



INSTITUTE FOR LOGIC,
LANGUAGE AND COMPUTATION

Lecture Notes
An Introduction to Prolog Programming

Ulle Endriss

UNIVERSITEIT VAN AMSTERDAM

Exercises

Exercise 1.1. Try to answer the following questions first “by hand” and then verify your answers using a Prolog interpreter.

- (a) Which of the following are valid Prolog atoms?

`f`, `loves(john,mary)`, `Mary`, `_c1`, `'Hello'`, `this_is_it`

- (b) Which of the following are valid names for Prolog variables?

`a`, `A`, `Paul`, `'Hello'`, `a_123`, `_`, `_abc`, `x2`

- (c) What would a Prolog interpreter reply given the following query?

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

- (d) Would the following query succeed?

`?- loves(mary, john) = loves(John, Mary).`

Why?

- (e) Assume a program consisting only of the fact

`a(B, B).`

has been consulted by Prolog. How will the system react to the following query?

`?- a(1, X), a(X, Y), a(Y, Z), a(Z, 100).`

Why?

Exercise 1.2. Read the section on matching again and try to understand what's happening when you submit the following queries to Prolog.

- (a) `?- myFunctor(1, 2) = X, X = myFunctor(Y, Y).`
- (b) `?- f(a, _, c, d) = f(a, X, Y, _).`
- (c) `?- write('One '), X = write('Two ').`

Exercise 1.3. Draw the family tree corresponding to the following Prolog program:

```
female(mary).
female(sandra).
female(juliet).
female(lisa).
male(peter).
male(paul).
male(dick).
male(bob).
male(harry).
parent(bob, lisa).
parent(bob, paul).
parent(bob, mary).
parent(juliet, lisa).
parent(juliet, paul).
parent(juliet, mary).
parent(peter, harry).
parent(lisa, harry).
parent(mary, dick).
parent(mary, sandra).
```

After having copied the given program, define new predicates (in terms of rules using `male/1`, `female/1` and `parent/2`) for the following family relations:

- (a) father
- (b) sister
- (c) grandmother
- (d) cousin

You may want to use the operator `\=`, which is the opposite of `=`. A goal like `X \= Y` succeeds, if the two terms `X` and `Y` cannot be matched.

Example: `X` is the brother of `Y`, if they have a parent `Z` in common and if `X` is male and if `X` and `Y` don't represent the same person. In Prolog this can be expressed by means of the following rule:

```
brother(X, Y) :-  
    parent(Z, X),  
    parent(Z, Y),  
    male(X),  
    X \= Y.
```

Exercise 1.4. Recall our big animal program consisting of four facts and two rules. Change the order of the two subgoals of the second rule. What happens when you execute the following query, asking for all possible alternative solutions using the semicolon key? Briefly explain *why* this happens.

```
?- is_bigger(A, donkey).
```

Advice: Try to predict what will happen before trying it on the computer.

Exercise 1.5. Most people will probably find all of this rather daunting at first. Read the chapter again in a few weeks' time when you will have gained some programming experience in Prolog and enjoy the feeling of enlightenment. The part on the syntax of the Prolog language and the stuff on matching and goal execution are particularly important.

2.4 Exercises

Exercise 2.1. Write a Prolog predicate `analyse_list/1` that takes a list as its argument and prints out the list's head and tail on the screen. If the given list is empty, the predicate should put out a message reporting this fact. If the argument term isn't a list at all, the predicate should just fail. Examples:

```
?- analyse_list([dog, cat, horse, cow]).  
This is the head of your list: dog  
This is the tail of your list: [cat, horse, cow]  
Yes
```

```
?- analyse_list([]).  
This is an empty list.  
Yes
```

```
?- analyse_list(sigmund_freud).  
No
```

Exercise 2.2. Write a Prolog predicate `membership/2` that works like the built-in predicate `member/2` (without using `member/2`).

Hint: This exercise, like many others, can and should be solved using a recursive approach and the head/tail-pattern for lists.

Exercise 2.3. Implement a Prolog predicate `remove_duplicates/2` that removes all duplicate elements from a list given in the first argument and returns the result in the second argument position. Example:

```
?- remove_duplicates([a, b, a, c, d, d], List).  
List = [b, a, c, d]  
Yes
```

Exercise 2.4. Write a Prolog predicate `reverse_list/2` that works like the built-in predicate `reverse/2` (without using `reverse/2`). Example:

```
?- reverse_list([tiger, lion, elephant, monkey], List).  
List = [monkey, elephant, lion, tiger]  
Yes
```

Exercise 2.5. Consider the following Prolog program:

```
whoami([]).
```

```
whoami([_, _ | Rest]) :-
    whoami(Rest).
```

Under what circumstances will a goal of the form `whoami(X)` succeed?

Exercise 2.6. The objective of this exercise is to implement a predicate for returning the last element of a list in two different ways.

- (a) Write a predicate `last1/2` that works like the built-in predicate `last/2` using recursion and the head/tail-pattern for lists.
- (b) Define a similar predicate `last2/2` solely in terms of `append/3`, without explicitly using recursion yourself.

Exercise 2.7. Write a predicate `replace/4` to replace all occurrences of a given element (second argument) by another given element (third argument) in a given list (first argument). Example:

```
?- replace([1, 2, 3, 4, 3, 5, 6, 3], 3, x, List).
List = [1, 2, x, 4, x, 5, 6, x]
Yes
```

Exercise 2.8. Prolog lists without duplicates can be interpreted as sets. Write a program that given such a list computes the corresponding power set. Recall that the power set of a set S is the set of all subsets of S . This includes the empty set as well as the set S itself.

Define a predicate `power/2` such that, if the first argument is instantiated with a list, the corresponding power set (i.e., a list of lists) is returned in the second position. Example:

```
?- power([a, b, c], P).
P = [[a, b, c], [a, b], [a, c], [a], [b, c], [b], [c], []]
Yes
```

Note: The order of the sub-lists in your result doesn't matter.

Exercise 2.9. Write a predicate `longer/2` that takes two lists as arguments and succeeds if the second is longer (has more elements) than the first. Implement your solution using only the tools and techniques introduced so far (in particular, do not make use of any arithmetic expressions, i.e., do not make use of any of the material to be covered in the next chapter). Examples:

```
?- longer([dog,cat,snake], [giraffe,elephant,lion,tiger]).
Yes

?- longer([1,2,3,4,5], []).
No
```

Exercise 2.10. This exercise is about numbers. You are used to representing, say, the number *twelve* using the decimal system, in which it is written as ‘12’. But you could also use the *unary* system, in which it can be written as ‘111111111111’. In the next chapter we will see how to work with numbers in the usual decimal system, but you actually already know everything you need to know to work with numbers in the unary system. Your task will be to implement some basic arithmetical operations for working with unary numbers. We will represent unary numbers as lists of *x*’s of the appropriate length. Thus, *five* would be `[x,x,x,x,x]`, *twelve* would be `[x,x,x,x,x,x,x,x,x,x,x,x]`, and *zero* would be `[]`. In the sequel, all numbers are understood to be such non-negative integers given in unary notation.

- (a) The *successor* of a number is the number we obtain if we add *one* to it. Thus, for example, the successor of *five* is *six*. Write a predicate called `successor/2` that will return, in the second argument position, the successor of the number provided in the first argument position. Examples:

<code>?- successor([x, x, x], Result).</code>	<code>?- successor([], Result).</code>
<code>Result = [x, x, x, x]</code>	<code>Result = [x]</code>
<code>Yes</code>	<code>Yes</code>

- (b) Implement a predicate `plus/3` to compute the sum of two given numbers. Example:

```
?- plus([x, x], [x, x, x, x], Result).
Result = [x, x, x, x, x, x]
Yes
```

- (c) Implement a predicate `times/3` to multiply two given numbers. Examples:

```
?- times([x, x], [x, x, x, x], Result).
Result = [x, x, x, x, x, x, x, x]
Yes

?- times([x, x, x], [x, x, x, x, x], Result), write(Result).
[x, x, x, x, x, x, x, x, x, x, x, x, x, x, x]
Result = [x, x, x, x, x, x, x, x, x, x|...]
Yes
```

Note that in the last example, the result (a list of fifteen *x*’s) is too long for Prolog to print, so we force printing using the `write`-command at the end of our query. Make sure your predicate works correctly also when one of the numbers is *zero*.

Hint: You don’t need to use any “normal” numbers in your program and you should not use any arithmetic operations provided by Prolog (to be covered in the next chapter).

Exercises

Exercise 3.1. Write a Prolog predicate `distance/3` to calculate the distance between two points in the 2-dimensional plane. Points are given as pairs of coordinates. Examples:

```
?- distance((0,0), (3,4), X).
```

```
X = 5.0
```

```
Yes
```

```
?- distance((-2.5,1), (3.5,-4), X).
```

```
X = 7.810249675906654
```

```
Yes
```

Exercise 3.2. Write a Prolog program to print out a square of $n \times n$ given characters on the screen. Call your predicate `square/2`. The first argument should be a (positive) integer, the second argument the character (any Prolog term) to be printed. Example:


```
?- square(5, '* ').
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
```

Yes

Exercise 3.3. Write a Prolog predicate `fibonacci/2` to compute the n th Fibonacci number. The Fibonacci sequence is defined as follows:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \quad \text{for } n \geq 2 \end{aligned}$$

Examples:

```
?- fibonacci(1, X).
X = 1
Yes

?- fibonacci(3, X).
X = 2
Yes

?- fibonacci(7, X).
X = 13
Yes
```

While some authors define the sequence slightly differently (with $F_0 = 1$), your implementation should conform to the definition given above.

Exercise 3.4. This exercise assumes you have already solved the previous one. In fact, it is not too difficult to translate the mathematical definition of the Fibonacci sequence into a working Prolog predicate for computing the n th Fibonacci number. However, the most straightforward implementation is not very efficient at all and will run out of memory for larger numbers (try it!). Examples:

```
?- fibonacci(10, X).
X = 55
Yes

?- fibonacci(20, X).
```

```
X = 6765
```

```
Yes
```

```
?- fibonacci(50, X).
```

```
ERROR: Out of local stack
```

Briefly explain what the source of this problem is. Then write a Prolog predicate `fastfibonacci/2` that can compute any of the first 100 Fibonacci numbers in under 100th of a second. Examples:

```
?- fastfibonacci(50, X).
```

```
X = 12586269025
```

```
Yes
```

```
?- fastfibonacci(100, X).
```

```
X = 354224848179261915075
```

```
Yes
```

What is the 42nd Fibonacci number?

Exercise 3.5. Write a Prolog predicate `element_at/3` that, given a list and a natural number n , will return the n th element of that list. Examples:

```
?- element_at([tiger, dog, teddy_bear, horse, cow], 3, X).
```

```
X = teddy_bear
```

```
Yes
```

```
?- element_at([a, b, c, d], 27, X).
```

```
No
```

Exercise 3.6. Write a Prolog predicate `mean/2` to compute the arithmetic mean of a given list of numbers. Example:

```
?- mean([1, 2, 3, 4], X).
```

```
X = 2.5
```

```
Yes
```

Exercise 3.7. Write a Prolog predicate `minimum/2` to find the smallest number within a given list of numbers. Example:

```
?- minimum([4, 6, 8, 3, 5, 7], Result).
```

```
Result = 3
```

```
Yes
```

What does your predicate do when given the empty list as input? Is that the correct answer? Why?

Exercise 3.8. Write a predicate `range/3` to generate all integers between a given lower and a given upper bound. The lower bound should be given as the first argument, the upper bound as the second. The result should be a list of integers, which is returned in the third argument position. If the upper bound specified is lower than the given lower bound, the empty list should be returned. Examples:

```
?- range(3, 11, X).
X = [3, 4, 5, 6, 7, 8, 9, 10, 11]
Yes

?- range(7, 4, X).
X = []
Yes
```

Exercise 3.9. This exercise demonstrates how to implement a simple database in Prolog. Copy the following list of eight facts (the data) into a program file:

```
born(jan, date(20,3,1977)).
born(jeroen, date(2,2,1992)).
born(joris, date(17,3,1995)).
born(jelle, date(1,1,2004)).
born(jesus, date(24,12,0)).
born(joop, date(30,4,1989)).
born(jannecke, date(17,3,1993)).
born(jaap, date(16,11,1995)).
```

That is, we are representing dates as terms of the form `date(Day,Month,Year)`.

- (a) Write a predicate `year/2` to retrieve all people born in a given year (through repeated backtracking). Example:

```
?- year(1995, Person).
Person = joris ;
Person = jaap ;
No
```

- (b) Implement a predicate `before/2` that, when given two `date`-expressions, will succeed if the first expression represents a date before the date represented by the second expression (you may assume that the user will only ask for well-formed dates, e.g., not for the 31st of April, and so forth). Example:

```
?- before(date(31,1,1990), date(7,7,1990)).
Yes
```

- (c) Implement a predicate `older/2` that succeeds in case the person given first is (strictly) older than the person given second. Example:

```
?- older(jannecke, X).
X = joris ;
X = jelle ;
X = jaap ;
No
```

You should get 28 solutions for the query `older(X, Y)`. Explain why.

Exercise 3.10. Imagine you have built a robot that can execute three different commands: turn *right* (by 90 degrees), turn *left* (by 90 degrees), and *move* forward (by 1 metre). Suppose you place your robot on a grid at position (0,0), facing north. Your ultimate task is to write a Prolog predicate `status/3` that will return the robot's position and orientation after having executed a given list of commands. For example, if your robot first moves forward twice, then turns right, and then moves three more times, then it will be at position (3,2), facing east. Examples:

```
?- status([move, move, right, move, move, move], Position, Orientation).
Position = (3,2)
Orientation = east
Yes
```

```
?- status([], Position, Orientation).
Position = (0,0)
Orientation = north
Yes
```

```
?-status([left, left, move], Position, Orientation).
Position = (0,-1)
Orientation = south
Yes
```

Start by writing a predicate `execute/5` for executing a single command: it should take the current position, the current orientation, and a single command (one of the atoms `right`, `left`, `move`) as input in the first three argument positions, and return the new position and orientation in the last two argument positions. Note that positions are pairs of the form (X,Y) , with X and Y representing integers, while the orientation has to be one of the four atoms `north`, `south`, `west`, `east`.

Then implement a predicate `status/5` that takes as input the current position, the current orientation, and the list of commands still to be executed, and that returns the final position and final orientation. That is, this predicate is like the predicate `status/3` you are ultimately supposed to implement, except that it also includes the current position and orientation as input. Finally, implement the predicate `status/3` as specified above.

Exercise 3.11. Polynomials can be represented as lists of pairs of coefficients and exponents. For example the polynomial

$$4x^5 + 2x^3 - x + 27$$

can be represented as the following Prolog list:

```
[(4,5), (2,3), (-1,1), (27,0)]
```

Write a Prolog predicate `poly_sum/3` for adding two polynomials using that representation. Try to find a solution that is independent of the ordering of pairs inside the two given lists. Likewise, your output doesn't have to be ordered. Examples:

```
?- poly_sum([(5,3), (1,2)], [(1,3)], Sum).
Sum = [(6,3), (1,2)]
Yes

?- poly_sum([(2,2), (3,1), (5,0)], [(5,3), (1,1), (10,0)], X).
X = [(4,1), (15,0), (2,2), (5,3)]
Yes
```

Hints: Before you even start thinking about how to do this in Prolog, recall how the sum of two polynomials is actually computed. A rather simple solution is possible using the built-in predicate `select/3`. Note that the list representation of the sum of two polynomials that don't share any exponents is simply the concatenation of the two lists representing the arguments.

Exercise 3.12. Recall that the set of prime numbers is $\{2, 3, 5, 7, 11, 13, 17, \dots\}$, i.e., the set of numbers with exactly two divisors each (namely 1 and the number itself). Write a Prolog predicate `prime/1` to check whether given number is prime. Examples:

```
prime(17).    prime(18).
Yes           No
```

Exercise 3.13. In 1742, in a letter to the famous mathematician Leonhard Euler, Christian Goldbach conjectured that every even integer greater than 2 can be expressed as the sum of two prime numbers. In his own words (quote taken from Wikipedia, consulted on 3 August 2015):

“Dass ... ein jeder numerus par eine summa duorum primorum sey, halte ich für ein ganz gewisses theorema, ungeachtet ich dasselbe nicht demonstrieren kann.”

To this date, nobody has been able to prove the truth of this statement for *all* integers, although it has been verified for very many of them with the help of computers.

Write a predicate called `goldbach/2` that, when given an even integer greater than 2 in the first argument position, will return an expression of the form `A + B`, such that both `A` and `B` are prime numbers and their sum is equal to the input number. Examples:

```
?- goldbach(30, Solution).  
Solution = 7+23  
Yes  
  
?- goldbach(17420000, Solution).  
Solution = 109+17419891  
Yes
```

Start by implementing a predicate `prime/1` to test whether a given number is prime. Then think about what you need to do to find two prime numbers that add up to a given number N . First you need to choose the first number A , which can be any number between 2 and $N/2$ (think about why these are the correct bounds!). Then you need to check whether A really is prime. Then you need to compute B as the difference of N and A , and finally you also need to check whether B is prime. You may find the built-in predicate `between/3` useful.

Exercise 3.14. One of the major news stories involving AI in recent years has been about *IBM Watson*, a computer program that successfully competed in the American television game show *Jeopardy!* in 2011, beating the very best human contestants. Another famous television game show is the British *Countdown* (also known as *Cijfers en Letters* in the Netherlands and as *Des Chiffres et des Lettres* in France, where it had been broadcast first). The purpose of this exercise is to see whether we can win this one for AI as well. We will focus on the *Letters Game* of the *Countdown* show. In this game, we are given nine letters of the alphabet (possibly including some repetitions). The goal then is to construct the longest possible word from these letters. Your score is the length of your word (provided it is a valid word of the English language).

Your ultimate task is to write a Prolog predicate `topsolution/3` to play this game. When given a list of nine letters in the first argument position, it should return as good a solution as possible, consisting of a word of the English language that can be constructed from those letters, in the second argument position and the length of that word (i.e., the score) in the third argument position. Example:

```
?- topsolution([g,i,g,c,n,o,a,s,t], Word, Score).  
Word = agnostic,  
Score = 8  
Yes
```

Start by downloading the file `words.pl` from <http://tinyurl.com/prolog-words> and put it in the same directory as your program file. This is a list of a little over 350,000 of the most common words of the English language, from *a* to *zyzzyva*, presented as a sequence of facts, such as “`word(agnostic).`”, etc. Include the line “`:- consult(words).`” in your program to make these facts available to you. Then proceed as follows.

First, search your Prolog reference manual for a built-in predicate for decomposing an atom into a list of characters. Use it to implement a predicate `word_letters/2` for converting a word (i.e., a Prolog atom) into a list of letters. Example:

```
?- word_letters(hello, X).
X = [h, e, l, l, o]
Yes
```

As an aside, note that you can use this predicate to find words with 45 letters:

```
?- word(Word), word_letters(Word, Letters), length(Letters, 45).
Word = pneumonoultramicroscopicsilicovolcanoconiosis,
Letters = [p, n, e, u, m, o, n, o, u|...]
Yes
```

Second, write a predicate `cover/2` that, given two lists, checks whether the second list “covers” the first. That is, it checks whether every item that occurs k times in the first list also occurs at least k times in the second. Examples:

```
?- cover([a,e,i,o], [m,o,n,k,e,y,b,r,a,i,n]).
Yes

?- cover([e,e,l], [h,e,l,l,o]).
No
```

Third, write a predicate `solution/3` that, when given a list of letters as the first argument and a desired score as the third argument, returns a word covered by that list that would provide the given score (i.e., the length of which is equal to the desired score). Example:

```
?- solution([g,i,g,c,n,o,a,s,t], Word, 3).
Word = act
Yes
```

Finally, implement `topsolution/3`. Document your program by showing how it performs for at least three different lists of letters. One of them should be `[y,c,a,l,b,e,o,s,x]`. This was one of the lists used in the edition of *Countdown* aired in Britain on 18 December 2002, when Julian Fell achieved the best overall score in the history of the show. He found the word *cables*, earning him a score of 6. Can your program beat the champion?

Exercise 3.15. The purpose of this exercise is to develop a system for plotting text-based graphical representations of simple functions (and, more generally, of relations). We assume that we are given a predicate `point/3`, with `point(D,X,Y)` succeeding if and only if we want to draw a point at position (X,Y) on a grid of size $D \times D$. For example, the function $f(x) = x$ is represented as follows (in this case, D is irrelevant, so we use the anonymous variable):

```
point(_, X, Y) :- X == Y.
```

Your ultimate task is to implement a predicate `plot/1`, taking as its only argument the dimension `D` of the graph, that will plot the relation represented by the predicate `point/3` currently compiled as part of your program. Examples:

```
?- plot(4).
      *
    *
  *
*
Yes

?- plot(6).
      *
    *
  *
 *
*
*
*
Yes
```

You will have to draw `*`'s and empty spaces as you go along, from position $(1,D)$ (upper lefthand corner) down to $(D,1)$ (lower righthand corner).

Start by implementing a predicate `next/3` that, given the dimension `D` and the current position (X,Y) , generates the next position. Keep in mind that you have to follow the y -axis in reverse order. Examples:

```
?- next(10, (8,3), Pos).
Pos = (9,3)
Yes

?- next(10, (9,3), Pos).
Pos = (10,3)
Yes

?- next(10, (10,3), Pos).
Pos = (1,2)
Yes
```

Now implement a predicate `plot/2` that takes as arguments a dimension `D` and a position (X,Y) and that (a) draws a `*` in case `point(D,X,Y)` succeeds (and an empty space otherwise), and that (b) recursively calls itself with the same dimension and the next position (using `next/3`). Make sure you define an appropriate base case to ensure the recursion terminates once your picture is complete. Finally, you can easily implement your main predicate `plot/1` by using `plot/2`.

Below are some further examples. We can instruct Prolog to draw a circle by asking it to mark all points with distance at most $D/2$ from the centre of the grid:

```
point(D, X, Y) :- (X-D/2) ** 2 + (Y-D/2) ** 2 <= (D/2) ** 2.
```

Now, once you have replaced the original definition of `point/3` with the one above, Prolog should react to your queries as follows:


```
?- plot(13).
```

```

      * * * * *
    * * * * * * * *
  * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
* * * * * * * * * *
  * * * * * * * *
    * * * * * * *
      * * * * *

```

```
Yes
```

Here are three more examples for the output we get for different definitions of `point/3`:

```
point(D, X, Y) :- X + Y > D.   point(D, X, Y) :- X + Y > D.   point(_, _, _).
                                point(_, X, Y) :- X >= Y.
```

```
?- plot(5).
```

```

* * * *
  * * *
   * *
    *

```

```
Yes
```

```
?- plot(5).
```

```

* * * *
  * * *
   * *
    * *
  * * *

```

```
Yes
```

```
?- plot(5).
```

```

* * * *
* * * *
* * * *
* * * *
* * * *

```

```
Yes
```

Exercise 3.16. Your task for this exercise will be to develop a Prolog predicate called `roman2arabic/2` to translate from Roman to the usual Arabic numerals. Examples:

```

?- roman2arabic('XXI', Number).   ?- roman2arabic('MCMXCIX', Number).
Number = 21                        Number = 1999
Yes                                Yes

```

Before you start thinking about how to implement this, remind yourself how Roman numerals work exactly, using appropriate online resources (such as Wikipedia).

The key idea we will use to approach this problem is to work with an intermediate representation consisting of lists of numbers corresponding to the symbols used for Roman numerals. For example, the number XXI will be represented as the list `[10, 10, 1]`: each number corresponds to one of the symbols, and the sum of the numbers is 21, the Arabic equivalent of XXI. In fact, because Roman numerals make use of the so-called *subtractive notation* (writing, for instance, IX rather than VIII for 9), it will get a little more complicated than this. For example, rather than representing XIX as `[10, 1,`

10], the sum of which would be 21 rather than the 19 we need, we will represent XIX as [10, 9]. That is, sometimes two Roman symbols are represented by a single number in the list (e.g., the I and X together are represented by the 9). Follow these steps to implement `roman2arabic/2`:

- (a) Implement a predicate `symbol/2` to retrieve for each symbol used in the Roman system the corresponding numerical value. You should cover I, V, X, L, C, D, and M. Example:

```
?- symbol('X', Value).
Value = 10
Yes
```

- (b) Now implement a predicate `symbols2numbers/2` to translate a given list of symbols to the corresponding list of its numerical values. Your initial version of the predicate should ignore the complication of the subtractive notation and simply translate each symbol separately. Examples:

```
?- symbols2numbers(['M', 'M', 'X', 'V'], Values).
Values = [1000, 1000, 10, 5]
Yes
```

```
?- symbols2numbers(['X', 'L', 'I', 'I'], Values).
Values = [10, 50, 1, 1]
Yes
```

Once this is working, add a further rule to your implementation to handle the subtractive notation. For the last example above, you should now get the following output (because XL represents the number 40, not the two numbers 10 and 50):

```
?- symbols2numbers(['X', 'L', 'I', 'I'], Values).
Values = [40, 1, 1]
Yes
```

To achieve this, as you go through the input list, you need to always first scan the *two* initial symbols in the list. Whenever the first of these two symbols has a value that is less than the value of the second symbol, you know that you have encountered an instance of the subtractive notation and you can calculate the corresponding single number.

- (c) Now write a predicate `sum/2` to compute the sum of a list of numbers provided in the first argument position. Example:

```
?- sum([1, 2, 3, 4, 5], Sum).
Sum = 15
Yes
```

- (d) Now put everything together to implement `roman2arabic/2`. You will need to be able to break up an atom such as `'XXI'` into a list of characters (e.g., `['X', 'X', 'I']`). Check the reference manual of your Prolog system for how to do that.

Exercises

Exercise 4.1. Consider the following operator definitions:

```
:- op(100, yfx, plink),  
   op(200, xfy, plonk).
```

- (a) Copy the operator definitions into a program file and compile it. Then run the following queries and explain what is happening.
- (i) `?- tiger plink dog plink fish = X plink Y.`

(ii) `?- cow plonk elephant plink bird = X plink Y.`

(iii) `?- X = (lion plink tiger) plonk (horse plink donkey).`

- (b) Write a Prolog predicate `pp_analyse/1` to analyse `plink/plonk`-expressions. The output should tell you what the principal operator is and which are the two main subterms. If the main operator is neither `plink` nor `plonk`, then the predicate should fail. Examples:

```
?- pp_analyse(dog plink cat plink horse).
Principal operator: plink
Left subterm: dog plink cat
Right subterm: horse
Yes
```

```
?- pp_analyse(dog plonk cat plonk horse).
Principal operator: plonk
Left subterm: dog
Right subterm: cat plonk horse
Yes
```

```
?- pp_analyse(lion plink cat plonk monkey plonk cow).
Principal operator: plonk
Left subterm: lion plink cat
Right subterm: monkey plonk cow
Yes
```

Exercise 4.2. Consider the following operator definitions:

```
:- op(100, fx, the),
   op(100, fx, a),
   op(200, xfx, has).
```

- (a) Indicate the structure of this term using parentheses and name its principal functor:

```
claudia has a car
```

- (b) What would Prolog reply when presented with the following query?

```
?- the lion has hunger = Who has What.
```

- (c) Explain why the following query would cause a syntax error:

```
?- X = she has whatever has style.
```

Exercise 4.3. Define operators in Prolog for the connectives of propositional logic. Use the following operator names:

- Negation: `neg`

- Conjunction: `and`
- Disjunction: `or`
- Implication: `implies`

Think about what precedences and associativity patterns are appropriate. In particular, your declarations should reflect the precedence hierarchy of the connectives as they are defined in propositional logic. Define all binary logical operators as being left-associative. Your definitions should allow for double negation without parentheses (see examples).

Hint: You can easily test whether your operator declarations work as intended. Recall that Prolog omits all redundant parentheses when it prints out the answer to a query. That means, when you ask Prolog to match a variable with a formula whose structure you have indicated using parentheses, those that are redundant should all disappear in the output. Parentheses that are necessary, however, will be shown. Examples:

```
?- Formula = a implies ((b and c) and d).
Formula = a implies b and c and d
Yes
```

```
?- AnotherFormula = (neg (neg a)) or b.
AnotherFormula = neg neg a or b
Yes
```

```
?- ThirdFormula = (a or b) and c.
ThirdFormula = (a or b)and c
Yes
```

Exercise 4.4. A formula of propositional logic (involving only negation, conjunction, and disjunction, but not, e.g., implication) is said to be in *negation normal form* (NNF) if it is the case that every subformula that is negated is a negative literal (i.e., the negation of an atomic proposition). That is, for example, $(p \vee \neg q) \wedge \neg r$ is in NNF, while $p \wedge \neg(q \vee r)$ and $\neg\neg p$ are not.

Define appropriate Prolog operators for negation, conjunction, and disjunction (as in the exercise above). Then write a predicate `nnf/1` that takes a formula of propositional logic (involving only these three operators) as an argument and that succeeds if and only if the formula provided is in NNF. Examples:

```
?- nnf((p or neg q) and neg r).
Yes
```

```
?- nnf(p and neg (q or r)).
No
```

```
?- nnf(neg neg p).
```

```
No
```

Hint: Propositional atoms correspond to atoms in Prolog. You can test whether a given term is a valid Prolog atom by using the built-in predicate `atom/1`.

Note: If you are looking for a challenge, you could also try to implement a predicate for translating a given formula into an equivalent formula in NNF (using de Morgan's laws).

Exercise 4.5. Write a Prolog predicate `cnf/1` to test whether a given formula of propositional logic is in conjunctive normal form (CNF), using the operators you defined for Exercise 4.3. Examples:

```
?- cnf((a or neg b) and (b or c) and (neg d or neg e)).
```

```
Yes
```

```
?- cnf(a or (neg b)).
```

```
Yes
```

```
?- cnf((a and b and c) or d).
```

```
No
```

```
?- cnf(a and b and (c or d)).
```

```
Yes
```

```
?- cnf(a).
```

```
Yes
```

```
?- cnf(neg neg a).
```

```
No
```

Exercise 4.6. Using the operators for the logical connectives defined in Exercise 4.3, but this time also covering an operator `iff` for bi-implications, implement a Prolog predicate `cnf/2` to compute the CNF of a given formula. Examples:

```
?- cnf(p iff neg neg q, CNF).
```

```
CNF = (neg p or q) and (neg q or p)
```

```
Yes
```

```
?- cnf(p and q or r and s, CNF).
```

```
CNF = (p or r) and (p or s) and (q or r) and (q or s)
```

```
Yes
```

Hints: For this kind of problem, it is tempting to define lots of redundant cases, resulting in a messy program. So try to be concise and systematic in your presentation, and only

include rules that are actually required. It's a good idea to first implement a predicate to eliminate any occurrences of `implies` and `iff` from the input formula.

Note: You could also think about how to simplify a given formula in CNF. For instance, you could try to remove redundant disjuncts or conjuncts (e.g., $P \wedge (P \vee Q)$ simplifies to P), or you could remove disjunctions containing complementary literals.

Exercises

Exercise 5.1. Type the following queries into a Prolog interpreter and explain what happens.

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

Exercise 5.2. Consider the following Prolog program:

```
result([_, E | L], [E | M]) :- !,
    result(L, M).

result(_, []).
```

- (a) After having consulted this program, what would Prolog reply when presented with the following query? Try answering this question first without actually typing in the program, but verify your solution later on using the Prolog system.

```
?- result([a, b, c, d, e, f, g], X).
```

- (b) Briefly describe what the program does and how it does what it does when the first argument of the `result/2`-predicate is instantiated with a list and a variable is given in the second argument position, i.e., as in item (a). Your explanations should include answers to the following questions:
 - What case(s) is/are covered by the Prolog fact?
 - What effect has the cut in the first line of the program?
 - Why has the anonymous variable been used?

Exercise 5.3. Implement Euclid's algorithm to compute the greatest common divisor (GCD) of two non-negative integers. This predicate should be called `gcd/3` and, given two non-negative integers in the first two argument positions, should match the variable in the third position with the GCD of the two given numbers. Examples:

```
?- gcd(57, 27, X).
X = 3
Yes
```

```
?- gcd(1, 30, X).
```

```
X = 1
```

```
Yes
```

```
?- gcd(56, 28, X).
```

```
X = 28
```

```
Yes
```

Make sure your program behaves correctly also when the semicolon key is pressed.

Hint: Recall that, using Euclid's algorithm, the GCD of two numbers a and b (with $a \geq b$) is computed by recursively substituting a with b , and b with the rest of the integer division of a and b . Make sure you define the right base case(s).

Exercise 5.4. Implement a Prolog predicate `occurrences/3` to count the number of occurrences of a given element in a given list. Make sure there are no wrong alternative solutions. Example:

```
?- occurrences(dog, [dog, frog, cat, dog, dog, tiger], N).
```

```
N = 3
```

```
Yes
```

Exercise 5.5. Write a Prolog predicate `divisors/2` to compute the list of all divisors for a given natural number. Example:

```
?- divisors(30, X).
```

```
X = [1, 2, 3, 5, 6, 10, 15, 30]
```

```
Yes
```

Make sure your program doesn't give any wrong alternative solutions and doesn't fall into an infinite loop when the user presses the semicolon key.

Exercise 5.6. Write a predicate `factor/2` to compute the prime factorisation of a given integer ≥ 2 . Use the same notation as in the following examples:

```
?- factor(30, X).
```

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

```
Yes
```

```
?- factor(300, X).
```

```
X = [2^2, 3, 5^2]
```

```
Yes
```

```
?- factor(1024, X).
```

```

X = [2^10]
Yes

?- factor(17, X).
X = [17]
Yes

```

Use your program to compute the prime factorisations of 7777777 and 12345654321.

Exercise 5.7. In the Treaty of Rome (1957) the six founding countries of the European Union specified the voting rule to decide on proposals in the Council of the European Commission. To pass, a proposal has to reach the threshold of 12 votes. The large countries (France, Germany, and Italy) each have 4 votes; the medium-sized countries (Belgium and the Netherlands) each have 2 votes; Luxembourg has 1 vote. Let us represent these facts in Prolog:

```

countries([belgium, france, germany, italy, luxembourg, netherlands]).
weight(france, 4).
weight(germany, 4).
weight(italy, 4).
weight(belgium, 2).
weight(netherlands, 2).
weight(luxembourg, 1).
threshold(12).

```

This may suggest that, say, Germany has twice as much voting power as the Netherlands, which in turn have twice as much power as Luxembourg. But, as we shall see, this would be a rather naïve interpretation of the rule.

A *coalition* of countries (a subset of the six countries) is called *winning*, if their sum of weights is at least equal to the threshold; otherwise it is called a *losing* coalition. Write a predicate `winning/1` that, when given a list of countries, succeeds if and only if that list constitutes a winning coalition. Examples:

```

?- winning([belgium, france, germany, netherlands]).
Yes

?- winning([belgium, netherlands, luxembourg]).
No

```

Let us say that a given country x is *critical* for a given coalition C if (i) C does not include x , (ii) C alone is not winning, but (iii) C together with x is winning. Implement a predicate `critical/2` to check whether a given country is critical for a given coalition. Examples:

```

?- critical(netherlands, [belgium, france, germany]).
Yes

```

```
?- critical(netherlands, [france, germany, italy]).
No
```

Next we want to find a way to generate *all* coalitions that are critical for a given country. To this end, implement a predicate `sublist/2` that succeeds when its first argument matches a sublist of the list given as the second argument. It should be possible to use it like this:

```
?- sublist(X, [a, b, c]).
X = [a, b, c] ;
X = [a, b] ;
X = [a, c] ;
X = [a] ;
X = [b, c] ;
X = [b] ;
X = [c] ;
X = [] ;
No
```

Now we can generate all critical coalitions for a given country through enforced backtracking:

```
?- countries(All), sublist(Coalition, All), critical(netherlands, Coalition).
All = [belgium, france, germany, italy, luxembourg, netherlands],
Coalition = [belgium, france, germany, luxembourg] ;
All = [belgium, france, germany, italy, luxembourg, netherlands],
Coalition = [belgium, france, germany] ;
All = [belgium, france, germany, italy, luxembourg, netherlands],
Coalition = [belgium, france, italy, luxembourg] ;
All = [belgium, france, germany, italy, luxembourg, netherlands],
Coalition = [belgium, france, italy] ;
All = [belgium, france, germany, italy, luxembourg, netherlands],
Coalition = [belgium, germany, italy, luxembourg] ;
All = [belgium, france, germany, italy, luxembourg, netherlands],
Coalition = [belgium, germany, italy] ;
No
```

That is, the 6 different lists bound to the variable `Coalition` above represent the 6 coalitions for which the Netherlands is critical. Let us define the *voting power* of a country as the number of coalitions for which it is critical (i.e., the voting power of the Netherlands is 6).³ Write a predicate `voting_power/2` to compute a given country's voting power (at this point you will need to make use of a built-in predicate called `findall/3`, which you can look up in your Prolog reference manual). Example:

³If you are interested in finding out more about this topic, search the Internet for “weighted voting games” and “Banzhaff power index” (what we have called the voting power of a country is a simplified version of the so-called Banzhaff power index used widely in political science and economics).

```
?- voting_power(netherlands, Power).  
Power = 6  
Yes
```

What is the voting power of Germany? How about Luxembourg? Explain what this means for the voting rule used.

Note: The only place in your program referring to the specific countries or the specific threshold mentioned in the text above should be the Prolog facts given at the very start. That is, it should be possible to re-use your program for later incarnations of the European Union (with more countries, different weights, and a different threshold) by only changing those facts.

Exercise 5.8. Check some of your old Prolog programs to see whether they produce wrong alternative solutions or even fall into a loop when the user presses ; (semicolon). Fix any problems you encounter using cuts (*one* will often be enough).

Exercises

Exercise 6.1. Translate the following Prolog program into a set of first-order logic formulas:

```
parent(peter, sharon).
parent(peter, lucy).

male(peter).

female(lucy).
female(sharon).

father(X, Y) :-
    parent(X, Y),
    male(X).

sister(X, Y) :-
    parent(Z, X),
    parent(Z, Y),
    female(X).
```

Exercise 6.2. Type the following query into Prolog and try to explain what happens:

```
?- X = f(X).
```

Hint: This example shows that matching (Prolog) and unification (logic) are in fact not exactly the same concept. Take your favourite Prolog book and read about the “occurs check” to find out more about this.

Exercise 6.3. As we have seen in this chapter, the goal execution process in Prolog can be explained in terms of the resolution method. (By the way, this also means, that a Prolog interpreter could be based on a resolution-based automated theorem prover implemented in a low-level language such as Java or C++.)

Recall the mortal Socrates example from the introductory chapter (page 10) and what has been said there about Prolog’s way of deriving a solution to a query. Translate that

program and the query into first-order logic and see if you can construct the corresponding resolution proof. Compare this with what we have said about the Prolog goal execution process when we first introduced the Socrates example. Then, sit back and appreciate what you have learned.