Logic Programming with PROLOG

5

Compared to classical programming languages such as C or Pascal, Logic makes it possible to express relationships elegantly, compactly, and declaratively. Automated theorem provers are even capable of deciding whether a knowledge base logically entails a query. Proof calculus and knowledge stored in the knowledge base are strictly separated. A formula described in clause normal form can be used as input data for any theorem prover, independent of the proof calculus used. This is of great value for reasoning and the representation of knowledge.

If one wishes to implement algorithms, which inevitably have procedural components, a purely declarative description is often insufficient. Robert Kowalski, one of the pioneers of logic programming, made this point with the formula

$$Algorithm = Logic + Control.$$

This idea was brought to fruition in the language PROLOG. PROLOG is used in many projects, primarily in AI and computational linguistics. We will now give a short introduction to this language, present the most important concepts, show its strengths, and compare it with other programming languages and theorem provers. Those looking for a complete programming course are directed to textbooks such as [Bra11, CM94] and the handbooks [Wie04, Dia04].

The syntax of the language PROLOG only allows Horn clauses. Logical notation and PROLOG's syntax are juxtaposed in the following table:

PL1 / clause normal form	PROLOG	Description
$(\neg A_1 \vee \ldots \vee \neg A_m \vee B)$	`B :- A1 , ... , Am.`	Rule
$(A_1 \wedge \ldots \wedge A_m) \Rightarrow B$	`B :- A1 , ... , Am.`	Rule
A	`A.`	Fact
$(\neg A_1 \vee \ldots \vee \neg A_m)$	`?- A1 , ... , Am.`	Query
$\neg(A_1 \wedge \ldots \wedge A_m)$	`?- A1 , ... , Am.`	Query

W. Ertel, *Introduction to Artificial Intelligence*,
Undergraduate Topics in Computer Science,
DOI 10.1007/978-0-85729-299-5_5, © Springer-Verlag London Limited 2011

```
 1 child(oscar,karen,frank).
 2 child(mary,karen,frank).
 3 child(eve,anne,oscar).
 4 child(henry,anne,oscar).
 5 child(isolde,anne,oscar).
 6 child(clyde,mary,oscarb).
 7
 8 child(X,Z,Y) :- child(X,Y,Z).
 9
10 descendant(X,Y) :- child(X,Y,Z).
11 descendant(X,Y) :- child(X,U,V), descendant(U,Y).
```

Fig. 5.1 PROLOG program with family relationships

Here $A_1, \ldots, A_m, A, B$ are literals. The literals are, as in PL1, constructed from predicate symbols with terms as arguments. As we can see in the above table, in PROLOG there are no negations in the strict logical sense because the sign of a literal is determined by its position in the clause.

5.1 PROLOG Systems and Implementations

An overview of current PROLOG systems is available in the collection of links on this book's home page. To the reader we recommend the very powerful and freely available (under GNU public licenses) systems GNU-PROLOG [Dia04] and SWI-PROLOG. For the following examples, SWI-PROLOG [Wie04] was used.

Most modern PROLOG systems work with an interpreter based on the *Warren abstract machine* (*WAM*). PROLOG source code is compiled into so-called WAM code, which is then interpreted by the WAM. The fastest implementations of a WAM manage up to 10 million logical inferences per second (LIPS) on a 1 GHz PC.

5.2 Simple Examples

We begin with the family relationships from Example 3.2 on page 35. The small knowledge base *KB* is coded—without the facts for the predicate `female`—as a PROLOG program named `rel.pl` in Fig. 5.1.

The program can be loaded and compiled in the PROLOG interpreter with the command

```
?- [rel].
```

An initial query returns the dialog

```
?- child(eve,oscar,anne).

Yes
```

with the correct answer `Yes`. How does this answer come about? For the query "`?- child(eve,oscar,anne).`" there are six facts and one rule with the same predicate in its clause head. Now unification is attempted between the query and each of the complementary literals in the input data in order of occurrence. If one of the alternatives fails, this results in backtracking to the last branching point, and the next alternative is tested. Because unification fails with every fact, the query is unified with the recursive rule in line 8. Now the system attempts to solve the subgoal `child(eve,anne,oscar)`, which succeeds with the third alternative. The query

```
?- descendant(X,Y).

X = oscar
Y = karen

Yes
```

is answered with the first solution found, as is

```
?- descendant(clyde,Y).

Y = mary

Yes
```

The query

```
?- descendant(clyde,karen).
```

is not answered, however. The reason for this is the clause in line 8, which specifies symmetry of the child predicate. This clause calls itself recursively without the possibility of termination. This problem can be solved with the following new program (facts have been omitted here).

```
1 descendant(X,Y) :- child(X,Y,Z).
2 descendant(X,Y) :- child(X,Z,Y).
3 descendant(X,Y) :- child(X,U,V), descendant(U,Y).
```

But now the query

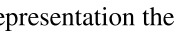
```
?- child(eve,oscar,anne).
```

is no longer correctly answered because the symmetry of `child` in the last two variables is no longer given. A solution to both problems is found in the program

```
 1 child_fact(oscar,karen,franz).
 2 child_fact(mary,karen,franz).
 3 child_fact(eva,anne,oscar).
 4 child_fact(henry,anne,oscar).
 5 child_fact(isolde,anne,oscar).
 6 child_fact(clyde,mary,oscarb).
 7
 8 child(X,Z,Y) :- child_fact(X,Y,Z).
 9 child(X,Z,Y) :- child_fact(X,Z,Y).
10
11 descendant(X,Y) :- child(X,Y,Z).
12 descendant(X,Y) :- child(X,U,V), descendant(U,Y).
```

By introducing the new predicate `child_fact` for the facts, the predicate `child` is no longer recursive. However, the program is no longer as elegant and simple as the—logically correct—first variant in Fig. 5.1 on page 68, which leads to the infinite loop. The PROLOG programmer must, just as in other languages, pay attention to processing and avoid infinite loops. PROLOG is just a programming language and not a theorem prover.

We must distinguish here between *declarative* and *procedural semantics* of PROLOG programs. The declarative semantics is given by the logical interpretation of the horn clauses. The procedural semantics, in contrast, is defined by the execution of the PROLOG program, which we wish to observe in more detail now. The execution of the program from Fig. 5.1 on page 68 with the query `child(eve,oscar,anne)` is represented in Fig. 5.2 on page 71 as a search tree.[1] Execution begins at the top left with the query. Each edge represents a possible SLD resolution step with a complementary unifiable literal. While the search tree becomes infinitely deep by the recursive rule, the PROLOG execution terminates because the facts occur before the rule in the input data.

With the query `descendant(clyde,karen)`, in contrast, the PROLOG execution does not terminate. We can see this clearly in the *and-or tree* presented in Fig. 5.3 on page 71. In this representation the branches, represented by  , lead from the head of a clause to the subgoals. Because all subgoals of a clause must be solved, these are *and branches*. All other branches are *or branches*, of which at least one must be unifiable with its parent nodes. The two outlined facts represent the

[1] The constants have been abbreviated to save space.

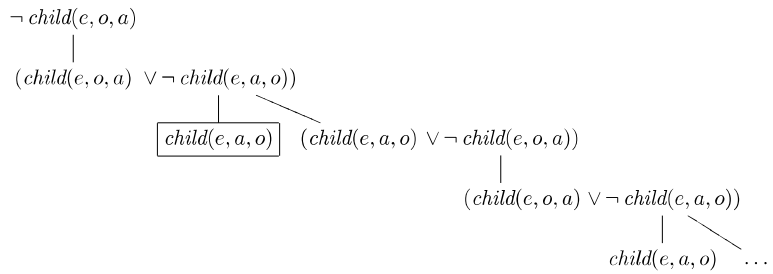


Fig. 5.2 PROLOG search tree for `child(eve,oscar,anne)`

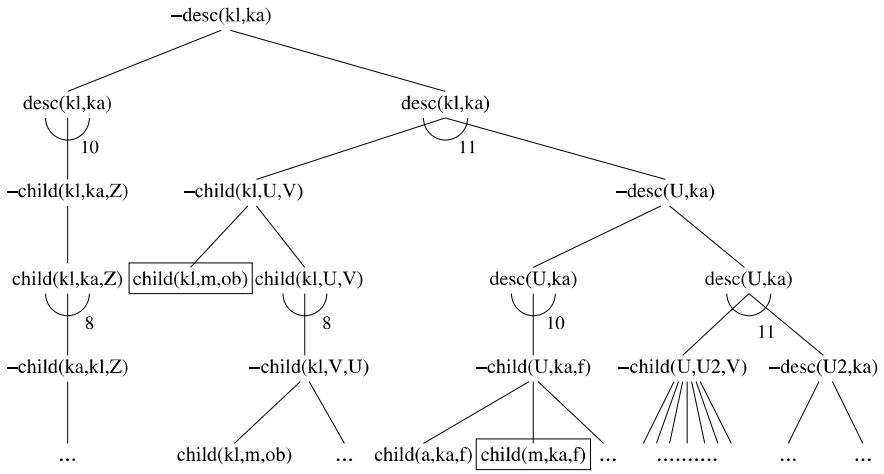


Fig. 5.3 And-or tree for `desc(clyde,karen)`

solution to the query. The PROLOG interpreter does not terminate here, however, because it works by using a depth-first search with backtracking (see Sect. 6.2.2) and thus first chooses the infinitely deep path to the far left.

5.3 Execution Control and Procedural Elements

As we have seen in the family relationship example, it is important to control the execution of PROLOG. Avoiding unnecessary backtracking especially can lead to large increases in efficiency. One means to this end is the *cut*. By inserting an exclamation mark into a clause, we can prevent backtracking over this point. In the following program, the predicate `max(X,Y,Max)` computes the maximum of the two numbers X and Y.

```
1 max(X,Y,X) :- X >= Y.
2 max(X,Y,Y) :- X < Y.
```

If the first case (first clause) applies, then the second will not be reached. On the other hand, if the first case does not apply, then the condition of the second case is true, which means that it does not need to be checked. For example, in the query

```
?- max(3,2,Z), Z > 10.
```

backtracking is employed because `Z = 3`, and the second clause is tested for `max`, which is doomed to failure. Thus backtracking over this spot is unnecessary. We can optimize this with a cut:

```
1 max(X,Y,X) :- X >= Y, !.
2 max(X,Y,Y).
```

Thus the second clause is only called if it is really necessary, that is, if the first clause fails. However, this optimization makes the program harder to understand.

Another possibility for execution control is the built-in predicate `fail`, which is never true. In the family relationship example we can quite simply print out all children and their parents with the query

```
?- child\_fact(X,Y,Z), write(X), write(' is a child of '),
write(Y), write(' and '), write(Z), write('.'), nl, fail.
```

The corresponding output is

```
oscar is a child of karen and frank.
mary is a child of karen and frank.
eve is a child of anne and oscar.
...
No.
```

where the predicate `nl` causes a line break in the output. What would be the output in the end without use of the `fail` predicate?

With the same knowledge base, the query "`?- child_fact(ulla,X,Y).`" would result in the answer `No` because there are no facts about `ulla`. This answer is not logically correct. Specifically, it is not possible to prove that there is no object with the name `ulla`. Here the prover E would correctly answer "`No proof found.`" Thus if PROLOG answers `No`, this only means that the query Q cannot

be proved. For this, however, $\neg Q$ must not necessarily be proved. This behavior is called *negation as failure*.

Restricting ourselves to Horn clauses does not cause a big problem in most cases. However, it is important for procedural execution using SLD-resolution (Sect. 2.5). Through the singly determined positive literal per clause, SLD resolution, and therefore the execution of PROLOG programs, have a unique entry point into the clause. This is the only way it is possible to have reproducible execution of logic programs and, therefore, well-defined procedural semantics.

Indeed, there are certainly problem statements which cannot be described by Horn clauses. An example is Russell's paradox from Example 3.7 on page 45, which contains the non-Horn clause $(shaves(barber, X) \vee shaves(X, X))$.

5.4 Lists

As a high-level language, PROLOG has, like the language LISP, the convenient generic list data type. A list with the elements A, 2, 2, B, 3, 4, 5 has the form

```
[A,2,2,B,3,4,5]
```

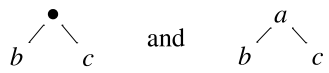
The construct `[Head|Tail]` separates the first element (Head) from the rest (Tail) of the list. With the knowledge base

```
list([A,2,2,B,3,4,5]).
```

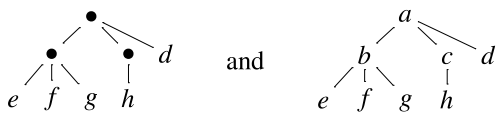
PROLOG displays the dialog

```
?- list([H|T]).

H = A
T = [2, 2, B, 3, 4, 5]

Yes
```

By using nested lists, we can create arbitrary tree structures. For example, the two trees



can be represented by the lists `[b,c]` and `[a,b,c]`, respectively, and the two trees



by the lists `[[e,f,g],[h],d]` and `[a,[b,e,f,g],[c,h],d]`, respectively. In the trees where the inner nodes contain symbols, the symbol is the head of the list and the child nodes are the tail.

A nice, elegant example of list processing is the definition of the predicate `append(X,Y,Z)` for appending list `Y` to the list `X`. The result is saved in `Z`. The corresponding PROLOG program reads

```
1 append([],L,L).
2 append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

This is a declarative (recursive) logical description of the fact that `L3` results from appending `L2` to `L1`. At the same time, however, this program also does the work when it is called. The call

```
?- append([a,b,c],[d,1,2],Z).
```

returns the substitution `Z = [a, b, c, d, 1, 2]`, just as the call

```
?- append(X,[1,2,3],[4,5,6,1,2,3]).
```

yields the substitution `X = [4, 5, 6]`. Here we observe that `append` is not a two-place function, but a three-place relationship. Actually, we can also input the "output parameter" `Z` and ask whether it can be created.

Reversing the order of a list's elements can also be elegantly described and simultaneously programmed by the recursive predicate

```
1 nrev([],[]).
2 nrev([H|T],R) :- nrev(T,RT), append(RT,[H],R).
```

which reduces the reversal of a list down to the reversal of a list that is one element smaller. Indeed, this predicate is very inefficient due to calling `append`. This program is known as *naive reverse* and is often used as a PROLOG benchmark (see Exercise 5.6 on page 81). Things go better when one proceeds using a temporary store, known as the accumulator, as follows:

List	Accumulator
`[a,b,c,d]`	`[]`
`[b,c,d]`	`[a]`
`[c,d]`	`[b,a]`
`[d]`	`[c,b,a]`
`[]`	`[d,c,b,a]`

The corresponding PROLOG program reads

```
1 accrev([],A,A).
2 accrev([H|T],A,R) :- accrev(T,[H|A],R).
```

5.5 Self-modifying Programs

PROLOG programs are not fully compiled, rather, they are interpreted by the WAM. Therefore it is possible to modify programs at runtime. A program can even modify itself. With commands such as `assert` and `retract`, facts and rules can be added to the knowledge base or taken out of it.

A simple application of the variant `asserta` is the addition of derived facts to the beginning of the knowledge base with the goal of avoiding a repeated, potentially time-expensive derivation (see Exercise 5.8 on page 81). If in our family relationship example we replace the two rules for the predicate descendant with

```
1 :- dynamic descendant/2.
2 descendant(X,Y) :- child(X,Y,Z), asserta(descendant(X,Y)).
3 descendant(X,Y) :- child(X,U,V), descendant(U,Y),
4 asserta(descendant(X,Y)).
```

then all derived facts for this predicate are saved in the knowledge base and thus in the future are not re-derived. The query

```
?- descendant(clyde, karen).
```

leads to the addition of the two facts

```
descendant(clyde, karen).
descendant(mary, karen).
```

By manipulating rules with `assert` and `retract`, even programs that change themselves completely can be written. This idea became known under the term *genetic programming*. It allows the construction of arbitrarily flexible learning programs. In practice, however, it turns out that, due to the huge number of senseless possible changes, changing the code by trial and error rarely leads to a performance increase. Systematic changing of rules, on the other hand, makes programming so much more complex that, so far, such programs that extensively modify their own code have not been successful. In Chap. 8 we will show how machine learning has been quite successful. However, only very limited modifications of the program code are being conducted here.

```
 1 start :- action(state(left,left,left,left),
 2                 state(right,right,right,right)).
 3
 4 action(Start,Goal):-
 5         plan(Start,Goal,[Start],Path),
 6         nl,write('Solution:'),nl,
 7         write_path(Path).
 8 %         write_path(Path), fail.    % all solutions output
 9
10 plan(Start,Goal,Visited,Path):-
11       go(Start,Next),
12       safe(Next),
13       \+ member(Next,Visited),       % not(member(...))
14       plan(Next,Goal,[Next|Visited],Path).
15 plan(Goal,Goal,Path,Path).
16
17 go(state(X,X,Z,K),state(Y,Y,Z,K)):-across(X,Y). % farmer, wolf
18 go(state(X,W,X,K),state(Y,W,Y,K)):-across(X,Y). % farmer, goat
19 go(state(X,W,Z,X),state(Y,W,Z,Y)):-across(X,Y). % farmer, cabbage
20 go(state(X,W,Z,K),state(Y,W,Z,K)):-across(X,Y). % farmer
21
22 across(left,right).
23 across(right,left).
24
25 safe(state(B,W,Z,K)):- across(W,Z), across(Z,K).
26 safe(state(B,B,B,K)).
27 safe(state(B,W,B,B)).
```

Fig. 5.4 PROLOG program for the farmer–wolf–goat–cabbage problem

5.6 A Planning Example

Example 5.1 The following riddle serves as a problem statement for a typical PROLOG program.

> A farmer wants to bring a cabbage, a goat, and a wolf across a river, but his boat is so small that he can only take them across one at a time. The farmer thought it over and then said to himself: "If I first bring the wolf to the other side, then the goat will eat the cabbage. If I transport the cabbage first, then the goat will be eaten by the wolf. What should I do?"

This is a planning task which we can quickly solve with a bit of thought. The PROLOG program given in Fig. 5.4 is not created quite as fast.

The program works on terms of the form `state(Farmer,Wolf,Goat,Cabbage)`, which describe the current state of the world. The four variables with possible values `left, right` give the location of the objects. The central recursive predicate `plan` first creates a successor state `Next` using `go`, tests its safety with `safe`, and repeats this recursively until the start and goal states are the same (in program line 15). The states which have already been visited are stored in the third argument of `plan`. With the built-in predicate `member` it is tested whether the state `Next` has already been visited. If yes, it is not attempted. The definition of the predicate `write_path` for the task of outputting the plan found is missing here. It is suggested as an exercise for the reader (Exercise 5.2 on page 80). For initial pro-

gram tests the literal `write_path(Path)` can be replaced with `write(Path)`. For the query "`?- start.`" we get the answer

```
Solution:
Farmer and goat from left to right
Farmer from right to left
Farmer and wolf from left to right
Farmer and goat from right to left
Farmer and cabbage from left to right
Farmer from right to left
Farmer and goat from left to right

Yes
```

For better understanding we describe the definition of `plan` in logic:

$$\forall z\ plan(z,z) \wedge \forall s\ \forall z\ \forall n\ [go(s,n) \wedge safe(n) \wedge plan(n,z) \Rightarrow plan(s,z)]$$

This definition comes out significantly more concise than in PROLOG. There are two reasons for this. For one thing, the output of the discovered plan is unimportant for logic. Furthermore, it is not really necessary to check whether the next state was already visited if unnecessary trips do not bother the farmer. If, however, `\+ member(...)` is left out of the PROLOG program, then there is an infinite loop and PROLOG might not find a schedule even if there is one. The cause of this is PROLOG's backward chaining search strategy, which, according to the depth-first search (Sect. 6.2.2) principle, always works on subgoals one at a time without restricting recursion depth, and is therefore incomplete. This would not happen to a theorem prover with a complete calculus.

As in all planning tasks, the state of the world changes as actions are carried out from one step to the next. This suggests sending the state as a variable to all predicates that depend on the state of the world, such as in the predicate `safe`. The state transitions occur in the predicate `go`. This approach is called *situation calculus* [RN10]. We will become familiar with an interesting extension to learning action sequences in partially observable, non-deterministic worlds in Chap. 10.

5.7 Constraint Logic Programming

The programming of scheduling systems, in which many (sometimes complex) logical and numerical conditions must be fulfilled, can be very expensive and difficult with conventional programming languages. This is precisely where logic could be useful. One simply writes all logical conditions in PL1 and then enters a query. Usually this approach fails miserably. The reason is the penguin problem discussed in Sect. 4.3. The fact `penguin(tweety)` does ensure that `penguin(tweety)` is

true. However, it does not rule out that `raven(tweety)` is also true. To rule this out with additional axioms is very inconvenient (Sect. 4.3).

Constraint Logic Programming (*CLP*), which allows the explicit formulation of constraints for variables, offers an elegant and very efficient mechanism for solving this problem. The interpreter constantly monitors the execution of the program for adherence to all of its constraints. The programmer is fully relieved of the task of controlling the constraints, which in many cases can greatly simplify programming. This is expressed in the following quotation by Eugene C. Freuder from [Fre97]:

> Constraint programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it.

Without going into the theory of the constraint satisfaction problem (CSP), we will apply the CLP mechanism of GNU-PROLOG to the following example.

Example 5.2 The secretary of Albert Einstein High School has to come up with a plan for allocating rooms for final exams. He has the following information: the four teachers Mayer, Hoover, Miller and Smith give tests for the subjects German, English, Math, and Physics in the ascendingly numbered rooms 1, 2, 3 and 4. Every teacher gives a test for exactly one subject in exactly one room. Besides that, he knows the following about the teachers and their subjects.

1. Mr. Mayer never tests in room 4.
2. Mr. Miller always tests German.
3. Mr. Smith and Mr. Miller do not give tests in neighboring rooms.
4. Mrs. Hoover tests Mathematics.
5. Physics is always tested in room number 4.
6. German and English are not tested in room 1.

Who gives a test in which room?

A GNU-PROLOG program for solving this problem is given in Fig. 5.5 on page 79. This program works with the variables `Mayer, Hoover, Miller, Smith` as well as `German, English, Math, Physics,` which can each take on an integer value from 1 to 4 as the room number (program lines 2 and 5). A binding `Mayer = 1` and `German = 1` means that Mr. Mayer gives the German test in room 1. Lines 3 and 6 ensure that the four particular variables take on different values. Line 8 ensures that all variables are assigned a concrete value in the case of a solution. This line is not absolutely necessary here. If there were multiple solutions, however, only intervals would be output. In lines 10 to 16 the constraints are given, and the remaining lines output the room numbers for all teachers and all subjects in a simple format.

The program is loaded into GNU-PROLOG with "`['raumplan.pl'].`", and with "`start.`" we obtain the output

```
[3,1,2,4]
[2,3,1,4]
```

```
 1 start :-
 2   fd_domain([Mayer, Hoover, Miller, Smith],1,4),
 3   fd_all_different([Mayer, Miller, Hoover, Smith]),
 4
 5   fd_domain([German, English, Math, Physics],1,4),
 6   fd_all_different([German, English, Math, Physics]),
 7
 8   fd_labeling([Mayer, Hoover, Miller, Smith]),
 9
10   Mayer #\=4,                 % Mayer not in room 4
11   Miller #= German,           % Miller tests German
12   dist(Miller,Smith) #>= 2,   % Distance Miller/Smith >= 2
13   Hoover #= Math,             % Hoover tests mathematics
14   Physics #= 4,               % Physics in room 4
15   German #\=1,                % German not in room 1
16   English #\=1,               % English not in room 1
17   nl,
18   write([Mayer, Hoover, Miller, Smith]), nl,
19   write([German, English, Math, Physics]), nl.
```

Fig. 5.5 CLP program for the room scheduling problem

Represented somewhat more conveniently, we have the following room schedule:

Room num.	1	2	3	4
Teacher	Hoover	Miller	Mayer	Smith
Subject	Math	German	English	Physics

GNU-PROLOG has, like most other CLP languages, a so-called *finite domain constraint solver*, with which variables can be assigned a finite range of integers. This need not necessarily be an interval as in the example. We can also input a list of values. As an exercise the user is invited, in Exercise 5.9 on page 81, to create a CLP program, for example with GNU-PROLOG, for a not-so-simple logic puzzle. This puzzle, supposedly created by Einstein, can very easily be solved with a CLP system. If we tried using PROLOG without constraints, on the other hand, we could easily grind our teeth out. Anyone who finds an elegant solution with PROLOG or a prover, please let it find its way to the author.

5.8 Summary

Unification, lists, declarative programming, and the relational view of procedures, in which an argument of a predicate can act as both input and output, allow the development of short, elegant programs for many problems. Many programs would be significantly longer and thus more difficult to understand if written in a procedural language. Furthermore, these language features save the programmer time. Therefore PROLOG is also an interesting tool for rapid prototyping, particularly for AI

applications. The CLP extension of PROLOG is helpful not only for logic puzzles, but also for many optimization and scheduling tasks.

Since its invention in 1972, in Europe PROLOG has developed into one of Europe's leading programming languages in AI, along with procedural languages. In the U.S., on the other hand, the natively invented language LISP dominates the AI market.

PROLOG is not a theorem prover. This is intentional, because a programmer must be able to easily and flexibly control processing, and would not get very far with a theorem prover. On the other hand, PROLOG is not very helpful on its own for proving mathematical theorems. However, there are certainly interesting theorem provers which are programmed in PROLOG.

Recommended as advanced literature are [Bra11] and [CM94], as well as the handbooks [Wie04, Dia04] and, on the topic of CLP, [Bar98].

5.9 Exercises

Exercise 5.1 Try to prove the theorem from Sect. 3.7 about the equality of left- and right-neutral elements of semi-groups with PROLOG. Which problems come up? What is the cause of this?

Exercise 5.2

(a) Write a predicate `write_move(+State1, +State2)`, that outputs a sentence like "Farmer and wolf cross from left to right" for each boat crossing. `State1` and `State2` are terms of the form `state(Farmer, Wolf, Goat, Cabbage)`.

(b) Write a recursive predicate `write_path(+Path)`, which calls the predicate `write_move(+State1, +State2)` and outputs all of the farmer's actions.

Exercise 5.3

(a) At first glance the variable `Path` in the predicate `plan` of the PROLOG program from Example 5.1 on page 76 is unnecessary because it is apparently not changed anywhere. What is it needed for?

(b) If we add a `fail` to the end of `action` in the example, then all solutions will be given as output. Why is every solution now printed twice? How can you prevent this?

Exercise 5.4

(a) Show by testing out that the theorem prover E (in contrast to PROLOG), given the knowledge base from Fig. 5.1 on page 68, answers the query "`?- descendant(clyde, karen).`" correctly. Why is that?

(b) Compare the answers of PROLOG and E for the query "`?- descendant(X, Y).`".

Exercise 5.5 Write as short a PROLOG program as possible that outputs 1024 ones.

❄ **Exercise 5.6** Investigate the runtime behavior of the naive reverse predicate.

(a) Run PROLOG with the trace option and observe the recursive calls of `nrev`, `append`, and `accrev`.
(b) Compute the asymptotic time complexity of `append(L1,L2,L3)`, that is, the dependency of the running time on the length of the list for large lists. Assume that access to the head of an arbitrary list takes constant time.
(c) Compute the time complexity of `nrev(L,R)`.
(d) Compute the time complexity of `accrev(L,R)`.
(e) Experimentally determine the time complexity of the predicates `nrev`, `append`, and `accrev`, for example by carrying out time measurements (`time(+Goal)` gives inferences and CPU time.).

Exercise 5.7 Use function symbols instead of lists to represent the trees given in Sect. 5.4 on page 73.

❄ **Exercise 5.8** The Fibonacci sequence is defined recursively by $fib(0) = 1$, $fib(1) = 1$ and $fib(n) = fib(n-1) + fib(n-2)$.

(a) Define a recursive PROLOG predicate `fib(N,R)` which calculates $fib(N)$ and returns it in `R`.
(b) Determine the runtime complexity of the predicate `fib` theoretically and by measurement.
(c) Change your program by using `asserta` such that unnecessary inferences are no longer carried out.
(d) Determine the runtime complexity of the modified predicate theoretically and by measurement (notice that this depends on whether `fib` was previously called).
(e) Why is `fib` with `asserta` also faster when it is started for the first time right after PROLOG is started?

❄ **Exercise 5.9** The following typical logic puzzle was supposedly written by Albert Einstein. Furthermore, he supposedly claimed that only 2% of the world's population is capable of solving it. The following statements are given.

- There are five houses, each painted a different color.
- Every house is occupied by a person with a different nationality.
- Every resident prefers a specific drink, smokes a specific brand of cigarette, and has a specific pet.
- None of the five people drinks the same thing, smokes the same thing, or has the same pet.
- Hints:
  - The Briton lives in the red house.
  - The Swede has a dog.
  - The Dane likes to drink tea.
  - The green house is to the left of the white house.

- The owner of the green house drinks coffee.
- The person who smokes Pall Mall has a bird.
- The man who lives in the middle house drinks milk.
- The owner of the yellow house smokes Dunhill.
- The Norwegian lives in the first house.
- The Marlboro smoker lives next to the one who has a cat.
- The man with the horse lives next to the one who smokes Dunhill.
- The Winfield smoker likes to drink beer.
- The Norwegian lives next to the blue house.
- The German smokes Rothmanns.
- The Marlboro smoker has a neighbor who drinks water.

Question: To whom does the fish belong?

(a) First solve the puzzle manually.

(b) Write a CLP program (for example with GNU-PROLOG) to solve the puzzle. Orient yourself with the room scheduling problem in Fig. 5.5 on page 79.