12

The Prolog Programming Language

We complete the book by a short discussion of an inference mechanism that is very different from the one presented in Chapter 7. It is only applicable to normal logic programs (*nlp*s),[1] and although sound with respect to answer set semantics of *nlp*s, it is not complete and might not even terminate; however, it has a number of advantages. In particular, it is applicable to *nlp*s with infinite answer sets, and it does not require grounding. The algorithm is implemented in interpreters for programming language Prolog. The language, introduced in the late 1970s, is still one of the most popular universal programming languages based on logic. Syntactically, Prolog can be viewed as an extension of the language of *nlp*s by a number of nonlogical features. Its inference mechanism is based on two important algorithms called *unification* and *resolution*, implemented in standard Prolog interpreters. **Unification** is an algorithm for matching atoms; **resolution** uses unification to answer queries of the form "find X such that $q(X)$ is the consequence of an *nlp* Π."

We end this chapter with several examples of the use of Prolog for finding declarative solutions to nontrivial programming problems. Procedural solutions to these problems are longer and much more complex.

12.1 The Prolog Interpreter

We start with defining unification and **SLD resolution** – an algorithm used by Prolog interpreters to answer queries to definite programs (i.e., *nlp*s without default negation). Then we look at an example of computing answers

[1] Recall from Chapter 3 that an *nlp* is a logic program without classical negation and disjunction; i.e., a program consisting of rules of the form

$$a_0 \leftarrow a_1, \ldots, a_m, \ not\ a_{m+1}, \ldots, \ not\ a_n$$

where as are atoms.

to queries to an *nlp* with default negation using **SLDNF resolution**.[2] (The accurate description of the algorithm is nontrivial, but we believe that the example is sufficient to illustrate the process for its practical understanding.)

12.1.1 Unification

The unification algorithm forms the core of resolution. Given a collection of atoms it checks if these atoms can be made identical by substitutions of terms for variables. For instance, atoms $p(X, a)$ and $p(b, Y)$ can be made identical by substitution $\alpha = [X{=}b, Y{=}a]$, which replaces X by b and Y by a; atoms $p(a)$ and $p(b)$ cannot be made identical by any substitution. Unexpectedly this seemingly simple procedure becomes nontrivial in more complex examples. Before giving a general unification algorithm we need the following definitions.

Definition 12.1.1. *A* **substitution** *is a finite mapping from variables to terms. We only consider substitutions α such that for any variable X, X does not occur in $\alpha(X)$.*

A substitution α defines a mapping on arbitrary expressions: For any expression A, $\alpha(A)$ is defined as the result of simultaneously applying α to all occurrences of variables in A. For instance, if $\alpha = [X{=}b, Y{=}a]$ and $A = p(X, Y, f(X))$, then $\alpha(A) = p(b, a, f(b))$.

Definition 12.1.2. *A substitution α is called a* **unifier** *of expressions A and B if $\alpha(A) = \alpha(B)$.*

Definition 12.1.3. *A substitution α is called a* **most general unifier (mgu)** *of A and B if*

1. *$\alpha(A) = \alpha(B)$ (i.e., α is a unifier).*
2. *α is more general than any other unifier β of A and B; i.e., there is a substitution γ such that for any variable X from the domain of α, $\beta(X) = \gamma(\alpha(X))$.*

It is possible to prove that most general unifiers only differ by the names of variables, but we do not do this here for the sake of brevity. Due to this result, we sometimes say "the" mgu instead of "an" mgu.

Here are a few examples to illustrate the definitions.

[2] The NF in SLDNF resolution stands for negation as failure – an alternative name for default negation.

1. Mappings $\beta = [X{=}a, Y{=}a]$ and $\alpha = [X{=}Y]$ are unifiers of expressions $f(X)$ and $f(Y)$. Mapping β says that you can unify $f(X)$ and $f(Y)$ by replacing X by a and Y by a. Mapping α says that you can unify these expressions by simply replacing X in $f(X)$ by Y; it requires one less step. It is easy to see that α is more general than β since $\beta(X) = \gamma(\alpha(X))$ for $\gamma = [Y{=}a]$. In fact (as we show later), α is a most general unifier.
2. Mapping $[X{=}a, Y{=}f(b)]$ is an mgu of expressions $f(X, f(b))$ and $f(a, Y)$.
3. Expressions $f(X)$ and $f(f(X))$ are not unifiable.

Theorem 3. *(Unification Theorem)*
(Herbrand, Robinson, Martelli, and Montanari)
There exists an algorithm that for any two atoms produces their most general unifier if they are unifiable and otherwise reports nonexistence of a unifier.

Proof: Notice that atoms are unifiable only if they start with the same predicate symbol. Therefore it suffices to describe a unification method for atoms $p(t_1, \ldots, t_n)$ and $p(s_1, \ldots, s_n)$; i.e., to find a unifier for a set of equations

$$S_0 = \{t_1{=}s_1,\ \ldots,\ t_n{=}s_n\}.$$

This can be done by the following algorithm. For the purpose of this algorithm, treat constants as function symbols of arity 0. Nondeterministically select the appropriate form of equation from the following table and perform the associated action:

	Equation	**Action**
(1)	$f(t_1, \ldots, t_n) = f(s_1, \ldots, s_n)$	replace by the set $\{t_i = s_i : i \in [1..n]\}$
(2)	$f(t_1, \ldots, t_n) = g(s_1, \ldots, s_m)$	stop with failure
(3)	$X = X$	delete the equation
(4)	$t = X$ where t is not a variable	replace by $X = t$
(5)	$X = t$ where t is not X, and X has another occurrence in the set of equations	if X occurs in t, then stop with failure else replace X by t in every other equation

Let $X_1, \ldots, X_n$ be the set of variables occurring in S_0. We now show that the algorithm stops with failure or produces a substitution of the form

$[X_1{=}t_1, \ldots, X_n{=}t_n]$. The key is to show that the algorithm always terminates.

Consider a trace of the algorithm consisting of sets $S_0, S_1, S_2, \ldots$ where S_i is obtained from S_{i-1} by an application of one of the rules in the table. Notice that a successful application of rule (5) to variable X_i reduces the number of occurrences of X_i to one and that application of no other rule increases this number. Since S_0 contains a finite number of variables and since rule (5) is not applicable to a set with no multiple occurrences of variables, we can conclude that rule (5) can only occur in a trace a finite number of times. Let S_n be the set obtained by the last application of rule (5). Since an application of rule (1) to a set S decreases the number of occurrences of function symbols in S and no other rule except (5) increases this number, there can only be a finite number of applications of rule (1) in a trace after S_n. A similar argument can be used for other rules, and hence any trace of the algorithm is finite. From the definition of the algorithm's rules it is clear that for every rule r, substitution α is a unifier of a set E of equations iff α is a unifier of the result of applying r to E. This implies that if the two atoms are unifiable, the last collection of equations is their most general unifier. As mentioned earlier, such a unifier is unique modulo renaming of variables.

Let's apply the algorithm to our previous examples.

Example 12.1.1.

- Unify $p(f(X))$ and $p(f(Y))$.

We record the trace of the algorithm by listing equations in each set formed by the iterations.

$$S_0 \ \{\, f(X) = f(Y)$$

Rule (1) gives us

$$S_1 \ \{\, X = Y$$

No other rules are applicable. Hence, S_1 is an mgu of $p(f(X))$ and $p(f(Y))$.

Example 12.1.2.

- Unify $p(f(X, f(b)))$ and $p(f(a, Y))$.

$$S_0 \ \{\, f(X, f(b)) = f(a, Y)$$

Rule (1) gives us

$$S_1 \ \begin{cases} X = a \\ f(b) = Y \end{cases}$$

Rule (4) requires that we flip the second equation:

$$S_2 \begin{cases} X = a \\ Y = f(b) \end{cases}$$

S_2 is the mgu.

Example 12.1.3.

- Unify $p(f(X))$ and $p(f(f(X)))$.

The algorithm forms equation $X{=}f(X)$ by rule (1) and stops with failure by rule (5).

A few other examples follow.

Example 12.1.4.

- Unify $p(a)$ and $p(b)$.

Applying rule (1) of the algorithm forms equation $a = b$. Since the algorithm treats constants as function symbols of arity 0, it stops with failure by rule (2).

Example 12.1.5.

- Unify $p(X, f(f(a, b), X))$ and $p(g(c), f(Y, Z))$.

(The equation selected for the next step is indicated by an arrow.)

$$S_0 \begin{cases} X = g(c) \\ f(f(a, b), X) = f(Y, Z) \quad \longleftarrow \end{cases}$$

Rule (1) gives us the following set of equations:

$$S_1 \begin{cases} X = g(c) \quad \longleftarrow \\ f(a, b) = Y \\ X = Z \end{cases}$$

Rule (5) applied to $X = g(c)$ tells us to replace other occurrences of X by $g(c)$:

$$S_2 \begin{cases} X = g(c) \\ f(a, b) = Y \\ g(c) = Z \quad \longleftarrow \end{cases}$$

Rule (4) makes $g(c) = Z$ into $Z = g(c)$:

$$S_3 \begin{cases} X = g(c) \\ f(a,b) = Y \\ Z = g(c) \end{cases} \longleftarrow$$

and $f(a,b) = Y$ into $Y = f(a,b)$:

$$S_4 \begin{cases} X = g(c) \\ Y = f(a,b) \\ Z = g(c) \end{cases}$$

The equations in S_4 describe the mgu.

12.1.2 SLD Resolution

The Prolog interpreter uses resolution-based algorithms as its method for finding answers to queries to a logic program. In this section we define **SLD resolution** used by Prolog interpreters to answer queries to definite programs. We give definitions of the basic notions of SLD resolution – SLD inference rule and SLD derivation. Theorems 4 and 5 show soundness and completeness of SLD derivation with respect to answer-set semantics of definite programs, thereby showing the correctness of answers to queries derived using SLD derivation.

We start with basic definitions:

Definition 12.1.4. *Let $Q = [q_0, \ldots, q_m]$ (where the qs are atoms) be a* **query** *to a definite program Π. An* **answer** *to Q by Π is a substitution α such that*

$$\Pi \models \forall\, \alpha(Q)$$

(i.e., every ground instance of $\alpha(Q)$ belongs to the answer set of Π).

For example, consider Π_1:

$$p(a).$$
$$q(a).$$
$$q(b).$$

and query $Q = [p(X), q(X)]$. An answer to Q by Π is $[X{=}a]$. You can see that the ground instance of $\alpha(Q)$ is $\{p(a), q(a)\}$ and that it belongs to the answer set of Π.

Now we describe the SLD resolution method for answering queries to definite programs. For the purpose of our resolution algorithm, we represent the original query Q by a rule

$$yes(X_1, \ldots, X_k) \leftarrow Q \tag{12.1}$$

where $X_1, \ldots, X_k$ is the list of variables of Q and yes is a new predicate symbol not appearing in the program. (In our later discussion we refer to rules containing the yes predicate symbol in the head as **query rules**.) It is easy to see that α is an answer to a query Q by Π_1 iff α is an answer to a query $yes(X_1 \ldots, X_k)$ by $\Pi_1 \cup$ (12.1).

A solution α to a query Q is then found by a series of consecutive transformations of the initial query rule, which leads to a single atom $yes(t_1, \ldots, t_k)$ where $\alpha = [X_1{=}t_1, \ldots, X_k{=}t_k]$.

For example, consider program Π_2 consisting of three rules

$$p(X, Y) \leftarrow q(X, b), r(f(X), Y). \tag{Π_2:1}$$

$$q(a, b). \tag{Π_2:2}$$

$$r(f(a), b). \tag{Π_2:3}$$

and query $Q = [p(X, Y)]$. The initial query rule, denoted by r_1, has the form

$$yes(X, Y) \leftarrow p(X, Y). \tag{r_1}$$

Rule (Π_2:1) is used to transform r_1 into another query rule r_2 of the form

$$yes(X, Y) \leftarrow q(X, b), r(f(X), Y). \tag{r_2}$$

The new rule is obtained by replacing $p(X, Y)$ by the body of (Π_2:1). Intuitively it is clear that if α is a solution to query $yes(X, Y)$ with respect to program $\Pi_2 \cup \{r_2\}$, then it is a solution to the same query with respect to program $\Pi_2 \cup \{r_1\}$. The second rule of Π_2 is used to transform r_2 into the next query rule r_3

$$yes(a, Y) \leftarrow r(f(a), Y). \tag{r_3}$$

This time the transformation is slightly more involved. It consists of two steps. During the first step, query $q(X, b)$ is unified with $q(a, b)$ – the head of rule (Π_2:2). The corresponding substitution is applied to the whole rule, transforming it into $yes(a, Y) \leftarrow q(a, b), r(f(a), Y)$. The second step replaces $q(a, b)$ by the body of rule (Π_2:2) – in our case the empty set. Finally, a similar transformation that uses the last rule (Π_2:3) is applied to transform r_3 into r_4:

$$yes(a, b). \tag{r_4}$$

Clearly $\alpha = [X{=}a, Y{=}b]$ is a solution of Q with respect to the original program Π_2.

In general, each step of this process is accomplished by the following rule called the **Resolution Rule**:

Definition 12.1.5. *(Resolution Rule)*
Consider a query rule r_q

$$yes(t_1, \ldots, t_k) \leftarrow q_0, \Delta$$

where q_0 *is the first atom in the body and* Δ *represents the rest of the query atoms, and a program rule* r_p

$$p_0 \leftarrow \Gamma$$

such that q_0 *and* p_0 *are unifiable and* r_q *and* r_p *do not have common variables.*

Rule r

$$\alpha(yes(t_1, \ldots, t_k) \leftarrow \Gamma, \Delta).$$

where α *is the mgu of* p_0 *and* q_0 *is called the* **resolvent** *of rules* r_q *and* r_p. *A triple* $\langle r_q, r_p, r\rangle$ *is referred to as the* **SLD resolution inference rule** *with premises* r_q, r_p *and conclusion* r.

It can be shown that if α is a solution to query $yes(X, Y)$ with respect to program $\Pi \cup \{r_q\}$, then it is a solution to the same query with respect to program $\Pi \cup \{r\}$.

The following is an example of the application of the resolution inference rule. Consider a query rule r_q,

$$yes(X, Y) \leftarrow p(X, Y), s(X)$$

and a program Π_3 consisting of a single rule r_p,

$$p(a, Y_1) \leftarrow r(f(a), Y_1).$$

The first rule, r_q, has the form

$$yes(X, Y) \leftarrow q_0, \Delta$$

where q_0 is $p(X, Y)$ and Δ is $s(X)$. The second, r_p, is of the form

$$p_0 \leftarrow \Gamma$$

where p_0 is $p(a, Y_1)$ and Γ is $r(f(a), Y_1)$. Clearly, the two rules have no common variables, and $\alpha = [X{=}a, Y{=}Y_1]$ is an mgu of q_0 and p_0. Hence the resolvent of r_q and r_p is r:

$$yes(a, Y_1) \leftarrow r(f(a), Y_1), s(a).$$

Now we define the notion of SLD derivation, which serves as the basis of the resolution algorithm.

Note that a rule r_2 obtained from a rule r_1 by renaming its variables is called a **variant** of r_1.

Definition 12.1.6. *(SLD Derivation)*
An **SLD derivation** *of Q from Π is a sequence $r_0, \ldots, r_n$ of query rules such that*

- *$r_0 = \{yes(X_1, \ldots, X_k) \leftarrow Q\}$ where $X_1, \ldots, X_k$ is the list of variables in Q.*
- *r_i is obtained by resolving r_{i-1} with a variant r of some rule of Π such that r and r_{i-1} have no common variables.*
- *r_n has the empty body.*

The following theorems explain the importance of SLD derivation.

Theorem 4. *Soundness of SLD Derivation*
If $yes(t_1, \ldots, t_k)$ is the last rule in the SLD derivation of $Q(X_1, \ldots, X_k)$ from Π, then the substitution $[X_1{=}t_1, \ldots, X_k{=}t_k]$ is an answer to Q by Π.

Theorem 5. *Completeness of SLD Derivation*
If a ground query $Q(c_1, \ldots, c_k)$ is true in the answer set of Π, then there is an SLD derivation of $Q(X_1, \ldots, X_k)$ from Π with the last rule $yes(t_1, \ldots, t_k)$, such that $Q(c_1, \ldots, c_k)$ is a ground instantiation of $Q(t_1, \ldots, t_k)$.

Clearly the sequence of rules r_q, r_p, r given earlier is an SLD derivation of query $[p(X, Y), s(X)]$ from program Π_3. Similarly, the sequence r_1, r_2, r_3 of rules illustrates the derivation of query $p(X, Y)$ from program Π_2. There is a subtle point we need to make here. Even though r_1, r_2, r_3 is an SLD derivation, our explanation of its construction is not, strictly speaking, accurate. A careful reader will notice that, in this construction, something similar to the resolution rule was applied to rules (Π_2:1) and r_1 that have common variables – this action is prohibited by the definition of SLD derivation. Of course, the problem can be easily remedied by renaming variables of one of the rules. So why does the definition include such a

requirement? The next example gives an answer to this question and shows that such renaming is indeed necessary.

Consider a program Π_4 that consists of the rule

$$p(f(X)).$$

Suppose we have query $p(X)$. If we ignore the requirement and attempt to apply the resolution to the program's rule and the first query rule

$$yes(X) \leftarrow p(X). \qquad (r_1)$$

we fail. (This, of course, happens because $p(X)$ and $p(f(X))$ are not unifiable.) However, $p(X)$ and $p(f(Y))$ are unifiable. Resolving query rule r_1 with variant $p(f(Y))$ of the program rule produces resolvent r_2

$$yes(f(Y)). \qquad (r_2)$$

Since this rule has an empty body, r_1, r_2 is our SLD derivation. The answer to query $p(X)$ is substitution $[X{=}f(Y)]$. Without renaming variables we would fail to produce this answer.

12.1.3 Searching for SLD Derivation – Prolog Control

An algorithm implemented in the interpreter for definite Prolog programs can be viewed as a depth-first search for an SLD derivation of Q from Π in the space of "candidate derivations." In this search a "candidate rule" r_i of this derivation is always obtained by resolving the candidate rule r_{i-1} with the topmost available rule of Π. For example, let Π be a program consisting of the following three rules:

$$p(a). \qquad (1)$$

$$p(b). \qquad (2)$$

$$q(b). \qquad (3)$$

To answer query $Q(X) = [p(X), q(X)]$ the Prolog interpreter creates the first query rule r_0

$$yes(X) \leftarrow p(X), q(X) \qquad (r_0)$$

and resolves it with the topmost rule (1) of the program. The body, $q(a)$, of the resulting query rule

$$yes(a) \leftarrow q(a) \qquad (r_1)$$

is, however, not unifiable with any rule of the program. Hence, no further resolution is possible. Since the body of (r_1) is not empty, the algorithm **backtracks**. Now (r_0) is resolved with rule (2) because it is the first untried rule of the program. The resulting query rule

$$yes(b) \leftarrow q(b) \qquad (r_2)$$

is further resolved with rule (3), which obtains

$$yes(b). \qquad (r_3)$$

Clearly, $(r_0), (r_2), (r_3)$ is an SLD derivation, and the answer to our query is

$$X = b.$$

The use of depth-first search allows the Prolog interpreter to find answers without using a large amount of memory (as opposed to the use of breadth-first search). It is, however, responsible for the nontermination of programs, which causes substantial problems for beginning Prolog programmers.

Consider, for instance, a program

$$\Pi_1 \begin{cases} p \leftarrow p. \\ p. \end{cases}$$

and a query p. Clearly the answer to the query according to Prolog semantics is *yes*. However, it is not difficult to see that the search mechanism implemented in the Prolog interpreter cannot produce this answer. After producing the query rule

$$yes \leftarrow p$$

the interpreter resolves it with the topmost possible rule $p \leftarrow p$ of the program. The result is again

$$yes \leftarrow p.$$

The interpreter is unable to find its way out of this loop.

There, of course, is an SLD derivation of a rule

$$yes$$

that can be obtained by resolving the query rule with the second rule of the program, but this derivation is never found.

The same query to the semantically equivalent program

$$\Pi_2 \begin{cases} p. \\ p \leftarrow p. \end{cases}$$

will return *yes*, but will go into the loop if the interpreter is forced to backtrack.

The limited control strategy of Prolog does not allow it to be viewed as a fully declarative language. If it were, termination properties of its programs would not be dependent on the rules' order or other semantically equivalent transformations.

There are modifications of control strategies implemented in Prolog interpreters that try to alleviate this problem. For instance, the interpreter of the XSB logic programming system tables information about the query-answering process. Given the program Π_1 described earlier and a query p, the XSB interpreter resolves the query rule with the first rule of the program and records this fact in a table. If the backtracking requires resolution of the same query rule (as in our example), XSB will consult the table, discover that this query rule has already been resolved with the first rule of Π_1 and will not attempt to repeat the process. Instead it will resolve the query rule with the second rule p of Π_1, obtaining the desired answer *yes*.

If the program contains variables and negation, the tabling mechanism is deeply nontrivial, but even in this case XSB manages to avoid a large number of potential loops. It is important to realize, however, that the complete elimination of loops is impossible. This follows immediately from the fact that Prolog is a universal programming language and can therefore represent any algorithm. Hence, according to the famous undecidability result by Turing, it is impossible to design an algorithm capable of detecting all the loops in an arbitrary program.

12.1.4 SLDNF Resolution

In this section we discuss the Prolog interpreter for programs containing default negation *not*. Let us first expand the definition of a query from the section on SLD resolution. Now by a **query** we mean a sequence $Q = [q_0, \ldots, q_m]$ where qs are atoms or atoms preceded by the default negation *not*. To answer queries for *nlp*s with *not*, the Prolog interpreter implements a version of an extension of SLD resolution. The precise definition of this algorithm, called SLDNF resolution, is somewhat complex so we explain it by way of example.

Let us consider a program Π consisting of the following four rules:

$$p(a). \qquad (\Pi{:}1)$$

$$p(b). \qquad (\Pi{:}2)$$

$$q(a). \tag{Π:3}$$

$$r(X) \leftarrow p(X),\ not\ q(X). \tag{Π:4}$$

and a query $r(X)$ represented as a rule:

$$yes(X) \leftarrow r(X). \tag{r_0}$$

The resolution rule from Definition 12.1.5 can be naturally expanded to rules in which atoms in the body can be preceded by *not*. SLDNF resolution uses this rule to resolve query r_0 with program rule (Π:4). The resulting query rule,

$$yes(X) \leftarrow p(X),\ not\ q(X) \tag{r_1}$$

is resolved with program rule (Π:1) to obtain the next query rule

$$yes(a) \leftarrow\ not\ q(a). \tag{r_2}$$

The first atom in the body of this rule is preceded by default negation, and hence the resolution rule is not applicable. To proceed we need a new inference rule called **Negation as Finite Failure (NAF)**: *A query rule*

$$yes(t_1, \ldots, t_k) \leftarrow\ not\ q, \Delta$$

where q is a ground atom derives a query rule

$$yes(t_1, \ldots, t_k) \leftarrow \Delta$$

if SLDNF resolution can consider all possible ways to prove q and demonstrate that all of them fail.[3]

If we go back to our example we see that SLDNF's attempt to prove $q(a)$ immediately succeeds. Neither the resolution rule nor NAF can use rule r_2, and hence the algorithm is forced to backtrack to rule (r_1). By resolving (r_1) and (Π:2) it finds another candidate rule:

$$yes(b) \leftarrow\ not\ q(b). \tag{r_3}$$

It is easy to see that the query $q(b)$ cannot be resolved with anything else and cannot therefore be proven by SLDNF resolution. At this point the resolution algorithm uses NAF and removes $not\ q(b)$ from the body of (r_3),

[3] This description is admittedly vague. Since NAF is used in the definition of SLDNF resolution, the description is circular and requires clarification. It is exactly this problem that makes the precise description of SLDNF resolution nontrivial. In most cases, however, this circularity does not cause any problem.

producing the rule

$$yes(b) \qquad (r_4)$$

which contains the answer to the initial query.

Notice that the algorithm only works if the corresponding negated query is ground. Otherwise the interpreter is said to **flounder**, and its actual behavior depends on the implementation.

It can be shown that *if the algorithm does not flounder then it is sound with respect to the answer set semantics*. It is also shown to be complete for a comparatively large class of programs.

12.2 Programming in Prolog

In this section we briefly discuss data representation in Prolog and give examples of using the language for solving several nontrivial programming problems. It is not our purpose to teach Prolog programming; many good books have already been written on that subject. Rather, we want to give readers a feel for the language and the types of solutions its use can produce. We use actual Prolog programming systems (such as SICStus, SWI, GNU, etc.) that include several "meta-level" constructs that do not fit into the syntax of *nlp* rules we discussed so far. We give intuitive explanations of these Prolog features without going deeply into the precise definition of the semantics so as not to digress from the main goal with a lengthy discussion. As examples of Prolog use, we present solutions to two classical problems. The first one, referred to as the *parts inventory problem*, consist of finding the type and quantity of basic parts needed to assemble a given number of products (in this case, bicycles). The second is that of *computing derivatives of polynomials.*

Note that many versions of Prolog use the built-in operator `\+` to denote default negation.

12.2.1 The Basics of Prolog Programming

We start with a simple program stating that two terms, a and b, satisfy property p.

```
1 % First Prolog Program
2 p(a).
3 p(b).
```

Let us assume that the program is stored in file `first.pl`.

Let's run the program in Prolog. You can use any Prolog system mentioned earlier. After installing the system, invoke it from the command line. (Make sure you are in the directory where your program resides.) You should see a question mark prompt. For instance, if you type

```
swipl
```

for SWI Prolog, this prompt appears:

```
?-
```

You then need to tell the interpreter to load the program. This is called "consulting" the file. If you use extension `.pl` in naming your file, you can omit it when consulting and just type

```
?- [first].
```

Prolog should respond with a message telling you that it has compiled your program. (If you do not use `.pl` but some other extension, say `.ext`, you have to type `['first.ext']`.

Next, type in the following query to Prolog and hit return. It will respond with `true` and return you to a Prolog prompt.

```
?- p(a).
true
?-
```

The query $p(X)$ produces the following:

```
?- p(X).
X = a
```

At this point you have a choice. If you hit the return/enter key, Prolog will say `true`, return to the Prolog prompt, and wait for further input. Sometimes, however, you will want to force the interpreter to backtrack to see if there are any more possible answers to the query. If you type in a semicolon, you will do just that. If there is another answer, Prolog will return the first one it finds and, again, wait to see if you want more. In our case we have

```
?- p(X);
X = a ;
X = b ;
false
```

Answer `false` indicates that there is no other correct answer to the query. To exit the Prolog environment, type `halt.`

Consider now program

```
% Second Prolog Program
p(a).
p(f(X)) :-
    p(X).
```

stored in file `second.pl`. If you consult this program and ask it query $p(X)$, Prolog will return `X = a`. If prompted to backtrack, it will return `p(f(a))`, `p(f(f(a)))`, etc. Since the answer set of the program contains an infinite set of answers to our query, it will be up to you to stop the process.

For the next example, consider a directed acyclic graph represented by relation $edge(X, Y)$ and relation $connected(X, Y)$, which holds if there is a path in the graph from node X to node Y.

```
% Graph Connectivity
edge(a,b).
edge(a,c).
edge(b,d).
connected(X,Y) :-
    edge(X,Y).
connected(X,Y) :-
    edge(X,Z),
    connected(Z,Y).
```

The program will respond to query $connected(X, Y)$ as follows:

```
X = a,
Y = b ;

X = a,
Y = c ;

X = b,
Y = d ;

X = a,
Y = d ;

true
```

correctly returning all the pairs satisfying our relation. Note, however, that if we replace the last rule of the program by equivalent rule

```
connected(X,Y) :-
    connected(Z,Y),
    edge(X,Z).
```

and force Prolog to backtrack, it will quickly run out of memory. (In our experiments, Prolog found the four answers, but instead of answering *false*, it returned an error message.) This happens because, due to the control strategy of the Prolog interpreter, it will go into an infinite loop. As we mentioned in Section 12.1.3, Prolog programmers cannot fully rely on the declarative meaning of Prolog statements and need to firmly keep in mind the basic strategy of Prolog control.

In addition to terms, Prolog has another useful datatype called the **list**. A list of objects can be formed by using square brackets. An expression $[a, b, c]$ defines a list of three terms, a, b, and c; [ ] denotes the empty list. A special expression $[F|T]$ denotes a list with head F and tail T. Here F is the first element of the list, and T is the list consisting of the rest of the elements. For instance, list $[a, b, c]$ can be written as $[a|[b, c]]$; list $[a]$ as $[a|[]]$, and so on. This definition is reflected in the unification algorithm. For instance, list $[F|T]$ is unified with a list $[a, b, c]$ by substitution $F{=}a, T{=}[b, c]$. The head and tail notation for lists is very convenient for writing recursive definitions of list properties and relations. For example, the following program tests the membership relation in lists:

```
% Program in1.pl
is_in(X,[X|_]).
is_in(X,[Y|T]) :-
    is_in(X,T),
    diff(X,Y).
eq(X,X).
diff(X,Y) :-
    \+ eq(X,Y).  % i.e., not eq(X,Y)
```

In the first rule '_' stands for an arbitrary term and is often called an "anonymous variable." (Any variable except X can be used instead, but it is easier to write '_'.)

The definition can be used to check if a ground term t belongs to a ground list l. For instance, to check membership of a in $[a, b, c]$ we ask a query $is\_in(b, [a, b, c])$, which, as expected, will be answered by `true`; query $is\_in(d, [a, b, c])$ will be answered by `false`.

In addition to checking if an element belongs to a list, the definition of is_in can be used to find list members. This can be done by a query of the form $is\_in(X, list)$ where $list$ is ground. For example,

```
?- is_in(X,[a,b,a]).
X = a ;
X = b ;
false.
```

Sometimes it is useful to use a more complex nonground term as the first parameter. For instance, to answer query $is\_in([X, b], [[a, b], c, [d, b]])$ Prolog does the necessary matching and responds as follows:

```
?- is_in([X,b],[[a,b],c,[d,b]]).
X = a ;
X = d ;
false.
```

These examples show that the program is capable of answering queries whose parameters are an arbitrary term and a ground term, respectively. (In fact, it can be proven that such queries are always correctly answered by the program.) But such good behavior is not guaranteed if the second parameter is not ground.

For instance, a query $is\_in(a, X)$ will be answered by `X = [a|_A]?`, read as "any list that starts with a'. This is, of course, a correct answer, but if prompted for another answer, the program will go into an infinite loop (which, most likely, will be indicated by a message mentioning "insufficient memory").

To avoid using a Prolog definition for answering unintended queries, Prolog programmers are strongly advised to mention the type of expected queries in documentation. In what follows we use symbol $+$ if the corresponding parameter must be ground, $-$ if it contains variables, and $?$ if the parameter is arbitrary (i.e., if it may or may not contain variables). For instance, the definition of is_in in program `in1.pl` would be preceded by a comment line

```
% is_in(?,+).
```

which indicates that the second parameter of the query formed by is_in must be ground. Sometimes a mapping of parameters into $\{+, -, ?\}$ is called the **mode** of a definition.

One might ask why we did not use a definition of is_in given by the following rules:

```
1 % Program in2.pl
2 is_in(X,[X|_]).
3 is_in(X,[Y|T]) :-
4     is_in(X,T).
```

This would work too, but there would be slight differences between the two definitions. For instance, let us run Prolog on the new program with query $is\_in(X, [a, b, a])$

```
?- is_in(X,[a,b,a]).
X = a ;
X = b ;
X = a ;
false.
```

The last answer is redundant and was not produced by the first program. To understand the difference let us trace the execution of both programs. For simplicity we drop the extension pl and refer to the programs as `in1` and `in2`. On the first call to program `in2`, Prolog instantiates X with a using the first rule. It does so similarly for `in1`. When asked to backtrack in `in2`, Prolog uses the second rule of `in2`, instantiates Y with a and T with $[b, a]$, and calls $is\_in(X, [b, a])$. Backtracking in `in1` is almost identical except the last call is $is\_in(X, [b, a]), diff(X, a)$. Both programs return the second answer $X = b$. One more backtrack shows us the difference: `in2` backtracks to $is\_in(X, [b, a])$ and returns the third answer $X = a$; `in1` backtracks to $is\_in(X, [b, a]), diff(X, a)$. As in `in2`, the call to $is\_in(X, [b, a])$ returns $X = a$, but $diff(a, a)$ fails and `in1` exits without returning a redundant answer.

This happens because the two rules in the definition of `in2` are not mutually exclusive, whereas the corresponding rules of `in1` are. It is good programming practice to keep rules in the definition of a relation mutually exclusive to avoid unintended consequences.

Note also that, although it would be logical to write

```
is_in(X,[Y|T]) :-
    diff(X,Y),  % causes floundering!
    is_in(X,T).
```

we cannot, because doing so would cause floundering as mentioned in Section 12.1.4. (Recall that `\+` is used in the definition of $diff$.) This undesirable behavior of the interpreter can be avoided if we put the is_in predicate first because it would serve to ground the variable and $diff$ would

no longer be called incorrectly. Relation is_in is similar to relation $member$, which is predefined in some Prologs and is part of a list library in others.

There are many other useful relations on lists that have been implemented as part of the Prolog systems in various ways; e.g.,

$length(L, N)$ iff N is the length of list L.
$append(L_1, L_2, L_3)$ iff L_3 is the result of appending L_1 and L_2.

Others can be defined by the programmer; e.g., relation $rm\_dupl(L_1, L_2)$ with mode $rm\_dupl(+, -)$ finds a list L_2 that is obtained from a ground list L_1 by removing duplicate elements. The relation is defined by the following rules:

```
% rm_dupl(+,-)
rm_dupl([],[]).
rm_dupl([X|T], L2) :-
    is_in(X,T),
    rm_dupl(T,L2).
rm_dupl([X|T],[X|T1]) :-
    \+ is_in(X,T),
    rm_dupl(T,T1).
```

The program answers query $rm\_dupl([a, a, b, c, b], X)$ as follows:

```
?- rm_dupl([a,a,b,c,b],X).
X = [a, c, b] ;
false
```

In other words the above definition allows us to find only one answer to that query. Another valid answer, [a,b,c], will not be found. If one desires to get all the answers, one can modify the definition using relation $permutation$, which is available in the list libraries of most Prologs, as follows:

```
% rm_dupl1(+,?)
rm_dupl1(L1,L2) :-
    rm_dupl(L1,L),
    permutation(L,L2).
```

Lists in Prolog can be created with the help of several important 'meta-level' relations. When relation $findall(Obj, Goal, Res)$ is called, it produces a list Res of all the objects Obj that satisfy goal $Goal$. It is a built-in predicate in many Prolog systems.

Consider a knowledge base containing the test grades of students $s1, s2, s3$:

```
grade(s1,98).
grade(s2,45).
grade(s3,99).
```

The following rule defines the number of students who received an A on their test:

```
num_of(a,N) :-
    findall(X, (grade(X,Y), Y>=90), L),
    length(L,N).
```

Note that (G1,G2) denotes a conjunction of goals G1 and G2. Since L is a list and as such may contain duplicate entries, accidental duplication of records in the knowledge base could cause an incorrect answer. To avoid this we can replace $findall$ by relation set_of_all defined as

```
set_of_all(X,G,L) :-
    findall(X,G,L1),
    rm_dupl(L1,L).
```

12.2.2 Parts Inventory Program

In this section we present a more advanced Prolog program that solves the *parts inventory problem*. Any business that makes something from parts that are in turn made from other parts is faced with the need to know how many of what part to keep in stock. The program in this section deals with a small number of bicycle parts, but it can be easily expanded to large, realistic examples while preserving the program structure. We simply need to add more facts to our knowledge base and tailor them to the business at hand. The following problem domain and program (with minor changes) came from William F. Clocksin and Christopher S. Mellish's book *Programming in Prolog: Using the ISO Standard* Clocksin and Mellish (2003), Fifth Edition, 2003, Ch. 3: Using Data Structures, pp. 64–66, and is used with kind permission of Springer Science+Business Media.

We are given a knowledge base that consists of a collection of basic bicycle parts together with the list of compound parts followed by their immediate components. We need to find the type and quantity of each basic part required to assemble more-complex ones. More precisely, we assume that our knowledge base contains a complete list of basic parts; e.g.,

```
basic_part(rim).
basic_part(nut).
basic_part(spoke).
```

```
basic_part(frame).
basic_part(brakes).
basic_part(tire).
```

By a lists of parts we mean a list of the form $[[P_1, K_1] \ldots [P_n, K_n]]$ where Ps are names of parts, if $i \neq j$ then $P_i \neq P_j$, and Ks are positive integers. $[P, K]$ is read as "K items of a part P." If all Ps of L are basic parts, we say that L is a list of basic parts. We also assume that the knowledge base contains a complete collection of compound parts together with information about the names and quantities of their immediate components; e.g,

```
components(bike,[[wheel,2],[frame,1],[steering,1]]).
components(wheel,[[spoke,4],[rim,1],[tire,1],[nut,5]]).
components(steering,[[brakes,2],[nut,10]]).
```

The first parameter of this relation is a compound part P, whereas the second is a list L of its immediate subparts (i.e., $[P_i, K]$ is in L iff P has exactly K immediate subparts of type P_i).

We are interested in defining a relation $parts\_required(N, P, L)$ that satisfies the following two conditions: For any positive integer N and a part P

1. there is a list L such that $parts\_required(N, P, L)$ is true;
2. if $parts\_required(N, P, L)$ is true then L is a list of basic parts required to assemble N parts of type P.

We start our design by defining $parts\_required(N, P, L)$ with the mode $parts\_required(+, +, -)$. The definition is given by two rules

```
parts_required(N,P,[[P,N]]) :-
    basic_part(P).
parts_required(N,P,L) :-
    components(P,C),
    parts_for_list(C,L1),
    times(N,L1,L).
```

where the relation $parts\_for\_list(C, L)$ satisfies the following conditions. For any list C of parts,

1. there is an L such that $parts\_for\_list(C, L)$ holds;
2. if $parts\_for\_list(C, L)$ holds, then L is a list of basic parts required to assemble parts from C, and relation $times(N, L_1, L_2)$ constructs list L_2 by replacing every element $[P, M]$ of L_1 by $[P, N * M]$.

To define $parts\_for\_list(C, L)$ with mode $parts\_for\_list(+, -)$ we need the following rules:

```
parts_for_list([ ],[ ]).
parts_for_list([[P,K]|T],L) :-
    parts_required(K,P,Lp),
    parts_for_list(T,Lt),
    combine(Lp,Lt,L).
```

Here $combine(L_1, L_2, L)$ is a new relation that appends two lists L_1 and L_2 of parts, transforms the result into a list of parts, and stores it in L.

To define relation $combine$, we first introduce an auxiliary relation $insert\_one(X, L_1, L_2)$, which inserts a list X of the form $[P, N]$ into a list L_1 of parts. The result L_2 should be a list of parts. For example,

```
insert_one([wheel,2],
           [[frame,1],[wheel,3]],
           [[frame,1],[wheel,5]]).
insert_one([wheel,2],
           [[frame,1]],
           [[wheel,2],[frame,1]]).
```

Here is the relation:

```
% mode insert_one(+,+,-)

insert_one(X, [], [X]).
insert_one([P,N1], [[P,N2] | T],  [[P,N] | T]) :-
    N is N1 + N2.
insert_one([P1, N1], [[P2,N2] | T1], [[P2,N2] | T ]) :-
    diff(P1,P2),
    insert_one([P1,N1], T1, T).
```

(You can see that arithmetic operators are defined in Prolog, as well as assignment (`is`)).

Now we can define relation $combine(L_1, L_2, L)$ such that

1. For every lists L_1 and L_2 of parts there is a list L such that $combine(L_1, L_2, L)$ holds.
2. If $combine(L_1, L_2, L)$ holds, then $[P, N]$ belongs to L iff the list of elements of L_1 and L_2 of the form $[P, K1], \ldots, [P, Km]$ is not empty and $N = K1 + \cdots + Km$; e.g.,

$$combine([[wheel, 2], [spoke, 5]], [[spoke, 1], [wheel, 3]],$$
$$[[spoke, 6], [wheel, 5]])$$

Note that the actual order of elements in the constructed list may differ.

```
% mode combine(+,+,-)

combine([],L,L).
combine([H | T], L1, L) :-
    insert_one(H,L1,L2),
    combine(T,L2,L).
```

To complete the assignment we define the relation $times(N, L_1, L_2)$ that satisfies the following two conditions:

1. For any positive number N and a list L_1 of parts, there is a list L_2 such that $times(N, L_1, L_2)$ holds.
2. If $times(N, L_1, L_2)$ holds, then $[P, K]$ is in L_2 iff $[P, K/N]$ is in L_1.

```
% mode times(+,+,-)

times(_,[],[]).
times(N,[[P,K]|T],[[P,M]|TN ]) :-
    M is K*N,
    times(N,T,TN).
```

Finally, we define $diff(X, Y)$:

```
eq(X,X).
diff(X,Y) :-
    \+ eq(X,Y).
```

To run this program in Prolog, collect all the typewritten code into a file, say `parts.pl`. Launch Prolog from the directory that your code is in, consult the program, and ask it how many basic parts are required to assemble 10 bicycles or, say, 100 wheels. Here is a sample run:

```
?- [parts].
% parts compiled 0.00 sec, 7,504 bytes
true.

?- parts_required(10, bike, L).
L = [[spoke, 80], [rim, 20], [tire, 20],
     [frame, 10], [brakes, 20], [nut, 200]] .

?- parts_required(100, wheel, L).
L = [[spoke, 400], [rim, 100],
     [tire, 100], [nut, 500]] .

?- halt.
```

12.2.3 (*) Finding Derivatives of Polynomials

Here is another example of using Prolog to solve complex problems with elegance. In this section we write a program computing derivatives of polynomials – a typical and important example of *symbolic computation.* By polynomials we mean functions constructed from a variable x, integers, function symbols $+$,$-$,$*$, exponent `^`, and parentheses. Note that X^N is defined for a nonnegative integer N only. Among other things the program is aimed to illustrate the use of basic Prolog data structures – terms and lists. Those who use Sictus Prolog need to include a library module for list operations. For SWI Prolog this is not needed.

The derivative is computed using relation $derivative(F, DF)$ that, given a polynomial F, computes the canonical form DF of its derivative. For instance, both queries

```
derivative(3*x^2+4*x+1,DF).
derivative(4*x+3*x^2+1,DF).
```

are answered by

```
DF = 6*x+4
```

Queries

```
derivative(3*x^2-4*x+1,DF).
derivative((2*x+3)^2,DF).
```

are answered by

```
DF = 6*x-4
```

and

```
DF = 8*x+12
```

respectively.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% COMPUTING DERIVATIVES of POLYNOMIALS
% For the purpose of this program polynomials are
% functions constructed from a variable x, integers,
% function symbols +, -, * and ^, and parentheses.
% Exponentiation X^N is defined for non-negative integer
% N. The derivative is only computed once. Multiple
% calls can lead to an infinite loop.

```

```
:-use_module(library(lists)). % not needed for SWI
                              % Prolog
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% derivative(F,DF) computes a derivative DF of a
% polynomial F. The result is given in canonical form.
% type F, DF - polynomials.
% mode derivative(+,-)

derivative(F,DF) :-
    der(F,G),
    simplify(G,DF).
```

The derivative is computed in two steps. First der applies the rules of derivation to F to produce a polynomial G, which is then reduced to its canonical form DF by $simplify$.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% der(F,DF) computes a derivative DF of a polynomial F.
% type F, DF - polynomials.
% mode der(+,-).

der(N,0) :-
    number(N).
der(x,1).
der(-x,-1).
der(F^0,0).
der(F^N, N*F^M*DF) :-
    N>=1,
    M is N-1,
    der(F,DF).
der(E1*E2, E1*DE2 + E2*DE1) :-
    der(E1,DE1),
    der(E2,DE2).
der(E1+E2, DE1+DE2) :-
    der(E1,DE1),
    der(E2,DE2).
der(E1-E2,DE1-DE2) :-
    der(E1,DE1),
    der(E2,DE2).
der(-(E),DE) :-
    der((-1)*E,DE).
```

One can now test the definition of der. Use your favorite Prolog system, consult the program, and ask the query `der(x^2,D)`. Prolog will answer with `D = 2*x^1*1`. Clearly, the answer is correct, but to have a decent output it should be simplified and reduced to canonical form `2*x`.

To simplify the resulting derivative we represent polynomials as lists of the form `[...[Ci,Ni]...]` where `[Ci,Ni]` corresponds to the term `Ci*x^Ni`. Polynomials written in this form are called **l-polynomials**. The derivative DF of F computed by `der(F,DF)` is translated into its equivalent l-polynomial, then written in canonical form, and, finally, transformed back into term form. A list `L = [[C1,N1],...,[Ck,Nk]])` is called the **list-representation** of polynomial `T = C1*x^N1 +...+ Ck*x^Nk`. This change of representation greatly simplifies the reduction to canonical form.

Relation $simplify(P, SP)$

1. transforms polynomial P into equivalent l-polynomial SP;
2. combines like terms;
3. removes terms whose coefficients are 0;
4. sorts terms in decreasing order of exponents; and
5. converts the resulting l-polynomial back to canonical form.

For instance, query

```
simplify(3*x^2+x^2-2*x^2+0*x^3+x^4,SP).
```

is answered by

```
SP = x^4+2*x^2
```

The complete definition follows.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% simplify(P,SP) iff SP is a canonical form of P
% type: P, SP - polynomials
% mode simplify(+,?)

simplify(P,SP) :-
    transform(P,TP),
    add_similar(TP,S),
    remove_zero(S,S1),
    poly_sort(S1,S2),
    back_to_terms(S2,SP).
```

Next we describe each step in detail. Relation $transform(P, L)$ translates polynomial P into an equivalent l-polynomial L. For instance, query

```
transform(3*x^2+4*x+1,L).
```

is answered by

```
L = [[3,2],[4,1],[1,0]]
```

Here is the definition:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% transform(T,[[C1,N1],...,[Ck,Nk]]) implies that
% T = C1*x^N1 +...+ Ck*x^Nk.
% type: T - polynomial, C - integer, N - nonnegative integer.
% mode: transform(+,?).

transform(C,[[C,0]]) :-
    integer(C).
transform(x,[[1,1]]).
transform(-x,[[-1,1]]).
transform(A*B,L) :-
    transform(A,L1),
    transform(B,L2),
    multiply(L1,L2,L).
transform(A^N,L) :-
     transform(A,L1),
     get_degree(L1,N,L).
transform(A+B,RT) :-
     transform(A,RA),
     transform(B,RB),
     append(RA,RB,RT).
transform(A-B,RT) :-
     transform(A,RA),
     transform(-B,RB),
     append(RA,RB,RT).
transform(-(A),R) :-
     transform((-1)*A,R).
```

The definition uses two new relations, $multiply(L_1, L_2, L)$ and $get\_degree(L_1, N, L)$. The former simply multiplies two polynomials written in list form. For instance, a query

```
multiply([[1,2],[3,4]],[[2,3],[3,2]],Y).
```

is answered by

```
Y = [[2,5],[3,4],[6,7],[9,6]]
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% multiply(L1,L2,L) L is the product of L1 and L2
% type: L1,L2,L - l-polynomials
% mode: multiply(+,+,-)

multiply_one(_,[],[]).
multiply_one([C1,N1],[[C2,N2]|R],[[C,N]|T]) :-
    C is C1*C2,
    N is N1+N2,
    multiply_one([C1,N1],R,T).

multiply([],_,[]).
multiply([H1|T1],T2,T) :-
    multiply_one(H1,T2,R1),
    multiply(T1,T2,R2),
    append(R1,R2,T).
```

The second relation, $get\_degree(L_1, N, L)$, used in $transform$, raises polynomial L_1 to the Nth degree. L is the nonsimplified result of this operation. Both polynomials are written in list form. For instance, query

```
get_degree([[3,2],[1,1]],2,L)
```

is answered by

```
L = [[9,4],[3,3],[3,3],[1,2]]
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% get_degree(L1,N,L) L is L1^N
% type: L1,L - l-polynomials, N - nonnegative integer
% mode: get_degree(+,+,-)

get_degree(L1,0,[[1,0]]).
```

```
get_degree(L1,1,L1).
get_degree(L1,N,L) :-
    N > 1,
    M is N-1,
    get_degree(L1,M,L2),
    multiply(L1,L2,L).
```

Returning to the definition of $simplify$, we define the second relation, $add\_similar(L_1, L_2)$, which simplifies l-polynomial L_1 by combining terms with like degrees. For instance, query

```
add_similar([[9,4],[3,3],[3,3],[1,2]],L)
```

is answered by

```
L = [[9,4],[6,3],[1,2]]
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% add_similar(L1,L2) implies that L2 is the result of
% adding up similar terms of L2.
% type: L1 and L2 are l-polynomials.
% mode: add_similar(+,?)

add_similar([],[]).
add_similar([[C1,N]|T],S) :-
    append(A,[[C2,N]|B],T),
    C is C1+C2,
    append(A,[[C,N]|B],R),
    add_similar(R,S).

add_similar([[C,N]|T],[[C,N]|ST]) :-
    not_in(N,T),
    add_similar(T,ST).
```

The relation $not\_in(N, L)$ holds if l-polynomial L does not contain a term with degree N. For instance, a query

```
not_in(2,[[1,2],[3,4]]).
```

is answered by *no*, whereas a query

```
not_in(1,[[1,2],[3,4]]).
```

returns *yes*.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% not_in(N,L) holds if an l-polynomial L does not
% contain a term with degree N
% type: N - non-negative integer, L l-polynomial
% mode: not_in(+,+)

not_in(_,[]).
not_in(_,[[]]).
not_in(N,[[_,M]|T]) :-
      diff(N,M),
      not_in(N,T).
```

The third relation of $simplify$ is $remove\_zero(L_1, L_2)$. It holds if L_2 is obtained from L_1 by removing terms whose coefficients are 0. For instance, query

```
remove_zero([[3,4],[0,3],[0,2],[3,0]],X).
```

is answered by

```
X = [[3,4],[3,0]]
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% remove_zero(L1,L2) iff list L2 is obtained from
% list L1 by removing terms of the form [0,N].
% type: L1,L2  l-polynomilas
% mode: remove_zero(+,?)

remove_zero([],[]).
remove_zero([[0,_]|T],T1) :-
    remove_zero(T,T1).
remove_zero([[C,N]|T],[[C,N]|T1]) :-
    diff(C,0),
    remove_zero(T,T1).
```

The next step in our description of $simplify$ is to sort terms in decreasing order of exponents. Sorting is needed for computing the canonical form of the derivative, and it is easier to do it while the derivative is still in list form. Here is an example. Query

```
poly_sort([[3,4],[5,1],[2,5]],X).
```

returns

```
X = [[2, 5], [3, 4], [5, 1]]
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% poly_sort(L1,L2) sorts l-polynomial L1 in decreasing
% order of exponents and stores the result in L2.
% type: L1,L2  l-polynomilas
% mode: poly_sort(+,-)

poly_sort([X|Xs],Ys) :-
     poly_sort(Xs,Zs),
     insert(X,Zs,Ys).
poly_sort([],[]).

insert(X,[],[X]).
insert(X,[Y|Ys],[Y|Zs]) :-
     greater(Y,X),
     insert(X,Ys,Zs).
insert(X,[Y|Ys],[X,Y|Ys]) :-
     greatereq(X,Y).

greater([_,N1],[_,N2]) :-
     N1 > N2.

greatereq([_,N1],[_,N2]) :-
     N1 >= N2.
```

To complete the definition of $simplify$ we define relation $back\_to\_terms(L, T)$, which translates l-polynomial L into its canonical equivalent, T. For instance, a query

```
back_to_terms([[3,2],[2,1],[1,0]],T).
```

is answered by

```
T = 3*x^2 + 2*x + 1
```

The definition of $back\_to\_terms$ uses recursion on the length of l-polynomial L. The base case is an l-polynomial of length 1. Note, that instead of a simple rule

```
back_to_terms([[C,N]],C*x^N)
```

we have seven different rules. This happens because we are not merely translating back to terms, but are also doing some simplification along the way. For instance, instead of translating $[[1,1]]$ as `1*x^1`, we translate it simply as `x`. (We do not need to take care of the case where $C = 0$ because we have already removed these terms.) The last two rules of the definition take care of the inductive step. Relation *append* discussed earlier in this chapter is used to split the list L into list L_1 and a singleton $[[C,N]]$. If $C > 0$ then the translation of L is the sum of the translations of L_1 and $[[C,N]]$; otherwise, it is the difference of the translations of L_1 and $[[AC,N]]$ where AC is the absolute value of C.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% back_to_terms(L,T) computes a term-representation T
% of L. L is an l-polynomial sorted in decreasing order
% of exponents and not containing terms of the form [0,N].
% type: L l-polynomial, T polynomial.
% mode: back_to_terms(+,?)

back_to_terms([[1,1]],x).
back_to_terms([[-1,1]],-x]).
back_to_terms([[C,0]],C) :-
     integer(C).
back_to_terms([[1,N]],x^N):-
      diff(N,0),
      diff(N,1).
back_to_terms([[-1,N]],-x^N):-
      diff(N,0),
      diff(N,1).
back_to_terms([[C,1]],C*x):-
      diff(C,1),
      diff(C,-1).
back_to_terms([[C,N]],C*x^N):-
     diff(N,1),
     diff(C,1),
     diff(C,-1).
back_to_terms(L,F+T) :-
     append(L1,[[C,N]],L),
     diff(L1,[]),
     C > 0,
```

```
202       back_to_terms(L1,F),
203       back_to_terms([[C,N]],T).
204 back_to_terms(L,F-T) :-
205       append(L1,[[C,N]],L),
206       diff(L1,[]),
207       C < 0,
208       abs(C,AC),
209       back_to_terms(L1,F),
210       back_to_terms([[AC,N]],T).
```

Finally we need

```
211 eq(X,X).
212 diff(X,Y) :-
213     \+ eq(X,Y).
214 abs(X,X) :-
215     integer(X),
216     X >= 0.
217 abs(X,Y) :-
218     integer(X),
219     X < 0,
220     Y is (-1)*X.
```

This completes our program. We hope that the examples of Prolog programs given in this chapter convince the reader that it is a valuable and interesting language in its own right. We encourage those interested in becoming proficient Prolog programmers to continue their study with several of the excellent textbooks currently available on the subject.

Summary

In this chapter we introduced important reasoning algorithms based on unification and resolution and demonstrated their use in interpreters for Prolog. The more general versions of resolution are used in automatic theorem proving and other important areas of logic-based reasoning.

The fact that resolution can be used to find elegant solutions to diverse computational problems is a consequence of the deep relationship between constructive mathematical proofs and algorithms. Recall that the proof is called **constructive** if it demonstrates existence of a mathematical object by exhibiting or providing a method for exhibiting such an object. As a result, an algorithm for building an object is often simply a part of a constructive

proof of the object's existence (or can be automatically extracted from such a proof). That is exactly what is done by the Prolog interpreter, which seeks a constructive logical proof of the existence of an object X satisfying query $q(X)$. The proof uses the resolution and the rules of a program. The answer returned by the interpreter can be viewed as a conclusion reached by a logical reasoner about the truth or falsity of a statement given that reasoner's knowledge about the world. Similar relationships between constructive proofs and computation are used in a number of other systems (see, for instance, Mints and Tyugu (1988), Miglioli, Moscato, and Ornaghi (1988), Constable et al. (1986)).

References and Further Reading

Resolution for the full first-order logic was introduced in a seminal article by John Alan Robinson (1965). The SLD resolution first appeared in Kowalski and Kuehnm (1971). SLDNF resolution first appeared in Clark (1978). A somewhat more accurate version of the definition can be found in Apt and Doets (1994). There are a number of very good books on Prolog. A good practical introduction can be found in Clocksin and Mellish (2003). As a source for more advanced material see, for instance, Sterling and Shapiro (1994) and Covington, Nute, and Vellino (1997). Poole, Mackworth, and Goebel (1998) give a good introduction to artificial intelligence based on Prolog.

Exercises

1. If possible, unify the following pairs of atoms; otherwise, explain why they will not unify.
   (a) $p(X)$ and $r(Y)$.
   (b) $p(X, Y)$ and $p(a, Z)$.
   (c) $p(X, X)$ and $p(a, b)$.
   (d) $r(f(X), Y, g(Y))$ and $r(f(X), Z, g(X))$.
   (e) $ancestor(X, Y)$ and $ancestor(bill, father(bill))$.
   (f) $ancestor(X, father(X))$ and $ancestor(david, george)$.
   (g) $q(X)$ and $q(a)$.
   (h) $p(X, a, Y)$ and $p(Z, Z, b)$.
   (i) $p(f(X))$ and $p(g(a))$.
   (j) $p(X, X)$ and $p(f(Y), g(Z))$.
   (k) $p(f(X, g(X), g(g(Y))))$ and $p(f(g(X), V, g(Y)))$.

2. Trace the Prolog interpreter for query $p(Y)$ to the following program:

$$p(f(X)) \leftarrow q(X), r(X).$$
$$q(a).$$
$$q(b).$$
$$r(b).$$

3. Write definitions for the following list operations in Prolog. Do not use built-in predicates that are not mentioned in this chapter.
   (a) $reverse(L, R)$ where R is the list containing all elements of L in reverse order.
   (b) $last(X, L)$ where X is the last element of L.
   (c) $second(X, L)$ where X is the second element of L.
   (d) $remove(X, L, NoX)$ where NoX is the same as list L with all occurrences of element X removed.

4. We say that a list $[A_1, \ldots, A_n]$ where all As are different represents a set $\{A_1, \ldots, A_n\}$. When we talk about set operations we simply mean to treat the lists as representations of sets. Write definitions for the following predicates in Prolog. X, Y, and Z are lists. Do not use built-in predicates that are not mentioned in this chapter.
   (a) $union(X, Y, Z)$ where Z is a list representing the union of sets represented by X and Y.
   (b) $subset(X, Y)$ where X is a subset of Y.
   (c) $intersection(X, Y, Z)$ where Z is the intersection of X and Y.