COMP97083/70043 Prolog

Dr Timothy Kimber

Autumn 2022

Contents

List of Program Listings

Chapter 1

Introduction

1.1 The Course

This is a laboratory module: a programming course. The majority of your time will be spent writing code. After some unassessed exercises to get you started, there will be five programming assignments to hand in: one per week once they begin. There is one lecture and initially one lab session per week. Later in the term, the lectures get shorter and the lab hours increase. The lab sessions are for you to work on the next assignment, and receive help if you need it. The lectures will be based around these notes, and later on to provide feedback on your submitted work.

1.2 Course Materials

These notes contain all the primary reading matter. You should read the relevant parts of the notes as preparation for the lecture each week, so that you can ask questions. This will also give you the necessary background for the next coursework. The notes also contain brief exercises. These will allow you to confirm that you understand what you have just read. If you get any of the exercises wrong, or do not understand them, then please ask for clarification in a lecture, or on the course discussion board.

There is also one short video to show you how to get started with running Prolog code. You should watch this video before the course starts, or during the first week.

1.3 Assessment

The total course mark is made up as follows.

- Five coursework exercises combined: 20%
- Timed programming test (January 2023): 80%

The five pieces of assessed coursework (programming assignments) are handed in during the term. In January, there will be a final test. This test will last 3 hours, and will also be a programming assignment. The reason for structuring the assessments and marks this way is that completing the five courseworks is intended to help prepare you for the final test. So, you get some marks for them, but their value is also the practice that you get, and the feedback that you will receive.

1.4 Intended Learning Outcomes

Learning should have an end product. A better *understanding* of the subject should always result. The question is how this understanding is supposed to manfiest itself, i.e. what should you be able to do once you have undergone the learning. In this case, at the end of the course you will be expected to be able to:

1. Write logic programs using the (Sicstus) Prolog language.
2. Apply good programming practice to the logic programming setting.
3. Apply a declarative or procedural style as appropriate.
4. Make proper use of relational evaluation.
5. Use built-in control techniques and design patterns to optimize execution.
6. Solve AI–related problems.

You will be assessed on your ability to do these things.

1.5 Programming Environment

There are several Prolog distributions. This course uses Sicstus Prolog:

`https://sicstus.sics.se/`

The general principles of the language described in the course apply to any distribution, but if you use something other than Sicstus the code examples may not work, and your answers to the lab exercises might not run.

SICStus Prolog is proprietary software and you need a valid licence in order to use it. The Department of Computing has a student licence. You can download and install Sicstus Prolog from

`https://www.doc.ic.ac.uk/csg-old/software/sicstus`

Choose the latest version available and see the relevant README file for licence information (required during installation).

1.6 Further Reading

The Sicstus Prolog User's Manual is an excellent source of information about the Prolog language, as well as documentation on the predefined predicates available in the Sictus distribution:

`https://sicstus.sics.se/sicstus/docs/latest4/html/sicstus.html/`

Another recommended secondary source is:

- *Prolog Programming for Artificial Intelligence*, Bratko, 4th edn, 2012.

Chapter 2

The Prolog Language

Prolog is short for "**Pro**gramming in **Log**ic".

Prolog statements correspond to sentences in *first-order predicate logic* (FOPL), which is a mathematical formalism used to state, and reason about, knowledge. This makes Prolog a *declarative* language in which we attempt to write succinct logical statements about the world. We can then *run* our statements, to find what ought to be true, or not. At least that is the theory. In practice we need to apply considerable programming skill and understanding of how Prolog works *procedurally*, to be able to write successful programs.

2.1 First-Order Logic

2.1.1 Propositional Logic

The foundation of first-order logic is *propositional logic*. Propositional logic sentences are made up of *propositions* and *connectives*. A proposition is a symbol that we define to represent some atomic item of information that can either be true or false. In the following sentence there are three propositions: *go_to_ic*, *study_prolog* and *best_time_ever* and two connectives. You may have already worked out its intended meaning.

$$go\_to\_ic \wedge study\_prolog \rightarrow best\_time\_ever$$

The connectives $\wedge$ and $\rightarrow$ represent logical conjunction (read as "and" in natural language) and implication ("then") respectively. There are five connectives. The other three are:

- $\vee$ for disjunction ("or")
- $\neg$ for negation ("not")
- $\leftrightarrow$ for equivalence ("if and only if").

To start reasoning we need to define some formulas to be true. If the following formulas are all asserted true:

$$go\_to\_ic \tag{2.1}$$

$$study\_prolog \tag{2.2}$$

$$go\_to\_ic \wedge study\_prolog \rightarrow best\_time\_ever \tag{2.3}$$

then we can conclude that *best_time_ever* is also true using a rule of logical inference.

2.1.2 Predicate Logic

In order to reason about non-logical objects using propositional logic, we would need to define separate propositions about each object:

$$tim\_goes\_to\_ic \tag{2.4}$$
$$tim\_studies\_prolog \tag{2.5}$$
$$tim\_goes\_to\_ic \land tim\_studies\_prolog \rightarrow tim\_has\_best\_time\_ever \tag{2.6}$$
$$fidelis\_goes\_to\_ic \tag{2.7}$$
$$\neg\, fidelis\_has\_best\_time\_ever \tag{2.8}$$
$$fidelis\_goes\_to\_ic \land \neg\, fidelis\_had\_best\_time\_ever \rightarrow \neg\, fidelis\_studies\_prolog \tag{2.9}$$

First-Order Logic provides a way to express the same knowledge more succinctly by generalising propositions to *predicates*, and adding *variables* and *quantifiers*:

$$\forall x\, (goes\_to\_ic(x) \land studies\_prolog(x) \rightarrow has\_best\_time\_ever(x)) \tag{2.10}$$
$$goes\_to\_ic(tim) \tag{2.11}$$
$$studies\_prolog(tim) \tag{2.12}$$
$$goes\_to\_ic(fidelis) \tag{2.13}$$
$$\neg\, has\_best\_time\_ever(fidelis) \tag{2.14}$$

Formulas 2.10–2.14 use three predicates: *go_to_ic*, *studies_prolog* and *has_best_time_ever*. Whereas a proposition is a single piece of logical information that is itself true or false, a predicate is a function that maps objects to $\{true, false\}$. So, $has\_best\_time\_ever(tim)$ and $has\_best\_time\_ever(fidelis)$ are true or false, and can be joined by connectives to make other formulas.

In formula 2.10, x is a variable that represents an object and the symbol $\forall$ *quantifies* x. There are two quantifiers: the universal quantifier $\forall$ (natural language reading: "for all"), and the existential quantifier $\exists$ ("there exists"). So, 2.10 is a statement about all objects. If 2.10 is true then, by substituting for x, we can deduce that any instance of 2.10 like:

$$goes\_to\_ic(tim) \land studies\_prolog(tim) \rightarrow has\_best\_time\_ever(tim)$$

is also true.

In general, a predicate is applied to an ordered tuple of objects. For instance, we might have a binary predicate *teaches* that is applied to pairs of objects:

$$teaches(tim, prolog) \tag{2.15}$$
$$teaches(fidelis, python) \tag{2.16}$$

(So, propositions are predicates applied to tuples of 0 objects.) A formula $teaches(tim, prolog)$, if true, says that the objects *tim* and *prolog* are related (by *teaches*). This gives rise to the notion of a *relation*. The relation *teaches* is the set of ground (variable-free) tuples, like $(tim, prolog)$, that *teaches* maps to true, or equivalently the set of ground formulas like $teaches(tim, prolog)$ that evaluate to true.

2.2 Prolog Program Statements

A Prolog program is made up of statements, called *clauses*. Each clause represents a first-order logic formula that the program asserts to be true.

2.2.1 A First Prolog Clause

Formula 2.10:

$$\forall x\,(goes\_to\_ic(x) \land studies\_prolog(x) \rightarrow has\_best\_time\_ever(x)) \tag{2.10}$$

can be written in Prolog as:

```
has_best_time_ever(X) :-
  goes_to_ic(X),
  studies_prolog(X).
```

Translation of the formula into Prolog starts by dropping the universal quantifier, which is implicit in Prolog:

$$goes\_to\_ic(x) \land studies\_prolog(x) \rightarrow has\_best\_time\_ever(x) \tag{2.17}$$

Next, we write the implication from right to left:

$$has\_best\_time\_ever(x) \leftarrow goes\_to\_ic(x) \land studies\_prolog(x) \tag{2.18}$$

Then we replace the mathematical $\leftarrow$ with `:-` and $\land$ with `,`.

```
has_best_time_ever(x) :- goes_to_ic(x), studies_prolog(x)
```

This is not yet Prolog code. In Prolog we have to use upper case for the variable:

```
has_best_time_ever(X) :- goes_to_ic(X), studies_prolog(X)
```

This is still not Prolog code. We also have to terminate the clause with a full stop:

```
has_best_time_ever(X) :- goes_to_ic(X), studies_prolog(X).
```

This is now Prolog code that can be loaded by the Prolog interpreter. Even so, as programmers we know that our code needs to follow sensible style conventions, so we add line breaks and indentation.

```
has_best_time_ever(X) :-
    goes_to_ic(X),
    studies_prolog(X).
```

Prolog statements represent logic formulas, but we are now writing a program, and the requirement to make it readable applies just as for any programming language. If you have used Prolog before, and have not used line breaks and indentation to make your code readable, now is the time to fix your bad habits.

2.2.2 A Program is a Set of Clauses

We can add clauses for formulas 2.11–2.13 to our Prolog program as follows:

```
goes_to_ic(tim).
goes_to_ic(fidelis).

studies_prolog(tim).
```

The objects *tim* and *fidelis* are represented by the *constants* `tim` and `fidelis`. A constant is an alphanumeric sequence of characters that begins with a lower case letter.

This program can now be *queried* in the Prolog interpreter to see what the given statements imply.

```
| ?- has_best_time_ever(tim).
yes
| ?- has_best_time_ever(fidelis).
no
```

The `yes` and the `no` shown are the ouptut produced when the queries are executed. The clauses in the program are given, true formulas. Using these clauses, Prolog is able to determine that the first queried formula is true and the second is false.

2.2.3 Clauses in General

All Prolog clauses must be of the form already seen. That is to say, they must be a right-to-left implication with a single atomic formula on the left, and a conjunction of zero or more atomic formulas on the right.

```
h(...) :-
  b1(...),
  b2(...),
  ... .
```

The formula that comes before the `:-` is called the *head* of the clause, and the conjunction after the `:-` is called the *body* of the clause. If the body is empty then the `:-` is omitted:

```
h(...).
```

It might appear that Prolog clauses, strictly called *Horn clauses* after mathematician Alfred Horn, are very restricted compared to the variety of first-order logic formulas that can be written. However, Horn clauses are Turing complete, meaning that we can define any computable function with them. We just have to find the appropriate clauses. Using only Horn clauses has many advantages. It simplifies the task of writing a program. Crucially, it also means that Prolog can use a single inference rule to derive new Horn clauses that are implied by the initial program. This derivation mechanism, called *SLD Resolution*, is introduced in Section 2.5.

Exercise 2.1

Write Prolog clauses that represent the following natural language statements.

(a) The dog is black.

(b) Darth is the father of Luke.

(c) A car is polluting if it uses petrol.

(d) All fish can swim.

(e) A tricycle has 3 wheels.

(f) A general outranks a captain.

2.3 Programs Define True Information

Prolog clauses are written as right-to-left implications because the emphasis is on the head of the clause, and its predicate. Each clause defines some condition under which the head predicate is *true*. Together, clauses with the same head predicate form a definition for that predicate.

For instance, an expanded definition of `has_best_time_ever` might have four clauses:

```
has_best_time_ever(X) :-
  goes_to_ic(X),
  studies_prolog(X).

has_best_time_ever(X) :-
  goes_to_vegas(X).

has_best_time_ever(alice).
has_best_time_ever(bob).
```

A Prolog program defines *only* the ways in which the head predicates can be true. So, formula 2.14:

$$\neg\, has\_best\_time\_ever(fidelis) \tag{2.14}$$

cannot be written as a Prolog clause. Although Prolog has a form of negation (see Chapter 5) the head of a clause cannot be negated. Instead, any formula that is not made true by some clause, or set of clauses, in the program is assumed to be false. So, even though there is no representation of formula 2.14 in the program, the example query in Section 2.2.2 reports that "no", `has_best_time_ever(fidelis)` is not true.

2.3.1 Declarative Reading of a Clause

Given the form of a Prolog clause its logical, or declarative, reading is that that the head is true *if* the body is true. So, the first `has_best_time_ever` clause above reads "X has the best time ever, if X goes to IC, and X studies Prolog".

Exercise 2.2

```
1  % parent(P, C). P is a parent of C.
2  parent(tom,bob).
3  parent(tom,liz).
4  parent(pam,bob).
5  parent(bob,ann).
6  parent(bob,pat).
7  parent(pat,jim).
8
9  % grandparent(G, C). G is a grandparent of C.
10 grandparent(G,C) :-
11   parent(G,P),
12   parent(P,C).
```

Given the Prolog program above, would you expect the following queries to ouput `yes` or `no`?

```
(a) ?- parent(bob,ann).
(b) ?- parent(tom,bob),parent(bob,pat).
(c) ?- grandparent(tom,jim).
(d) ?- grandparent(jenny,jim).
(e) ?- grandparent(bob,jim).
```

2.4 Programs Compute Answers

A Prolog program will not just tell you if a ground query is true or false. If the query contains variables, the result will return values for those variables:

```
| ?- has_best_time_ever(X).
X = tim;
```

This still has a logical interpretation: the query formula is true *when X has the value tim*. However, there is an obvious procedural interpretation too: the value *tim* is the output of calling the procedure `has_best_time_ever()`. Now the motivation for the layout of our code should become completely clear. As well as a declarative (logical) definition, a clause is a procedure.

```
has_best_time_ever(X) :-
  goes_to_ic(X),
  studies_prolog(X).
```

When this procedure is called it executes by calling the `goes_to_ic()` procedure, and then the `studies_prolog()` procedure. Sometimes the procedural interpretation of Prolog is emphasized by the context of the program, and sometimes not. Part of the "art" of Prolog programming is to always keep both interpretations in mind.

2.4.1 Modes

The procedure `has_best_time_ever()` operates in two different *modes*:

```
| ?- has_best_time_ever(tim).
yes

| ?- has_best_time_ever(X).
X = tim;
```

In the first query it receives an input and provides a yes/no response, in the second it receives no input and outputs a return value. In other languages one might achieve something similar by setting default values for inputs, or overloading the procedure definition. Those techniques are choices that a programmer can make explicitly. By contrast, all Prolog procedures operate like this automatically. This is a direct effect of the way Prolog evaluates queries, which is explained in Section 2.5.

Exercise 2.3

```
% parent(P, C). P is a parent of C.
parent(tom,bob).
parent(tom,liz).
parent(pam,bob).
parent(bob,ann).
parent(bob,pat).
parent(pat,jim).

% grandparent(G, C). G is a grandparent of C.
grandparent(G,C) :-
  parent(G,P),
  parent(P,C).
```

Given the same program as Exercise 2.2, what would be the output of the following queries?

```
(a) ?- grandparent(bob,X).
(b) ?- grandparent(liz,X).
(c) ?- parent(tom,X),parent(X,jim).
(d) ?- parent(bob,X),parent(X,jim).
(e) ?- parent(X,Y),parent(Y,jim).
(f) ?- parent(X,Y),parent(X,jim).
```

2.5 Prolog Computations

Prolog execution involves solving (evaluating) queries. A query can include multiple atomic formulas, called *subgoals*. The query is proved only if all its subgoals can be proved.

2.5.1 Queries as Logic

Just like program clauses, in logic a query is a Horn clause. The difference is that the query is a Horn clause with an empty head. So, the query

```
| ?- has_best_time_ever(tim).
```

represents the first-order formula:

$$\leftarrow \mathit{has\_best\_time\_ever}(\mathit{tim}) \tag{2.19}$$

which has an equivalent form:

$$\neg\, \mathit{has\_best\_time\_ever}(\mathit{tim}) \tag{2.20}$$

So, why is a query a negated formula? Because Prolog uses *proof by contradiction*. To prove that a program implies *has_best_time_ever*(*tim*) is true, the Prolog interpreter assumes ¬ *has_best_time_ever*(*tim*) and attempts to show that there is now a contradiction.

2.5.2 Derived Queries and Unification

Continuing the example above, to show a contradiction the interpreter must look for a clause that could make the subgoal *has_best_time_ever*(*tim*) true. This must be a clause that

- has `has_best_time_ever` as its head predicate
- has a head that can be *unified* with the *subgoal* `has_best_time_ever(tim)`

Two atomic formulas can be unified if there is a way to replace the variables in the formulas that makes them identical. Crucially, only variables can be replaced, so the clauses

```
has_best_time_ever(alice).
has_best_time_ever(bob).
```

cannot be used. The heads of these clauses are not identical to `has_best_time_ever(tim)` and neither the query nor the clauses contain variables, so there is no way to make them identical. However, the clause

```
has_best_time_ever(X) :-
  goes_to_ic(X),
  studies_prolog(X).
```

can be used. To unify the head of this clause with the query, the variable `X` must be replaced with the constant `tim`. This substitution has to be applied to the entire clause, giving:

```
has_best_time_ever(tim) :-
  goes_to_ic(tim),
  studies_prolog(tim).
```

Following the substitution, the clause and the query can be *resolved*, resulting in a derived query. The resolved subgoal is removed and replaced with the body of the resolved clause. In our example the derived query is therefore

```
:- goes_to_ic(tim), studies_prolog(tim).
```

The new query is logically implied by the previous query and the program clause. No contradiction has yet been shown, but further resolution steps are possible. Execution continues by applying the same process to `goes_to_ic(tim)`, the first subgoal in the derived query.

Exercise 2.4

```
% parent(P, C). P is a parent of C.
parent(tom,bob).
parent(tom,liz).
parent(pam,bob).
parent(bob,ann).
parent(bob,pat).
parent(pat,jim).

% grandparent(G, C). G is a grandparent of C.
grandparent(G,C) :-
  parent(G,P),
  parent(P,C).
```

Given the program above and the following queries, what will be the next, derived query in each case?

```
(a) :- parent(pam,X), parent(X,Y)
(b) :- grandparent(tom,X)
(c) :- parent(pat,X), parent(X,bob)
(d) :- grandparent(pam,jim)
```

2.5.3 Successful Computations

A sequence of Prolog resolution steps is called a *computation*. If the query is reduced to empty, then a contradiction has been shown, the formula that was queried is true, and the computation is *successful*.

Continuing the previous example, the head of the program clause

```
goes_to_ic(tim).
```

is identical to the next subgoal in the current query. The body of the clause is empty, and replacing the subgoal by empty gives the next derived query:

```
:- studies_prolog(tim).
```

The one remaining subgoal can be resolved with another unconditional clause producing an empty query. So, this computation is successful, which is immediately reported to the user:

```
| ?- has_best_time_ever(tim).
yes
```

2.5.4 Unsuccessful Computations

If, at any point in a computation, there is no program clause whose head can be unified the next subgoal, then the computation fails. Our program contains only one clause for `studies_prolog`:

```
studies_prolog(tim).
```

so the following query:

```
:- studies_prolog(fidelis).
```

fails immediately as it cannot be unified with anything.

```
| ?- studies_prolog(fidelis).
no
```

In this example the failure to unify the current query with a clause caused the initial query to fail entirely. There is one other possibility: the interpreter can *backtrack* to find an alternative way to solve the query.

2.5.5 Backtracking

We have seen that there can be multiple clauses with the same head predicate, meaning multiple sets of conditions under which the predicate is true. To ensure that a given query is correctly and fully evaluated, all the clauses may need to be tried. This happens through a backtracking mechanism. To understand this mechanism, let us now consider the evaluation of the following query:

```
:- has_best_time_ever(X).
```

with the program as shown in Listing 2.1. The full evaluation of the query is depicted in Fig. 2.1. Evaluation begins at the top of the diagram, where the single subgoal `has_best_time_ever(X)` needs to be solved. The program contains four clauses defining `has_best_time_ever` and each one can be unified with the query. This causes a branch point (1) in the query execution. All four clauses are used — in the order they are given in the program (left-to-right in the diagram).

Execution proceeds in a depth-first manner. So, a new query:

```
:- goes_to_ic(X), studies_prolog(X).
```

is derived using the first clause, and then this computation (the leftmost branch in the diagram) is followed until it ultimately succeeds at point 3. At this point execution is suspended and the successful value for `X` is reported:

```
1  % has_best_time_ever(?X).
2  %   Succeeds iff X has had a provably great time.
3  has_best_time_ever(X) :-
4    goes_to_ic(X),
5    studies_prolog(X).
6
7  has_best_time_ever(X) :-
8    goes_to_vegas(X).
9
10 has_best_time_ever(alice).
11 has_best_time_ever(bob).
12
13 % goes_to_vegas(?X).
14 %   Data on known trips to Sin City.
15 goes_to_vegas(maria).
16
17 % goes_to_ic(?X).
18 %   Student data for Imperial College.
19 goes_to_ic(tim).
20 goes_to_ic(fidelis).
21
22 % studies_prolog(?X).
23 %   Once a student of Prolog, always a student of Prolog.
24 studies_prolog(tim).
```

Listing 2.1: *The example program in full.*

```
| ?- has_best_time_ever(X).
X = tim ?
```

The variable `X` acquires the value `tim` through a unifying substitution within the branch. Substitutions are shown in the diagram as a set of bindings `{Var/Value, ...}`.

The user now has the option of ending the evaluation or continuing. If they choose to continue, execution will restart at the most recent point where an alternative clause could have been unified with the query (point 2). A new computation (the branch ending at point 4) is constructed from there. This computation fails when a clause to resolve with `:- studies_prolog(fidelis)` cannot be found. When this happens execution immediately backtracks and restarts at point 1, with no suspension. The interpreter is only suspended when evaluation reaches point 5, having found the next solution:

```
| ?- has_best_time_ever(X).
X = tim ? ;
X = maria ?
```

If the query is fully evaluated, four branches will eventually succeed, each for a different value of `X`:

```
| ?- has_best_time_ever(X).
X = tim ? ;
X = maria ? ;
X = alice ? ;
X = bob ? ;
no
```

A final `no` shows that there are no more solutions - no other ways of backtracking exist.

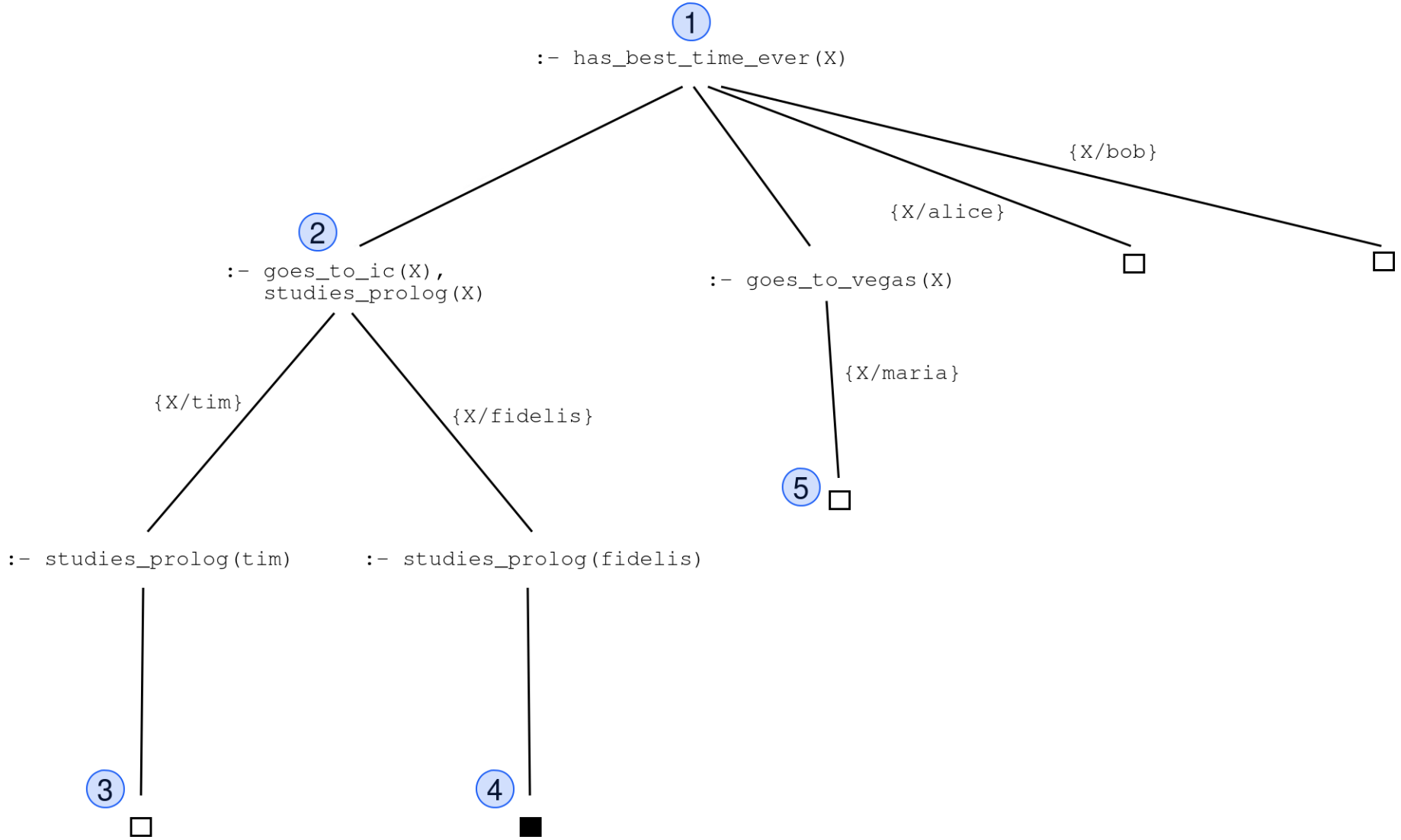


Figure 2.1: *Execution of the query* `:-has_best_time_ever(X)`*. Branches are followed depth-first, from left to right.* □ *successful computation;* ■ *unsuccessful computation.*

Exercise 2.5

```
% parent(P, C). P is a parent of C.
parent(tom,bob).
parent(tom,liz).
parent(pam,bob).
parent(bob,ann).
parent(bob,pat).
parent(pat,jim).

% grandparent(G, C). G is a grandparent of C.
grandparent(G,C) :-
  parent(G,P),
  parent(P,C).
```

Given the program above, draw the Prolog evaluation tree of the following queries.

```
(a) ?- grandparent(bob,X).
(b) ?- parent(X,Y),parent(Y,jim).
```

2.6 Solutions to Exercises in Chapter 2

Solution 2.1

```
% (a)
is_black(the_dog).

% (b)
father(darth, luke).

% (c)
polluting(X):-
  car(X),
  uses_petrol(X).

% (d)
swim(X):-
  fish(X).

% (e)
wheels(X,3):-
  tricycle(X).

% (f)
outranks(X, Y):-
  general(X),
  captain(Y).
```

Solution 2.2

The outputs will be (a) `yes`, (b) `yes`, (c) `no`, (d) `no`, (e) `yes`.

Solution 2.3

The outputs will be (a) `X = jim`, (b) `no`, (c) `no`, (d) `X = pat`, (e) `X = bob, Y = pat`, (f) `X = pat, Y = jim`.

Solution 2.4

The derived queries will be:

```
(a) :- parent(bob,Y)
(b) :- parent(tom,P), parent(P,X)
(c) :- parent(jim,bob)
(d) :- parent(pam,P), parent(P,jim)
```

Solution 2.5

Will be discussed in a Q&A session.

Chapter 3

Programming Techniques

Prolog is a programming language. Like any other programming language it is a tool to perform practical computations and solve problems. This chapter focuses on how to design practical programs, and the Prolog-specific techniques you will need.

3.1 Using Variables

The way variables are used to manipulate data in Prolog is quite unlike it is in imperative languages. We we will start by implementing two predicates:

- `min_of_two` will find the minimum of two given numbers;
- `min_of_three` will find the minimum of three given numbers.

3.1.1 Inputs and Outputs

First, we need to decide on the *arity* of the predicates — how many arguments should they have? This may seem like a strange question, but it can be a source of confusion when dealing with predicates like `min_of_two` and `min_of_three` with such a clearly procedural purpose.

We want `min_of_two` to take two numbers and output the minimum. So, it has two inputs and an output. As seen in Chapter 2, Prolog returns information by assigning values to variables that occur within a query. So, predicates need arguments for inputs *and* arguments for outputs. This means that `min_of_two` needs three arguments.

```
| ?- min_of_two(2, 3, 2).  % Does the min of 2 and 3 equal 2?
yes.
| ?- min_of_two(2, 10, 6).  % Does the min of 2 and 10 equal 6?
no.
| ?- min_of_two(2, 3, Min).  % What Min is the min of 2 and 3?
Min = 2 ?
```

Prolog documentation uses the notation `min_of_two(+X, +Y, -Min)`, to show which arguments must be provided as inputs, and which will return output values. Arguments that can be either an input or an output are written with a `?`.

```
1  % min_of_two(+X, +Y, ?Min)
2  %   Succeeds iff Min is the lower of the numbers X and Y.
3  min_of_two(X, Y, X):-
4    X =< Y.
5  min_of_two(X, Y, Y):-
6    X > Y.
7
8  % min_of_three(+X, +Y, +Z, ?Min)
9  %   Succeeds iff Min is the lowest of the numbers X, Y and Z.
10 min_of_three(X, Y, Z, Min):-
11   min_of_two(X, Y, MinXY),
12   min_of_two(MinXY, Z, Min).
```

Listing 3.1: *A Prolog program to find the minimum of three numbers.*

3.1.2 Unifying Outputs

A variable acquires its value through unification, which is just another word for copying data. A simple way to give an output variable a value is to copy an input value. This is just what we need to do in `min_of_two`. The output `Min` will get the value of one of the inputs `X` and `Y`. To do this in Prolog we just write a clause where the output is the same as one input:

```
min_of_two(X, Y, X):-
  X =< Y.
```

This has an intuitive declarative reading, and also works procedurally. To use this clause to solve a query:

```
| ?- min_of_two(2, 3, Min).
```

a unifying substitution must be found to make the head of the clause and the subgoal identical. The head of the clause causes this substitution to copy the value of the first input into the output variable. Finally, we need a condition in the body of the clause to ensure it only succeeds when `X` actually is the minimum.

It is more likely that some intermediate computation will be needed to find an output. This is the case with `min_of_three`:

```
min_of_three(X, Y, Z, Min):-
  min_of_two(X, Y, MinXY),      % find min of X and Y
  min_of_two(MinXY, Z, Min).    % use MinXY to find Min
```

This clause calls `min_of_two` as a subprocedure, passing in two of its inputs. This subprocedure query will compute a value for the intermediate result `MinXY`, which is then passed into the next subprocedure call as an input. The output of the second call to `min_of_two` is the value we need as the overall output. Just as with `min_of_two`, the necessary copying of the value is achieved by simply using `Min` as the variable in both cases.

Exercise 3.1

```
% teaches(?PersonID, ?CourseID)
teaches(tk106,70043).
...
% person(?ID, ?First, ?Last)
person(tk106,'Timothy','Kimber').
...
```

Given a file of ground clauses for the predicates shown in the snippet above, define a predicate `course_teacher` that accepts a course id as its input, and returns the first and last name of the teacher of the course.

3.1.3 Backtracking Through Outputs

Section 2.5.5 shows a top-level query that returns different values for a variable via backtracking. This will also happen when a predicate is called as a subprocedure. So, when `factor` is called in:

```
prime_factor(N, Factor):-
  factor(N, Factor),
  is_prime(Factor).
```

we assume it will gradually backtrack through all factors of `N`. There might be all sorts of branching and backtracking going on inside `factor` to find one factor. This is of no concern. Somehow a value for `Factor` will be found. This value is passed into `is_prime`. If `is_prime(Factor)` fails, execution automatically backtracks to `factor`, generates the next value, and proceeds onto `is_prime` again. So, the procedure will keep checking factors until one is found that is prime, which is returned. If `prime_factor` is the top-level query being solved, then the user now has the option to continue execution, whereupon the process starts up again with the next untried factor.

Exercise 3.2

```
% teaches(?PersonID, ?CourseID)
teaches(tk106,70043).
...
% course(?ID, ?Title)
course(70043,'Prolog').
...
% role(?PersonID, ?Role)
role(tk106,'TF').
...
```

Given a file of ground clauses for the predicates shown in the snippet above, define a predicate `taught_by_tf(-Course)` that succeeds when `Course` (an output) is the name of a course taught by a Teaching Fellow (TF).

3.1.4 Singleton Variables

Normally, variables in clauses will occur at least twice. For example, `min_of_three` pairs up variables to pass data from one place to another. If variables occur that are not paired in this way, then there is a likelihood that there has been a typing mistake that will cause a bug:

```
min_of_three(X, Y, Z, Min):-      % Min is a 'singleton'
  min_of_two(X, Y, MinXY),
  min_of_two(MinXY, Z, Minimum).  % Minimum is a 'singleton'
```

To help you spot such bugs Prolog will issue a warning if singleton variables exist in your code:

```
 | ?- consult(min).
 % consulting min.pl...
 * [Min,Minimum] - singleton variables
 * Approximate lines: 10-13, file: 'min.pl'
```

3.1.5 Anonymous Variables

Sometimes there is a good reason for a singleton variable to exist.

```
% beats(+X, +Y)
% Succeeds if X beats Y (scoring method not shown).
beats(X, Y):-
  score(X, XS),
  score(Y, YS),
  XS > YS.
beats(chuck_norris, Y).  %% Chuck Norris beats everyone.
```

Since a singleton is not automatically a bug, it is possible to ignore the warning and execute the code. This is very bad practice! The correct approach is to replace any genuine singletons with '_', which is a special *anonymous* variable:

```
beats(chuck_norris, _).  %% Chuck Norris beats everyone. Really.
```

This signifies that the variable is an intentional singleton, and the warning is not generated. Using this approach, when warnings appear they must be caused by accidental singletons, so they represent bugs that need to be fixed.

Anonymous variables are always singletons, so care must be taken. Every occurrence of '_' is a different variable, so changing the first clause above to be

```
beats(_, Y):-     % First singleton
  score(_, XS),   % Second, different singleton
  score(Y, YS),
  XS > YS.
```

would break the code because it is equivalent to:

```
beats(X, Y):-           % First singleton
  score(Anyone, XS),    % Second, different singleton
  score(Y, YS),
  XS > YS.
```

Other variables that start with '_', not '_' itself, confuse the situation somewhat. Such a variable will also prevent singleton warnings:

```
beats(chuck_norris, _All).  %% No singleton warning
```

This can be used as a way to give the variable a more meaningful name and still avoid the warning. However, only '_' is treated as a singleton when the program is executed. Two occurrences of a variable like `_All` represent the same variable. So,

```
beats(_X, Y):-      % First occurence of _X
  score(_X, XS),    % Same variable, second occurrence
  score(Y, YS),
  XS > YS.
```

is equivalent to the original version of the clause, using `X`, not the version using `_`. The conclusion of this is that using any variable starting with `_` is pointless, even prone to errors, unless the variable is a genuine singleton. If the variable is supposed to be paired, then use a name that starts with a capital letter. If you do not you will not see any warnings, even if you mistype your variable names and create accidental singletons.

Exercise 3.3

```
% teaches(?PersonID, ?CourseID)
teaches(tk106,70043).
...
% person(?ID, ?First, ?Last)
person(tk106,'Timothy','Kimber').
...
% role(?PersonID, ?Role)
role(tk106,'TF').
...
```

Given a file of ground clauses for the predicates illustrated above, define a predicate `ra_lecturer(-First, -Last)` that succeeds when `First` and `Last` are the names of a Research Associate (role `'RA'`) who teaches any course. Use anonymous variables where appropriate.

3.2 Using Clauses

3.2.1 One Clause per Case

The clause discussed in Section 3.1.2 only defines one case of `min_of_two`. The definition is completed by adding another clause for the second case (see Listing 3.1). These two clauses might appear,

on first inspection, like separate unconnected statements. However, they are brought together to form a single procedure by the backtracking mechanism.

3.2.2 Disjunction

If you need to add a condition "`A` or `B`" to a predicate definition:

```
has_superheroes(X):-
  made_by(X, marvel) OR??? made_by(X, dc).
```

then you simply need two clauses:

```
has_superheroes(X):-
  made_by(X, marvel).

has_superheroes(X):-
  made_by(X, dc).
```

Backtracking will ensure that `has_superheroes(X)` will be true for any `X` made by either `dc` or `marvel`. There is an explicit way to write "or" in Prolog, but it has no advantages, and introduces several code smells. Prolog is meant to be written in clauses — they are more declarative and keep statements succinct. In short — use clauses, not or.

3.2.3 Variable Scope

The scope of a variable is a single clause. So, variable names can be reused in different clauses for the same predicate without affecting the program.

3.2.4 Pattern Matching Inputs

Non variables can be written into the heads of clauses to restrict the inputs that the clauses accepts:

```
% transform(+Image, +Transformation, -Transformed)
%  Transformed is Image after applying the Transformation operation.
transform(Image, reflect, Transformed):-
  % code to perform reflection ...

transform(Image, gray, Transformed):-
  % code to convert to grayscale ...

transform(Image, blur, Transformed):-
  % code to blur image ...
```

Exercise 3.4

Assume you have a full implementation of the image transformation program above, and assume that the predicate `load_image(+Filepath, -Image)` reads an image file and returns an image object `Image`. What query will produce a grayscale version of the image in `' /picture.jpg'`?

3.3 Structuring Code with Subprocedures

Prolog does not have control flow statements found in imperative languages. Instead, code must be carefully structured using subprocedures. So, a disjunction in the middle of a clause:

```
man_in_tights(X):-
  in_movie(X, M),
  (made_by(M, marvel) OR??? made_by(M, dc)),
  male(X),
  fights_crime(X).
```

can still be implemented as two clauses, but these clauses must now define a subprocedure:

```
man_in_tights(X):-
  in_movie(X, M),
  comic_movie(M),
  male(X),
  fights_crime(X).

comic_movie(M):-
  made_by(M, marvel).
comic_movie(M):-
  made_by(M, dc).
```

Exercise 3.5

```
transform(Image, gray, Transformed):-
  copy_pixels(Image, Copy),
  colors_to_black(Copy, Gray),
  save_transformed_image(Gray, Transformed).
transform(Image, reflect, Transformed):-
  store_pixels(Image, Buffer),
  pad_image(Buffer, SquareImage),
  copy_pixels(SquareImage, Copy),
  reverse_pixels(Copy, Reversed),
  save_transformed_image(Reversed, Transformed).
transform(Image, blur, Transformed):-
  store_pixels(Image, Buffer),
  pad_image(Buffer, SquareImage),
  copy_pixels(SquareImage, Copy),
  blur_pixels(Copy, Blurred),
  save_transformed_image(Blurred, Transformed).
```

Refactor the program above to reduce the amount of repeated code. (The code has been made up purely for this exercise. Do not try to run it - it will not work. Your task is simply to rewrite it to make something equivalent.)

3.4 Determinacy

In general, a Prolog query that contains variables has 0 to many solutions. Through backtracking, all possible solutions will be found, so it is important to ensure that only correct solutions will be returned. In particular, there will often be situations where a predicate, like `min_of_two` should succeed just once. Such a predicate is called *determinate*. Even if we attempt to get more answers after the first, there should be none:

```
| ?- min_of_two(2, 3, Min).
Min = 2 ? ;
no
```

Of course, Prolog does not enforce this — it is the programmer's job, and attention must be paid to this or bugs will ensue. In particular, it is not sufficient for the first solution returned to be correct, if others follow:

```
| ?- min_of_two(2, 3, Min).
Min = 2 ? ;
Min = 3 ? ;      % NOT acceptable
no
```

To avoid this it is important to remember that Prolog clauses do not implement an "if-else" relationship — one clause succeeding does not prevent the next one succeeding too. Even worse (well, certainly no better), allowing queries to backtrack into extra clauses can lead to infinite loops:

```
| ?- min_of_two(2, 3, Min).
Min = 2 ? ;
                % Er, what happened?
```

Looking at the definition of `min_of_two` (Listing 3.1), it has two clauses, so it has the potential to return more than one solution. Furthermore, there are no patterns in the heads of the clauses to restrict the accepted inputs. So, every query for `min_of_two` can be unified with both clauses. To make sure that only one of the clauses succeeds for any value of `X` and `Y`, both must include sufficient conditions to make them mutually exclusive.

3.5 Recursion

Prolog does not have loops ("do loops" have recently been added to some versions of Prolog, but are still not part of the ISO standard) which means that recursion is used extensively. If you have not written a lot of recursive code, then you will need to revise how. The basic principles are:

- there must be one or more simple (non-recursive) "base" cases;
- there must be one or more recursive or "general" cases;
- a recursive case makes a recursive call to solve a *simplified* version of the problem.

Recursion is implemented using one clause per case. This is illustrated by the factorial program in Listing 3.2, which has several other notable points.

```
1  % factorial(+N, -Fac)
2  %   Fac is N factorial.
3  factorial(0, 1).
4  factorial(N, Fac):-
5    N > 0,
6    M is N - 1,
7    factorial(M, MFac),
8    Fac is N * MFac.
```

Listing 3.2: *A Prolog program to compute $n!$ (first attempt).*

The `factorial` predicate should be determinate, and should succeed for all values of `N >= 0`. The base case should (only) succeed when `N=0`, and this is implemented by putting 0 into the head of the first clause. The recursive clause should (only) succeed for positive `N`. The head of the recursive clause cannot impose any restriction, so the condition `N > 0` is necessary to make the clauses mutually exclusive, and avoid an infinite loop.

The recursive clause needs to decrement the value of its input before making the recursive call. In Prolog, this cannot be done by updating the value of the variable `N`. Prolog variables acquire values when they are substituted, and having got a value they cannot be substituted again — they are no longer variables. So, to pass the value `N-1` into the recursive call, we need a new variable, `M`. Prolog arithmetic and the `is` predicate are explained in Chapter 4.

3.5.1 Tail Recursion

Recursive programs that are determinate should be made *tail recursive*. A tail recursive Prolog program will be optimised into a loop by the compiler: each stack frame overwrites the previous one. This avoids the problem of deep recursion using up lots of memory. There are two conditions for a program for a predicate p to be tail recursive.

Definition 3.5.1 (Tail Recursion)**.** A program for a predicate p is *tail recursive* if:

1. Recursive calls only ever occur at the very end (tail) of a clause.
2. No further backtracking is possible, within the evaluation of the call to p, when a recursive call occurs.

The `factorial` program in Listing 3.2 is not tail recursive because Condition 1 is not satisfied. The recursive call is not at the end of the clause, so the program is not equivalent to a loop and cannot be optimised.

Condition 2 is satisfied, and is fairly easy to satisfy in general. Firstly, the recursive `factorial` clause is written after the base case. So, by the time this clause is used other `factorial` branches will have already been followed, and presumably failed. Secondly, the calls that occur in the recursive clause before the recursive call are all determinate — they cannot backtrack. Given these two, there cannot be any remaining backtrack points. In general, satisfying this condition is simplest if there is just one recursive clause. Use subprocedures to allow you to structure your code this way.

Condition 1 can be satisfied by rewriting the `factorial` program. The new version is shown in Listing 3.3. It puts most of the code into a new auxiliary predicate: `factorial/4`. The rewritten `factorial/2` predicate is now simply an entry point into `factorial/4` that initialises two of

```
1  % factorial(+N, -NFac)
2  %   NFac is N!
3  factorial(N, NFac):-
4    factorial(N, 0, 1, NFac).
5
6  % factorial(+N, +I, +Acc, -Fac)
7  %   Tail recursive helper for factorial/2.
8  %   NFac is N!, assuming Acc is I! and 0<=I<=N.
9  factorial(N, N, Acc, Acc).
10 factorial(N, I, Acc, NFac):-
11   I < N,
12   J is I + 1,
13   JAcc is J * Acc,
14   factorial(N, J, JAcc, NFac).
```

Listing 3.3: *An improved, tail-recursive Prolog program to compute $n!$*

its inputs. These new inputs provide the missing parts of the loop implementation, as described in the next section.

3.5.2 Accumulators

The `factorial/4` predicate in Listing 3.3 uses an *accumulator* argument `Acc`. An accumlator is a running total, or more generally a partial solution. It is updated in each iteration. In this case, `Acc` is used to store the value of `I!`, where `I` is a counter that is incremented from 0 to `N`. The features of this design that always apply are:

- the accumulator is an *input* argument;
- the accumulator must be initialised when calling the recursive predicate;
- each iteration makes a new value for the accumulator and passes it into the recursive call;
- a separate output argument is needed to return the solution;
- the solution is copied from the accumulator to the output when the problem is solved — in the base case.

Using an accumulator means each iteration can solve part of the problem before making a recursive call, rather than after. This enables tail recursive optimisation of the program.

Exercise 3.6

Write a tail recursive program to implement `mult(+X, +Y, -Z)`, where `X >= 0` is an integer, `Y >= 0` is an integer and `Z` is `X` multiplied by `Y`. You may use the `+` operator, but not the `*` operator. Either pattern match with Listing 3.3 or read Section 4.4 for help with arithmetic.

3.6 Failing By Design

The factorial function is undefined for $N < 0$. In imperative or functional languages this means that something special, or maybe exceptional, would need to be written into the code to handle invalid input. In Prolog this is not necessary. Prolog has a trivially simple way of handling this: the query just fails. So, either of the `factorial` programs above will give the following output, without needing any modification:

```
| ?- factorial(-1, Fac).
no
```

As an answer to the logical query this is correct: is there a value of `Fac` such that `Fac` is factorial of -1? No, there is not. We get this correct behaviour by providing an absence of code. No clause makes `factorial(-1, Fac)` true. Rather than being an error, this is good design. The query is evaluated normally, and fails. In other words, it evaluates to *false*. We should not be put off by the word "fail". In Prolog, failure is frequently the correct result.

Exercise 3.7

My lucky number is 70043. Write a program `lucky(+N)` that succeeds if `N` is 70043 and fails if `N` is any other number.

3.7 Supporting Modes

A predicate will not always work in all possible modes. Each predicate will have one or more normal modes of operation, which should be shown in documentation. For `factorial/2` this is given as `factorial(+N, -Fac)`. If the predicate is used in some other mode, it might or might not work:

```
| ?- factorial(3, 6).
yes
| ?- factorial(N, 6).
! Instantiation error in argument 2 of (<)/2
! goal:  0<_415
```

This is fine. Prolog does not enforce specific modes, so the queries will be executed, but if they do not use the documented mode then there should be no surprise if the result is an error. The question for a programmer is — what modes should work? If it is important for a predicate to work in multiple modes then this needs to be factored into the code, and the testing. To make sure you have not forgotten any vital modes, one place to check carefully is your own code. It is not unknown to find you are trying to use your own predicate in an unsupported mode.

Exercise 3.8

Does your answer for Exercise 3.6 work correctly in the following modes?

(a) `mult(+X, +Y, +Z)`

(b) `mult(-X, +Y, +Z)`

(c) `mult(-X, -Y, +Z)`

3.8 Solutions to Exercises in Chapter 3

Solution 3.1

```
% course_teacher(+CourseID, -First, -Last)
course_teacher(CID, First, Last):-
  teaches(PID, CID),
  person(PID, First, Last).
```

Solution 3.2

```
% taught_by_tf(-Course)
taught_by_tf(Course):-
  course(CID, Course),
  teaches(PID, CID),
  role(PID, 'TF').
```

Solution 3.3

```
% ra_lecturer(?First, ?Last)
ra_lecturer(First, Last):-
  role(PID, 'RA'),
  teaches(PID, _),
  person(PID, First, Last).
```

Solution 3.4

```
 | ?- load_image('~/picture.jpg', Image),
      transform(Image, gray, GrayImage).
```

Solution 3.5

```
transform(Image, Transformation, Transformed):-
  transform_image(Image, Transformation, NewImage),
  save_transformed_image(NewImage, Transformed).


transform_image(Image, gray, NewImage):-
  copy_pixels(Image, Copy),
  colors_to_black(Copy, NewImage).

transform_image(Image, reflect, Reflection):-
  padded_buffered_image(Image, Buffer),
  reverse_pixels(Buffer, Reflection).

transform_image(Image, blur, Blurred):-
```

```
  padded_buffered_image(Image, Buffer),
  blur_pixels(Buffer, Blurred).


padded_buffered_image(Image, NewImage):-
  store_pixels(Image, Pixels),
  pad_image(Pixels, PaddedImage),
  copy_pixels(PaddedImage, NewImage).
```

Solution 3.6

```
% mult(+X, +Y, -Z)
%   Z = X * Y, where X >= 0 and Y >= 0.
mult(X, Y, Z):-
  X >= 0,
  Y >= 0,
  mult(X, Y, 0, Z).

% mult(+X, +Y, +Acc, -Z)
%   Z = X * Y + Acc, where X >= 0, Y >= 0, and Acc >= 0.
mult(_, 0, Acc, Acc).
mult(X, Y, Acc, Z):-
  Y > 0,
  Y1 is Y - 1,
  NewAcc is Acc + X,
  mult(X, Y1, NewAcc, Z).
```

Solution 3.7

```
lucky(70043).
```

Solution 3.8

(a) Yes, it should.

(b) I doubt it.

(c) I doubt it!

Chapter 4

Data

A practical language must have data structures. After briefly reviewing data primitives this chapter discusses Prolog data structures, and how to use them.

4.1 Terms

A piece — any piece — of Prolog data is called a *term*. So, given a clause

```
H:-
  B1,
  ...,
  BM.
```

`H, B1, ..., BM` are each formulas of the form

```
p(T1,...,TN)
```

where `p` is a predicate of arity `N` and `T1,...,TN` are terms. The terms seen in the preceding chapters have all been variables, atoms or numbers. Prolog also has *compound terms* which enable data structures to be constructed. Compound terms are introduced in Section 4.2.

4.1.1 Variables

Variables have been discussed thoroughly in Chapters 2 and 3. Syntactically, a variable is an alphanumeric sequence where the first character is either a capital letter or '_'.

4.1.2 Atoms

A Prolog atom is non-numeric constant, e.g. `tim`. Atoms can take a variety of forms. There are reserved atoms, like the empty list `[]` (see Section 4.3). The two main user-defined forms are:

- Any sequence of alphanumeric characters or '_', starting with a lowercase letter.
- Any sequence of characters in single quotes. Examples: `'Tim Kimber'`, `'my-prolog-file.pl'`

4.1.3 Numbers

Numbers, either integers or floating point numbers are also Prolog constants. Arithmetic expressions and their evaluation is discussed in Section 4.4.

4.2 Compound Terms

Just as propositions in Chapter 2 were generalised to predicates with arguments, when arguments are added to atoms they become compound terms. So, `person(tim,kimber)` is a compound term in which the atom `person` is called the *functor* of the term, and the atoms `tim` and `kimber` are the arguments of the term.

In general a term is a formula

```
f(t1,...,tN)
```

where the functor `f`, of arity `N`, is an atom and the arguments `t1,...,tN` are terms.

Obviously, the syntax of compound terms and simple logical formulas is identical. This can be confusing when considering formulas in isolation, but the difference should be clear in the context of a program. So, within a program, the line:

```
teaches(person(tim,kimber),prolog).
```

is a clause defining a predicate `teaches/2` that contains the terms `person(tim,kimber)` and `prolog`. Predicates can only be applied to terms, never to other predicate calls, so `person(tim,kimber)` cannot be a logical formula in this context. It must be a piece of data.

4.2.1 Data Structures

Compound terms allow data structures to be built. So, a term `person(tim,kimber)` can be used to represent a `person` object with attributes `tim` and `kimber`. The attributes can even be named explicitly with their own functors for added clarity:

```
person(given_name(tim),family_name(kimber))
```

Since the arguments of a compound term can themselves be compound terms, our data structures can be nested to any depth.

4.2.2 Pattern Matching and (De)Constructing Terms

Variables are terms. So, not only can compound terms contain constants and other compound terms, they can contain variables. Such terms are used within clauses to extend the usefulness of the pattern matching technique seen in Section 3.2.4. So, the `greet/2` program in Listing 4.1 only succeeds if its input is a `person/2` term:

```
| ?- greet(person('Tim','Kimber')).
Hello Tim Kimber!
yes

| ?- greet(boris).
no
```

Assuming the program is called with a `person/2` term, then the substitution sets the values of `Firstname` and `Lastname` to be the arguments within this input. So, the pattern serves as both

```
1  % greet(+Person)
2  %   Greets Person with a friendly message.
3  greet(person(Firstname,Lastname)):-
4    write('Hello '),
5    write(Firstname),
6    write(' '),
7    write(Lastname),
8    write('!'),nl.
```

Listing 4.1: *A Prolog program to output a greeting.*

a condition on the input, and a way of accessing the data inside the input. Compound terms can be constructed within a clause in the same way.

Something to keep in mind when using patterns to define acceptable inputs for a clause is that variables match any term. So, if `greet/1` is called with a variable it will not fail, it will accept the variable and continue executing:

```
| ?- greet(X).
Hello _1309 _1311!
X = person(_A,_B) ?
```

This is only a problem if the program is supposed to work when called with a variable.

4.2.3 Unifying Terms with =

Terms are unified as the result of procedure calls, but can also be unified explicitly using the built-in `=/2` predicate:

```
| ?- person('Tim','Kimber') = person(X, Y).
X = 'Tim',
Y = 'Kimber' ?
```

Note that `=/2` in Prolog is a predicate that succeeds and unifies its arguments if this unification is possible. It is not assignment as you would find in imperative languages. So, this query fails:

```
| ?- person('Tim','Kimber') = person(X, Y), X = 'Will'.
no
```

because the second unification is not possible after the first.

The counterpart of = is `\=` which succeeds if its arguments *cannot* be unified:

```
| ?- person('Tim','Kimber') \= fish.
yes
| ?- person('Tim','Kimber') \= person(X, Y).
no
```

Exercise 4.1

```
% student(?ID, ?First, ?Last, ?Address)
%   First Last is a student with id ID living at Address
%   where Address is a term address(Street, City, Country)
student(
  ppr120,
  'Pierre',
  'Programmer',
  address('1 Rue Eclaire', 'Le Havre', 'France')
).
...
```

Given a file containing ground clauses for `student/4`, as illustrated above, write a Prolog program `city(+ID, -City)` that returns the city in which a student lives, given their id.

Exercise 4.2

Write a Prolog program `city(+Student, -City)` that returns the city in which `Student` lives, where:

- `Student` is a term of the form `student(ID,First,Last,Address)`
- `Address` is a term of the form `address(Street,City,Country)`
- `ID, First, Last, Street, City` and `Country` are atoms.

Exercise 4.3

What is the outcome of the following queries?

```
(a) | ?- X = Y, Y = 5.
(b) | ?- X \= Y, X = 3, Y = 5.
```

4.2.4 Comparing Terms

Identity

Terms can be compared for identity without being unified. The built-in binary predicates == and \== succeed if their arguments are identical and not identical, respectively. Two different variables are not identical, so:

```
| ?- fish == fish.
yes
| ?- fish == chips.
no
| ?- person(First, Last) == person(X, Y).
no
| ?- 1 \== 2.
yes
```

Exercise 4.4

What is the outcome of the following queries?

```
(a) | ?- X == Y, X = 5.
(b) | ?- X = Y, X == Y.
```

Standard Order of Terms

Prolog defines a *standard order of terms*. This is a single sequence containing every term, as follows:

1. Variables, by age, oldest first; followed by
2. Floats, in numeric order; followed by
3. Integers, in numeric order; followed by
4. Atoms, in alphabetical order; followed by
5. Compound terms, ordered first by arity, then by the name of the principal functor, then by the arguments in left-to-right order.

The following inequality predicates compare terms according to this standard order:

- `+T1 @< +T2` succeeds if `T1` is before `T2` in the standard order;
- `+T1 @> +T2` succeeds if `T1` is after `T2` in the standard order;
- `+T1 @=< +T2` succeeds if `T1` is not after `T2` in the standard order;
- `+T1 @>= +T2` succeeds if `T1` is not before `T2` in the standard order.

Exercise 4.5

What is the outcome of the following queries?

```
(a) | ?- f(A) @< f(A, a).
(b) | ?- f([g(1),2,3]) @> f([a]).
```

4.3 Lists

A list is a very common data structure in Prolog with a special syntax and built-in support.

4.3.1 Declaring and Constructing Lists

A Prolog list is either the atom `[]`, which represents an empty list, or a compound term that is normally written like

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

"Under the hood", a non-empty list is actually a compound term '`./2`'. In this form, the two arguments of any non-empty list are the first element, or head, of the list (a term) and a list containing the remaining elements, or tail. So, `[1,2,3]` is syntactic sugar for `.(1,.(2,.(3,[])))`.

The underlying binary form makes it simple to split a list into its head and tail (or equivalently extend a list with a new head) by unifying it with a term containing two variables:

```
.(H,T)
```

The same thing can be done without needing to resort to the binary form, using the syntax

```
[H|T]
```

The latter term is exactly equivalent to the former. In both cases `H` will be assigned the head of the list and `T` will be assigned the tail:

```
| ?- [1,2,3] = .(H,T).
H = 1,
T = [2,3] ? ;
| ?- [1,2,3] = [H|T].
H = 1,
T = [2,3] ? ;
```

The `|` syntax in fact allows any number of elements to be extracted from the front of the list:

```
| ?- [a,b,c,d,e] = [H1,H2,H3|T].
H1 = a,
H2 = b,
H3 = c,
T = [d,e] ?
```

4.3.2 Patterns for Mapping Lists

Mapping one list to another is a ubiquitous operation, so it is worth looking at how to do it well. There are two design patterns that can be used. Both are, naturally, recursive. The choice depends on the order in which we want the output.

Reverse Mapping a List

If the order of the output list is not important, or if reversing the order relative to the input list is actually required, then we can use a list accumulator. A list accumulator is used to collect mapped elements as they are generated. Just like a numerical accumulator, it is an input argument that has to be initialised, often to `[]`, when the procedure is called. As the recursion proceeds the accumulator always holds the list of values mapped so far.

An example of this pattern is shown in Listing 4.2. This program follows all the principles of using accumulators set out in Section 3.5.2. The recursive predicate removes the next element `N` from the input list, maps it to `N * 2`, and inserts this value into the accumulator. The new value is inserted in front of the previously mapped elements in `Acc`, so the effect is to reverse the order of the data:

```
| ?- double([5,3,15,-1], D).
D = [-2,30,6,10] ? ;
```

Each step is a constant time operation, so the program takes $\Theta(n)$ time for a list of length n. Just as with the `factorial` program, the use of the accumulator makes `double` tail recursive.

In-Order Mapping a List

If it is important to map the list in order, then we have to use the alternative design pattern, as exemplified by the program in Listing 4.3. This program appears to be much simpler — there is no auxiliary predicate and no accumulator. The output for a non-empty input is defined by a list pattern

```
[NSq|Squares]
```

in the head of the recursive clause. This output pattern is filled in when the calls in the body of the clause find values for the two variables.

```
| ?- square([5,3,15,-1], S).
S = [25,9,225,1] ? ;
```

The `square/2` program maps the input to the output in order, because the mapped value for the first element `N` is inserted in front of the mapped values for the tail list `Ns`. The mapping and list de/construction steps are constant time operations, so the overall time is $\Theta(n)$ for a list of length n.

```
1  % double(+L, -Doubles)
2  %   Doubles is the list containing every number in list L
3  %   multiplied by 2, in reverse order.
4  double(L, Doubles):-
5    double(L, [], Doubles).
6
7  % double(+L, +Acc -Doubles)
8  %   List Doubles contains every number in list L
9  %   multiplied by 2, in reverse order,
10 %   followed by the elements of list Acc.
11 double([], Acc, Acc).
12 double([N|Ns], Acc, Doubles):-
13   N2 is N * 2,
14   double(Ns, [N2|Acc], Doubles).
```

Listing 4.2: *Example of the reverse-map design pattern.*

```
1  % square(+L, -Squares)
2  %   List Squares contains every number in L squared.
3  square([], []).
4  square([N|Ns], [NSq|Squares]):-
5    NSq is N * N,
6    square(Ns, Squares).
```

Listing 4.3: *Example of the in-order map design pattern.*

The only remaining question is whether this program is tail recursive. The answer is yes. Clearly, the recursive call is written last and there can be no backtracking. So far, so good. The only doubt is over the list construction step. Only the tail of the mapped list is computed recursively. If the head of the output is inserted after this it would break tail recursion. Prolog supports a special version of tail recursion called *tail recursion modulo cons* to handle this situation. The output list is constructed outside the stack and a pointer is used to keep track of where to write the next mapped element.

4.3.3 Built-In List Predicates

Prolog distributions come with libraries of pre-defined predicates that can be imported into a program, and so-called *built-in* predicates that are always available, without import. Different distributions vary over what is available, and what is built-in.

Every Prolog distribution includes countless pre-defined predicates that perform list operations. Many of these will be found in the `lists` library. The ones described below are normally built-in. They are all useful, but also subject to over-use.

`sort/2`

A call `sort(List, Sorted)` sorts `List` according to the standard order of terms. It also removes duplicates from the sorted list.

```
| ?- sort([3,g(1),TWO,3], Sorted).
Sorted = [TWO,3,g(1)] ?
```

`length/2`

A call `length(List, Len)` succeeds when `Len` (an integer) is the length of `List`. The predicate works in all modes.

```
| ?- length([3,g(1),TWO,3], Len).
Len = 4 ?
| ?- length(L, 2).
L = [_A,_B] ?
| ?- length([a], 2).
no
```

`member/2`

A call `member(E,L)` succeeds when `E` is a member of the list `L`. This can be used to check if a given `E` is in `L`, but also to find all members of `L`:

```
| ?- member(a, [a,b,c]).
yes
| ?- member(X, [a,b,c]).
X = a ? ;
X = b ? ;
X = c ? ;
```

As you would expect, `member/2` takes linear time.

append/3

A call `append(L1,L2,L3)` succeeds when: `L1`, `L2` and `L3` are all lists; and concatenating `L1` and `L2` gives `L3`. It is possible to use `append` in just about every conceivable mode:

```
| ?- append([a,b], [c,d,e], L).
L = [a,b,c,d,e] ? ;

| ?- append(L, [c,d,e], [a,b,c,d,e]).
L = [a,b] ? ;

| ?- append(L1, L2, [a,b]).
L1 = [],
L2 = [a,b] ? ;
L1 = [a],
L2 = [b] ? ;
L1 = [a,b],
L2 = [] ? ;
```

`append/3` is also linear — it takes $\Theta(n)$ time where n is the length of the first list (the prefix).

Try to avoid writing every program that involves lists as a series of calls to `member` and `append`. This approach (which I like to call "appendicitis") can lead to very strange algorithms. `member` and `append` can be defined very simply, and if you have a task that they don't quite fit you will be better off writing your own predicate(s).

Exercise 4.6

(Without using `append`!) Write a program `my_append/3` that behaves exactly like `append/3` described above.

Exercise 4.7

```
% bad_add(+L1, +Elem, -L2)
%   L2 is the list L1 after adding a new element Elem.
bad_add(L1, Elem, L2):-
  append(L1, [Elem], L2).
```

Improve the program above.

4.4 Expressions and Arithmetic

Prolog has a standard set of built-in arithmetic operators including:

```
+   -   *   /   //   div   rem   mod   **
```

For the full list, and the precise behaviour of `//`, `div`, `rem` and `mod`, see online documentation.

These operators are, in fact, an alternative notation for some compound terms. So, the expression

```
3 + 5
```

represents the term

```
+(3,5)
```

4.4.1 Evaluation of Expressions

To evaluate arithmetic expressions they must be passed into specific built-in arithmetic predicates. In keeping with expressions these predicates can be written in infix form.

The most common arithmetic predicate is `is/2`. A call to `is` will evaluate its second argument (an expression) and unify the resulting value with its first argument (normally a variable):

```
| ?- X is 3 + 5.
X = 8 ? ;
```

Expressions are also evaluated by the comparison predicates

```
=:=   =\=   <   >   =<   >=
```

where `=:=/2` and `=\=/2` succeed if their arguments do, and do not, evaluate to the same value, respectively. In all cases both arguments will be evaluated before their values are compared:

```
| ?- 3 + 5 =:= 2 ** 3.
yes
| ?- 4 * 6 =\= 12 * 2.
no
```

When they are called, it must be possible for the predicates above to completely evaluate the expressions, which must therefore be ground (contain no unbound variables). If not, an error will be generated:

```
| ?- Y is X * 2.
! Instantiation error in argument 2 of (is)/2
! goal:  _407 is _413*2
```

Finally, expressions are unsimplified compound terms right up to the point that one of the arithmetic predicates is executed. In particular, predicates like `=/2` and `==/2` *do not* evaluate expressions:

```
| ?- 4 = 2 + 2.
no
| ?- X == 2 + 2.
no
```

Exercise 4.8

What is the result of the following queries?

```
(a) | ?- X = Y, X =:= Y.
(b) | ?- X = 5, X =:= 5.
(c) | ?- X = 2 + 2.
(d) | ?- 4 is 2 + 2.
```

4.5 Solutions to Exercises in Chapter 4

Solution 4.1

```
city(ID, City):-
  student(ID, _, _, address(_, City, _)).
```

Solution 4.2

```
city(Student, City):-
  Student = student(_, _, _, address(_, City, _)).
```

Solution 4.3

(a) Succeeds with `X = 5, Y = 5`.

(b) `no`.

Solution 4.4

(a) `no`.

(b) Succeeds with `Y = X`.

Solution 4.5

(a) `yes`.

(b) `yes`.

Solution 4.6

```
my_append([], L, L).
my_append([X|Xs], Y, [X|Zs]):-
  my_append(Xs, Y, Zs).
```

Solution 4.7

```
% good_add(+L1, +Elem, -L2)
%   L2 is the list L1 after adding a new element Elem.
good_add(L1, Elem, [Elem|L1]).
```

Solution 4.8

(a) Error.

(b) Succeeds with `X = 5`.

(c) Succeeds with `X = 2+2`.

(d) `yes`.

Chapter 5

Negation

5.1 The \+ Predicate

"Not" is implemented by the unary \+ predicate.

```
| ?- \+(member(a, [1,2,3])).
yes
```

In order to avoid an extra set of parentheses \+ can also be written as a prefix operator:

```
| ?- \+ member(a, [1,2,3]).
yes
```

The argument of \+ is a "Goal", meaning that it can be a conjunction of subgoals just as you would enter as a top-level query. If the goal has more than one subgoal, then it must be placed inside parentheses:

```
| ?- \+ (member(X, [1,2,3]), X == a).
yes
```

Now there must be a space between the operator and the argument, or the code will be interpreted as a call to a non-existent predicate \+ of arity $n > 1$.

5.1.1 Negation in Programs

As explained in Section 2.3, negation can be used in the body, but not in the head of program clauses. So, here is a predicate that succeeds for any list containing a, as long as there is no b after it.

```
% a_not_b(+List)
%   Succeeds if List contains a, unless there is a b after it.
a_not_b(List):-
  append(_, [a|Tail], List),
  \+ member(b, Tail).
```

5.1.2 The Meaning of \+

The \+ operator implements a form of negation called *Negation as Failure*. Procedurally, a call

```
\+ Goal
```

where Goal is a goal as described above will succeed if there is no solution for the query

```
| ?- Goal.
```

So, in order to evaluate `\+ Goal` the interpreter has to suspend normal evaluation and start a new query for `Goal`. If this query has no solutions, then `\+ Goal` in the original computation succeeds and normal evaluation resumes.

5.2 Variables within Negation

Care must be taken with variables inside negation. The program in Listing 5.1 gives different results for the following two queries:

```
| ?- \+ disappears(X), character(X).
no

| ?- character(X), \+ disappears(X).
X = bart ?
```

because the queries are performing negation as failure, and under negation as failure they have different meanings.

```
1  character(harry).
2  character(bilbo).
3  character(bart).
4
5  has_cloak(harry).
6  has_ring(bilbo).
7
8  disappears(X) :-
9    has_cloak(X).
10 disappears(X) :-
11   has_ring(X).
```

Listing 5.1: *Example of a program queried using negation.*

The first query will only succeed if the query

```
| ?- disappears(X).
```

has no solutions. So, the first query is confirming that no `X` disappears, and then looking for any character. The second query will find a value for `X` first, and then determine if this `X` disappears. Assuming that the second query is the one with the intended meaning, one way of spotting the problem with the first is by remembering a fairly simple fact about negation:

> *If there are variables that are unbound when a negated call occurs, and the negation succeeds, then those variables will still be unbound.*

The point of negation is that there are no solutions. In which case no variable without a value is going to end up with one if the negation succeeds. For the first example query to work, `X` would somehow have to become `bart` within `\+ disappears(X)`. This is impossible.

Although we have to take care with variables, do not be overly cautious. As the `a_not_b` program in Section 5.1.1 demonstrates, variables can be used to good effect within negation. Negation is a very useful tool, that can make programs simpler and more declarative.

Exercise 5.1

```
person(john).
person(mary).
person(jane).

% likes(?X, ?Y).
%   X likes Y.
likes(mary, jane).
likes(john, mary).
likes(jane, mary).
```

Given the clauses above for `person/1` and `likes/2`, write a definition for `sad(-P)`. A person is sad if there is no person that likes them.

5.3 Solutions to Exercises in Chapter 5

Solution 5.1

```
% sad(-P).
%   P is a sad person.
sad(P):-
  person(P),
  \+ likes(_, P).
```

Chapter 6

Controlling Query Evaluation

6.1 Pruning Computations with !

The basic Prolog query evaluation mechanism can result in programs backtracking and running unnecessary computations. Prolog provides an operator to allow the programmer to prune these computations out of the evaluation tree. The operator is written `!` and called *cut*.

6.1.1 The Effect of !

The `!` operator can be added to the body of a clause, and prevents a query backtracking once it has reached this point. Given a program:

```
p(...):-
   ...

p(...):-
   cond_1(...),
   ...,
   cond_m(...),
   !,                % <--- CUT
   ...

p(...):-
   ...

 ...
```

that includes a cut in some clause, and:

- the clause containing the cut is used to solve a query; and
- conditions `cond1(...)` to `condm(...)` in the body of the clause up to the cut all succeed;

then, whether the query eventually succeeds or fails:

- no subsequent clause for `p` will be used to (re-)solve the query; and
- any backtracking points that could be used to re-solve conditions `cond_1(...)` to `cond_m(...)` will be discarded.

```
1  % send(+Customer, +Balance, -Message)
2  %   Picks correct Message for Customer, according to Balance.
3  send(Cust, Balance, Message) :-
4    Balance =< 0,
5    warning(Cust, Message).
6  send(Cust, Balance, Message) :-
7    Balance > 0,
8    Balance =< 50000,
9    credit_card_info(Cust, Message).
10 send(Cust, Balance, Message) :-
11   Balance > 50000,
12   investment_offer(Cust, Message).
```

Listing 6.1: *Example of a program with unnecessary backtracking.*

In other words, the first solution of `cond_1(...)` to `cond_m(...)` is used, if any, and any others are ignored.

6.1.2 Unnecessary Computations

The program in Listing 6.1 generates one of three possible messages (not shown) to send to a customer of a bank. We can imagine this predicate being called within a larger program that backtracks through all customers. In order to avoid generating two or three messages for the same customer when the program backtracks, the condition in the first clause:

```
Balance =< 0
```

has to be inverted in the second clause (`Balance > 0`), and the same is true of the condition

```
Balance =< 50000
```

for clauses 2 and 3. This is effectively checking the same conditions twice. Since the program will backtrack and try to re-solve a `send` subgoal if it can, these conditions will be checked and re-checked for all inputs, regardless of the value of `Balance`. In this toy example the effect of these repeated conditions may well be small (how many customers does the bank have?), but other programs might repeat very expensive computations in this situation. Cuts can be added to the program to prevent this. Listings 6.2 and 6.3 show two ways in which this can be done.

6.1.3 Placing Cuts

The cuts in both Listings 6.2 and 6.3 are placed in the first and second clauses, immediately after the balance conditions have been checked. This placement is important. It eliminates backtracking as soon as possible. The cut cannot come before the balance conditions, because we want backtracking into the correct clause to occur if a balance condition fails. Nor should it come later in the clause, because failure of some later step would then incorrectly trigger backtracking.

The cuts ensure that once the correct clause has been found, no others will be tried. So if `Balance` is −100, the cut in the first clause will be executed, and there will be no attempt to use the second and third clauses to re-solve the query for the same customer.

```
1   send(Cust, Balance, Message) :-
2     Balance =< 0,
3     !,
4     warning(Cust, Message).
5   send(Cust, Balance, Message) :-
6     Balance > 0,
7     Balance =< 50000,
8     !,
9     credit_card_info(Cust, Message).
10  send(Cust, Balance, Message) :-
11    Balance > 50000,
12    investment_offer(Cust, Message).
```

Listing 6.2: *Version of the bank account program using "green" cuts.*

```
1   send(Cust, Balance, Message) :-
2     Balance =< 0,
3     !,
4     warning(Cust, Message).
5   send(Cust, Balance, Message) :-
6     Balance =< 50000,
7     !,
8     credit_card_info(Cust, Message).
9   send(Cust, Balance, Message) :-
10    investment_offer(Cust, Message).
```

Listing 6.3: *Version of the bank account program using "red" cuts.*

6.1.4 Green Cuts

Listing 6.2 uses *green cuts*. Green cuts do not affect the correctness of the program: the output is the same with or without them. So, removing the cuts from Listing 6.2 gets us back to Listing 6.1. The difference between these programs is that the version with cuts will not backtrack once a cut has been reached. However, both versions include all the same conditions, so Listing 6.2 has to retain the inverted balance conditions in the second and third clauses. If `Balance` is +100, then both `Balance =< 0` and `Balance > 0` will still be checked. Unnecessary backtracking has been eliminated, but conditions are still repeated.

6.1.5 Red Cuts

Unlike Listing 6.2, the correctness of the version of the program in Listing 6.3 depends on the presence of the cuts. This means they are *red cuts*. The inverted balance conditions have been removed. With the cuts in place this is fine. The second and third clauses can only be reached if the conditions in the previous clauses failed. So, the red cuts both eliminate unnecessary backtracking, and allow repeated conditions to be removed.

6.1.6 Green Cuts or Red Cuts?

Red cuts pose a problem when it comes to the declarative reading of a program. A cut has no logical meaning, only a procedural effect. So, the logical reading of a program is the same with or without cuts. Consequently, a program with red cuts, like the one in Listing 6.3 is by definition logically incorrect. For instance, reading the second clause of Listing 6.3 as a logical statement, the formula

```
send(alice, -100, "Would you like to have a credit card?")
```

is true (assume that this is the credit card message). However, if the formula is executed as a Prolog query it will not succeed. In other words, the program is functionally correct even though it appears to be logically incorrect.

This apparent contradiction means there is a design choice to be made. If efficiency and software engineering is the priority then we should use red cuts. Listing 6.3 is a perfectly good program. From a software engineering point of view it is the best version. If, on the other hand, the logical reading of the program is the priority then we should use green cuts, or no cuts at all. This choice is context dependent. Which approach to use will vary between programs, maybe even between predicates within a single program.

6.1.7 Pitfalls and Bad Practice

Red cuts can lead to bugs if you check a condition at the wrong point in the code. This is a particular danger when unification introduces a "hidden condition". See Exercise 6.1.

Cuts can also be used poorly, effectively hiding bugs. If a predicate is producing duplicate or incorrect answers on backtracking, the answer is to fix this predicate, *not* to suppress the incorrect answers by scattering cuts around the rest of the code. See Exercise 6.2.

Exercise 6.1

```
% min_of_two(+X, +Y, ?Min)
%   Succeeds iff Min is the lower of the numbers X and Y.
min_of_two(X, Y, X):-
  X =< Y,
  !.
min_of_two(X, Y, Y).
```

The version of `min_of_two` above has a red cut, and a bug in it. Find the bug and fix it, without removing the cut. (Make sure to consider both modes in which the predicate is supposed to work.)

```
1   send(Cust, Balance, Message) :-
2     (
3       Balance =< 0
4     ->
5       warning(Cust, Message)
6     ;
7       (
8         Balance =< 50000,
9       ->
10        credit_card_info(Cust, Message)
11      ;
12        investment_offer(Cust, Message)
13      )
14    ).
```

Listing 6.4: *The bank program, written using the if-then-else operator.*

Exercise 6.2

```
% min_of_two(+X, +Y, -Min)
%   Returns the lower of X and Y as Min.
min_of_two(X, Y, X):-
  X =< Y.
min_of_two(X, Y, Y).

% min_of_three(+X, +Y, +Z, -Min)
%   Returns the lowest of the X, Y and Z as Min.
min_of_three(X, Y, Z, Min):-
  min_of_two(X, Y, MinXY), !,
  min_of_two(MinXY, Z, Min), !.
```

The version of `min_of_three` above correctly finds the lowest of three given numbers, using cuts to make it determinate. Improve the code by changing the placement of the cuts.

6.1.8 An Alternative Red Cut

Prolog provides an if-then-else operator that produces code equivalent to a set of clauses with red cuts. The syntax of this operator is

```
(P -> Q ; R)
```

which will be executed by first trying to solve the goal `P`; then if `P` succeeds solve goal `Q` without backtracking to re-solve `P`; otherwise (if `P` fails) attempt to solve goal `R`. Listing 6.4 contains a version of the bank program written using if-then-else. This is equivalent to Listing 6.3.

Of the two versions, Listing 6.3 is strongly preferred. The if-then-else syntax in Listing 6.4 has completely destroyed the clausal structure of the code. It can be argued that the red cuts in Listing 6.3

have already taken away its declarative meaning. Even so, it is better to be consistent and define each case in its own clause. As Listing 6.4 illustrates, if-then-else (and the related "or") encourage longer, nested statements. This is something to avoid in any language.

6.2 Collecting All Solutions

Most Prolog distributions provide built-in predicates that allow you to collect all the solutions to some goal (list of subgoals) into a list.

6.2.1 `findall/3`

The built-in predicate `findall(+Template, +Goal, -List)` constructs a list, where each element is an instance of the term `Template`, and there is one element for each solution of `Goal`. So, if we have the following facts

```
% friend(?P, ?Q)
%   P is a friend of Q.
friend(alice, bob).
friend(alice, clare).
friend(bob, clare).
```

then we can obtain a list containing all friends of `clare` with the query

```
| ?- findall(F, friend(F, clare), Friends).
Friends = [alice,bob] ? ;
```

The element template can be any term and the goal can be any goal.

Exercise 6.3

Write a query to construct a list containing all the possible pairs `(X,Y)` such that concatenating `X` and `Y` gives the list `[a,b,c]`.

Exercise 6.4

```
person(alice).
person(bob).
person(clare).

friend(alice, bob).
friend(alice, clare).
friend(bob, clare).
```

Assuming a file containing the data above, write a program for `all_friends(-List)` that returns a list containing all the pairs `(Name,Num)` such that `Name` is a person with `Num` friends. (You may find the built-in predicate `length(List, Len)` that finds the length of a list helpful.)

6.2.2 setof/3

The built-in predicate `setof{+Template, +Goal, -List}` behaves in much the same way as `findall/3`, with some critical differences.

- The `List` generated is sorted and duplicate-free. Sorting follows the *standard total order* of Prolog terms (see Section 4.2.4).
- Unbound variables that appear in the `Goal` but not in the `Template` are treated as *free variables*. This means that alternative solutions of a `setof` call can occur on backtracking, whereas `findall` is always determinate.

The second difference is best illustrated with some queries:

```
| ?- findall(F, friend(F, P), Fs).
Fs = [alice,alice,bob] ?

| ?- setof(F, friend(F, P), Fs).
P = bob,
Fs = [alice] ? ;
P = clare,
Fs = [alice,bob] ? ;
```

In the `findall` query the variable `P` is *existentially quantified*. So, all values of `F` for which there exists some `P` that makes the goal true are added to the list. Also note that, even though it appears in the query, `P` is not part of the answer displayed — because the list is a combined result from different `P`s.

In contrast, in the `setof` query `P` is not treated as existentially quantified — its actual value is significant. So, the query returns a separate list for each `P`, and `P` is part of the answer displayed.

If existential quantification is needed within a `setof`, then it can be applied to specific variables. The variable(s) should be given as a prefix to the goal, with each one followed by the existential quantifier `^`.

```
| ?- setof(F, P^friend(F,P), Fs).
Fs = [alice,bob]
```

Note that the list is still sorted and duplicate-free.

Exercise 6.5

Assuming a file containing ground clauses for `friend/2` as described above, write a program for `all_friendly(-Set)` that returns a sorted, duplicate-free list `Set` containing the name of everyone that is a friend of somebody. As a challenge, you may not use the existential quantifier in your program. (Consider the initial description of a what makes a free variable.)

6.3 Exploring All Solutions

It can be useful to find all solutions to some goal without actually needing to collect anything into a list. For instance, given data about students and courses they are registered for

```
student(abc01).
...
registered(abc01, 70043).
...
```

the program in Listing 6.5 determines whether every student is registered for at least one course. This program uses a double negation pattern that is generally applicable, and so worth studying. It has both a logical and a procedural basis. Logically, we want to implement a formula

$$all\_registered \leftarrow \forall s\,(student(s) \rightarrow \exists c\, registered(s,c)) \tag{6.1}$$

This has no direct translation into Prolog, but it is equivalent to:

$$all\_registered \leftarrow \neg\,\exists s\,(student(s) \wedge \neg\,\exists c\, registered(s,c)) \tag{6.2}$$

which gives us the Prolog implementation. Procedurally, to solve the outer negation the interpreter has to attempt to find a student that is not registered to some course. If such a student is found,

```
1  % all_registered
2  %   Succeeds if all students are registered
3  %   for at least one course.
4  all_registered :-
5    \+ (
6      student(S),
7      \+ registered(S,_)
8    ).
```

Listing 6.5: *Example of a program using the forall double negation pattern.*

```
% forall(+P, +Q)
%   Succeeds iff goal Q is true
%   for every true instance of goal P.
forall(P, Q):-
  \+ (P, \+ Q).
```

Listing 6.6: *Implementation of meta-program* `forall/2`.

the negation immediately fails and `all_registered` is not true. Only if there is no such student, in other words `registered(S,_)` is found to succeed for every student, can the outer negation succeed. This is the check that we were attempting to perform, and negation as failure provides a succinct way to run it.

This double negation pattern provides a general implementation of "forall". So, the meta-program in Listing 6.6 can be used whenever you want "if `P` then `Q`" to be universally true.

6.4 Solutions to Exercises in Chapter 6

Solution 6.1

```
min_of_two(X, Y, Min):-
  X =< Y,
  !,
  Min = X.
min_of_two(X, Y, Y).
```

Solution 6.2

```
min_of_two(X, Y, Min):-
  X =< Y,
  !,
  Min = X.
min_of_two(X, Y, Y).

min_of_three(X, Y, Z, Min):-
  min_of_two(X, Y, MinXY),
  min_of_two(MinXY, Z, Min).
```

Solution 6.3

```
| ?- findall((X,Y), append(X, Y, [a,b,c]), Pairs).
```

Solution 6.4

```
  % all_friends(-List)
  %   List is a list of all pairs (Name, Num) where
  %   Name is a person that has Num friends.
  all_friends(List):-
    findall(
      (Name, NumFriends),
      friends(Name, NumFriends),
      List
    ).

  % friends(?Name, ?Num)
  %   Name is a person that has Num friends.
  friends(Name, Num):-
    person(Name),
    findall(F, friend(F, Name), Fs),
    length(Fs, Num).
```

Solution 6.5

```
% all_friendly(-Set)
%   Set is a duplicate-free, sorted list of
%   everyone that is a friend of somebody else.
all_friendly(Fs):-
  setof(F, friendly(F), Fs).

% friendly(-F)
%   F is a friend of somebody.
friendly(F):-
  friend(F, _).
```

Chapter 7

Metaprogramming

Metaprogramming concerns programs that variously evaluate or manipulate *other* programs. Prolog provides specific support for:

- constructing clauses and queries at runtime;
- executing dynamically generated queries using metapredicates;
- modifying the currently loaded program at runtime;
- making some of the currently defined predicates available as an object program for meta-interpreter code.

7.1 Queries as Data

Metaprogramming in Prolog is simplified by the fact that compound terms are syntactically identical to the subgoals that are the building blocks of clauses and queries. So, terms representing queries can occur within a metaprogram, and then be executed at the appropriate time.

7.1.1 Built-In Metapredicates

We have already seen the built-in metapredicates:

- `\+/1`
- `findall/3`
- `setof/3`

Each one accepts an argument that is a query (a tuple of subgoals). The query is evaluated during the execution of each predicate to produce a desired result. To this list we can add the predicate `call/1`, which simply evaluates its argument (also a query):

```
| ?- call(member(X, [a,b])).
X = a ;
X = b ;
```

7.1.2 Subgoals and Queries are Terms

In the example above, `call(member(X, [a,b]))` is a syntactically correct query because `member(X, [a,b])` is a syntactically correct compound term, meaning it can be passed as an argument. As a compound term it has functor `member` and inner terms `X` and `[a,b]`. It happens that this term also corresponds to a subgoal that Prolog knows how to evaluate.

Since any subgoal is a compound term, then any tuple of subgoals is also a compound term. (A tuple is the name used for a compound term without a functor. Tuples are terms.) So, metapredicates can also be passed tuples of subgoals:

```
| ?- call((member(X, [a,b]), Y = [X,z])).
X = a,
Y = [a,z] ? ;
X = b,
Y = [b,z] ? ;
```

7.1.3 (De)Constructing Compound Terms

The arguments of compound terms can be manipulated using the standard unification method. The following built-in predicates provide further ways of constructing and deconstructing terms.

`functor/3`

A call `functor(Term, Functor, Arity)` can either construct a new term with `Arity` uninstantiated arguments, or deconstruct a given term:

```
| ?- functor(T, factorial, 2).
T = factorial(_A,_B) ? ;

| ?- functor(factorial(3, 6), Func, Arity).
Func = factorial,
Arity = 2 ? ;
```

`arg/3`

A call `arg(ArgNum, Term, Arg)` allows one to extract an argument from a given term according to its numerical position (starting from 1):

```
| ?- arg(2, factorial(3, 6), Fac).
Fac = 6 ? ;
```

`=../2`

The `=..` predicate is normally used in infix form and converts between a compound term and a list containing the functor followed by the arguments in order:

```
| ?- factorial(3, 6) =.. Parts.
Parts = [factorial,3,6] ? ;

| ?- Goal =.. [append,[a],[b],L].
Goal = append([a],[b],L) ? ;
```

Exercise 7.1

```
% Some binary functions defined as ternary predicates ...

plus(X, Y, Z) :-
  Z is X + Y.

min(X, Y, Z) :-
  X < Y,
  !,
  Z = X.
min(_, Y, Y).

% etc.
```

Assume a set of ternary predicates defining various binary functions, such that the predicate arguments always follow the order: (left operand, right operand, value), as exemplified above. Write a definition of `evaluate(+Funcs,+Args,-Results)` such that the *n*th element of the list `Results` is the result of applying the *n*th function in the list `Funcs` (e.g. `'plus'`) to the *n*th ordered pair of operands in the list `Args` (e.g. `(2,2)`).

7.2 Clauses as Data

Clauses can also be arguments within metaprograms. Such clauses can then be used to update the currently loaded program. So, we can write applications that generate programs, e.g. through a learning process, as a translation into Prolog from some other language, or some other application.

7.2.1 Clauses are also Terms

Clauses (up to but not including the terminal `.`) are terms with functor `:-/2`. For instance, a clause

```
strong(X) :-
  lifts(X, rock).
```

is a term with inner terms `strong(X)` and `lifts(X, rock)`, as confirmed by this unification:

```
| ?- (strong(X) :- lifts(X,rock)) = :-(P,Q).
P = strong(X),
Q = lifts(X,rock) ? ;
```

7.2.2 Dynamic and Static Predicates

Clauses can be added to and removed from the currently loaded program at runtime. A predicate that is undefined in code loaded from source files is automatically *dynamic*, and can be freely modified. Predicates that are defined by clauses in loaded source code are *static* by default. Such predicates can only be redefined if they are explicitly made dynamic using a declaration:

```
:- dynamic factorial/2, min_of_two/3.
```

7.2.3 Adding and Removing Clauses

A clause defining a dynamic predicate can be added using `assert/1`:

```
| ?- person(X,Y).
no
| ?- assert(person(tim, kimber)).
yes
| ?- person(X,Y).
X = tim,
Y = kimber ? ;
```

The built-in predicate `retract/1` will remove from the current program the *first* clause that matches its argument:

```
| ?- assert(person(tim,kimber)).
yes
| ?- assert(person(tim,berners-lee)).
yes
| ?- retract(person(tim, Surname)).
Surname = kimber ? ;
```

Whereas `retractall/1` will remove all matching clauses:

```
| ?- retractall(person(_, _)).
yes
| ?- person(X, Y).
no
```

```
% solve(+Goals)
%   Goals is a tuple of one or more compound terms
%   of arity N >= 0, and Goals is a logical consequence
%   of the current dynamic program, under first-order logic.
solve(true) :- !.

solve((G, Gs)) :-
  !,
  solve(G),
  solve(Gs).

solve(G) :-
  clause(G, Body),
  solve(Body).
```

Listing 7.1: *The Prolog in Prolog meta-interpreter.*

7.3 Meta-interpreters

A meta-interpreter is a program that evaluates a language similar or identical to its own implementation language. The ease with which clauses and goals can be treated as data makes Prolog well-suited to writing meta-interpreters. We will look at a simple "Prolog in Prolog" interpreter as an example. This program is shown in Listing 7.1.

The Prolog in Prolog program defines a single predicate, `solve/1`. This predicate treats its argument as a Prolog query to be evaluated against the current dynamic clauses. This provides a convenient separation between the static meta-interpreter code, and a dynamic object program.

```
| ?- assert(person(tim,kimber)).
yes
| ?- solve(person(First,Last)).
First = tim,
Last = kimber ? ;
```

7.3.1 Searching Dynamic Clauses: `clause/2`

The operation of `solve/1` is based on the built-in predicate `clause/2`. A call `clause(Head, Body)` searches for a dynamic clause whose head matches the compound term `Head` and whose body matches the tuple `Body`. If `Head` matches an unconditional clause, then Body is the atom `true`.

```
| ?- clause(person(X,Y), Body).
X = tim,
Y = kimber,
Body = true ? ;
```

The predicate of `Head` must be given. All matching clauses will be found on backtracking.

```
% solve(+Goals)
%   Goals is a list of zero or more compound terms
%   of arity N >= 0, and Goals is a logical consequence
%   of the current dynamic program, under first-order logic.
solve([]).

solve([G|Gs]) :-
  clause_list([G|Body]),
  append(Body, Gs, DerivedGs),
  solve(DerivedGs).
```

Listing 7.2: *An alternative Prolog in Prolog meta-interpreter.*

```
| ?- assert((min_of_two(X,Y,X) :- X < Y)).
yes
| ?- assert((min_of_two(X,Y,Y) :- X >= Y)).
yes
| ?- clause(min_of_two(2,3,Min), Body).
Min = 2,
Body = (2<3) ? ;
Min = 3,
Body = (2>=3) ? ;
```

The third clause of the `solve` program uses `clause/2` to evaluate its input query. The first and second clauses deal with the cases when the query is the atom `true`, and when it contains at least two subgoals, respectively. Having eliminated these cases the input to the third clause can only be a single subgoal, which can therefore be passed into `clause`.

7.3.2 Beyond Prolog in Prolog

Having defined the most basic meta-interpreter, Prolog in Prolog, it can serve as a starting point for extensions and variations. For example, an alternative version of Prolog in Prolog is shown in Listing 7.2.

In Listing 7.2 the definition of `solve` has been simplified by changing the representation of queries from tuples to lists. The structure of the original program, with three clauses and two cuts, was forced by the different forms that `Body` can take in `clause(Head,Body)`. This can be reduced to just two cases, the usual empty and nonempty lists, by avoiding the use of `clause/2` altogether. It is replaced by a user-defined alternative, `clause_list/1`. (See Exercise 7.2.)

Exercise 7.2

```
member(X, [X|_]).
member(X, [_|Tail]) :-
  member(X, Tail).
```

The definition of `member/2` is shown above. Define `clause_list/1` so that the meta-interpreter in Listing 7.2 can solve queries with `member/2` subgoals.

The basic interpreter can also be extended to provide extra features, if they are appropriate for a particular application. For example, the depth-first, left-to-right search strategy of Prolog could be overridden, information about a proof could be collected, or the user could even be treated as a source of extra knowledge — an "oracle" — when a goal cannot be proved using the saved program.

7.4 Solutions to Exercises in Chapter 7

Solution 7.1

```
% evaluate(+Funcs, +Args, -Results)
%  The nth element of the list Results is the result
%  of applying the nth binary function in Functions
%  to the nth pair of operands in the list Inputs.
evaluate([], _, []).
evaluate([Fun|Funs], [(X,Y)|Inputs], [Z|Zs]) :-
  Goal =.. [Fun,X,Y,Z],
  call(Goal),
  evaluate(Funs, Inputs, Zs).
```

Solution 7.2

```
:- dynamic clause_list/1.

clause_list([member(X, [X|_])]).
clause_list([member(X, [_|Tail]), member(X, Tail)].
```

Chapter 8

Definite Clause Grammars

Language processing and formal grammars are closely linked to the origins of Prolog. The pioneers of Prolog, in particular Robert Kowalski, a former Head of the Department of Computing at Imperial College London, and Alain Colmerauer of the Université d'Aix Marseilles, were all interested in such applications [1]. Kowalski and Colmerauer showed that grammar rules translate directly into Prolog clauses, providing a simple means of writing a language parser [2]. Such a program is called a *definite clause grammar* (DCG), because a Prolog program clause without negation is called a definite clause. This chapter describes how to define and use different kinds of DCG.

8.1 Grammars

A formal grammar, like the example in Fig. 8.1 is a set of rules, called *production rules* or *rewrite rules*, that defines every possible string in a language. Each rule is of the form:

⟨*N*⟩ ::= S

where ⟨*N*⟩ is a *non-terminal* symbol. Non-terminals do not appear in the strings of the language, they are only used in the process of applying the rules. The right-hand side of the rule, S, can contain both non-terminals and *terminal* symbols. The strings of the language are strings of terminal symbols, and every one can be generated as follows:

1. Initialise the string to the *start symbol* (⟨*sentence*⟩ in Fig. 8.1).
2. For every non-terminal symbol ⟨*N*⟩ in the string, apply the rule with ⟨*N*⟩ on the left by replacing ⟨*N*⟩ with one of the alternatives on the right. (Alternatives are separated by |.)
3. Repeat (2) until the string contains only terminal symbols.

For instance, some of the possible strings in the language defined in Fig. 8.1 are:

- `the man eats an apple`
- `a man sings`
- `the dog eats`

⟨*sentence*⟩ ::= ⟨*noun_phrase*⟩ ⟨*verb_phrase*⟩

⟨*noun_phrase*⟩ ::= ⟨*determiner*⟩ ⟨*noun*⟩

⟨*verb_phrase*⟩ ::= ⟨*verb*⟩
| ⟨*verb*⟩ ⟨*noun_phrase*⟩

⟨*determiner*⟩ ::= `the`
| `a`
| `an`

⟨*noun*⟩ ::= `man`
| `apple`
| `dog`

⟨*verb*⟩ ::= `eats`
| `sings`

Figure 8.1: *A grammar for a simple English-like language.*

8.2 Syntax Checking

The first programming task when dealing with language texts is to confirm whether the text can be generated by the grammar defining the language. Continuing with the example language from Section 8.1, we want to design a program that will confirm that "the man eats an apple" is a sentence, but "sings a dog" is not.

In the remainder of the chapter a text will be represented by a list of Prolog atoms like `[the, man, eats, an, apple]`. We will assume the existence of a method of translating other forms of text, such as a string of characters, into this form.

8.2.1 Experiments with `append/3`

The production rules in Fig. 8.1 already closely resemble Prolog clauses. It is easy to translate them into a working program that correctly recognises sentences in our list representation. All that is needed is to append lists to one another in the correct order. Listing 8.1 contains such a program.

However, there is a problem with this approach. It is not terribly efficient (see Exercise 8.1).

Exercise 8.1

Consider the evaluation of the query

```
| ?- sentence([sings,a,dog]).
```

by the program in Listing 8.1. Give a succinct description of what happens.

Any program to solve this problem must compare the input text to legitimate phrases in the language. The problem with this particular program is that the comparison only happens once full sentences have been constructed. So, even though an input like `[sings,a,dog]` does not start with a valid noun phrase, the program will not just try each possible noun phrase, it will also combine each one with every possible verb phrase before finally deciding `[sings,a,dog]` is not a

```
% sentence(?S)
%   The list of atoms S is a sentence.
sentence(S) :-
  noun_phrase(NP),
  verb_phrase(VP),
  append(NP,VP,S).

% noun_phrase(?NP)
%   The list of atoms NP is a noun phrase.
noun_phrase(NP) :-
  determiner(D),
  noun(N),
  append(D,N,NP).

% verb_phrase(?VP)
%   The list of atoms VP is a verb phrase.
verb_phrase(V) :-
  verb(V).
verb_phrase(VP) :-
  verb(V),
  noun_phrase(NP),
  append(V,NP,VP).

determiner([the]).
determiner([a]).
determiner([an]).

noun([man]).
noun([apple]).

verb([eats]).
verb([sings]).
```

Listing 8.1: *The first implementation of the English-like grammar, using* `append`.

sentence. There must be a better way.

As you have no doubt observed, Listing 8.1 is really a sentence generation program. No input is passed into `noun_phrase` or `verb_phrase` to be checked. The program can simply be reorganised to change this. The definition of `sentence` would become:

```
sentence(S) :-
  append(NP,VP,S),
  noun_phrase(NP),
  verb_phrase(VP).
```

This is an improvement of sorts, but there are still problems. Rather than guessing sentences of the language and comparing them to the input, this program is now guessing parts (prefixes) of the input, that might be a correct input for `noun_phrase`.

8.2.2 Definite Clause Grammars

A definite cause grammar combines the best parts of the two approaches using `append`. The aim is to generate parts of the language, but in small chunks, not whole sentences, starting with those parts that can occur at the start of a sentence. These chunks are compared to the start of the input. If a match is found, then the chunk — an actual part of a sentence in the language — is split off the input, rather than splitting off a guess.

A DCG version of the program is shown in Listing 8.2. The first obvious difference is that every predicate now has two arguments. Although the program can generate sentences too, it is easiest to start by thinking about the case when an input is being checked. The first argument of each clause is the input. The second argument is the remainder of the input after some prefix has been removed if the clause succeeded. The prefix removed must be of the correct type. So, `noun_phrase/2` removes a noun phrase, `determiner/2` removes a determiner, etc. The program can be called as follows:

```
| ?- sentence([sings,the,man], R).
no
| ?- sentence([the,man,eats,an,apple], R).
R = [an,apple] ? ;
R = [] ? ;
```

The second query has succeeded twice, having found two different prefixes that are sentences, with different remainders.

8.2.3 Difference Lists

The declarative reading of the DCG clauses is that they are true when the *difference* between their arguments is a valid phrase of the correct type. This is referred to as the *difference list* representation. So, the DCG program does not define `[the,man]` as a noun phrase, it defines the pair of lists `([the,man,eats],[eats])` as a noun phrase.

8.2.4 DCG Syntax

DCG clauses all follow the same pattern, with one argument for the input and one for the remainder (or the difference list pair, if you prefer). To simplify the task of writing DCG programs, Prolog has a special syntax that avoids listing all these repetitive arguments.

```
% sentence(?S, ?R)
%   The result of removing the list R from
%   the list S is a sentence.
sentence(S, R) :-
  noun_phrase(S, VP),
  verb_phrase(VP, R).

% noun_phrase(?NP, ?R)
%   The list NP-R is a noun phrase.
noun_phrase(NP, R) :-
  determiner(NP, N),
  noun(N, R).

% verb_phrase(?VP, ?R)
%   The list VP-R is a verb phrase.
verb_phrase(VP, R) :-
  verb(VP, R).
verb_phrase(VP, R) :-
  verb(VP, NP),
  noun_phrase(NP, R).

determiner([the|R], R).
determiner([a|R], R).
determiner([an|R], R).

noun([man|R], R).
noun([apple|R], R).
noun([dog|R], R).

verb([eats|R], R).
verb([sings|R], R).
```

Listing 8.2: *The English-like grammar, implemented as a DCG.*

```
1   % sentence(?S, ?R).                    % Expands to ...
2   %   S-R is a sentence.
3   sentence -->                           % sentence(X, Y) :-
4     noun_phrase,                         %   noun_phrase(X, X1),
5     verb_phrase.                         %   verb_phrase(X1, Y).
6
7   % noun_phrase(?NP, ?R).
8   %   NP-R is a noun phrase.
9   noun_phrase -->                        % noun_phrase(X, Y) :-
10    determiner,                          %   determiner(X, X1),
11    noun.                                %   noun(X1, Y).
12
13  % verb_phrase(?VP, ?R)
14  %   VP-R is a verb phrase.
15  verb_phrase -->                        % verb_phrase(X, Y) :-
16    verb.                                %   verb(X, Y).
17  verb_phrase -->                        % verb_phrase(X, Y) :-
18    verb,                                %   verb(X, X1),
19    noun_phrase.                         %   noun_phrase(X1, Y).
20
21  determiner --> [the].                  % determiner(X, Y) :-
22  determiner --> [a].                    %   X = [the|Y].
23  determiner --> [an].                   % etc.
24
25  noun --> [man].                        % noun(X, Y) :-
26  noun --> [apple].                      %   X = [man|Y].
27  noun --> [dog].                        % etc.
28
29  verb --> [eats].                       % verb(X, Y) :-
30  verb --> [sings].                      %   X = [eats|Y].   etc.
```

Listing 8.3: *The DCG grammar program, using DCG syntax.*

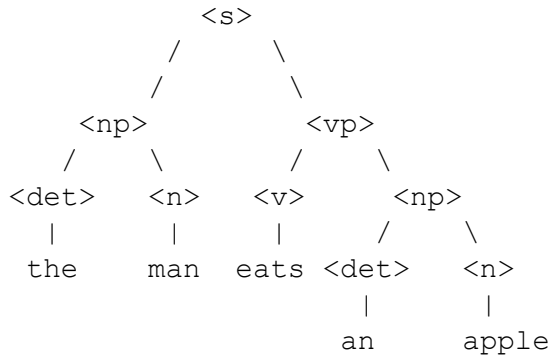
Listing 8.3 shows a version of the program using DCG syntax. The important features are that the non-terminals are written without their difference list arguments and terminals are written within lists. When this code is loaded it is automatically expanded into an ordinary program that is virtually identical to Listing 8.2. The expanded program can be queried as you would any program. So, having loaded the code in Listing 8.3, the queries in Section 8.2.2 will work. Alternatively, the built-in predicate `phrase/[2,3]`, can be used to query the unexpanded program as shown below.

```
| ?- phrase(sentence, [the,man,eats,an,apple]).
yes
| ?- phrase(noun_phrase, [the,man,eats,an,apple], R).
R = [eats,an,apple] ? ;
```

8.3 Extending the Syntax Checker

8.3.1 Outputting a Syntax Tree

Having successfully solved the syntax checking problem, we want to extend the program to perform other parsing tasks, such as building a syntax tree:



This parser will copy words in the text into compound terms that form the tree. A tree representing a sentence will be a term `s(NP,VP)` where `NP` is a tree `np(Det,N)` representing a noun phrase, and `VP` is a tree `vp(V)` or `vp(V,NP)` representing a verb phrase, and so on.

The rule for `determiner` in this parser is:

```
% determiner(?Tree, ?X, ?Y)
%   Tree is a syntax tree representing the determiner X-Y.
determiner(det(D)) -->
  [D],
  {determiner(D)}.
```

This illustrates several other features of DCG rules. In the body, `[D]` has replaced more specific terms like `[the]`. This will still match a terminal symbol (a word in the text), but now we use `D` to match whatever is the first word of the input. It is followed by `{determiner(D)}` which checks that the word `D` is correct. Code in DCG syntax enclosed in `{ }` is left unchanged when the rules are expanded into standard Prolog syntax, so `{determiner(D)}` is not calling the DCG rule, which has three arguments when it is expanded, not one. It might be helpful to see the expanded rule at this point:

```
% determiner(?Tree, ?X, ?Y)
%   Tree is a syntax tree representing the determiner X-Y.
determiner(det(D), X, Y) :-
  X = [D|Y],
  determiner(D).
```

The role of `determiner(D)` is to look up `D` in a "dictionary" of `determiner/1` facts:

```
% determiner(?D)
determiner(the).
determiner(an).
determiner(a).
```

Putting the terminals into `determiner/1` like this is just a way of keeping the program tidy — the DCG rule is now longer and doing this means we now only need one, plus the dictionary of facts.

Going back to the DCG rule, the last thing to note is the head of the rule, which now has an argument `det(D)`. DCG rules do not have to be limited to the two text input/ouput (difference list) arguments. These two arguments will always be added when the rule is expanded, and they will be in addition to any arguments given in the DCG syntax rule. The new argument is the one used to build the syntax tree. The terminal `D` is copied into this term, which is the tree representation of a single determiner. The new rule can be called with phrase as before.

```
| ?- phrase(determiner(ST),[the,man,eats], R).
ST = det(the),
R = [man,eats] ?
```

Exercise 8.2

Write the DCG rules for `sentence`, `noun_phrase` and `verb_phrase` for the syntax tree parser. These rules should succeed if the text has a prefix that satisfies the grammar production rules in Fig. 8.1, and return appropriate syntax tree structures, as described in Section 8.3.1.

8.3.2 Making the Parser Case-Sensitive

The grammar in Fig. 8.1 is *context-free*: a phrase generated by a production rule can always be included in a sentence regardless of the phrases around it. This becomes a problem when more words such as `men` and `eat` are added to the language. The language now includes sentences such as:

- "The men eats an apple"
- "A dog eat"

To solve this problem we start by adding case information to the dictionary facts:

```
% determiner(?D, ?Case)
determiner(the,_).
determiner(an,s).      % s is `singular'
determiner(a,s).
```

```
noun(man,s).
noun(men,p).            % p is `plural'
noun(apple,s).
```

and then return this information in the DCG rule.

```
% determiner(?Tree, ?Case, ?X, ?Y)
%   Tree is a syntax tree representing
%   the determiner X-Y, which has case Case.
determiner(det(D), Case) -->
  [D],
  {determiner(D, Case)}.
```

After completing the program (see Exercise 8.3) it should generate output like this:

```
| ?- phrase(sentence(ST), [the,men,eats,the,apple]).
no
| ?- phrase(sentence(ST), [the,men,eat,the,apple]).
ST = s(np(det(the),n(men)),vp(v(eat),np(det(the),n(apple)))) ?
```

Exercise 8.3

Write the DCG rules for `sentence`, `noun_phrase` and `verb_phrase` for the case-sensitive parser. These rules should generate syntax trees, and ensure that sentences in the language are case-sensitive. (Think carefully about the second verb phrase rule.)

8.4 Further Examples

The Sicstus manual includes some other useful examples.

This rule shows how terminals and non-terminals can be mixed in a single rule, and how sequences of terminals can be specified to occur together.

```
                        %   Expands to ...
p(X) -->                %   p(X, S, R) :-
  [go, to],             %     S = [go, to|S1],
  q(X),                 %     q(X, S1, S2),
  [stop].               %     S2 = [stop|R].
```

And the program in Listing 8.4 parses and evaluates arithmetic expressions containing single digit numbers and operators. Note that the text parsed by this program is a list of character codes, which are converted into integers by the last clause. Such a list in Prolog can be written as a double-quoted string containing the characters. So, the string `"2+2*5"` is the same as the list `[50,43,50,42,53]`. The program can be called using `phrase/[2,3]` in the usual way.

```
% expr(Z, X, Y).
%   Z is the value of the expression text X-Y.
expr(Z) --> term(X), "+", expr(Y), {Z is X + Y}.
expr(Z) --> term(X), "-", expr(Y), {Z is X - Y}.
expr(X) --> term(X).

% term(Z, X, Y).
%   Z is the value of the term text X-Y.
term(Z) --> number(X), "*", term(Y), {Z is X * Y}.
term(Z) --> number(X), "/", term(Y), {Z is X / Y}.
term(Z) --> number(Z).

% number(Z, X, Y).
%   Z is the value of the number text X-Y.
number(C) --> "+", number(C).
number(C) --> "-", number(X), {C is -X}.
number(X) --> [C], {"0"=<C, C=<"9", X is C - "0"}.
```

Listing 8.4: *An interpreter for strings containing arithmetic expressions, from the Sictsus manual.*

```
| ?- phrase(expr(Z), "1-3+2*2").
Z = -6 ?
```

References

[1] Robert A. Kowalski. “The early years of logic programming”. In: *Communications of the ACM* 31.1 (1988), pp. 38–43.

[2] Fernando C.N. Pereira and David H.D. Warren. “Definite clause grammars for language analysis—a survey of the formalism and a comparison with augmented transition networks”. In: *Artificial intelligence* 13.3 (1980), pp. 231–278.

8.5 Solutions to Exercises in Chapter 8

Solution 8.1

Every sentence in the language will be generated and compared to `[sings,a,dog]`.

Solution 8.2

```
% sentence(?Tree, ?S, ?R)
%   Tree is a compound term representing
%   the sentence S-R as a syntax tree.
sentence(s(NP,VP)) -->
  noun_phrase(NP),
  verb_phrase(VP).

% noun_phrase(?Tree, ?NP, ?R)
%   Tree is a syntax tree for the noun phrase NP-R.
noun_phrase(np(D,N)) -->
  determiner(D),
  noun(N).

% verb_phrase(?Tree, ?VP, ?R)
%   Tree is a syntax tree for the verb phrase VP-R.
verb_phrase(vp(V)) -->
  verb(V).
verb_phrase(vp(V,NP)) -->
  verb(V),
  noun_phrase(NP).
```

Solution 8.3

```
% sentence(?Tree, ?S, ?R)
%   Tree is a compound term representing
%   the sentence S-R as a syntax tree.
sentence(s(NP,VP)) -->
  noun_phrase(NP, Case),
  verb_phrase(VP, Case).

% noun_phrase(?Tree, ?Case, ?NP, ?R)
%   Tree is a syntax tree for the noun phrase NP-R,
%   with case Case.
noun_phrase(np(D,N), Case) -->
  determiner(D, Case),
  noun(N, Case).

% verb_phrase(?Tree, ?Case, ?VP, ?R)
%   Tree is a syntax tree for the verb phrase VP-R,
%   with case Case.
verb_phrase(vp(V), Case) -->
```

```
    verb(V, Case).
verb_phrase(vp(V,NP), Case) -->
    verb(V, Case),
    noun_phrase(NP, _).     % Case does not have to match.
```