unbounded minimum, then the original state is rewritten to a fail state. Be warned that if $\langle G \mid C_1 \rangle$ has an infinite derivation tree, evaluation of the minimization literal may not terminate.

Optimization need not be used only in the goal. We can also use it in rules, allowing us to define more complex user-defined constraints.

A *straddle* option involves buying a call and a put option at the same exercise price. Its behaviour can be modelled as

```
straddle(S, C, E, P)  :-  Buy = 1, C = C1 + C2, P = P1 + P2,
                          call_option(Buy, S, C1, E, P1),
                          put_option(Buy, S, C2, E, P2).
```

The following rule defines the best profit that can be made from a straddle.

```
best_straddle(C, E, P) :- minimize(straddle(S, C, E, P), -P).
```

5.5 Summary

Modelling in a CLP language is quite different to programming in a traditional programming language and requires an ability to reason about a problem in terms of constraints. At first, this may seem both difficult and unintuitive. However, because the resulting program uses constraints rather than Boolean tests and assignments, it is considerably more flexible than a program written in a conventional language, and may be used to answer a wide variety of questions.

We have seen how predicates with multiple rule definitions allow us to model choice, conditional statements and fixed tables of data (that is, relations). Recursion allows us to model more complex problems in which the number of constraints and their form depends on the input parameters of the problem. Thus recursion provides the functionality of the more standard iteration constructs of traditional programming languages. For the novice programmer, recursion is often the largest stumbling block. Knowledge of a functional language may help, but the best way to learn is by programming.

With experience, a constraint programmer is able to immediately define their problem in constraint terms. The most important words of advice are—practice, practice, and then practice some more.

5.6 Practical Exercises

Unfortunately, the $CLP(\mathcal{R})$ system does not provide any minimization constructs, while both SICStus Prolog and ECLiPSe provide minimization but only for finite domain constraints. Therefore, none of the examples of minimization given in this chapter are directly executable. The use of minimization in both SICStus Prolog and ECLiPSe requires minimization literals `minimize(G, E)` to be used in such a way that any answer to the goal G fixes E to a particular value. Only one minimal answer is returned. In Chapter 9 we show how to implement minimization as defined in this chapter using the library functions of these languages.

P5.1. Try the goal corresponding to the second question about crossing the river in Section 5.1. Can you explain the answer? Note that the faster the rowing speed of the traveller the better the chance of crossing the river. Can you give a different goal which can give a useful answer to the second question?

P5.2. Actually in the wet season there are no crocodiles and the traveller does not need to travel directly across the river. She may set out at an angle θ to the direction straight across (this could be negative if she sets off heading downstream). This is illustrated in Figure 5.5. Give a model of the river crossing constraint which takes into account this new freedom. Using this model, determine how wide the river must be if she rows at 1.5 m/s at an angle of $\pi/6$ upstream, the river flows at 1 m/s and she lands 20 m downstream from the setting off point.

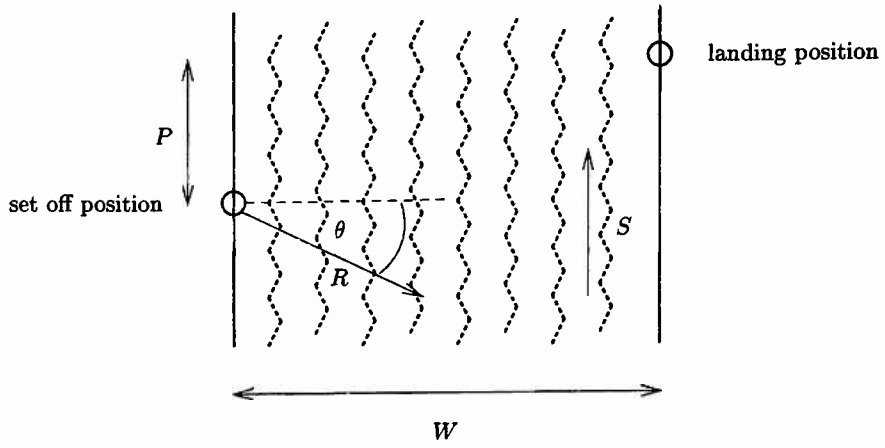

Figure 5.5 River crossing.

P5.3. A spring with force constant k exerts a force of kx if compressed by distance x. It exerts no force if it is not compressed. Write a program defining the relationship between a force constant, compression distance and exerted force for a spring.
A spring holder is shown in Figure 5.6 where the length of the springs are $L1$ and $L2$ and the width of the spring holder is L. Given that the spring constants are $K1$ and $K2$ respectively, define a predicate `stable` so that the user-defined constraint `stable(L, L1, K1, L2, K2, W, X)` holds when X is any distance from the left wall of the placement of the left hand side of an object of width W so that the object will be stable, that is, the force on it from both springs will be the same.
Find where an object of width 10 cm will be stable if the left spring is 10 cm long and has spring constant 1 N/cm and the right spring is 6 cm long and has spring constant 2 N/cm, assuming the gap is 20 cm wide. Find where an object of width 2 cm will be stable, in the same circumstances.

P5.4. A butterfly combination of options involves a call of cost $C1$ at exercise price $E1$, and another call of cost $C3$ at a higher exercise price $E3$, as well as selling two calls for $C2$ each at exercise price $E2$ between the other two exercise prices. A butterfly combination is only sensible if it can make a profit. Write a program to determine which butterfly combinations are sensible. Test the following combinations:

(a) $C1 = 100 \wedge E1 = 4 \wedge C2 = 200 \wedge E2 = 6 \wedge C3 = 300 \wedge E3 = 8$,

(b) $C1 = 100 \wedge E1 = 3 \wedge C2 = 200 \wedge E2 = 7 \wedge C3 = 300 \wedge E3 = 8$,

(c) $C1 = 100 \wedge E1 = 1 \wedge C2 = 300 \wedge E2 = 5 \wedge C3 = 100 \wedge E3 = 6$.

P5.5. Write a predicate using `minimize` to determine the maximum loss for a butterfly combination defined as in the previous question. Find the maximum loss for each of the combinations in the previous question.

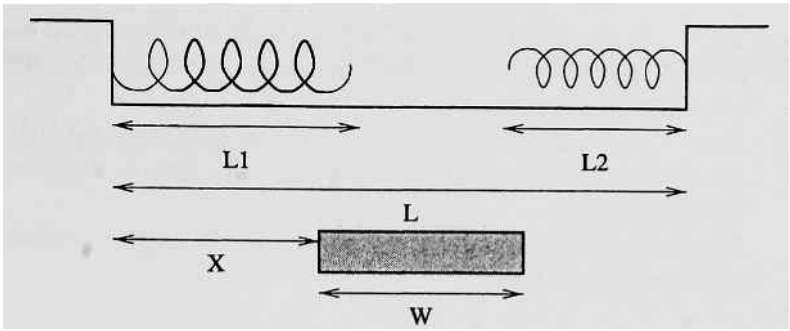

Figure 5.6 A spring holder.

P5.6. Write rules which define the following family relationships:

(a) X is a grandparent of Y,

(b) X is an uncle of Y.

P5.7. The relation `flight(S,E,D)` details the flights from a start city S to an end city E with distance D.

```
flight(melbourne, sydney, 1000).
flight(melbourne, perth, 4000).
flight(perth, singapore, 5000).
flight(sydney, singapore, 8500).
flight(melbourne, cairns, 4000).
flight(sydney, cairns, 3200).
flight(cairns, singapore, 4900).
flight(singapore, london, 10000).
flight(perth, bahrain, 8000).
flight(bahrain, london, 6500).
```

Write a program defining the predicate `path(S, E, D)` which gives the distance D travelled from start city S to end city E using the flights described above. For example, given the goal

```
path(melbourne, singapore, D).
```

it should return answers $D = 9500$, $D = 9000$ and $D = 9900$ corresponding to flying via Sydney, Perth or Cairns respectively.

P5.8. Write a predicate `shortest_path(S, E, D)`, using `minimize`, that finds the shortest possible distance D that can be covered travelling from start city S to end city E using the flights described in Exercise P5.7.

P5.9. On the first day of Christmas, traditionally one gives one's true love one object, on the second day $1+2$ objects, on the third day $1+2+3$ objects. Write a program to define the predicate `gifts(N, T)` where T is the total number of gifts given in the first N days of Christmas.

P5.10. In the drinking game **CLP** one must drink one glass every time a number is reached which is divisible by 7 or divisible by 5, unless the previous drink was taken less than 8 numbers ago.
Write a predicate `divides(N,D)` which holds if D exactly divides N assuming both are positive integers. (Note this is trivial with integer arithmetic constraints, but not so for reals).
Write a predicate `drinks(N, C)` that counts the number of drinks C that are taken in counting from 1 to N.

5.7 Notes

Modelling of real-world problems using constraints is not a new activity. Mathematicians have been doing this since the very beginning of mathematics. Any introductory operations research text discusses modelling using arithmetic constraints.

The programming language $CLP(\mathcal{R})$ used in this chapter is essentially the CLP language developed by Jaffar and his students. One of the first CLP languages, it was introduced in [61, 69] and is described more fully in [75].

Using facts to represent tables and simple rules for querying these tables is a well-known programming technique from the logic programming community, where it is called (for obvious reasons) "database programming." For further examples, the reader is referred to any introduction to Prolog, for instance [125]. The relationship between CLP and databases will be explored more fully in Chapter 11.

The options trading modelling example is based on work of Lassez *et al* [82] who have developed a sophisticated expert system in $CLP(\mathcal{R})$ for reasoning about options trading. The example programs given here merely provide a brief introduction to the way the expert system works. Their system is considerably more powerful and employs nonlinear numerical modelling as well as rules modelling the strategies of option traders.

Translating iteration into recursion is a well known method for writing iterative programs in either logic or functional programming languages. The program for computing mortgage repayments is taken from [61].

Minimization primitives were first included in the constraint logic programming language CHIP [4]. Although we give only one minimization primitive, usually a number are provided by a CLP system which correspond to different implementations of the minimization search. Unfortunately, most implemented minimization primitives do not agree with the definition given here since they will only ever return a single answer. For more about this see the discussion in Chapter 10.