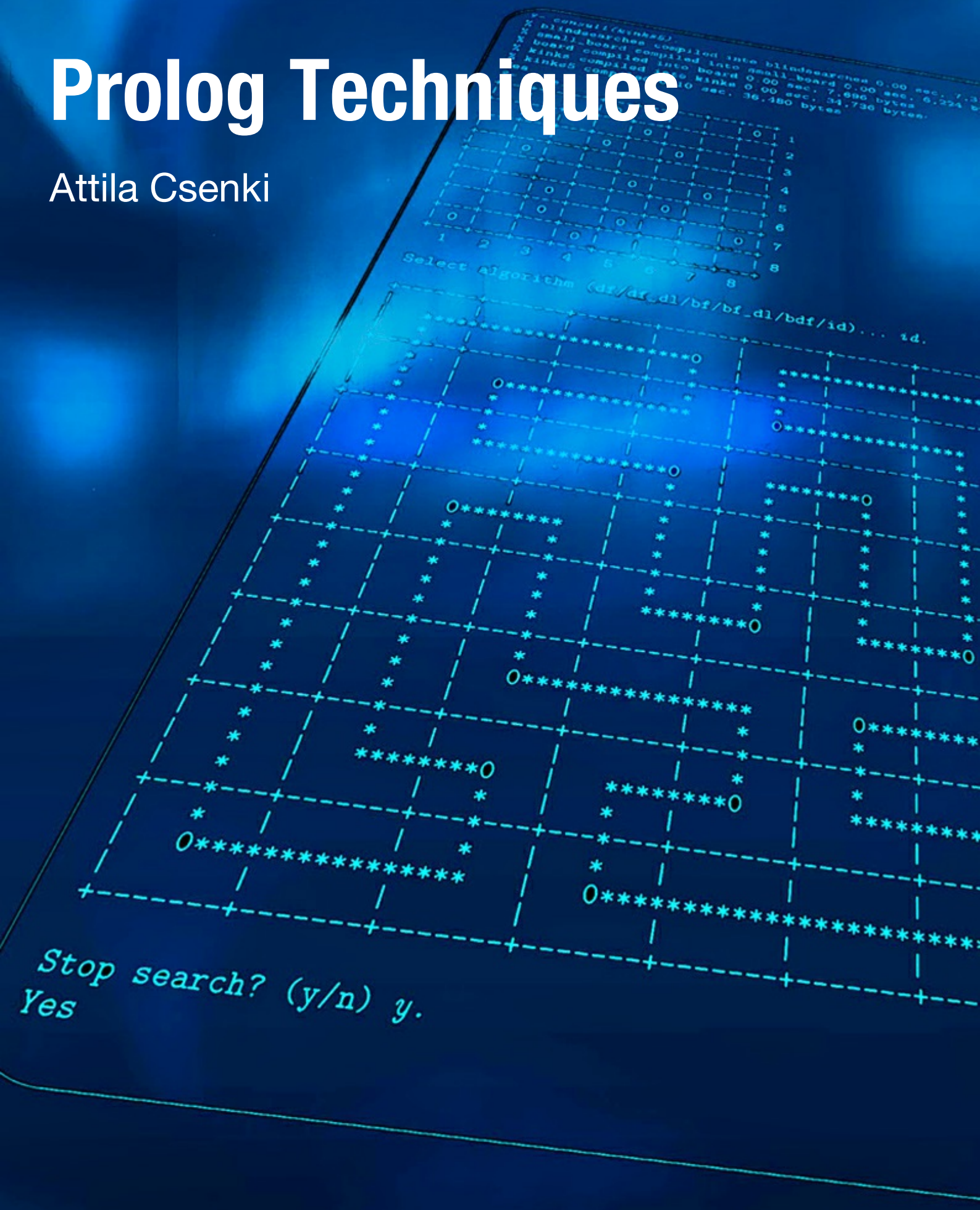


# Prolog Techniques

Attila Csenki



Attila Csenki

---

# Prolog Techniques

---

---

Prolog Techniques

© 2014 Attila Csenki & [bookboon.com](http://bookboon.com)

ISBN 978-87-7681-476-2

To my wife Ágnes who patiently endured me working on this book for most of my spare time during last two years.

# Contents

<b>Preface</b>	<b>11</b>
<b>1 Accumulator Technique</b>	<b>13</b>
1.1 A Simple Example	13
1.2 Hand Computations	14
1.3 Further Examples	14
1.4 Pseudocodes	23
1.5 Generalization	26
1.6 Case Study: The Perceptron Training Algorithm	27
1.6.1 Classification Problem	27
1.6.2 Algorithm	27
1.6.3 Implementation	29
<b>2 Difference Lists</b>	<b>37</b>
2.1 Implementations of List Concatenation	37
2.2 Implementations of List Flattening	42
2.2.1 Project: Lists as Trees & <i>flatten/2</i>	43
2.2.2 Flattening Lists by <i>append/3</i>	48
2.2.3 <i>flatten/2</i> by the Difference List Technique	49
2.2.4 Comparing Different Versions	49
2.3 Implementations of List Reversal	50
2.3.1 Program Transformations	51

---

2.3.2	Difference Lists as Accumulators	57
2.4	Case Study: Dijkstra's Dutch Flag Problem	57
2.4.1	Basic Implementation Using <i>append/3</i>	58
2.4.2	A More Concise Version	58
2.4.3	Using Difference Lists	59
2.5	Rotations	61
2.5.1	Rotating a List	61
2.5.2	The Perceptron Training Algorithm Revisited	64
2.5.3	Planar Rotations	65
2.5.4	Application: The Gauss–Seidel Method	69
<b>3</b>	<b>Program Manipulations</b>	<b>75</b>
3.1	Simple Database Operations	75
3.1.1	Basic Database Manipulation	79
3.1.2	Changing the Database	80
3.1.3	File Modifications	85
3.1.4	Updating <i>right_to/2</i> and <i>people.pl</i>	87
3.1.5	Automated Saving of Selected Predicates	87
3.1.6	Miniproject: Modelling a Stamp Collection	91
3.2	Case Study: Automated Unfolding	95
3.2.1	Elementary Unfolding	95
3.2.2	Complete One Step Unfolding	104
3.2.3	Rearranging Clauses	106
3.3	Dijkstra's Dutch Flag Problem Revisited	108

---

3.3.1	Problem Generalization and First Solution	108
3.3.2	Enhanced Implementations	111
<b>4</b>	<b>Exploratory Code Development</b>	<b>117</b>
4.1	A Nursery Rhyme	117
4.1.1	First Preliminary Implementation	119
4.1.2	Another Preliminary Implementation	124
4.1.3	The Final Version	125
4.1.4	Other Approaches	127
4.2	Project: ' <i>One Man Went to Mow . . .</i> '	132
4.3	Chapter Notes	139
<b>A</b>	<b>Solutions of Selected Exercises</b>	<b>141</b>
A.1	Chapter 1 Exercises	141
A.2	Chapter 2 Exercises	145
A.3	Chapter 3 Exercises	157
A.4	Chapter 4 Exercises	167
<b>B</b>	<b>Software</b>	<b>177</b>
<b>C</b>	<b>Glossary</b>	<b>179</b>
	<b>References</b>	<b>183</b>
	<b>Index</b>	<b>185</b>

## List of Figures

1.1	Hand Computations for <i>new sum/2</i>	15
1.2	Hand Computations for <i>rev/2</i>	16
1.3	Hand Computations for <i>min/2</i>	17
1.4	Suggested Hand Computations for <i>from to/3</i>	18
1.5	Hand Computations for <i>cnt/3</i>	19
1.6	Hand Computations for <i>palin/1</i> — <i>success</i>	21
1.7	Hand Computations for <i>palin/1</i> — <i>failure</i>	21
1.8	Typical Clause Structures of a Predicate with an Accumulator	27
1.9	Generalized Clause Structures	27
1.10	A Linearly Separable Data Set	28
1.11	Classifying a Point	29
1.12	A Single Updating Step	30
1.13	Applying the Perceptron Training Algorithm	31
2.1	Difference List	39
2.2	List Concatenation by Difference Lists	40
2.3	Tree Representation of $[a,[b,[],[c,a],e]]$	44
2.4	Declarative Reading of (P-2.3)	50
2.5	Illustrating Clause (b2) in (P-2.6)	56
2.6	Illustrating Exercise 2.9	57
2.7	Rotating by Difference Lists	61
2.8	Hand Computations for <i>averages/2</i>	62

2.9	Rotating a List with Four Entries	64
2.10	The Original List and its Rotated Image	65
2.11	The Original Matrix <b>A</b> and its Rotated Image <b>A</b> <sup>(rot)</sup>	66
2.12	Hand Computations for Rotation in the Plane	67
3.1	The Initial Seating Arrangement	76
3.2	Rectangular Table	77
3.3	After George's Departure	81
3.4	After Tracy's and Joe's arrival	82
3.5	File Organization for the Round Table Example	85
3.6	The File <code>people.pl</code> <i>after</i> the Interactive Session	87
3.7	The File <code>committee.pl</code>	89
3.8	The File <code>committee.pl</code>	89
3.9	Interactive Prolog-Assisted Program Transformation: Session I	97
3.10	Interactive Prolog-Assisted Program Transformation: Session II	98
3.11	Unfolding, Experiment 1: Disassembling clause 4 of <i>a/5</i>	99
3.12	Unfolding, Experiment 2: Disassembling clause 3 of <i>c/2</i>	99
3.13	Unfolding, Experiment 3: Experiments 1 & 2 followed by appropriate unification	100
3.14	Unfolding, Experiment 4: Experiment 3 followed by new clause creation and database update	101
3.15	Illustrative Example of Intended Database Updates	111
3.16	Top Level Definition of <i>def_encolour dl/1</i>	112
3.17	Example Session for Exercise 3.19	116



---

4.1	The Rhyme's Simplified Pattern	118
4.2	Exploring Details of the Rhyme's Structure	124
4.3	Desired Behaviour of <i>song/0</i>	133
A.1	Annotated Hand Computations for <i>from to/3</i>	141
A.2	Hand Computations for <i>mult/3</i>	144
A.3	Illustrating the Second Clause of <i>dl/2</i>	154
A.4	The Last Two Customers Swap Places	159
A.5	Automated Solution of Exercise 2.9, Part (c)	163
A.6	Database Changes Brought About by <i>cosu/3</i>	166
A.7	Search Tree of the Query <i>?- int(1,I)</i>	172

## List of Tables

1.1	Algorithm 1.4.1 and Related Hand Computations (Fig. 1.2)	24
1.2	Algorithm 1.4.2 and Related Hand Computations (Fig. 1.5)	26
1.3	Algorithm 1.4.3 and Related Hand Computations (Figs. 1.6 & 1.7)	26
1.4	Co-ordinates of Points in the Plane with Class Labels	27
2.1	Gauss–Seidel Iterations	70
3.1	Cases for <i>swap_neighbours/2</i>	84
4.1	Rhyme Structure	127
4.2	CPU Times for Versions of the Query <i>?- rhyme_prel( V, R)</i>	129
A.1	Algorithm A.1.1 & Prolog Clause Correspondence (Example 1.6)	144

# Preface

Prolog is considered difficult by students. Usually, by the time they learn Prolog, which is most likely to happen in preparation for a course in Artificial Intelligence (AI) or Expert Systems, they will have studied imperative programming and/or the object oriented paradigm. Unfortunately, this prior experience is not always conducive to learning Prolog. Even though there is a good provision of traditional Prolog textbooks (for example [2]), students still find it hard to write solutions in Prolog to problems of any notable complexity. In my experience this holds also (and in particular) for problems for which Prolog should be the natural choice.

This book is intended to relieve the problem by providing a good collection of programming projects, case studies and exercises of various complexity. It will be useful for three kinds of students.

- Those whose prime source of information is a traditional introductory lecture course in Prolog. For these people my book will serve to show in *context* how the various programming techniques and language elements may be employed. The book may be used to accompany such a course as a *workbook* and the student should find in it a wealth of information to answer questions concerning the aspects of Prolog taught in the course.
- Those who want to *refresh* and *extend* their knowledge of Prolog, perhaps with some field of application in mind.
- Students of AI learning about *search algorithms* in particular. Most AI books present search algorithms by pseudocode and are not concerned with details of implementation. In my experience, however, anything seen implemented is more likely to be retained (beyond the exam).

There is a deeper reason also why such a book is felt timely. Programming is a *creative* activity and it is an innate human need to take pleasure (and pride) in the object of one's creation, be it a sculpture, a painting, a piece of music, or indeed, a computer program. The opportunity is provided here for students to learn (and experience the said intellectual satisfaction) by creating their own solutions in Prolog to a host of interesting, challenging and varied programming problems. Many of the problems and the way they are approached here are believed to be novel.

Sadly, it is felt that the creative aspect of learning is not given enough room in today's educational environment in the UK.<sup>1</sup> It is hoped that this book will help the student to rediscover Prolog programming as a

---

<sup>1</sup>There is ample evidence to support this thesis. In degree courses, we tend to focus on the 'engineering' aspects of and tools for writing (large scale) software; this activity tends to be team-based, procedure-bound and offers little scope for the kind of pleasure felt by completing a working 'whole'. Learning by students tends to be assessment driven and many never experience the creative feedback. The tasks they have to complete for the exams are (by their very nature) not intended to create anything sizable or ambitious. Coursework assignments do not attract many marks for fear of plagiarism. Finally, modularization does not encourage students to take an interest beyond what is in the module descriptor. The only time where creativity is really called for will be the final year project by which time many will lack the practice to complete the task to their supervisor's (and, equally importantly, their *own*) satisfaction.

worthwhile and enjoyable activity.

The core of the material in this book grew out of laboratory classes and coursework prepared by the author for second year computer science students at Bradford University, as part of the lecture course *Symbolic and Declarative Computing – Artificial Intelligence*. This is a two-semester course with an introduction to Functional Programming with Haskell, Logic Programming with Prolog and the basics of AI. The choice of examples and topics for this book is of course tinged by the context in which Prolog was presented. For example, I discuss the functional programming style since it is useful in producing concise, readable and elegant implementations also in Prolog. The selection of topics for the examples was influenced in part by the AI element of the course though much new material has found its way into the book. To make set problems more easily accessible for the reader, I subdivide the overall task into manageable portions indicating in each the desired outcome (if applicable, in form of a sample session in Prolog) with suggestions for how best to attack the subtasks.

The working style advocated here is best described by the following attributes:

- example based,
- interactive,
- exploratory and experimental,
- incremental,
- progressing from the specific to the more general,
- identifying patterns of computation with a view to generalization.

It will be seen from the list of contents that the material, by its very nature, is not ordered in a linear fashion but is grouped in topics deemed important for programming in Prolog.

The work comprises two parts: the present volume *Prolog Techniques* and the forthcoming *Applications of Prolog*. This first volume is in four chapters and illustrates special Prolog programming techniques. The second volume will concentrate on applications of Prolog, mainly from Artificial Intelligence.

The order in which the books may be studied is fairly free even though an example introduced somewhere may serve in a later chapter to illustrate the generalization or improvement afforded by the material just covered.

The SWI-Prolog compiler is used throughout: it has been around for quite some time; it is well documented; it is free; and, it is being maintained with new, improved versions becoming available all the time. Furthermore, there is an object oriented extension to SWI-Prolog (XPCE) for building graphical applications, useful if one wants to pursue this line further.

Solutions for a selection of exercises are discussed in the appendices. All Prolog source code produced in the course of this book project (including model solutions for all the exercises) can be downloaded from the Ventus website.

I am grateful to Dr. Coxhead of Birmingham University for discussions and extensive comments on initial versions of several of the chapters. My colleague Dr. Fretwell gave me many tips concerning L<sup>A</sup>T<sub>E</sub>X, the typesetting system used to produce the books.

Bradford,  
April 2014

Attila Csenki

# Chapter 1

## Accumulator Technique

One of the features of Prolog which beginners may find difficult to cope with is the absence of a language construct for writing loops such as the *while* and *for* loops known from imperative programming. In Prolog, repetition is accomplished by *recursion* which holds some pitfalls for the novice user. In this chapter, we introduce the *accumulator technique* for defining predicates by recursion.

### 1.1 A Simple Example

Let us start with the simple problem of calculating the sum of the (integer) entries in a list. A naïve definition is as follows.

**Prolog Code P-1.1: Definition of *sum/2***

```

1 sum([],0).                                     % clause 1
2 sum([H|T],S) :- sum(T,S0), S is H + S0. % clause 2

```

The definition of *sum/2* in (P-1.1) is by *recursion*: clause 1 is the *Base Case*, clause 2 is the *Recursive Step*. It is a viable definition for lists of moderate length, as shown below.

```

?- from_to(1,100,L), sum(L,S).1
L = [1, 2, 3, 4, 5, 6, 7, 8, 9|...]
S = 5050

```

For longer lists, however, error by stack overflow is observed:

```

?- from_to(1,100000,L), sum(L,S).
ERROR: Out of local stack2

```

How should *sum/2* be restructured to avoid this problem? The answer lies in what is called a *tail recursive* definition:

<sup>1</sup>For a definition of the predicate *from\_to(+Low,+High,-List)*, see Exercise 1.1, p. 17 and the solution of Exercise 3.16, p. 167. *from\_to/3* returns in *List* the list of integers between the bounds *Low* and *High*.

<sup>2</sup>The query below shows that stack overflow is caused here by *sum/2* and *not* by *from\_to/3*.

```

?- from_to(1,100000,L).
L = [1, 2, 3, 4, 5, 6, 7, 8, 9|...]

```

For a predicate defined by a recursive clause, the self-invocation should be the *last* goal in its body. And, for the Prolog system to discard all references to the goals preceding the last one (and thereby freeing up memory), a cut (!) should be introduced just before the self-invocation.

There is no immediate way of rewriting the second clause of *sum/2* along these lines (The order of the goals in its body can't be interchanged since the tail needs summing before the final sum is computed.) The problem is solved by augmenting the old version by an *accumulator argument* for holding intermediate results of the computation. The new version, *sum/3*, is defined by

**Prolog Code P-1.2: Definition of *sum/3***

```

1 sum([],S,S). % clause 1
2 sum([H|T],Acc,S) :- NewAcc is Acc + H, !, sum(T,NewAcc,S). % clause 2

```

The second argument of *sum/3* serves as an *accumulator* that holds a value which could be termed 'the sum accrued thus far'. The third argument is carried (in clause 2) as an uninstantiated variable until eventually (in clause 1) it is unified with the accumulator. By the time clause 1 applies, the accumulator will have received the sum of all entries of the initial list provided that the accumulator argument has been *initialized* to zero; this latter step is carried out when invoking *sum/3*:

```

?- from_to(1,100000,L), sum(L,0,S).
L = [1, 2, 3, 4, 5, 6, 7, 8, 9|...]
S = 5.00005e+009

```

(There is no error due to stack overflow this time!)

We may define *new\_sum/2* by

**Prolog Code P-1.3: Definition of *new\_sum/2***

```

1 new_sum(L,S) :- sum(L,0,S). % clause 0

```

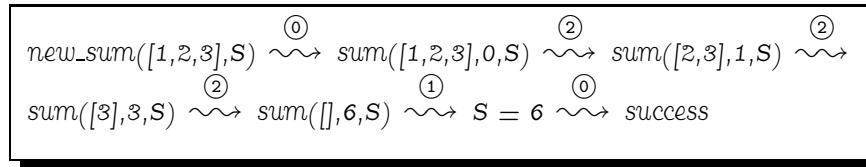
The predicate *sum/3* is used by *new\_sum/2* as an *auxiliary* predicate. The accumulator argument in *sum/3* is initialised by *new\_sum/2* in clause 1 to zero.

## 1.2 Hand Computations

It is instructive to examine the workings of *new\_sum/2* and *sum/3* by a sequence of *hand computations* (see, [3], pp. 116). To consider a specific case, we trace in Fig. 1.1 the computation by *new\_sum/2* of the sum of the entries of *[1,2,3]*. The wavy arrow ( $\rightsquigarrow$ ) is used to indicate transitions, interrelating one stage with the next. The details of how a transition is (or should be) accomplished are elaborated upon in the clause as marked above the arrow.

## 1.3 Further Examples

Hand computations can be carried out to test code already written but they are also useful for defining new predicates. It is this latter rôle in which we are going to illustrate their use here in several examples. The following steps will be involved.

Figure 1.1: Hand Computations for `new_sum/2`

- State the algorithm to be employed. This may take various forms, most likely, it will be in plain English.<sup>3</sup>
- Construct an example (or examples) typifying all conceivable situations.
- Carry out hand computations for the examples chosen. Transitions of a similar kind (i.e. those intended to be covered by the same clause) receive identical labels.
- Inspect the hand computations and define a clause for each label.

<sup>3</sup>In Sect. 1.4, *pseudocodes* will be introduced for describing algorithms.

**Example 1.1.** Define a new version of the built-in predicate *reverse/2* for reversing a list.

Our approach is easily visualized by thinking of the list entries as a pack of cards whose order has to be reversed. Put the pack, face down, on the table and build up a second pack by moving the cards from the top of the first, one by one, to the top of the second. The stopping criterion is also obvious: stop when the first pile is used up, i.e. if the first list is empty. The hand computations in Fig. 1.2 have been carried out using this idea. The definition (P-1.4) is based on them.

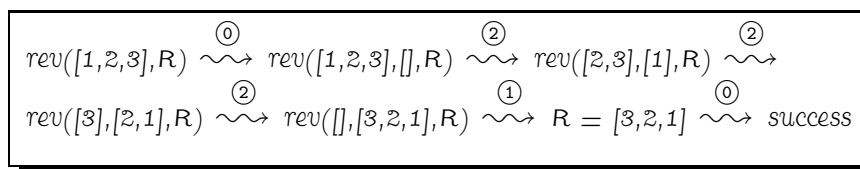


Figure 1.2: Hand Computations for *rev/2*

**Prolog Code P-1.4: Definition of *rev/2***

```

1 rev(L,R) :- rev(L,[],R).                % clause 0
2 rev([],R,R).                            % clause 1
3 rev([H|T],Acc,R) :- rev(T,[H|Acc],R). % clause 2

```

**Example 1.2.** Define a predicate *min/2* for computing the smallest entry of an (integer) list as shown below.

```

?- min([7,-3,2,5],S).
S = -3

```

The idea is again readily illustrated by using a pack of cards. We are now looking for the card with the smallest value.

1. Take the top one and set it aside.
2. Inspect the top card and compare its value with the one set aside. Retain the smaller of the two, set it aside while discarding the other.
3. Repeat step 2 until you run out of cards. The one set aside will be a one with the minimum value.

From the hand computations in Fig. 1.3 it is seen that there should be two recursive clauses: in the case marked ②, the head of the list is smaller than the current value of the accumulator and thus it will be replaced by the former; in the case marked ③, this condition does not apply and therefore the old accumulator value is retained. Fig. 1.3 also shows that the initial value of the accumulator in *min/3* is the head of the input list (step ①).<sup>4</sup>

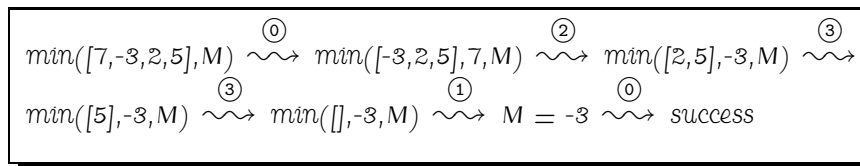
<sup>4</sup>There is an alternative to this. Use the built-in predicate *current\_prolog\_flag/2* to find the largest integer Prolog can represent and initialize the accumulator to this value:

```

?- current_prolog_flag(max_integer, Large), min([7,-3,2,5],Large,M).
Large = 2147483647
M = -3
Yes

```



Figure 1.3: Hand Computations for *min/2***Prolog Code P-1.5: Definition of *min/2***

```

1 min([H|T],M) :- min(T,H,M).           % clause 0
2 min([],M,M).                          % clause 1
3 min([H|T],Acc,M) :- H < Acc, !, min(T,H,M). % clause 2
4 min([_|T],Acc,M) :- min(T,Acc,M).      % clause 3

```

(Notice that in clause 3 the goal  $H \geq Acc$  is omitted as it would always succeed by the time that clause is tried. Here we rely on the clauses' particular order.)

**Exercise 1.1.** Define a predicate *from\_to*(?Low,?High,?List) for producing in *List* all the natural numbers in ascending order between *Low* and *High*. The various modes of operation of *from\_to*/3 are illustrated below.

```

?- from_to(6,9,L). 5
L = [6, 7, 8, 9]
?- from_to(6,9,[_,_,E|_]). 6
E = 8
?- from_to(Low,High,[6, 7, 8, 9]). 7
Low = 6 High = 9
?- from_to(6,9,[6, 7, 8, 9]).
Yes
?- from_to(9,6,L).
No

```

Some suggested hand computations are shown in Fig. 1.4.

**Example 1.3.** (*Several accumulators*) Define *cnt*(+Atom,-U,-L) for counting the number of upper and lower case letters in an atom. The query below illustrates the intended behaviour of *cnt*/3.

```

?- cnt('''The Magic Flute'' is Mozart''s last opera.',U,L). 8
U = 4
L = 27

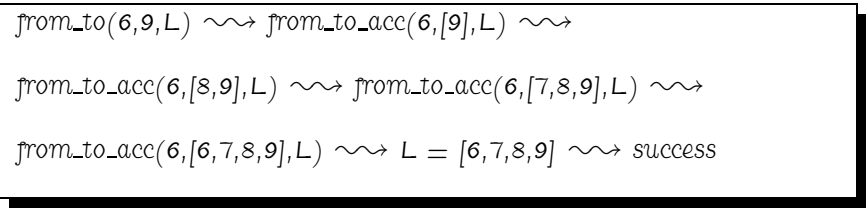
```

<sup>5</sup>Use the built-in predicates *var*/1 and *integer*/1 when implementing this functionality.

<sup>6</sup>Here we pick out the third entry of the result list by *pattern matching*. The built-in predicate *is\_list*/1 should be used to allow for such behaviour.

<sup>7</sup>To implement this functionality, you will need the built-in predicate *last*/2.

<sup>8</sup>The outside quotes mark the atom. Inside the atom, the characters '' stand for the quote.



```

from_to(6,9,L) ~~~> from_to_acc(6,[9],L) ~~~>
from_to_acc(6,[8,9],L) ~~~> from_to_acc(6,[7,8,9],L) ~~~>
from_to_acc(6,[6,7,8,9],L) ~~~> L = [6,7,8,9] ~~~> success

```

Figure 1.4: Suggested Hand Computations for *from\_to/3*

Instead of inspecting the atom's characters directly, we will convert them by the built-in predicate *atom\_codes/2* to the list of their corresponding ASCII values and then class each entry according to whether it is

- Between 65 and 90 ( $\Rightarrow$  upper case, increment first accumulator),
- Between 97 and 122 ( $\Rightarrow$  lower case, increment second accumulator),
- None of the above ( $\Rightarrow$  non-alphabetic, no incrementation).

**Built-in Predicate:** *atom\_codes(?Atom,?List)*

It converts *Atom* into the corresponding *List* of ASCII values and vice versa.  
Example:

```
?- atom_codes('Way Out',L).
L = [87, 97, 121, 32, 79, 117, 116]
```

Hand computations for *cnt/3* on the atom 'Way Out' are shown in Fig. 1.5. The code shown in (P-1.6)

```

cnt('Way Out',U,L) ① ~~~~~>
cnt([87,97,121,32,79,117,116],0,0,U,L) ② ~~~~~>
cnt([97,121,32,79,117,116],1,0,U,L) ③ ~~~~~>
cnt([121,32,79,117,116],1,1,U,L) ③ ~~~~~>
cnt([32,79,117,116],1,2,U,L) ④ ~~~~~> cnt([79,117,116],1,2,U,L) ② ~~~~~>
cnt([117,116],2,2,U,L) ③ ~~~~~> cnt([116],2,3,U,L) ③ ~~~~~>
cnt([],2,4,U,L) ① ~~~~~> U = 2, L = 4 ① ~~~~~> success

```

Figure 1.5: Hand Computations for *cnt/3*

mirrors the hand computations.

**Prolog Code P-1.6: Definition of *cnt/3***

```

1 cnt(Atom,U,L) :- atom_codes(Atom,Values),           % clause 0
2                   cnt(Values,0,0,U,L), !.             %
3 cnt([],U,L,U,L).                                     % clause 1
4 cnt([H|T],AccU,AccL,U,L) :- upper(H),                % clause 2
5                               NewAccU is AccU + 1,     %
6                               !, cnt(T,NewAccU,AccL,U,L). %
7 cnt([H|T],AccU,AccL,U,L) :- lower(H),                % clause 3
8                               NewAccL is AccL + 1,     %
9                               !, cnt(T,AccU,NewAccL,U,L). %
10 cnt([_|T],AccU,AccL,U,L) :- cnt(T,AccU,AccL,U,L).   % clause 4

```

The auxiliary predicates used in (P-1.6) are *upper/1* and *lower/1*; they are defined in (P-1.7).

**Prolog Code P-1.7: Definitions of *upper/1* and *lower/1***

```

1 upper(C) :- C >= 65, C <= 90.
2 lower(C) :- C >= 97, C <= 122.

```

**Example 1.4.** (*Grouping of arguments*) For better readability, arguments may be grouped by using compound terms. The name of the term's functor is chosen to reflect the arguments' common rôle. For example, a new version of *cnt/3* from Example 1.3, called *count/2*, is shown in (P-1.8)

**Prolog Code P-1.8: Definition of *count/2***

```

1 count(Atom,cases(U,L)) :- atom_codes(Atom,Values),           % clause 0
2                           count(Values,acc(0,0),acc(U,L)), !. %
3 count([],Acc,Acc).                                           % clause 1
4 count([H/T],acc(U,L),Result) :- upper(H),                   % clause 2
5                               NewU is U + 1, !,              %
6                               count(T,acc(NewU,L),Result).    %
7 count([H/T],acc(U,L),Result) :- lower(H),                   % clause 3
8                               NewL is L + 1, !,              %
9                               count(T,acc(U,NewL),Result).    %
10 count([_ /T],acc(U,L),Result) :- count(T,acc(U,L),Result). % clause 4

```

*count/2* will behave as *cnt/3* does:

```

?- count('Way Out',cases(U,L)).
U = 2
L = 4

```

*count/2* is essentially the same predicate as *cnt/3* but the number of arguments is reduced to two via the term *cases/2*. The auxiliary predicate *count/3* has been derived from *cnt/5* by merging the two accumulator arguments and the two output arguments each. The accumulators are grouped by the compound term *acc/2*; the now single output argument is reproduced in the recursive clauses 2–4 by a variable until upon satisfying the stopping criterion in clause 1, it is unified with the term in the accumulator argument. The call to *count/3* in clause 0 initialises the accumulator to *acc(0,0)*; the result is received in the variables *U* and *L* by unifying the third argument with the *acc(U,L)*.

This example shows that argument grouping allows the arity of a predicate to be reduced. This observation will be useful in our discussion of a generalization of the accumulator technique in Sect. 1.5.

By repeatedly applying this technique, more elaborate hierarchical groupings of arguments may be achieved by *nesting* terms.

**Example 1.5.** (*Test for success or failure only*) A palindrome is a list (of atoms) which is identical to its reverse. We can use *rev/2* from Example 1.1 to test if a list is a palindrome:

```

?- rev([m,a,d,a,m],[m,a,d,a,m]).
Yes
?- rev([a,d,a,m],[a,d,a,m]).
No

```

Clearly, in both cases the *entire* reverse of the first argument had to be computed for subsequent matching with the original by way of unification. (P-1.9) shows a more efficient solution ([8], p. 110).

**Prolog Code P-1.9: Definition of *palin/1***

```

1 palin(L) :- palin(L, []).           % clause 0
2 palin(L, L).                         % clause 1
3 palin([_|T], T).                     % clause 2
4 palin([H|T], Acc) :- palin(T, [H|Acc]). % clause 3

```

If *palin/1* succeeds, only the front of the list will be worked through as illustrated in Fig. 1.6. (Clauses like 1 and 2 will be executed by unification.) For cases which *fail*, still the whole list will have to be scanned (Fig. 1.7).

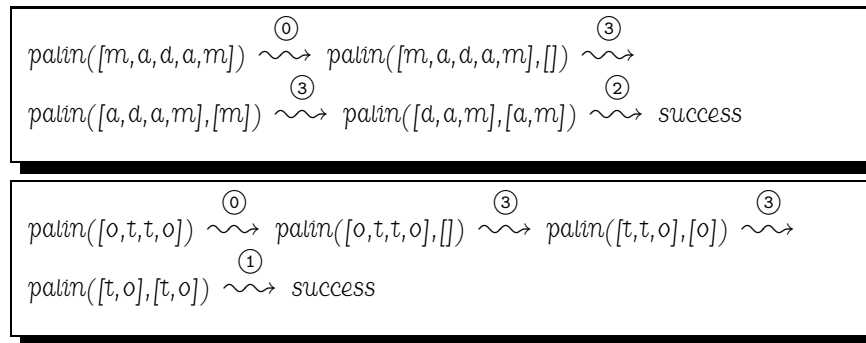


Figure 1.6: Hand Computations for *palin/1* — *success*

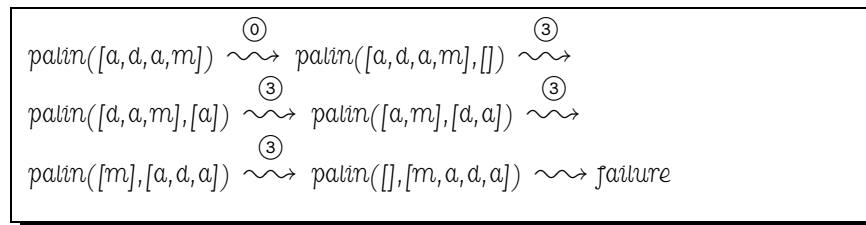


Figure 1.7: Hand Computations for *palin/1* — *failure*

**Example 1.6.** (*Switches as accumulators*) Define *numbers(+Atom, -N)* for finding out how many numbers there are in an *Atom*. Example:

```

?- numbers('Elisabeth the 1st reigned between 1558 and 1603.', N).
N = 3

```

This task can be solved in various ways but we are interested in a solution which makes only a *single* pass through the list of the (encoded) characters of *Atom*.

We can view our problem as having to count the number of *sequences of digits* in *Atom*. We shall of course work with the characters' encoded values; the ASCII values of the ten digits are 48, ..., 57. *digit/1* in (P-1.10) succeeds for encoded digits.

**Prolog Code P-1.10:** Definition of the auxiliary predicate *digit/1*

```
1 digit(C) :- 48 =< C, C =< 57.
```

As we progress through the list of (encoded) characters, the beginning of a new sequence of digits will be recognized by the condition

- The previous character was not a digit
- The current character is a digit.

We employ a dedicated, two-valued argument, called a *switch*, to save the information about the digit read. The switch has two alternative states: *digit* and *nodigit*; it will be initialized to *nodigit*. We also use an accumulator argument for the number of digit sequences 'encountered thus far'. The accumulator is incremented every time the switch changes state from *nodigit* to *digit*.

The predicate *numbers(+List,+Switch,+Acc,-N)* in (P-1.11) is an implementation of these ideas.<sup>9</sup>

**Prolog Code P-1.11: Definition of *numbers/2***

```

1 numbers(Atom,N) :- atom_codes(Atom,Values),           % clause 0
2                   numbers(Values,nodigit,0,N), !.      %
3
4 numbers([],_,N,N).                                     % clause 1
5 numbers([H|T],nodigit,Acc,N) :- digit(H),             % clause 2
6                               NewAcc is Acc + 1, !,    %
7                               numbers(T,digit,NewAcc,N). %
8 numbers([H|T],digit,Acc,N) :- digit(H), !,           % clause 3
9                               numbers(T,digit,Acc,N).  %
10 numbers([_|T],_,Acc,N) :- !, numbers(T,nodigit,Acc,N). % clause 4

```

**Exercise 1.2.** Solve the problem from Example 1.6 as before (i.e. by a single pass through the data), now without using a switch.<sup>10</sup>

## 1.4 Pseudocodes

Programming (in any language) is a creative activity and the accompanying thought processes may be difficult to formalize and will ultimately remain a personal experience. Nevertheless, there are tools intended to assist the programmer in the software production process. Here the notion of an *algorithm* plays a central rôle. Indeed, one view of the (procedural) software production process is that it is a series of steps in each of which an algorithm is derived from a previous one by refinement until a working implementation is obtained.

Ideally, when programming in Prolog we should be less concerned with algorithms and be allowed to concentrate on a declarative *description* of the problem in the hope that the Prolog system will arrive at a solution from our specification. In practice, however, both viewpoints are useful and the accumulator technique obviously favours the procedural style.

Therefore, as an adjunct to our discussion of the accumulator technique, we want to look at here a particular way of describing algorithms, namely by *pseudocodes*. Pseudocodes are of interest in particular when using Prolog as an implementation language for Artificial Intelligence (AI) since books in AI use pseudocode for specifying algorithms (e.g. [7, 13, 14]).

We start with the algorithm for reversing lists by *rev/2* in Example 1.1. Algorithm 1.4.1, shown below, is inspired by the hand computations in Fig. 1.2. It is formulated in terms of *iteration* and would be implemented by a *while* loop if we were to use an imperative programming language. It is seen that the pseudocode mimics the workings of an abstract procedural language and that the depth to which individual steps are detailed may be varied. When the pseudocode is finally ‘translated’ to Prolog, recursion is used to implement iteration. Table 1.1 interrelates the steps in the hand computations with the pseudocode statements.

<sup>9</sup>Reference will be made to (P-1.11) in Exercise 1.3, p. 26.

<sup>10</sup>*Hint.* Employ a ‘look ahead’ strategy to see what (encoded) character will be read *after* the present one. (This plan allows a concise implementation to be achieved.)

**Algorithm 1.4.1:** REVERSE(*List*)

```

Accumulator ← [] (1)
while List ≠ []
do { [H|T] ← List (2)
    Accumulator ← [H|Accumulator] (3)
    List ← T (4)
  }
Rev ← Accumulator (5)
return (Rev)

```

Statement	(1)	(2)	(3)	(4)	(5)
Hand Computation Step	①	②	②	②	①

Table 1.1: Algorithm 1.4.1 and Related Hand Computations (Fig. 1.2)

A slightly more complex case is illustrated by Example 1.3 whose pseudocode, inspired by the hand computations in Fig. 1.5, is shown as Algorithm 1.4.2. (The correspondence between pseudocode statements and steps in the hand computations is displayed in Table 1.2.)

These examples illustrate the following points.

- The *while* loop is implemented by recursion and by using Prolog's control flow model.
- The *if-then-else* construct is implemented by putting the clauses in the right order and by pattern matching using unification.
- Named memory locations (variables) in the pseudocode are implemented by specific arguments of predicates or of compound terms.
- Assignment (indicated in the pseudocode by  $\leftarrow$ ) is accomplished by unification.
- In general, Prolog implementations tend to be more concise than the corresponding program written in a conventional language.

We conclude this section with the pseudocode for Example 1.5, shown as Algorithm 1.4.3, p. 26. This is of special interest for two reasons. First, the algorithm is not expected to produce any 'output' in the procedural sense except for Prolog's *Yes-No* response. This should be no cause for concern, however; proceed as before except that the predicate now has no 'output' argument. The second noteworthy property of Algorithm 1.4.3 is that it contains a mid-loop exit and therefore it does not comply with the principles of *Structured Programming* (one entry – one exit), a style normally adhered to in procedural programming. Thus, *Nassi-Shneiderman Diagrams* (also called *Structograms*) [12], would not be a suitable alternative for specifying this algorithm even though *palin/1* is a good example of a perfectly acceptable Prolog definition. This shows that Prolog allows code to be written whose logic would be frowned upon under different circumstances and whose use would be out of bounds for users of Structograms.<sup>11</sup>

<sup>11</sup>We note in passing that the German Code of Practice DIN 66261 [6] describes the use of Structograms.



**Algorithm 1.4.2:** COUNT(*Atom*)

```
Values ← list of ASCII values of characters in Atom (1)
AccU ← 0 (2)
AccL ← 0 (3)
while Values ≠ []
do { [H|T] ← Values (4)
    if H is an upper case letter
    then { AccU ← AccU + 1 (5)
    else if H is a lower case letter
    then { AccL ← AccL + 1 (6)
    Values ← T (7)
Uppers ← AccU (8)
Lowers ← AccL (9)
return (Uppers, Loweres)
```

<i>Statement</i>	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)
<i>Hand Comp<sup>2</sup> Step</i>	①	①	①	② ③ ④	②	③	② ③ ④	①	①

Table 1.2: Algorithm 1.4.2 and Related Hand Computations (Fig. 1.5)

<b>Algorithm 1.4.3:</b> PALINDROME( <i>List</i> )	
<i>Accumulator</i> $\leftarrow []$	(1)
<b>while</b> <i>List</i> $\neq []$	(2)
<b>if</b> <i>List</i> = <i>Accumulator</i>	(2)
<b>then</b> { <b>return</b> ( <i>success</i> )	(3)
<b>do</b> {	(4)
<b>else</b> { <b>if</b> <i>T</i> = <i>Accumulator</i>	(4)
<b>then</b> { <b>return</b> ( <i>success</i> )	(5)
<i>List</i> $\leftarrow T$	(6)
<i>Accumulator</i> $\leftarrow [H Accumulator]$	(6)
<b>return</b> ( <i>failure</i> )	

<i>Statement</i>	(1)	(2)	(3)	(4)	(5)	(6)
<i>Hand Computation Step</i>	①	①	② ③	②	③	③

Table 1.3: Algorithm 1.4.3 and Related Hand Computations (Figs. 1.6 &amp; 1.7)

**Exercise 1.3.** Construct the pseudocode for the Prolog code in Example 1.6. Also establish the correspondence between the Prolog clauses and the statements of your pseudocode.

■

## 1.5 Generalization

Each clause of the predicates seen thus far with accumulator arguments fits one of the two *patterns* shown in Fig. 1.8.<sup>12, 13</sup>

Fig. 1.9 shows a more general scheme where we group *Input* and *Accumulator* into *Argument* which then is subjected to some transformations until a stopping criterion applies.

<sup>12</sup>For *palin/2* from Example 1.5 also to fit this mould, the *Result* argument is to be ignored.

<sup>13</sup>If necessary, apply first the technique from Example 1.4 to reduce the arity of *predicate* to 3.

```

predicate(Input,Accumulator,Accumulator) :-
    stopping-condition(Input,Accumulator).

predicate(Input,Accumulator,Result) :-
    carry-on-condition(Input,Accumulator),
    transform(Input,Accumulator,NewInput,NewAccumulator),
    !, predicate(NewInput,NewAccumulator,Result).

```

Figure 1.8: Typical Clause Structures of a Predicate with an Accumulator

```

predicate(Argument,Result) :- stopping-condition(Argument),
                               extract-info-from(Argument,Result)

predicate(Argument,Result) :- carry-on-condition(Argument),
                               transform(Argument,NewArgument),
                               !, predicate(NewArgument,Result).

```

Figure 1.9: Generalized Clause Structures

## 1.6 Case Study: The Perceptron Training Algorithm

### 1.6.1 Classification Problem

A basic problem in connectionist AI is that of finding a linear classifier for two groups of data in the space of  $n$ -tuples of real numbers. As an illustrative example, we consider the two-dimensional data in Table 1.4.

$x_1$	6.981	14.414	2.337	8.500	9.190	1.149	14.786	7.842
$x_2$	0.554	4.466	4.040	3.496	2.000	6.100	2.179	6.331
Label $d$	-1	+1	-1	+1	-1	-1	+1	+1

Table 1.4: Co-ordinates of Points in the Plane with Class Labels

Each of the 8 points belongs to one of the two classes labelled +1 or -1. A plot of the data with a separating straight line is shown in Fig. 1.10. The *Perceptron Training Algorithm* allows a separating straight line to be found if it exists (e.g. [7, 13, 14]); the data then is said to be *linearly separable*.

### 1.6.2 Algorithm

A simple decision rule for linearly separable data is based on the *perceptron* which in the two-dimensional case can be written in the form

$$d(x_1, x_2) = \begin{cases} +1 & \text{if } w_1x_1 + w_2x_2 \geq t, \\ -1 & \text{if } w_1x_1 + w_2x_2 < t, \end{cases} \quad (1.1)$$

with *weights*  $w_1, w_2$  and *threshold*  $t$ . The decision rule (1.1) generalizes for  $n$ -dimensional data to

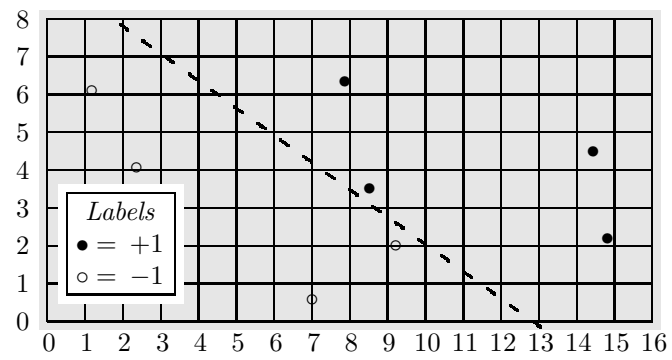


Figure 1.10: A Linearly Separable Data Set

$$d(x_1, \dots, x_n, x_{n+1}) = \begin{cases} +1 & \text{if } w_1x_1 + \dots + w_nx_n + w_{n+1}x_{n+1} \geq 0, \\ -1 & \text{if } w_1x_1 + \dots + w_nx_n + w_{n+1}x_{n+1} < 0, \end{cases} \quad (1.2)$$

with *weights*  $w_1, \dots, w_{n+1}$  and *unit bias*  $x_{n+1} = 1$ .<sup>14</sup> For later reference, (1.2) is presented in Fig. 1.11 as a procedure.

```

procedure CLASSIFY( $[x_1, \dots, x_{n+1}], [w_1, \dots, w_{n+1}]$ )
   $class \leftarrow \text{sign} \left( \sum_{k=1}^{n+1} w_k x_k \right)$ 
  return ( $class$ )

```

Figure 1.11: Classifying a Point

The *sign* function in Fig. 1.11 is defined by

$$\text{sign}(s) = \begin{cases} +1 & \text{if } s \geq 0, \\ -1 & \text{if } s < 0. \end{cases}$$

In Fig. 1.12 it is shown how a single updating step is carried out by the perceptron. It takes a sample point  $\mathbf{x}$  from the training data with the corresponding *desired* class label  $d$ , the current (list of) weights  $\mathbf{w}$  and returns the updated weights,  $\mathbf{w}^{(new)}$ . The positive constant  $c$ , the *learning rate*, is arbitrary but fixed throughout the whole training session.

To find a set of weights for which the decision rule correctly classifies *all* training points, the updating step from Fig. 1.12 is repeated as indicated in Fig. 1.13, p. 31. The weights' initial values and the learning rate are arbitrary; we have chosen  $\mathbf{w}^{(0)} = [-0.51, -0.35, 0.13]$  and  $c = 0.25$  in our example. After each iteration step, it is checked whether any of the training data points is misclassified, in which case iteration continues. Iteration is stopped as soon as all training data points are correctly classified. This is shown in Algorithm 1.6.3, p. 32.

### 1.6.3 Implementation

We represent (the by the unit bias augmented version of) Table 1.4 and the weights' initial values by the facts

```

ps([[ 6.981, 0.554, 1], ..., [ 7.842, 6.331, 1]]). % points
ds([-1, 1, -1, 1, -1, -1, 1, 1]). % classes
ws([-0.51, -0.35, 0.13]). % weights

```

The Perceptron Training Algorithm 1.6.3 will be implemented by the predicate *pta/6* with argument pattern

```

pta(+LearningRate, +Points, +DesiredOutputs, +Weights,
    -FinalWeights, -Iterations)

```

It calls in (P-1.12) the auxiliary predicate *pta/2*, which itself is structured according to Fig. 1.9.

#### Prolog Code P-1.12: Definition of *pta/6*

```

1 pta(LRate, Points, DesiredOutputs, Weights, FinalWeights, Iters) :-
2   pta(in(LRate, Points, DesiredOutputs, Weights, 0),
3     out(FinalWeights, Iters)).

```

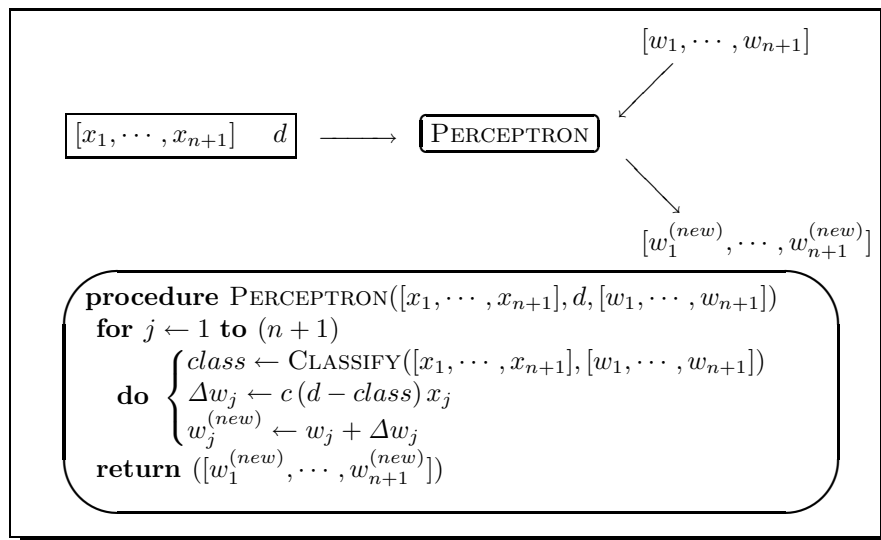


Figure 1.12: A Single Updating Step

The rôles of the arguments of *in/5* and *out/2* are obvious from the names chosen. (The last argument of *in/5* is an accumulator for the iteration number. It is initialized to zero in (P-1.12).) The definition in (P-1.13) follows the layout from Fig. 1.9.

**Prolog Code P-1.13: Definition of *pta/2***

```

1 pta(in(_,Ps,Ds,Ws,I),out(Ws,I)) :- classify_all(Ps,Ws,Ds), !. % clause 1
2 pta(Arg,Result)                    :- transform(Arg,NewArg),      % clause 2
3                                     !, pta(NewArg,Result).           %

```

With reference to Fig. 1.9 it is seen that

<sup>14</sup>Equation (1.2) thereby subsumes (1.1) by putting  $n = 2$  and  $t = -w_{n+1}$ .

<sup>15</sup>The symbol  $\text{++}$  stands for list concatenation.

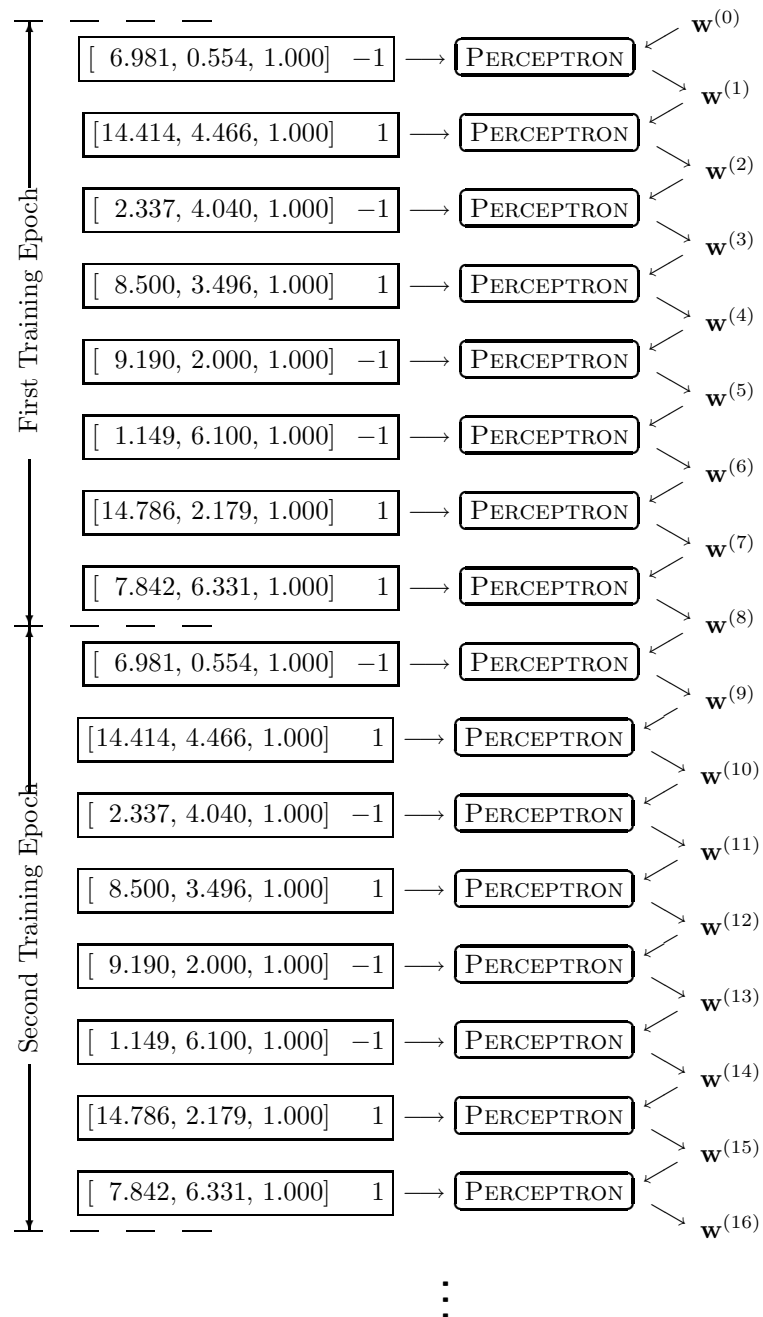


Figure 1.13: Applying the Perceptron Training Algorithm

**Algorithm 1.6.3:** PTA( $[\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(N)}], [d_1, \dots, d_N], \mathbf{w}$ )

**comment:** Perceptron Training Algorithm.

Iterate until all points are correctly classified.

**procedure** CLASSIFYALL( $[\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(N)}], \mathbf{w}$ )

**for**  $i \leftarrow 1$  **to**  $N$

**do**  $\{c_i \leftarrow \text{CLASSIFY}(\mathbf{x}^{(i)}, \mathbf{w})$

**return**  $([c_1, \dots, c_N])$

**main**

$Weights \leftarrow \mathbf{w}$

$Points \leftarrow [\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(N)}]$

$DesiredOutputs \leftarrow [d_1, \dots, d_N]$

$ActualOutputs \leftarrow \text{CLASSIFYALL}(Points, Weights)$

$Iterations \leftarrow 0$

**while**  $ActualOutputs \neq DesiredOutputs$

**do**  $\left\{ \begin{array}{l} [P|OtherPs] \leftarrow Points \\ [D|OtherDs] \leftarrow DesiredOutputs \\ Weights \leftarrow \text{PERCEPTRON}(c, P, D, Weights) \\ Points \leftarrow OtherPs \mathrel{+}^{15} [P] \\ DesiredOutputs \leftarrow OtherDs \mathrel{+} [D] \\ ActualOutputs \leftarrow \text{CLASSIFYALL}(Points, Weights) \\ Iterations \leftarrow Iterations + 1 \end{array} \right.$

**output**  $(Iterations, Weights)$



- In clause 1, *stopping-condition* is implemented by *classify\_all/3*, defined by recursion in (P-1.14).

**Prolog Code P-1.14: Definition of *classify\_all/3***

```

1 classify_all([],_,[]). % clause 1
2 classify_all([P|OtherPs],Weights,[Class|OtherCs]) :- % clause 2
3     classify(P,Weights,Class), !, %
4     classify_all(OtherPs,Weights,OtherCs). %

```

(The predicate *classify/3* is a straightforward implementation of the procedure in Fig. 1.11; for its definition, see the file *accumulator.pl*.)

- In clause 1, *extract-info-from* is realized by unification of the last two arguments of *in/5* with those of *out/2*.
- In clause 2, the *carry-on-condition* is implicitly defined by failure of the predicate *classify\_all/3* in clause 1.
- Finally, the predicate *transform/2* is defined by (P-1.15).

**Prolog Code P-1.15: Definition of *transform/2***

```

1 transform(in(C,[P|OtherPs],[D|OtherDs],Ws,Acc),
2         in(C,NewPs,NewDs,NewWs,NewAcc)) :-
3     append(OtherPs,[P],NewPs),
4     append(OtherDs,[D],NewDs),
5     perceptron(C,P,D,Ws,NewWs),
6     NewAcc is Acc + 1.

```

The predicate *perceptron/5* in (P-1.15), line 5, is a straightforward implementation of the weight updating step from Fig. 1.12. It is defined in (P-1.16).

**Prolog Code P-1.16: Definition of *perceptron/5***

```

1 perceptron(C,Point,D,Weights,NewWeights) :-
2     classify(Point,Weights,Class),
3     Const is C * (D - Class),
4     mult(Const,Point,DeltaWs),
5     add(Weights,DeltaWs,NewWeights).

```

The implementation thus defined we use to find after 801 iterations a correct classifier.

```

?- ps(_Ps), ds(_Ds), ws(W0), pta(0.25,_Ps,_Ds,W0,W,I).16
W0 = [-0.51, -0.35, 0.13]
W = [3.018, 4.1935, -39.87]
I = 801

```

While the initial weights give rise to some incorrect classifications,

```
?- ps(_Ps), classify_all(_Ps, [-0.51, -0.35, 0.13], Classes).
Classes = [-1, -1, -1, -1, -1, -1, -1, -1]
```

the new weights define a correct classifier,

```
?- ps(_Ps), classify_all(_Ps, [3.018, 4.1935, -39.87], Classes).
Classes = [-1, 1, -1, 1, -1, -1, 1, 1]
```

(The corresponding separating straight line

$$\{ (x_1, x_2) : 3.018x_1 + 4.1935x_2 - 39.87 = 0 \}$$

is shown in Fig. 1.10.)

---

<sup>16</sup>In the version of SWI-Prolog used here (version 3.4.5), variables whose name starts with an underscore (such as *\_Ps*) won't be displayed. Issue the query

```
?- set_prolog_flag(toplevel_print_anon, false).
Yes
```

at the beginning of the session to achieve the same effect with version 5.2.7 (the most recent version at the time of writing).

**Exercise 1.4.** To make the definition of *perceptron/5* in (P-1.16) complete, define *mult/3* and *add/3* thus implementing scalar multiplication and addition of vectors, respectively. Your definitions should be by both simple recursion and the accumulator technique. Reflect on the performance of each implementation. ■

**Exercise 1.5.** The Perceptron Training Algorithm may be carried out for a *fixed* number of iterations rather than until all points are correctly classified. Augment the definition of *pta/6* to cover this case too. Thus the argument pattern of *pta/6* is now

```
pta(+LearningRate,+Points,+DesiredOutputs,+Weights,
    -FinalWeights,?Iterations)
```

This modification is useful for instance in our example for confirming that the least number of iterations needed to classify all points correctly is indeed 801:<sup>17</sup>

```
?- ps(_Ps), ds(_Ds), ws(W0), pta(0.25,_Ps,_Ds,W0,W,800).
W0 = [-0.51, -0.35, 0.13]
W = [6.5085, 4.4705, -39.37]
?- ps(_Ps), classify_all(_Ps,[6.5085, 4.4705, -39.37],Classes).
Classes = [1, 1, -1, 1, 1, -1, 1, 1]
```

*Hint.* Use Algorithm 1.6.4 (p. 35) in lieu of Algorithm 1.6.3. A minimal change to clause 1 of *pta/2* (p. 30) will do. ■

**Algorithm 1.6.4:**  $\text{PTA}(c, [\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(N)}], [d_1, \dots, d_N], \mathbf{w}, m)$

**comment:** Perceptron Training Algorithm.  
Iterate  $m(> 0)$  number of times.

**main**

$\text{Weights} \leftarrow \mathbf{w}$

$\text{Points} \leftarrow [\mathbf{x}^{(1)}, \dots, \mathbf{x}^{(N)}]$

$\text{DesiredOutputs} \leftarrow [d_1, \dots, d_N]$

$\text{Iterations} \leftarrow m$

**repeat**

$\left\{ \begin{array}{l} [P|\text{OtherPs}] \leftarrow \text{Points} \\ [D|\text{OtherDs}] \leftarrow \text{DesiredOutputs} \\ \text{Weights} \leftarrow \text{PERCEPTRON}(c, P, D, \text{Weights}) \\ \text{Points} \leftarrow \text{OtherPs} \mathrel{++} [P] \\ \text{DesiredOutputs} \leftarrow \text{OtherDs} \mathrel{++} [D] \\ \text{Iterations} \leftarrow \text{Iterations} - 1 \end{array} \right.$

**until**  $\text{Iterations} = 0$

**output** ( $\text{Weights}$ )

<sup>17</sup>From the procedure PERCEPTRON in Fig. 1.12 it is seen that once a set of weights has been found which gives rise to correct classification for *all* points, further iterations won't change the weights' values. Thus, the fact that after 800 iterations some of the points are misclassified, shows that any lesser number of iterations won't do either. 801 is therefore the minimum number of iterations needed for correct classification.

## Chapter 2

# Difference Lists

Owing to the availability of unification in Prolog, there is a useful technique that allows predicates involving certain list operations to be implemented very efficiently. Because at the conceptual level the technique appears to be manipulating 'differences of lists', it is known as the *Difference List Technique*.

### 2.1 Implementations of List Concatenation

Suppose we want to concatenate the two lists `[a,b,c]` and `[d,e]` to give us the new list `[a,b,c,d,e]`; in other words, we want to *append* the list `[d,e]` to the list `[a,b,c]`. We can do this by the built-in predicate *append/3* as follows:

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

We use Prolog's *listing/1* to display the definition of *append/3*:

```
?- listing(append/3).
append([], A, A).
append([A|B], C, [A|D]) :- append(B, C, D).
```

Due to its recursive definition, *append/3* will be invoked four times when running our example. In general, the depth of the proof tree will be proportional to the length of the list in the first argument.

We want to explore a computationally more economical approach to the problem of list concatenation. Let us place in the database the following one-line definition of *app\_dl1/4*:<sup>1</sup>

```
app_dl1(A,B,B,A).
```

Let us carry out the following experiment:

```
?- app_dl1([a,b,c/X],X,[d,e],Z).
X = [d, e]
Z = [a, b, c, d, e]
```

---

<sup>1</sup>Notation: *app* stands for *append*; *dl* stands for *difference list*; and, 1 indicates that it is the first version – other (improved) versions soon to follow.

We have accomplished the intended *append* operation once again! Let us examine how. The following unifications have taken place:

1. A is unified with [a,b,c|X].
2. B is unified with X.
3. B is instantiated to [d,e].
4. A is unified with Z.

It is easily seen that the net result of 1–4 is that Z is instantiated to [a,b,c,d,e]. We now define a new predicate *app\_dl2/3* which is slightly different but still equivalent to *app\_dl1/4*:

*app\_dl2*(A-B,B,A).

(We have chosen, for reasons to be explained soon, to reduce the arity by one by 'merging' the first two arguments of *app\_dl1/4* to a hyphenated term.<sup>2</sup>) Let us see how *app\_dl2/3* behaves:

```
?- app_dl2([a,b,c|X]-X,[d,e],Z).
X = [d, e]
Z = [a, b, c, d, e]
```

We get the earlier response since the unification steps carried out are as before. The hyphen notation chosen in *app\_dl2/3* is more customary, however, and it lends itself to the following *interpretation*.

The term [a,b,c|X]-X is interpreted as a representation of the list [a,b,c] in *difference list notation*. The variable X stands for *any* list. If we unify this term with Y-[], then Y will be instantiated to [a,b,c] in the usual list notation:

```
?- [a,b,c|X]-X = Y-[] .
X = []
Y = [a, b, c] ;
No
```

Fig. 2.1 shows how the three conceptual lists are interrelated. It must be emphasized that the above interpretation is a mere working model for what is actually taking place inside Prolog. It turns out, however, that it is unnecessary to look beyond this conceptual model when working with 'difference lists'. To reinforce this point, let us consider yet another (the fourth) version of *append*:

*app\_dl4*(A-B,B-C,A-C).

---

<sup>2</sup>We could have chosen some other operator for the term in the first argument of the new predicate; for example, the same effect is achieved by:

```
:- op(50,xfx,&).
...
app_dl3(A&B,B,A).
```

The first line – a *directive* – declares & as an infix operator of precedence 50. In the first argument of *app\_dl3/3* a term A&B replaces the former A-B. The response will be as before:

```
?- app_dl3([a,b,c|X]&X,[d,e],Z).
X = [d, e]
Z = [a, b, c, d, e]
```

If the hyphen (-) is chosen to denote difference lists, however, no operator declaration is required since it is a Prolog built-in.

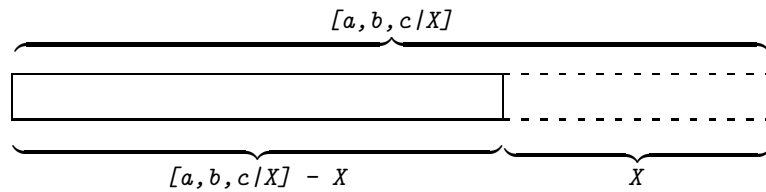


Figure 2.1: Difference List

All arguments of `app_dl4/3` are difference lists; the earlier query now reads as follows.

```
?- app_dl4([a,b,c/X]-X,[d,e/Y]-Y,Z1-Z2).
X = [d, e|_G370]
Y = _G370
Z1 = [a, b, c, d, e|_G370] Z2 = _G370 ;
No
```

The (difference) lists involved here are interrelated as shown in Fig. 2.2. The concatenated list is returned in the last argument of `app_dl4/3` in the form of `[a, b, c, d, e|_G370]-_G370`.

(`_G370` is some internally chosen variable name.) It is easily seen that this is accomplished in *one* unification

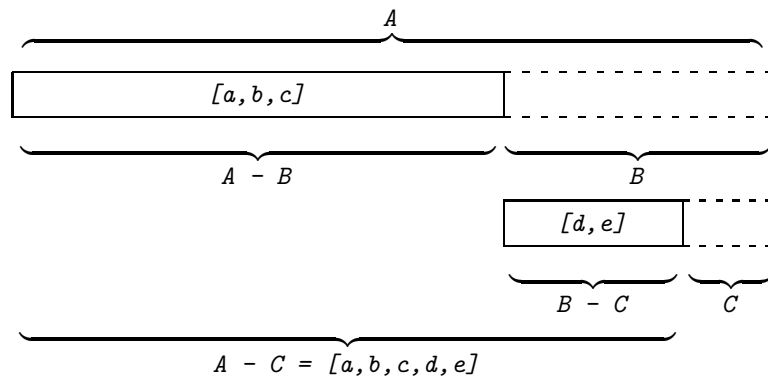


Figure 2.2: List Concatenation by Difference Lists

step *irrespective of the lengths of the lists to be concatenated*. (Appending difference lists is therefore a *constant time* operation.)

We now want to confirm all this experimentally, too. To get started, we need some method for creating difference lists. One way forward is by means of `append/3`. For example, in

```
?- setof(_N,between(1,5,_N),Ns), append(Ns,X,L), DL = L-X.
Ns = [1, 2, 3, 4, 5]
X = _G468
L = [1, 2, 3, 4, 5|_G468]
DL = [1, 2, 3, 4, 5|_G468]-_G468
```



---

**Built-in Predicates: *bagof/3* and *setof/3***

*bagof(+Item,+Goal,?Items)* is used to collect in the list *Items* instances of *Item* for which *Goal* is satisfied. Free variables in *Goal* will be instantiated to values for which *Goal* succeeds. Example: Throw two dice to record all possible results whose sum does not exceed 3.

```
?- bagof((_D1,_D2),(between(1,6,_D1), between(1,6,_D2),
                    S is _D1 + _D2, S =< 3), Pairs).
S = 2 Pairs = [ (1, 1)] ;
S = 3 Pairs = [ (1, 2), (2, 1)] ;
No
```

We collect the pairs *irrespective* of the values taken by *S* by

```
?- bagof((_D1,_D2),_S^(between(1,6,_D1), between(1,6,_D2),
                    _S is _D1 + _D2, _S =< 3), Pairs).
Pairs = [ (1, 1), (1, 2), (2, 1)] ;
No
```

*setof/3* is used in a similar fashion except that the entries in *Items* are sorted in ascending order and there are no multiple entries in *Items*.

---

the list `[1,2,3,4,5]` is written as a difference list DL using the internal variable `_G468`.

---

**Built-in Predicate: *between(+Low,+High,?Value)***

On backtracking, the variable *Value* is unified with all integer values between *Low* and *High*. Example:

```
?- between(-1,3,V).
V = -1 ;
V = 0 ;
...
```

---

We now *append* to DL the difference list form of `[d,e]` and also measure the number of inferences by *time/1*:

```
?- setof(_N,between(1,5,_N),Ns), append(Ns,X,L), DL = L-X,
   time(app_dl4(DL,[d,e|Y]-Y,Z1-Z2)).
% 1 inferences in 0.00 seconds (Infinite Lips)
Ns = [1, 2, 3, 4, 5]
X = [d, e|_G691]
L = [1, 2, 3, 4, 5, d, e|_G691]
DL = [1, 2, 3, 4, 5, d, e|_G691]-[d, e|_G691]
Y = _G691
Z1 = [1, 2, 3, 4, 5, d, e|_G691]
Z2 = _G691
```

We need one single inference step only. On the other hand, the corresponding operation with proper lists is more expensive (6 inferences):

---

```
?- setof(_N,between(1,5,_N),Ns), time(append(Ns,[d,e],Z)).
% 6 inferences in 0.00 seconds (Infinite Lips)
Ns = [1, 2, 3, 4, 5]
Z = [1, 2, 3, 4, 5, d, e]
```

(You may wish to repeat the experiment with larger lists by adjusting the second argument in *between/3* above.)

## 2.2 Implementations of List Flattening

Lists in Prolog can have a nested structure; for example, `[a,[b,[],[c,a],e]]` is a valid list. The built-in predicate *flatten/2* is designed to ‘linearize’ lists as indicated below:

```
?- flatten([a,[b,[],[c,a],e]],L).
L = [a, b, c, a, e]
```

In this section, we are going to explore several implementations of *flatten/2* the most efficient of which will turn out to be the one based on the difference list technique.

### 2.2.1 Project: Lists as Trees & *flatten/2*

The usual square bracket notation for lists is just a notational convenience. The underlying (but not immediately obvious) structure is that of a *term* with the *functor* `'.'` (dot). This may be demonstrated by using the triad of built-in predicates *functor/3*, *arg/3* and `=..`/<sup>3</sup>. For example,

```
?- functor([a,b,c],F,A).
F = '.,'
A = 2
```

shows that the list `[a,b,c]` (as any list) is represented as a term with arity 2 and functor `'.'`. We may find the values of the term's first and second argument respectively by

```
?- arg(1,[a,b,c],A).
A = a
```

and

```
?- arg(2,[a,b,c],A).
A = [b, c]
```

The same may be gleaned from using *univ*:

```
?- [a,b,c] =.. L.
L = ['.', a, [b, c]]
```

Finally, we may even use the dot-notation when working with lists; for example, `[b,c]` may be appended to `[a]` by

```
?- append(. (a, []), . (b, . (c, [])), L).
L = [a, b, c]
```

Even though lists are not written in practice in this way (since the square bracket notation is more suited to human use), the dot-notation is useful for representing the structure of lists (and that of *nested* lists in particular) as a tree of terms. As an example, the tree representation of the list `[a,[b,[],[c,a],e]]` is shown in Fig. 2.3. The following is easily observed:

- The flattened list `[a,b,c,a,e]` may be formed from the tree representation of `[a,[b,[],[c,a],e]]` by visiting all leaf terms in turn in a counter-clockwise direction and by collecting those leaves from left-hand branches which are not the empty list `[]`.

This process will flatten *any* list. Exercises 2.1– 2.3 below elaborate on this idea, leading to an implementation of *flatten/2*.

We can easily convert from the dot-notation to the square bracket notation; for example,

```
?- L = . (a, . (b, . ([], . (c, . (a, [])), . (e, []))), []).
L = [a, [b, [], [c, a], e]]
```

The reverse process has to be programmed.

**Exercise 2.1.** Define a predicate *sharp/2* for converting lists into *terms* with functor `#/2` as exemplified by the following query.<sup>4</sup>

<sup>3</sup>This is an infix predicate and is called *univ*.

<sup>4</sup>Ideally, we would like to have a predicate for converting lists in the square bracket notation to a (possibly nested) *term* with functor `'.'`. However, this is not immediately achievable since as soon as Prolog sees a term whose functor is `'.'` it will automatically display it in the square bracket notation.

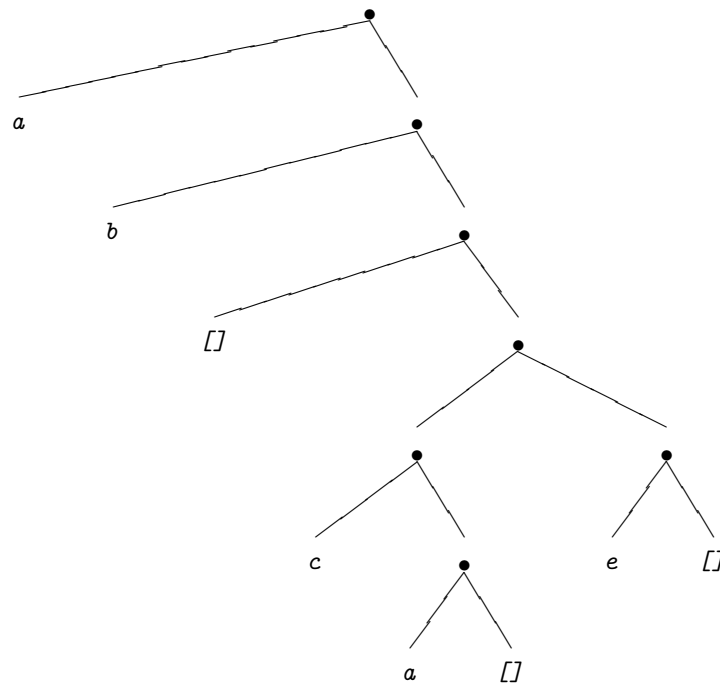


Figure 2.3: Tree Representation of  $[a, [b, [], [c, a], e]]$

```
?- sharp([a,[b,[],[c,a],e]],S).
S = #(a, #(b, #([], #(c, #(a, [])), #(e, []))), [])
```

■

*Hint.* The definition should be recursive and the ‘boundary case’ may be verified by using the built-in predicate *proper\_list/1*.

If we now had a predicate *lf/2* for returning the leaf nodes from the *#*-tree of a list (as specified earlier), we could easily implement *flatten/2*, as indicated by

```
?- sharp([a,[b,[],[c,a],e]],_S), bagof(_L, lf(_S,_L),Ls).
Ls = [a, b, c, a, e]
```

**Exercise 2.2.** Define a predicate *lf(+S, -L)* which on backtracking unifies *L* with the left-hand leaves (not equal to *[]*) of the *#*-tree *S*:

```
?- lf(#(a, #(b, #([], #(c, #(a, [])), #(e, []))), [],L).
L = a ;
L = b ;
L = c ;
L = a ;
L = e ;
No
```

■

*Note.* Your implementations of *sharp/2* and *lf/2* should be able to cope with lists involving variables, too:

```
?- sharp([a,[Y,[b,X]],c,f(X)],S).
Y = _G315
X = _G321
S = #(a, #(b, #(_G315, #(_G321, [])), []), #(c, #(_G321, [])))

?- sharp([a,[_Y,[b,_X]],c,f(_X)],_S), !, lf(_S,Leaf).
Leaf = a ;
Leaf = _G435 ;
Leaf = b ;
Leaf = _G441 ;
Leaf = c ;
Leaf = f(_G441) ;
No
```

**Exercise 2.3.** Now define a first version of *flatten/2*:

```
?- flatten_1([a,[b,[],[c,a],e]],L).
L = [a, b, c, a, e]
?- flatten_1([a,[Y,[b,X]],c,f(X)],L).
Y = _G339
X = _G345
L = [a, _G339, b, _G345, c, f(_G345)]
```

■

As indicated in Exercise 2.1, in the first instance Prolog won't convert a list to a term whose functor is the dot; more precisely, such a conversion won't be visible since Prolog automatically shows lists in the square bracket notation. There are two ways, however, to instruct the Prolog system to suppress this conversion automatism.

- The built-in predicate *write\_term/2* may be used to *display* a term such that any list within it will be shown in the generic term-representation using the '.' functor:

```
?- write_term([a, [b, [], fun([c, a]), e]], [ignore_ops = true]).
.(a, .(. (b, .([, .(fun(. (c, .(a, []))), .(e, []))), .([])))
```

The second argument of *write\_term/2* is a list-of-options where the flag *ignore\_ops* is set to *true*; the default is *false*.

- We may achieve the same effect for the *entire* interactive session by the built-in predicate *set\_prolog\_flag/2*; this is exemplified below:

```
?- L = [a, [b, [], fun([c, a]), e]].
L = [a, [b, [], fun([c, a]), e]]
?- set_prolog_flag(toplevel_print_options, [ignore_ops=true]).
Yes
?- L = [a, [b, [], fun([c, a]), e]].
L = .(a, .(. (b, .([, .(fun(. (c, .(a, []))), .(e, []))), .([])))
```

Once it has been set by the user with *set\_prolog\_flag/2*, the state of *ignore\_ops* is checked by the built-in predicate *current\_prolog\_flag/2*:

```
?- current_prolog_flag(toplevel_print_options, [ignore_ops=V]).
V = true
```

In the next exercise, you are asked to implement a predicate allowing lists to be shown in the dot-notation.

**Exercise 2.4.** Based on *sharp/2* from Exercise 2.1, define a predicate *dot/1* for *displaying* lists in the dot-notation as exemplified by the following query.

```
?- dot([a, [b, [], [c, a], e]]).
.(a, .(. (b, .([, .(. (c, .(a, []))), .(e, []))), .([])))
```

Thus the predicate *dot/1* will be something akin to *write\_term/2* (with the flag *ignore\_ops* set to *true*). However, lists within Prolog terms with other than the dot-functor should be displayed by *dot/1* in the square bracket notation:

```
?- dot([a, [b, [], fun([c, a]), e]]).
.(a, .(. (b, .([, .(fun([c, a]), .(e, []))), .([])))
```

*Hint.* Proceed along the following lines.

- Use the built-in predicate *term\_to\_atom/2* to convert the list in the sharp-notation to an atom.

---

**Built-in Predicate:** *term\_to\_atom(?Term,?Atom)*

The atom *Atom* corresponds to the term *Term*. Example:

```
?- term_to_atom(fun1(a,fun2(c),d),A).
A = 'fun1(a, fun2(c), d)'
```

---

- Convert the atom into a list of one-character atoms by using the built-in predicate *atom\_chars/2* (c.f. p. 126).
- Define a predicate *sharps\_to\_dots/2* by the accumulator technique for converting sharps to dots.<sup>5</sup> Example:

```
?- sharps_to_dots([#, '(', a, ', ', '[', ']', ')'],D).
D = ['. ', '(', a, ' ', '[', ']', ')']
```

---

<sup>5</sup>Alternatively, the built-in function *maplist/3* from p. 127 may be used to define *sharps\_to\_dots/2*.

- Finally, concatenate the list of one-character atoms thus obtained to an atom by using `concat_atom/2` from p. 126. Also show the result.

■

## 2.2.2 Flattening Lists by `append/3`

Another implementation<sup>6</sup> of `flatten/2`, proposed by Clocksin in [1], p. 58, uses the predicate `append/3`:

### Prolog Code P-2.1: Clocksin's definition of `flatten/2`

```

1 flatten_3([], []). % clause 1
2 flatten_3([H/T], L1) :- flatten_3(H, L2), % clause 2
3 flatten_3(T, L3), %
4 append(L2, L3, L1). %
5 flatten_3(X, [X]). % clause 3

```

This definition is easily understood through a *declarative* reading:

- Clause 1: This is the base case. It says that an empty list is flattened into an empty list.
- Clause 2: This is the recursive step. A list `[H/T]` (whose head `H` is possibly a list itself) is flattened in the following steps.
  1. Flatten the head `H`.
  2. Flatten the tail `T`.
  3. Concatenate the latter two flattened lists.
- Clause 3: The flattened version of a term that unifies neither with `[]` nor with `[H/T]` is the term itself. This clause is intended to cater for the case of list entries which are not themselves lists; a *ground* atom (i.e. a one without a variable) is an example thereof.

List flattening defined by (P-2.1) works as intended for (nested) lists whose tree representation has leaves which are ground atoms or are terms with other than the dot functor; for example,

```

?- flatten_3([a, [b, [f(X, d), []], [c, f(X), a], e]], L).
X = _G414
L = [a, b, f(_G414, d), c, f(_G414), a, e]

```

However, lists some of whose leaves are free variables, won't be correctly flattened by `flatten_3/2`:

```

?- flatten_3([a, [Y, [b, X]], c, f(X)], L).
Y = []
X = []
L = [a, b, c, f([])]

```

**Exercise 2.5.** Augment the definition of `flatten_3/2` such that it correctly handles also lists involving free variables. Another (though easy to rectify) shortcoming of `flatten_3/2` is that on backtracking it will return spurious solutions:

<sup>6</sup>We count this implementation as *version 3* as you will find, in connection with the solution of Exercise 2.3, a 'version 2' is discussed in Appendix A.2 on p. 147.



```
?- flatten_3([a, [b, []], [c, a], e],L).
L = [a, b, c, a, e] ;
L = [a, b, c, a, e, []]
```

Your improved implementation (version 4) should solve also this problem.

■

### 2.2.3 *flatten/2* by the Difference List Technique

(P-2.2) shows a clause-by-clause ‘translation’ of the definition of *flatten\_3/2* in terms of difference lists ([1], p. 58).

#### Prolog Code P-2.2: Difference list based definition of *flatten/2*

```
1 flatten_5(L,F) :- flatten_dl(L,F-[]), !.           % clause 1
2                                     %
3 flatten_dl([],L-L).                               % clause 2
4 flatten_dl([H|T],L1-L3) :- flatten_dl(H,L1-L2), % clause 3
5                               flatten_dl(T,L2-L3). %
6 flatten_dl(X,[X|Z]-Z).                             % clause 4
```

The *append* goal does not appear in (P-2.2) as list concatenation is now accomplished by difference lists. *flatten\_5/2* will behave identically to *flatten\_3/2* except that its solution is unique because of the *cut (!)* in clause 1.

**Exercise 2.6.** The predicate *flatten\_5/2* in (P-2.2) won’t correctly flatten lists involving free variables. Modify (P-2.2) to resolve this problem.

■

### 2.2.4 Comparing Different Versions

We have developed several versions of *flatten/2* in the previous section and now their relative performance will be assessed. To do this, we need a way of generating nested lists which are ‘complicated’ enough to cause a noticeable amount of computing time when flattened. A predicate *nested(+Num, -List)* will prove useful for this purpose: given the positive integer *Num*, *List* should be unified with a nested list in the following fashion:

```
?- nested(9,L).
L = [[[[[[[[[1], 2], 3], 4], 5], 6], 7], 8], 9]
```

**Exercise 2.7.** Define the predicate *nested/2* by the accumulator technique and then use it to time the performance of the various versions of *flatten/2* by the built-in predicate *time/1*.

■

## 2.3 Implementations of List Reversal

There are several ways we can define our own version of the built-in predicate *reverse/2*. Its first implementation (P-2.3) uses *append/2*.

**Prolog Code P-2.3: First implementation of *reverse/2***

```

1 reverse_1([], []). % clause 1
2 reverse_1([H|T], R) :- reverse_1(T, L), % clause 2
3                       append(L, [H], R). %

```

A declarative reading of clause 2 in (P-2.3) is suggested in Fig. 2.4.

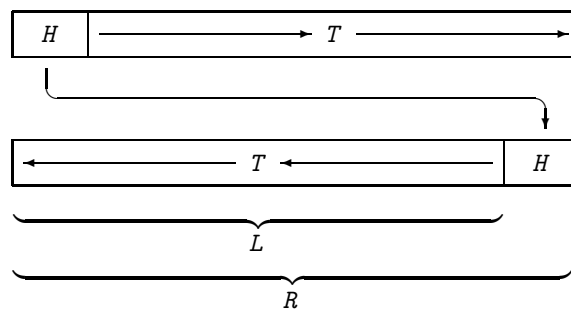


Figure 2.4: Declarative Reading of (P-2.3)

Another implementation of list reversal, now by the accumulator technique, is by (P-2.4) (see Example 1.1, p. 16):

**Prolog Code P-2.4: A second implementation of *reverse/2***

```

1 reverse([], R, R). % clause 1
2 reverse([H|T], Acc, R) :- reverse(T, [H|Acc], R). % clause 2
3 reverse_2(L, R) :- reverse(L, [], R). % clause 3

```

(P-2.3) may be rewritten in terms of difference lists as follows:

**Prolog Code P-2.5: Definition of *reverse/2* by difference lists**

```

1 rev_dl([], L-L). % clause (a1)
2 rev_dl([X], [X|L]-L). % clause (a2)
3 rev_dl([H|T], L1-L3) :- rev_dl(T, L1-L2), % clause (a3)
4                       rev_dl([H], L2-L3). %
5 reverse_3(L, R) :- rev_dl(L, R-[]), !.

```

Notice that clause (a2) in (P-2.5) does not directly correspond to any of the clauses in (P-2.3); it simply defines the difference list representation of (the reverse of) a list with a single entry.

### 2.3.1 Program Transformations

The performance of a predicate with a given definition can sometimes be enhanced by employing certain transformations leading to a new but logically equivalent form. Even though this topic is not directly related to the difference list technique, it is opportune to address this issue here. Specifically, we are going to demonstrate how the three clauses (a1)–(a3) in (P-2.5) can be transformed by *folding* and *unfolding* into the logically equivalent clauses (b1)–(b2) in (P-2.6):

**Prolog Code P-2.6: Concise definition of *rev\_dl/2***

```
1 rev_dl([],L-L). % clause (b1)
2 rev_dl([H/T],L1-L2) :- rev_dl(T,L1-[H/L2]). % clause (b2)
```

(For an in-depth exposition of both folding and unfolding, see [9].)

## Unfolding

Let us assume that we have in our Prolog knowledge base two clauses of the following form:

$$A \quad :- \quad B_1, \dots, B_m, C, B_{m+1}, \dots, B_n. \quad (2.1)$$

$$C \quad :- \quad D_1, \dots, D_k. \quad (2.2)$$

Then the clause

$$A \quad :- \quad B_1, \dots, B_m, D_1, \dots, D_k, B_{m+1}, \dots, B_n. \quad (2.3)$$

is a logical consequence of (2.1)–(2.2), inferred by an *Elementary Unfolding Operation*. Equation (2.3) is said to have been obtained by *unfolding* (2.1) upon the goal  $C$ . We note that,

- The requirement that the head of one clause be *identical* to one of the goals in the body of another clause can be relaxed to the two *unifying*. (This is a mere reflection on Prolog’s inference mechanism.)
- In general, the new clause (2.3) won’t be a replacement for (2.1) since in the database there may be other clauses whose head is identical to (or unifies with) the goal  $C$  in (2.1). To *replace* a clause like (2.1), we would have to carry out each and every possible elementary unfolding operation on the goal  $C$  in (2.1); in such a case, a *Complete One Step Unfolding* (COSU) is said to have been carried out.
- Finally, the two clauses (2.1) and (2.2) need not be distinct; they may be replicas of one and the same clause from the database. In fact, for a COSU, also such ‘self-unfoldings’ have to be considered. (This may be of interest for recursively defined predicates.)

Let us now turn to our specific example: we want to do a COSU on the call  $\text{rev\_dl}([H], L2-L3)$  in clause (a3) of (P-2.5). We represent the clauses (a1)–(a3) in (P-2.5) equivalently by (P-2.7)

**Prolog Code P-2.7: Equivalent form of (a1)–(a3) in (P-2.5)**

```

1 rev_dl([], L-L) :- true.
2 rev_dl([X], [X/L]-L) :- true.
3 rev_dl([U/V], W1-W3) :- rev_dl(V, W1-W2),
4                          rev_dl([U], W2-W3).
```

and then seek to unify in turn the head of each with the term  $\text{rev\_dl}([H], L2-L3)$ . This can be done ‘by hand’, or, more reliably, by using Prolog’s unification mechanism:

```

?- rev_dl([], L-L) = rev_dl([H], L2-L3).
No
?- rev_dl([X], [X/L]-L) = rev_dl([H], L2-L3).
X = _G372
L = _G376
H = _G372
L2 = [_G372|_G376]
L3 = _G376
Yes
?- rev_dl([U/V], W1-W3) = rev_dl([H], L2-L3).
U = _G372
```

```
V = []
W1 = _G375
W3 = _G376
H = _G372
L2 = _G375
L3 = _G376
Yes
```

The first unification attempt fails. The second unification succeeds and gives rise to the clause

```
rev_dl([_G372|T],L1-_G376) :- rev_dl(T,L1-[_G372|_G376]), true.
```

The third unification also succeeds, giving rise to the clause

```
rev_dl([_G372|T],L1-_G376) :- rev_dl(T,L1-_G375),
                               rev_dl([],_G375-W2),
                               rev_dl([_G372],W2-_G376).
```

(This last step is an instance of an elementary unfolding operation involving self-unfolding.) The one step unfolding operation is now complete and the last two clauses thus obtained may replace clause (a3) in (P-2.5). The new database is shown in (P-2.8).<sup>7</sup>

**Prolog Code P-2.8: Partially transformed clauses**

```
1 rev_dl([],L-L). % clause (a1)
2 rev_dl([X],[X|L]-L). % clause (a2)
3 rev_dl([H|T],L1-L2) :- rev_dl(T,L1-[H|L2]). % clause (a3.1)
4 rev_dl([H|T],L1-L3) :- rev_dl(T,L1-L2), % clause (a3.2)
5                          rev_dl([],L2-W), %
6                          rev_dl([H],W-L3). %
```

As is illustrated here, the new database after unfolding is not smaller than the initial one. We shall, however, shortly identify the clauses (a2) and (a3.2) in (P-2.8) as *redundant*.

Clause (a2) in (P-2.8) is redundant for it may be inferred from (a1) and (a3.1) in an elementary unfolding operation on the call `rev_dl(T,L1-[H|L2])` in clause (a3.1).<sup>8</sup> The requisite unification is

```
?- rev_dl([],L-L) = rev_dl(T,L1-[H|L2]).
L = [_G360|_G361]
T = []
L1 = [_G360|_G361]
H = _G360
L2 = _G361
Yes
```

It gives rise to the clause

```
rev_dl([_G360|[]],[_G360|_G361]-_G361) :- true.
```

which, after some variable renaming, is recognized as clause (a2) in (P-2.8).

It is seen that sometimes the database may be reduced by showing that one of its clauses can be inferred from the other ones by unfolding. Here, for a further reduction of the database we need another technique, called *folding*.

<sup>7</sup>Notice that some of the variables are renamed when writing down (P-2.8).

<sup>8</sup>As before, read clause (a1) in (P-2.8) as `rev_dl([],L-L):- true`.

## Folding

Let us assume that we have two clauses in the Prolog database that are of the form

$$A \quad :- \quad B_1, \dots, B_m, C, B_{m+1}, \dots, B_n. \quad (2.4)$$

$$D \quad :- \quad C. \quad (2.5)$$

Let us furthermore assume that (2.5) is the *only* clause in the database whose head is the term  $D$ . Then, if during the computation it is found that the goal  $D$  succeeds, we can infer that also  $C$  holds.<sup>9</sup> We can therefore augment the database by the clause

$$A \quad :- \quad B_1, \dots, B_m, D, B_{m+1}, \dots, B_n. \quad (2.6)$$

called the *folding* of clause (2.4). A more general formulation says that if some term  $D'$  is found to hold which *unifies with*  $D$ , then

$$A \quad :- \quad B_1, \dots, B_m, D', B_{m+1}, \dots, B_n.$$

may be inferred in lieu of clause (2.6).

We now want to apply these ideas to eliminate clause (a3.2) in (P-2.8). As a first step, we show that the clauses

```
L1 = L2 :- rev_dl([], L1-L2).      % clause (c1)
W1 = [E|W2] :- rev_dl([E], W1-W2). % clause (c2)
```

are a logical consequence of (a1) and (a3.1) in (P-2.8).<sup>10</sup>

To justify (c1), we observe that

- Clause (a1) is equivalent to

```
rev_dl([], L1-L2) :- L1 = L2. % clause (d)
```

- The term `rev_dl([], L1-L2)` does not unify with any of the heads in (a1) and (a3.1) hence we may infer clause (c1) from clause (d). (This reasoning is identical to that for justifying folding.)

To justify (c2), we observe that

- `rev_dl([E], W1-W2)` will unify with the head of clause (a3.1) only:

```
?- rev_dl([E], W1-W2) = rev_dl([H|T], L1-L2).
E = _G372
W1 = _G375
W2 = _G376
H = _G372
T = []
L1 = _G375
L2 = _G376
Yes
```

<sup>9</sup>In the absence of clause (2.5), the query `?- not(D).` would succeed by the *Closed World Assumption* which states that the negation of anything which cannot be inferred from the database is deemed **true**. Therefore,  $D$  can only hold if  $C$  holds.

<sup>10</sup>More precisely, (c1) and (c2) are a consequence of the *completion* of (a.1) and (a3.1).

- We may therefore infer the ‘reverse’ of clause (a3.1) with the above instantiation pattern as

$$\text{rev\_dl}([],\_G375-[_G372|_G376]) \text{ :- rev\_dl}([\_G372|[]],\_G375-\_G376).$$

or, in a more readable format,

$$\text{rev\_dl}([],W1-[E|W2]) \text{ :- rev\_dl}([E],W1-W2). \text{ \% clause (e)}$$

- Finally, we use clause (e) to obtain clause (c2) by unfolding on the call  $\text{rev\_dl}([],L1-L2)$  in clause (c1).

To infer now clause (a3.2) from (a1) and (a3.1) we *hypothesize* the body (i.e. the conjunction of the goals) of (a3.2):

$$\text{rev\_dl}(T,L1-L2), \text{ rev\_dl}([],L2-W), \text{ rev\_dl}([H],W-L3).$$

We infer by clause (c1) that

$$L2 = W.$$

and therefore

$$\text{rev\_dl}([H],L2-L3).$$

from which by clause (c2)

$$L2 = [H/L3].$$

and therefore

$$\text{rev\_dl}(T, L1 - [H/L3]) \text{ :- true.}$$

Unfold now clause (a3.1) to get

$$\text{rev\_dl}([H/T], L1 - L3).$$

which is indeed the head of clause (a3.2).<sup>11</sup>

An interpretation of clause (b2) in (P-2.6) is shown in Fig. 2.5. It admits the following declarative interpre-

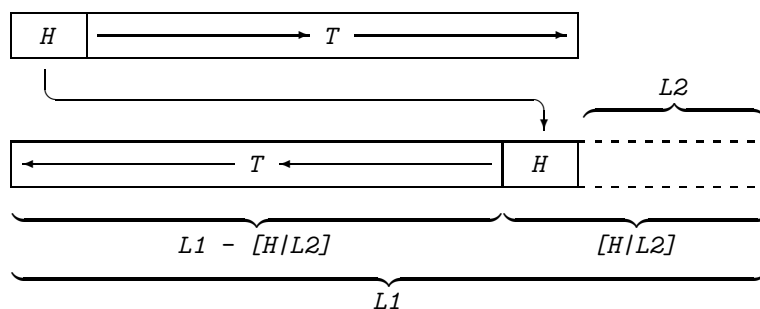


Figure 2.5: Illustrating Clause (b2) in (P-2.6)

tation:

The difference list  $L1 - L2$  is the reverse of the list  $[H/T]$  if the difference list  $L1 - [H/L2]$  is the reverse of  $T$ .

(This shows once again that we can think of difference lists as if they were true differences of lists!)

**Exercise 2.8.** Time the performance of the four versions of *reverse/2* and comment on the results. You should generate long lists (of consecutive integers) by using the built-in predicates *between/3* and *findall/3*.<sup>12</sup>

■

**Exercise 2.9.** Fig. 2.6 is an analogue of Fig. 2.5 for an enhanced implementation of *reverse/2*, also based on the difference list technique.

- Give a declarative reading of Fig. 2.6.
- Define a new version of *reverse/2* based on Fig. 2.6.
- Obtain your new version also by unfolding clause (b2).

<sup>11</sup>The foregoing reasoning is an instance of the application of the *Implication Introduction Rule* in Propositional Calculus.

<sup>12</sup>*findall/3* is identical to *bagof/3* (see p. 41) except that *findall/3* will return the empty list and succeed in cases where *bagof/3* fails.



- (d) Assess the new version's behaviour as in Exercise 2.8.
- (e) What would be a further enhancement to this implementation and how could the idea be generalized?

■

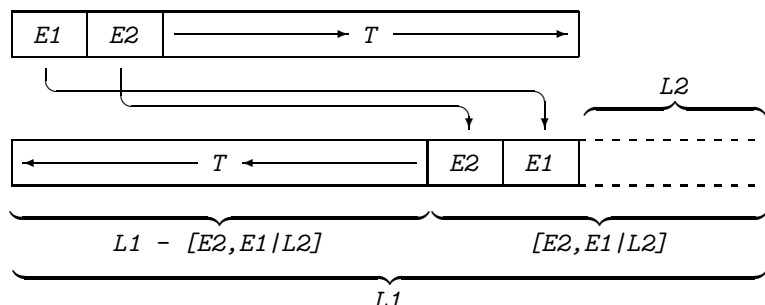


Figure 2.6: Illustrating Exercise 2.9

### 2.3.2 Difference Lists as Accumulators

Close inspection of clause (b2) in (P-2.6) reveals another interesting feature. If *rev\_dl* is interpreted as a predicate with arity 3 then its third argument may be thought of as an accumulator akin to the second argument of *reverse/3* in (P-2.4), p. 50. The other two arguments also correspond to each other accordingly. This shows, perhaps surprisingly, that two techniques based on entirely different approaches may result in the same implementation. (You will find some other examples on the similarity of the two techniques in [16], pp. 243–244.)

## 2.4 Case Study: Dijkstra's Dutch Flag Problem

We use Dijkstra's Dutch Flag Problem (e.g. [16]) to illustrate how a predicate defined in terms of *append/3* can be recast to a more efficient form by the difference list technique.

A list of terms of the form *col(Object, Colour)* is defined by the predicate *items/1* where *Colour* is one of the Dutch national colours, i.e. *red*, *white* or *blue*.

```
items([col(sky,blue), col(tomato,red), col(milk,white),
      col(blood,red), col(ocean,blue), col(cherry,red),
      col(snow,white)]).
```

We want to define a predicate *dijkstra/2* for arranging the items in the order of the Dutch flag's colours. Within each colour group, the original order should be retained:

```
?- items(_Items), dijkstra(_Items, Grouped).
Grouped = [col(tomato, red), col(blood, red), col(cherry, red),
          col(milk, white), col(snow, white), col(sky, blue),
          col(ocean, blue)]
```

### 2.4.1 Basic Implementation Using *append/3*

The idea for a basic version of *dijkstra/2* is as follows. We define three predicates — one for each colour — for returning the list of items of that particular colour. These lists are then concatenated to a list of grouped items.

Below is shown the definition of *reds(+Items,-Reds)*; the other two predicates are defined in an analogous manner.

**Prolog Code P-2.9: Definition of *reds/2***

```

1 reds([], []).                                     % clause 1
2 reds([col(Object,red)|T],[col(Object,red)|L]) :- reds(T,L). % clause 2
3 reds([col(_,Colour)|T],L) :- Colour \= red,                % clause 3
4                               reds(T,L).                    %

```

(P-2.9) is a straightforward recursive definition supported by the following declarative reading:

- Clause 1: If *Items* is the empty list then *Reds* will be empty.
- Clause 2: Assume that the list *L* comprises all red entries of *T*. Then, the same relationship holds for the lists *[Item/L]* and *[Item/T]* if *Item* is red.
- Clause 3: Assume again that the list *L* comprises all red entries of *T*. Also assume that *Item* is *not* red. Then, *L* comprises all red entries of the augmented list *[Item/T]*.

*reds/2* behaves as expected,

```
?- items(_Items), reds(_Items,Reds).
```

```
Reds = [col(tomato, red), col(blood, red), col(cherry, red)]
```

*dijkstra/2* may now be defined by (P-2.10).

**Prolog Code P-2.10: A first definition of *dijkstra/2***

```

1 dijkstra(Items,Grouped) :- reds(Items,R),
2                               whites(Items,W),
3                               blues(Items,B),
4                               append(R,W,RandW),
5                               append(RandW,B,Grouped).

```

### 2.4.2 A More Concise Version

The predicates *reds/2*, *whites/2* and *blues/2* from Sect. 2.4.1 are *structurally* identical; their structure is captured by that of *colour/3* in (P-2.11).

**Prolog Code P-2.11: Definition of *colour/3***

```

1 colour(_, [], []).
2 colour(Clr,[col(Object,Clr)|T],[col(Object,Clr)|L]) :- colour(Clr,T,L).
3 colour(Clr,[col(_,Colour)|T],L) :- Colour \= Clr,
4                               colour(Clr,T,L).

```

It is clear that once the first argument of *colour/3* is instantiated to a particular colour, it will behave as the predicate for the corresponding colour; for example,

```
?- items(_Items), colour(red,_Items,Reds).
```

```
Reds = [col(tomato, red), col(blood, red), col(cherry, red)]
```

This suggests a second implementation of *dijkstra/2*, shown in (P-2.12).

**Prolog Code P-2.12: A second definition of *dijkstra/2***

```
1 dijkstra(Items,Grouped) :- colour(red,Items,R),  
2                             colour(white,Items,W),  
3                             colour(blue,Items,B),  
4                             append(R,W,RandW),  
5                             append(RandW,B,Grouped).
```

### 2.4.3 Using Difference Lists

As *dijkstra/2* uses list concatenation by *append/3*, it is a candidate for being recast in terms of difference lists.

- First, we define *colour\_dl/3* in (P-2.13) by using difference lists.

**Prolog Code P-2.13: Definition of *colour\_dl/3***

```

1 colour_dl(_, [], L-L).
2 colour_dl(Clr, [col(Object,Clr)|T], [col(Object,Clr)|L1]-L2) :-
3     colour_dl(Clr, T, L1-L2).
4 colour_dl(Clr, [col(_,Colour)|T], L1-L2) :-
5     Colour \= Clr,
6     colour_dl(Clr, T, L1-L2).

```

- Then, we concatenate in (P-2.14) the three lists of groups by *dijkstra\_dl/2*.

**Prolog Code P-2.14: Definition of *dijkstra\_dl/2***

```

1 dijkstra_dl(Items, L1-L4) :- colour_dl(red, Items, L1-L2),
2                             colour_dl(white, Items, L2-L3),
3                             colour_dl(blue, Items, L3-L4).

```

- Finally, in (P-2.15) the grouped list *Grouped* (as a true list) is obtained by unifying the difference list with *Grouped-[]*.

**Prolog Code P-2.15: *dijkstra/2* based on difference lists**

```

1 dijkstra(Items, Grouped) :- dijkstra_dl(Items, Grouped-[]).

```

**Exercise 2.10.** All versions of *dijkstra/2* discussed thus far need three passes through the input list, one for each colour. This inefficiency is avoided by the version defined by (P-2.16)–(P-2.17).

**Prolog Code P-2.16: Definition of *colour/4***

```

1 colour([], [], [], []).
2 colour([col(Object,red)|T], [col(Object,red)|R], W, B) :- colour(T, R, W, B).
3 colour([col(Object,white)|T], R, [col(Object,white)|W], B) :- colour(T, R, W, B).
4 colour([col(Object,blue)|T], R, W, [col(Object,blue)|B]) :- colour(T, R, W, B).

```

**Prolog Code P-2.17: *dijkstra/2* based on *colour/4***

```

1 dijkstra(Items, Grouped) :- colour(Items, R, W, B),
2                             append(R, W, RandW),
3                             append(RandW, B, Grouped).

```

(*colour/4* features as an ‘amalgamation’ of the predicates *reds/2*, *whites/2* and *blues/2* from Sect. 2.4.1.)

- Rewrite *colour/4* and *dijkstra/2* (from (P-2.17)) by using difference lists. Compare the performance of all versions of *dijkstra/2* available thus far by using *time/1*.
- The version of *dijkstra/2* from (P-2.17) as well as its difference list based version from (a) will fail if one of the entries in *Items* is not coloured red, white or blue. Augment both predicates to avoid failure for such inputs. (As before, *Grouped* should comprise exactly the items in the Dutch national colours.)

■

## 2.5 Rotations

### 2.5.1 Rotating a List

Sometimes it is required to create a new (output) list by *rotating* some input list. We have met an example thereof in Sect. 1.6 where in the course of the Perceptron Training Algorithm, the predicate *transform/2*, defined in (P-1.15), p. 33, subjected some list of *Ps* to a rotation. This meant that if  $[P/OtherPs]$  is unified with the list of training points  $[p_1, p_2, \dots, p_N]$ , say, then *transform/2* will return in *NewPs* the 'rotated' list  $[p_2, \dots, p_N, p_1]$ . (The list of desired class labels  $[D/OtherDs]$  is subjected by *transform/2* to the same transformation.)

In (P-1.15), rotation was achieved by using *append/3*. Difference lists offer a constant-time alternative to accomplish the same (e.g. [1]) if the original list is a difference list; example:

```
?- [a1, a2, a3, a4/X]-X = [H/Y]-[H/Z], R = Y-Z.
```

```
X = [a1|_G397]
```

```
H = a1
```

```
Y = [a2, a3, a4, a1|_G397]
```

```
Z = _G397
```

```
R = [a2, a3, a4, a1|_G397]-_G397
```

Fig. 2.7 spells out how the above result can be modelled in terms of differences of lists.

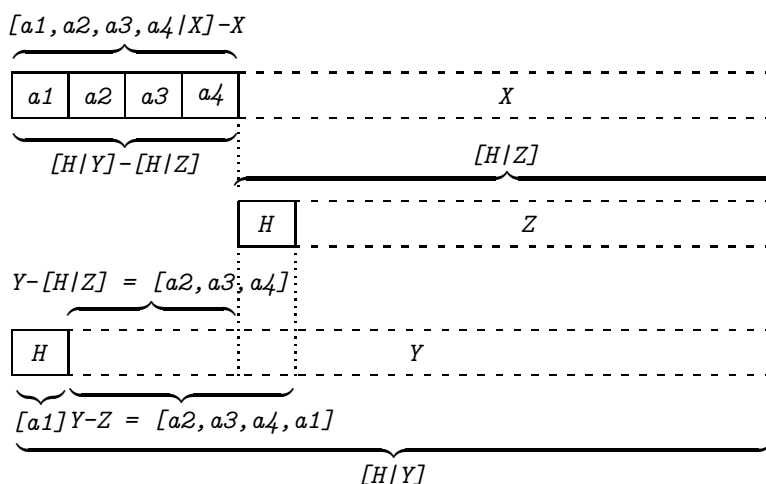


Figure 2.7: Rotating by Difference Lists

This idea easily carries over to more sophisticated schemes of computation where the result is based on some input from the 'front' being transformed and placed to the 'back'. For example, the core for computing the averages of consecutive entries in a list of numbers may look like this:

```
?- [1,2,3,4/X]-X = [H1,H2/Y]-[Last/Z], Last is (H1 + H2)/2,
```

```
R = [H2/Y]-Z.
```

```
...
```

```
R = [2, 3, 4, 1.5|_G574]-_G574
```

**Exercise 2.11.** Based on the above query, define *averages\_dl(+DL,-ADL)* for computing the pairwise averages of adjacent numbers in a list of *positive* integers. Both, *DL* and *ADL* are represented in the difference list format. Example:

```
?- averages_dl([4,8,16,32|_X]-_X,ADL).
ADL = [6, 12, 24|_G426]-_G426 ;
No
```

*Outline Idea.* A version based on *ordinary* lists is shown in (P-2.18).

**Prolog Code P-2.18: Definition of *averages/2***

```
1 averages(L,A) :- aver([-1,1|L],A), !.           % clause 1
2 aver([_,0,_|T],T).                             % clause 2
3 aver(X,Result) :- av_rotate(X,Y),              % clause 3
4                  aver(Y,Result).                %
5 av_rotate([H1,H2|Y],L) :- Last is (H1 + H2)/2, % clause 4
6                  append([H2|Y], [Last],L). %
```

The auxiliary predicate *av\_rotate/2* is the ordinary list based version of the ‘compute-the-average-and-rotate’ function. Let us show an example of how *averages/2* will behave:

```
?- averages([4,8,16,32],A).
A = [6, 12, 24] ;
No
```

It is seen that the list for which the averages are to be computed is first appended to  $[-1,1]$ . This augmented list is then transformed by repeated application of *av\_rotate/2* (via a recursive call to *aver/2*) until the zero (i.e. the average of the first two entries) moves to the second position. The final result is then obtained by removing the first three entries of the list thus returned. (See also the hand computations in Fig. 2.8.) Rewrite the above definition in terms of difference lists.

```
averages([4,8,16,32], A)  $\xrightarrow{\textcircled{1}}$ 
aver([-1,1,4,8,16,32], A)  $\xrightarrow{\textcircled{3}}$  aver([1,4,8,16,32,0], A)  $\xrightarrow{\textcircled{3}}$ 
aver([4,8,16,32,0,2.5], A)  $\xrightarrow{\textcircled{3}}$  aver([8,16,32,0,2.5,6], A)  $\xrightarrow{\textcircled{3}}$ 
aver([16,32,0,2.5,6,12], A)  $\xrightarrow{\textcircled{3}}$  aver([32,0,2.5,6,12,24], A)  $\xrightarrow{\textcircled{3}}$ 
aver([32,0,2.5|6,12,24], A)  $\xrightarrow{\textcircled{2}}$  A = [6,12,24]  $\xrightarrow{\textcircled{1}}$  success
```

Figure 2.8: Hand Computations for *averages/2*

*Notes.*

- ❶ We may use this definition to implement afresh the averaging of *ordinary* lists of positive integers. We do this by first converting the original list to a difference list by *dl/2*, defined in (P-2.19).

**Prolog Code P-2.19: *dl/2* for list to difference list**

```

1 dl([],L-L). % clause 1
2 dl([H|T],[H|L1]-L2) :- dl(T,L1-L2). % clause 2

```

Then, the list of averages may be computed thus.

```

?- dl([4,8,16,32],_DL), averages_dl(_DL,A-[]).
A = [6, 12, 24] ;
No

```

- ❷ The difference list based version is faster than the one using *append/3*. Faster still is the predicate defined by simple recursion in (P-2.20).

**Prolog Code P-2.20: averages/2 by recursion**

```

1 averages2([_], []).
2 averages2([H1,H2|T],[A|AS]) :- A is (H1 + H2) / 2,
3                               averages2([H2|T],AS).

```

**Exercise 2.12.** Give a pictorial illustration of clause 2 of *dl/2* in (P-2.19). Based on this illustration, give it a declarative reading.

The term ‘rotation’ is justified by the following consideration. We imagine the list entries to be labels to movable beads threaded onto a circular wire. Our ‘rotation’ corresponds to each bead moving one position to the left. The crucial step here is the identification (or ‘glueing together’) of both ends of the list. (See Fig. 2.9.)

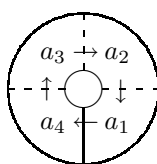


Figure 2.9: Rotating a List with Four Entries

## 2.5.2 The Perceptron Training Algorithm Revisited

As indicated before, there is scope for improving the Prolog implementation of the Perceptron Training Algorithm from Sect. 1.6 by using difference lists. Carrying out the two rotations via difference lists, we now have a new clause of *transform/2* in (P-2.21).<sup>13</sup>

**Prolog Code P-2.21: An additional clause for transform/2**

```

1 transform(in(C,[P|TP1]-[P|TP2],[D|TD1]-[D|TD2],Ws,Acc),
2           in(C,TP1-TP2,TD1-TD2,NewWs,NewAcc)) :-
3   perceptron(C,P,D,Ws,NewWs),
4   NewAcc is Acc + 1.

```

The stopping criterion, originally implemented by *classify\_all/3* in (P-1.14), p. 33, is also rewritten to accommodate difference lists; this is in (P-2.22).

**Prolog Code P-2.22: Additional clauses for classify/3**

```

1 classify_all(L-_,_,L1-L1) :- var(L).
2 classify_all([P|TP1]-TP2,Weights,[Class|TC1]-TC2) :-
3   classify(P,Weights,Class), !,
4   classify_all(TP1-TP2,Weights,TC1-TC2).

```

<sup>13</sup>See (P-1.15), p. 33, for the original definition of *transform/2*.



(P-2.21) and (P-2.22) are placed in the file where the earlier definitions are, as all previous definitions should still apply.<sup>14</sup> (The new clauses won't clash with existing definitions.) To convert the list of training points and the list of desired class labels to difference lists, we use the predicate *dl/2* from Exercise 2.11. With these additions then, we are now ready to run and confirm the computational advantage of the new version:<sup>15</sup>

```
?- ws(Ws), ps(_Ps), ds(_Ds), time(pta(0.25,_Ps,_Ds,Ws,W,801)).
% 41,335 inferences in 0.38 seconds (108776 Lips)
Ws = [-0.51, -0.35, 0.13] W = [3.018, 4.1935, -39.87]
?- ws(Ws), ps(_Ps), ds(_Ds), dl(_Ps,_PsDL), dl(_Ds,_DsDL),
   time(pta(0.25,_PsDL,_DsDL,Ws,W,801)).
% 28,519 inferences in 0.28 seconds (101854 Lips)
Ws = [-0.51, -0.35, 0.13] W = [3.018, 4.1935, -39.87]
```

(We have excluded from the timing the conversions to difference lists by *dl/2* as they present a constant computational overhead whose relative contributions will be negligible as the number of iterations is increased.)

### 2.5.3 Planar Rotations<sup>16</sup>

To extend the notion of ‘rotation’ from lists to matrices, we consider list rotations once again. One way to rotate the list  $L = [a_1, a_2, a_3, a_4]$  is indicated in Fig. 2.10:

1. Copy  $L$  infinitely many times along the line.
2. Shift the frame of  $L$  by one cell to the right. The framed entries form the rotated list.
3. Several successive rotations will be achieved by shifting the frame the requisite number of cells to the right.

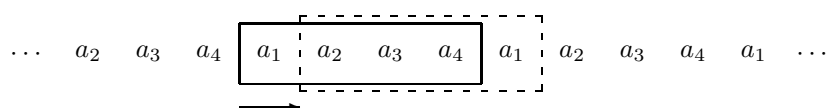


Figure 2.10: The Original List and its Rotated Image

We want to consider the analogous construction in the plane. A two-dimensional rectangular pattern (i.e. a *matrix*) of entries is given; this may be, for example, the three by four matrix

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \end{bmatrix} \quad (2.7)$$

We tile the entire plane with copies of  $\mathbf{A}$  and shift a three by four frame from  $\mathbf{A}$  to South-East to obtain the rotated matrix

$$\mathbf{A}^{(rot)} = \begin{bmatrix} a_{22} & a_{23} & a_{24} & a_{21} \\ a_{32} & a_{33} & a_{34} & a_{31} \\ a_{12} & a_{13} & a_{14} & a_{11} \end{bmatrix}$$

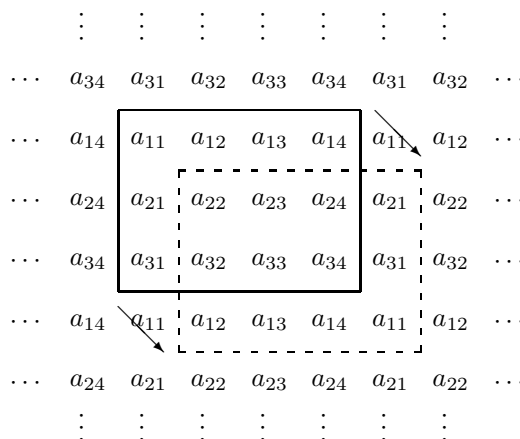


Figure 2.11: The Original Matrix  $\mathbf{A}$  and its Rotated Image  $\mathbf{A}^{(rot)}$

This is illustrated in Fig. 2.11. (Several such moves may be used for successive rotations.)

The argument to justify the term ‘rotation’ is now more involved. We first identify the two horizontal edges of the matrix and glue them together. The result is a tube which then is treated as a flexible pipe. Then, both ends of the pipe are glued together such that the first and last entries of each matrix row meet. What we then have is a *torus* covered with the mesh of the matrix entries. Our ‘rotation’ corresponds to each entry moving to its neighbouring North–Western cell.

## Implementation

Initially, a matrix will be represented as a list of its rows which themselves are written as lists. Therefore, for example, the matrix  $\mathbf{A}$  in (2.7) may be defined by (P-2.23).

**Prolog Code P-2.23: Definition of *matrix\_a/1***

```

1 matrix_a([[ a11, a12, a13, a14],
2           [ a21, a22, a23, a24],
3           [ a31, a32, a33, a34]]).
```

(This is then a list of lists of Prolog atoms.)

*Using Proper Lists.* Rotations will be carried out in two stages as indicated in Fig. 2.12. First, in step ①, the list representations of rows undergo a rotation each; this is implemented by *rot\_rows/2* in (P-2.24).

<sup>14</sup>All code pertinent to the Perceptron Training Algorithm is replicated in the file `d1.pl`.

<sup>15</sup>A similar result applies when calling *pta/6* with a *variable* in its last argument.

<sup>16</sup>This section and the next are based on [4]. The author thankfully acknowledges the permission by Elsevier to republish this material here.

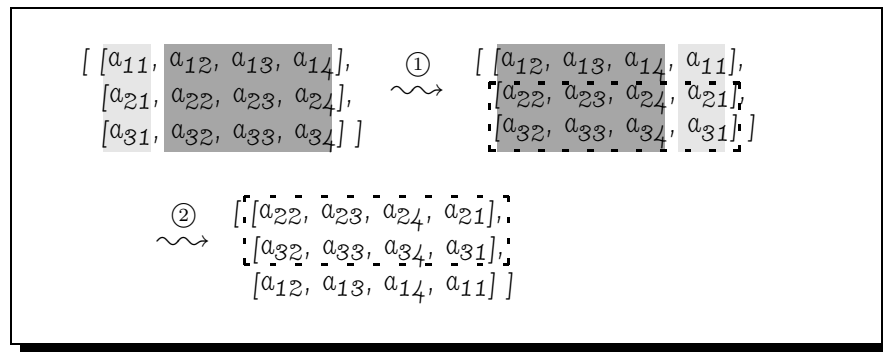


Figure 2.12: Hand Computations for Rotation in the Plane

**Prolog Code P-2.24: Definition of *rot\_rows/2***

```

1 rot_rows([], []). % clause 1
2 rot_rows([H/T|Ls], [R/Rs]) :- append(T, [H], R), !, % clause 2
3                               rot_rows(Ls, Rs). %

```

Then, in step ②, the ‘outside’ list is rotated by the predicate *rot\_matrix/2* in (P-2.25).

**Prolog Code P-2.25: Definition of *rot\_matrix/2***

```

1 rot_matrix(M, R) :- rot_rows(M, [H/T]), % clause 1
2                   append(T, [H], R). %

```

The timed rotation of **A** will look like this:

```

?- matrix_a(A), time(rot_matrix(A, R)).
% 20 inferences in 0.00 seconds (Infinite Lips)
A = [[a11,a12,a13,a14], [a21,a22,a23,a24], [a31,a32,a33,a34]]
R = [[a22,a23,a24,a21], [a32,a33,a34,a31], [a12,a13,a14,a11]]

```

*Using Difference Lists.* All lists will be replaced by difference lists; in particular, matrices are now difference lists of difference lists. We need a way of converting the old matrix representation to its new equivalent. This will be achieved by the predicate *dl2(+LOfLs, -DLOfDLs)* in (P-2.26).

**Prolog Code P-2.26: Definition of *dl2/2***

```

1 dl2([], L-L).
2 dl2([H/T], [HDL/L1]-L2) :- dl(H, HDL), !,
3                             dl2(T, L1-L2).

```

**Exercise 2.13.** Define a predicate *show\_matrix\_dl/1* for displaying the original matrix rows via the new difference list representation as shown below.

```

?- matrix_a(A), dl2(A, ADL), show_matrix_dl(ADL).
[a11, a12, a13, a14] [a21, a22, a23, a24] [a31, a32, a33, a34]

```

■

The new, difference lists based implementations (P-2.27) and (P-2.28) are obtained by a straightforward clause by clause ‘translation’ of (P-2.24) and (P-2.25), respectively.

**Prolog Code P-2.27: Definition of *rot\_rows\_dl/2***

```

1 rot_rows_dl(L-, Y-Y) :- var(L).
2 rot_rows_dl([H/T1]-[H/T2]|Ls1]-Ls2, [T1-T2|R1]-R2) :-
3   rot_rows_dl(Ls1-Ls2, R1-R2).

```

**Prolog Code P-2.28: Definition of *rot\_matrix\_dl/2***

```

1 rot_matrix_dl(MDL, T1-T2) :- rot_rows_dl(MDL, [H/T1]-[H/T2]).

```

The test below confirms the computational advantage of using difference lists.

```
?- matrix_a(A), dl2(A,DLA), time(rot_matrix_dl(DLA,DLR)),
    show_matrix_dl(DLR).
% 12 inferences in 0.00 seconds (Infinite Lips)
[a22, a23, a24, a21] [a32, a33, a34, a31] [a12, a13, a14, a11]
```

**Exercise 2.14.** Your predicate `show_matrix_dl/1` from Exercise 2.13 will in all likelihood interfere with predicates invoked *after* its call. You may find, for example, that you can't produce the rotated matrix after you have used `show_matrix_dl/1` for displaying the original matrix:

```
?- matrix_a(A), dl2(A,DLA), show_matrix_dl(DLA),
    rot_matrix_dl(DLA,DLR), show_matrix_dl(DLR).
[a11, a12, a13, a14] [a21, a22, a23, a24] [a31, a32, a33, a34]
No
```

What is the reason for this? Try to remedy the situation.

■

## 2.5.4 Application: The Gauss–Seidel Method

We want to solve *iteratively* the system of linear equations

$$u + \alpha v + \beta w = r \quad (2.8)$$

$$\gamma u + v + \delta w = s \quad (2.9)$$

$$\lambda u + \rho v + w = t \quad (2.10)$$

in the three unknowns  $u$ ,  $v$  and  $w$ . Given some initial approximate solutions  $u^{(0)}$ ,  $v^{(0)}$ ,  $w^{(0)}$ , we calculate a new value for  $u$  from (2.8) by

$$u^{(1)} = r - \alpha v^{(0)} - \beta w^{(0)} \quad (2.11)$$

This then is used with (2.9) to calculate a new value for  $v$ :

$$v^{(1)} = s - \gamma u^{(1)} - \delta w^{(0)} \quad (2.12)$$

Finally, an updated value for  $w$  is obtained by using  $u^{(1)}$ ,  $v^{(1)}$  in (2.10):

$$w^{(1)} = t - \lambda u^{(1)} - \rho v^{(1)} \quad (2.13)$$

We have thus completed one cycle of the iteration scheme known as the *Gauss–Seidel Method*<sup>17</sup> (e.g. [10], [17]).

In each updating step, one of the equations (2.11)–(2.13) is used to recompute the variable concerned. The following observations will be crucial.

- All three updating equations (2.11)–(2.13) take the form

$$x_1 = b_1 - a_{12}x_2 - a_{13}x_3 \quad (2.14)$$

if before each iteration step the system (2.8)–(2.10) is recast in matrix form as  $\mathbf{Ax} = \mathbf{b}$  where  $\mathbf{A}$ ,  $\mathbf{b}$  and  $\mathbf{x}$  are as shown in Table 2.1.

- In Table 2.1, each of the entries for  $\mathbf{A}$ ,  $\mathbf{b}$  and  $\mathbf{x}$  is obtained from the one above it by *rotation*.<sup>18</sup>

<i>Iterations</i>	<b>A</b>	<b>b</b>	<b>x</b>	<i>Updating ...</i>
1, 4, 7, ...	$\begin{bmatrix} 1 & \alpha & \beta \\ \gamma & 1 & \delta \\ \lambda & \rho & 1 \end{bmatrix}$	$\begin{bmatrix} r \\ s \\ t \end{bmatrix}$	$\begin{bmatrix} u \\ v \\ w \end{bmatrix}$	$u$
2, 5, 8, ...	$\begin{bmatrix} 1 & \delta & \gamma \\ \rho & 1 & \lambda \\ \alpha & \beta & 1 \end{bmatrix}$	$\begin{bmatrix} s \\ t \\ r \end{bmatrix}$	$\begin{bmatrix} v \\ w \\ u \end{bmatrix}$	$v$
3, 6, 9, ...	$\begin{bmatrix} 1 & \lambda & \rho \\ \beta & 1 & \alpha \\ \delta & \gamma & 1 \end{bmatrix}$	$\begin{bmatrix} t \\ r \\ s \end{bmatrix}$	$\begin{bmatrix} w \\ u \\ v \end{bmatrix}$	$w$

Table 2.1: Gauss–Seidel Iterations

The method and the above observations carry over to linear systems of any size. The  $n$ -dimensional analogue of (2.14) is

$$x_1 = b_1 - a_{12}x_2 - \dots - a_{1n}x_n \quad (2.15)$$

Equation (2.15) is the centrepiece in our formulation of the Gauss–Seidel algorithm and it is very easily implemented in Prolog. In fact, if **A**, **b** and **x** are respectively represented by  $[[First/Rest]/OtherRows]$ ,  $[B/OtherBs]$  and  $[X/OtherXs]$ , the code fragment implementing (2.15) will read

```
...
dot_product(Rest,OtherXs,P),
NewX is B - P,
...
```

where *dot\_product/3* defines the scalar product of two vectors (not shown here).

Algorithm 2.5.1 shows the pseudocode in the form ready for implementation in Prolog using the present formulation. (The output *Subscripts* indicates the permutation which the components of **x** have been put through and is the list of subscripts thereof.)

<sup>17</sup>The special feature of this iteration scheme is that updated values are used as soon as they become available.

<sup>18</sup>By observing the iteration numbers, row three is found to be ‘above’ row one.

**Algorithm 2.5.1:** GAUSS-SEIDEL( $\mathbf{A}, \mathbf{b}, \mathbf{x}, \mathbf{s}, i$ )

**comment:**  $\mathbf{A}$  is the  $n \times n$  coefficient matrix with unit diagonals.  
 $\mathbf{b}$  is the  $n$ -vector of r.h.s. constants.  
 $\mathbf{x}$  is the  $n$ -vector of guessed solutions.  
 $\mathbf{s}$  is the list of subscripts of the components of  $\mathbf{x}$ .  
 $i$  is the required number of iterations.

$Subscripts \leftarrow \mathbf{s}$

$Iterations \leftarrow i$

**while**  $Iterations \neq 0$

$\left\{ \begin{array}{l} \text{Update (the first entry of) } \mathbf{x} \text{ by (2.15)} \\ \mathbf{A} \leftarrow \text{ROTATEMATRIX}(\mathbf{A}) \\ \mathbf{b} \leftarrow \text{ROTATELIST}(\mathbf{b}) \\ \mathbf{x} \leftarrow \text{ROTATELIST}(\mathbf{x}) \\ Subscripts \leftarrow \text{ROTATELIST}(Subscripts) \\ Iterations \leftarrow Iterations - 1 \end{array} \right.$

**output**  $(\mathbf{x}, Subscripts)$

The core predicate in our implementation is *g\_seidel/2* with arguments *in/4* and *out/4*. It is defined in (P-2.29) and implements all but the last action specified inside the while loop in Algorithm 2.5.1.

**Prolog Code P-2.29:** Definition of *g\_seidel/2*

```

1 g_seidel(in([[First|Rest]|OtherRows],
2           [B|OtherBs], [_|OtherXs], [S|OtherSs]),
3           out(NewAs, NewBs, NewXs, NewSs)) :-
4   dot_product(Rest, OtherXs, P),
5   NewX is B - P,
6   rot_matrix([[First|Rest]|OtherRows], NewAs),
7   append(OtherBs, [B], NewBs),
8   append(OtherXs, [NewX], NewXs),
9   append(OtherSs, [S], NewSs).
```

*g\_seidel/2* is used by *g\_seidel/7*, the top level predicate defined in (P-2.30), to complete the requisite number of iterations.

**Prolog Code P-2.30:** Definition of *g\_seidel/7*

```

1 g_seidel(_, _, Xs, Ss, 0, Xs, Ss).
2 g_seidel(As, Bs, Xs, Ss, I, FinalXs, FinalSs) :-
3   g_seidel(in(As, Bs, Xs, Ss), out(NewAs, NewBs, NewXs, NewSs)),
4   NewI is I - 1, !,
5   g_seidel(NewAs, NewBs, NewXs, NewSs, NewI, FinalXs, FinalSs).
```

**Example 2.1.**<sup>19</sup> We want to solve the system  $\mathbf{Ax} = \mathbf{b}$  where

$$\mathbf{A} = \begin{bmatrix} 1 & -0.25 & -0.25 & 0 \\ -0.25 & 1 & 0 & -0.25 \\ -0.25 & 0 & 1 & -0.25 \\ 0 & -0.25 & -0.25 & 1 \end{bmatrix}, \mathbf{b} = \begin{bmatrix} 50 \\ 50 \\ 25 \\ 25 \end{bmatrix}.$$

The above system is defined by the Prolog facts

```
a([[ 1, -0.25, -0.25, 0],
   [-0.25, 1, 0, -0.25],
   [-0.25, 0, 1, -0.25],
   [ 0, -0.25, -0.25, 1]]).
```

and

```
b([50, 50, 25, 25]).
```

The initial approximate solution  $x_1^{(0)} = \dots = x_4^{(0)} = 100$  is defined in Prolog by

```
x0([100, 100, 100, 100]). s([1, 2, 3, 4]).
```

The exact solution,  $x_1 = x_2 = 87.5$ ,  $x_3 = x_4 = 62.5$ , is obtained after 50 iterations thus

---

<sup>19</sup>Source: [10].



```
?- a(A), b(B), x0(X), s(S), g_seidel(A,B,X,S,50,NewX,NewS).
A = [[1, -0.25, -0.25, 0], [-0.25, 1, 0, -0.25],
      [-0.25, 0, 1, -0.25], [0, -0.25, -0.25, 1]]
B = [50, 50, 25, 25]
X = [100, 100, 100, 100]
S = [1, 2, 3, 4]
NewX = [62.5, 62.5, 87.5, 87.5]
NewS = [3, 4, 1, 2]
```

■

**Exercise 2.15.** Re-implement Gauss-Seidel by using difference lists and compare the performances of the implementations. You should use the predicates *dl/2* (defined by (P-2.19) in Sect. 2.5.1) and *dl2/2* and *rot\_matrix\_dl/2* (defined respectively by (P-2.26) and (P-2.28) in Sect. 2.5.3).

■



## Chapter 3

# Program Manipulations

In Prolog, unlike in most other programming languages, there is no clear distinction between program code and data. In this chapter, we are going to demonstrate how this feature of Prolog can be made use of in practice. In Sect. 3.1 we discuss the built-in Prolog predicates for basic database maintenance work. In Sect. 3.2 we present a tool for automated program unfolding, a program transformation technique the ‘manual’ form of which we met in Sect. 2.3.1. Finally, in Sect. 3.3 we show how Prolog can be used to define a Prolog program some features of which are specified at runtime.

### 3.1 Simple Database Operations

In this section, we illustrate by a simple example how the Prolog database can be modified from within the Prolog system.

#### The Round Table

Six people are seated at a round table as shown in Fig. 3.1. The predicate *right\_to/2*, defined in (P-3.1) by six facts, describes the seating arrangement in an obvious fashion.

*Prolog Code P-3.1: Initial definition of right\_to/2*

```
1 right_to(martin,lisa).    right_to(lisa,george).  
2 right_to(george,clara).  right_to(clara,adam).  
3 right_to(adam,susan).    right_to(susan,martin).
```

**Exercise 3.1.** Write queries to answer the following questions:

- (a) Who is seated to the right of Adam?
- (b) To whom is Clara the right neighbour?
- (c) Who are the neighbours of George?

Define Prolog rules for

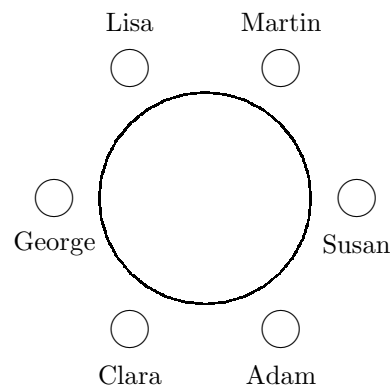


Figure 3.1: The Initial Seating Arrangement

- (d) "... is seated to the left of ..."
- (e) "... are the neighbours of ..."
- (f) "... is seated opposite to ..."

*Hints.* The envisaged solution for this exercise is elementary and concise and should make no use of lists. The following is suggested for solving part (f):

- If we want to find the person seated opposite to Adam, say, it will help to imagine that the party are seated not at a round table but at a long *rectangular* one at the head of which is seated Adam (Fig. 3.2).
- Define an auxiliary predicate *facing/3* returning all pairs of people facing each other from one particular person's point of view (here: Adam's), and, eventually, facing that person himself. *facing/3* should respond as follows.

```
?- facing(adam,Left,Right).
Left = clara   Right = susan ;
Left = george  Right = martin ;
Left = lisa    Right = lisa ;
No
```

- Now implement *opposite\_to/2* using *facing/3*.
- *opposite\_to/2* should fail if the number of people around the table is odd.

■

**Exercise 3.2.** Further useful predicates may be defined for the Round Table example.

- (a) Write a predicate *guests/0* for displaying the names of all those at the table. (Use a failure driven loop; see inset on p. 77.) *guests/0* should fail only if there aren't any people at the table.

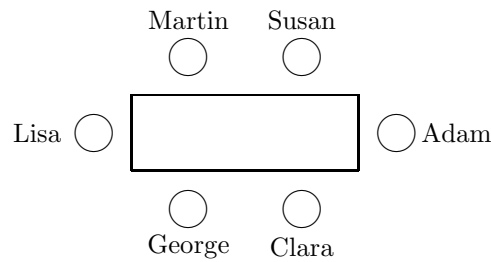


Figure 3.2: Rectangular Table

---

**Built-in Predicates: *fail/0* and *true/0***

*fail/0* always fails. *true/0* always succeeds. *Failure driven loops* may be defined by *fail/0*. Example:

```
?- right_to(_X,_), write(_X), write(' '), fail; true.
martin lisa george clara adam susan
Yes
```

---

- (b) Use a failure driven loop to define a predicate *opposites/0* for displaying all pairs seated opposite each other:

```
?- opposites.
martin, clara
lisa, adam
george, susan
adam, lisa
susan, george
clara, martin
Yes
?- joins(fred, clara, adam).1
fred has joined the table.
Yes
?- opposites.
No
```

- (c) Use the accumulator technique to define a predicate *look\_right(+Person)* for displaying all the guests' names counterclockwise, starting with a particular person. Example:

```
?- look_right(george).
```

---

<sup>1</sup>See Sects. 3.1.3 and 3.1.4 for how to implement *joins/3*. Here it is used only to indicate that *opposites/0* should fail for an odd number of guests in the database.

```
george clara adam susan martin lisa  
Yes
```



### Departures and Arrivals

Initially, we will have read the facts in (P-3.1), p. 75, into memory by *consult/1* (or by some equivalent thereof). It is important at this stage to remember that the *database* comprises all predicates loaded in memory; these will be those defined by the user as well as the built-in ones. Let us now assume that we want to model the departure from, and the arrival to, the table of people by updating the database.

*Departures.* Departures will obviously involve removal of clauses from the database. To model, for example, George's departure, we shall have to remove all facts referencing George. In addition, former neighbours of George will now be seated next to each other, necessitating additions to the database. Thus, to record departures, we shall need both deletion from, and addition to, the database.

*Arrivals.* Arrivals will clearly involve an augmentation of the definition of *right\_to/2* by new facts. To model for example the arrival of Tracy and Joe, to be seated between Adam and Susan, we will have to add the three facts in (P-3.2) to the database.

**Prolog Code P-3.2: New facts for *right\_to/2***

```
1 right_to(adam,tracy). right_to(tracy,joe). right_to(joe,susan).
```

And, we will have to remove the fact indicating that Susan is Adam's right-hand neighbour:

```
right_to(adam,susan).
```

Therefore, to account for arrivals, both deletion from, and addition to the database will need to be done.

### 3.1.1 Basic Database Manipulation

We now review a few basic built-in predicates for modifying the database.

- We use *retract/1* (or *retractall/1*) to remove a clause (or all clauses of a predicate) from the database. The predicate whose clause is *retracted* has to be declared *dynamic*, implemented either as a directive in one of the source files or by calling *dynamic/1* as a goal just before *retracting*. This is achieved in our example either by including in one of the files consulted the directive

```
:- dynamic(right_to/2).
```

or interactively by

```
?- dynamic(right_to/2), retract(right_to(X,Y)).
```

---

**Built-in Predicate: *retract(+Term)***

Removes from the database the *first* clause unifying with *Term*. Example:

```
?- listing(right_to(X,Y)).
right_to(martin, lisa).
right_to(lisa, george).
...
?- retract(right_to(_,_)).
Yes
?- listing(right_to(X,Y)).
right_to(lisa, george).
...
```

---

- We use *assert/1* to add a new clause to the database.

---

**Built-in Predicate: *assert(+Term)***

Adds to the database the clause in *Term*. Example: a possible (reasonable) definition by *assert/2* of a predicate *near/2* for the Round Table example may be achieved by<sup>2</sup>

```
?- assert(near(X,Y) :- (right_to(X,Z), right_to(Z,Y))).
?- assert(near(Y,X) :- (right_to(X,Z), right_to(Z,Y))).
```

Notice that, as shown above, the conjunctive body of the clause *asserted* should be written in parenthesis.

---

A predicate newly introduced by *assert/1* is deemed *dynamic*. An existing static (i.e. non-dynamic) predicate may be augmented by a new clause via *assert/1* only after declaring it *dynamic*.

- *retractall/1* is used to remove from the database *all* clauses whose head unifies with the pattern in its argument. As with *retract/1*, *retractall/1* may revoke dynamic predicates only.

---

**Built-in Predicate: *retractall(+Term)***

Removes from the database *all* clauses whose *head* unifies with *Term*. For example, both clauses of the predicate *near/2* *asserted* earlier may be removed in a single step by

```
?- retractall(near(_,_)).
Yes
```

---

### 3.1.2 Changing the Database

The following queries may be used to achieve the intended changes to the database.

- George leaves the table (Fig. 3.3).

```
?- dynamic(right_to/2), right_to(X,george),
   right_to(george,Y), assert(right_to(X,Y)),
   retract(right_to(X,george)), retract(right_to(george,Y)).
X = lisa
Y = clara
Yes
```

As is easily confirmed by the query *?- listing(right\_to/2).*, the predicate *right\_to/2* is now defined in the database by the facts in (P-3.3).

---

<sup>2</sup>An *interactive* definition of a clause by *assert/1* has the same effect as defining the same clause via *consult(user)* except that in the latter case a newly defined predicate is static.



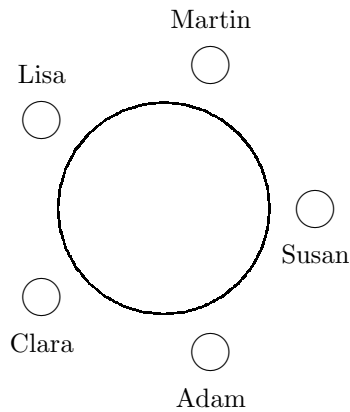


Figure 3.3: After George's Departure

**Prolog Code P-3.3: Updated definition of *right\_to/2***

```

1 right_to(martin, lisa).  right_to(clara, adam).
2 right_to(adam, susan).  right_to(susan, martin).
3 right_to(lisa, clara).

```

(Notice, however, that the definition of *right\_to/2* in its Prolog source file is not yet affected.)

- Tracy and Joe join the table and are seated between Adam and Susan (Fig. 3.4).

```

?- right_to(adam,X), retract(right_to(adam,X)),
   assert(right_to(adam,tracy)), assert(right_to(tracy,joe)),
   assert(right_to(joe,X)).
X = susan
Yes

```

Notice that due to the previous query the predicate *right\_to/2* is now dynamic. It is now defined in the database by the facts in (P-3.4).<sup>3</sup>

**Prolog Code P-3.4: Final definition of *right\_to/2***

```

1 right_to(martin, lisa).  right_to(clara, adam).
2 right_to(susan, martin). right_to(lisa, clara).
3 right_to(adam, tracy).  right_to(tracy, joe).
4 right_to(joe, susan).

```

It is seen that *assert/1* places the new clause *behind* the existing ones for the same predicate.<sup>4</sup>

<sup>3</sup>As before, we may confirm this by the query `?- listing(right_to/2).`

<sup>4</sup>The related predicate *asserta/1* (not used here) behaves exactly as *assert/1* except that it places the new clause *in front of* all existing ones for the same predicate.

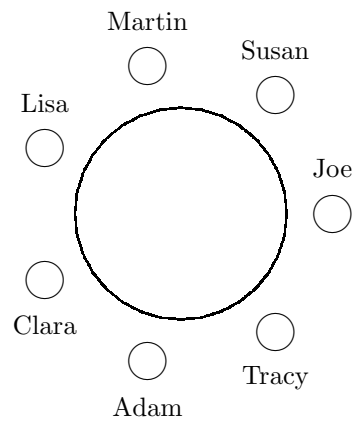


Figure 3.4: After Tracy's and Joe's arrival

**Exercise 3.3.** Thus far, we have carried out (for reasons of transparency) database changes *interactively* only. In this exercise, you are asked to define some predicates for manipulating the database.

- (a) Define a predicate *swap\_neighbours(+Left,+Right)* for recording in the database of two neighbours swapping places. (For this predicate to succeed, prior to the swap, the person named in *Left* should be seated to the left of the person named in *Right*.) If we assume, for example, that the seating arrangement is initially as shown in Fig. 3.1, then the swap of Clara and Adam will be accomplished by

```
?- swap_neighbours(clara,adam).
Yes
```

After this, the database will look as follows.

```
right_to(martin, lisa).  right_to(lisa, george).
right_to(susan, martin). right_to(adam, clara).
right_to(george, adam). right_to(clara, susan).
```

- (b) Define a predicate *swap(+Person1,+Person2)* for recording in the database of two people swapping places who need not be neighbours. To exemplify, assume again that the database is initially as shown in Fig. 3.1. Then, Adam and George's swap is carried out by

```
?- swap(adam,george).
Yes
```

upon which the database is as shown below.

```
right_to(martin, lisa).  right_to(susan, martin).
right_to(adam, clara).  right_to(lisa, adam).
right_to(george, susan). right_to(clara, george).
```

*Note.* You may use the predicate *swap\_neighbours/2* from part (a) in your definition of *swap/2*.

■

**Exercise 3.4.** (*Modelling a queue*)<sup>5</sup> A queue with at least two customers at a checkout is modelled by the Prolog predicate *behind/2* which is defined in the file `queue.pl` as shown below. (*behind/2* is declared a *dynamic* predicate in `queue.pl`.)

```
behind(lisa,george).  behind(george,clara).  behind(clara,adam).
behind(adam,susan).  behind(susan,peter).
```

(These facts have an obvious interpretation: the person named in the second argument stands *behind* the person named in the first argument.)

- (a) Define a predicate *swap\_neighbours(+Person1,+Person2)* for recording in the database of two *neighbours* swapping places. (For this predicate to succeed, prior to the swap, the person named in *Person2* should be standing *behind* the person named in *Person1*.) Example:

---

<sup>5</sup>The ideas involved here will be similar to those in Exercise 3.3 but now we have also to identify the first and the last person in the queue.

```
?- swap_neighbours(clara,adam).
Yes
```

After this query, the database will look as follows. After this, the database will look as follows.

```
behind(lisa,george).  behind(george,adam).  behind(adam,clara).
behind(clara,susan).  behind(susan,peter).
```

*Hint.* You should define *swap\_neighbours(+Person1,+Person2)* by four clauses, each of them covering one of the cases indicated in the Table 3.1 where the two questions concerned are defined by

1. Is *Person1* the *first* person in the queue? (Yes/No)
2. Is *Person2* the *last* person in the queue? (Yes/No)

'Yes' to 1	and	'Yes' to 2	'Yes' to 1	and	'No' to 2
'No' to 1	and	'Yes' to 2	'No' to 1	and	'No' to 2

Table 3.1: Cases for *swap\_neighbours/2*

- (b) (*Queue jumping*) Using *swap\_neighbours/2*, now define *by recursion* a predicate *to\_front(+P)* for recording in the database of person *P* moving to the front of the queue. Example:

```
?- to_front(adam).
Yes
```

After this query, the database will look as follows.

```
behind(adam,lisa).  behind(lisa,george).  behind(george,clara).
behind(clara,susan).  behind(susan,peter).
```

- (c) Define *by recursion* a predicate *before(+Person1,?Person2)* for finding the names of all those who will be served before *Person1*. On backtracking, *Person2* should be unified with the names of all those to be served before *Person1*. For example, assuming that the database is as given initially, we should find the names of all customers to be served before Adam by the query:

```
?- before(adam,P).
P = clara ; P = george ; P = lisa ;
No
```

You will find the solution of this exercise in `queue.pl`.

■

**Exercise 3.5.** The predicate *lives\_in/2* is defined by (P-3.5).

**Prolog Code P-3.5: Initial definition of *lives\_in/2***

```

1 lives_in(london, paul).      lives_in(birmingham, adam).
2 lives_in(leeds, susan).     lives_in(york, george).
3 lives_in(london, tracy).    lives_in(birmingham, david).
4 lives_in(york, peter).      lives_in(york, jane).
5 lives_in(leeds, joe).        lives_in(london, jack).

```

They form part of an employer's database concerning employees' locations. Let us now assume that the London branch and all its employees move to York due to relocation. Write a query which will change the Prolog database accordingly. After issuing the query, *lives\_in/2* is defined by (P-3.6).

**Prolog Code P-3.6: Final definition of *lives\_in/2***

```

1 lives_in(birmingham, adam). lives_in(leeds, susan).
2 lives_in(york, george).      lives_in(birmingham, david).
3 lives_in(york, peter).       lives_in(york, jane).
4 lives_in(leeds, joe).         lives_in(york, paul).
5 lives_in(york, tracy).        lives_in(york, jack).

```

### 3.1.3 File Modifications

We may want to modify clauses in the Prolog source file(s) as a *permanent* record of the changes in the database. With a view to doing this, we have distributed the Prolog source code to three separate files as shown in Fig. 3.5. It is seen that the Prolog source proper (in *party.pl*) is separated from what could

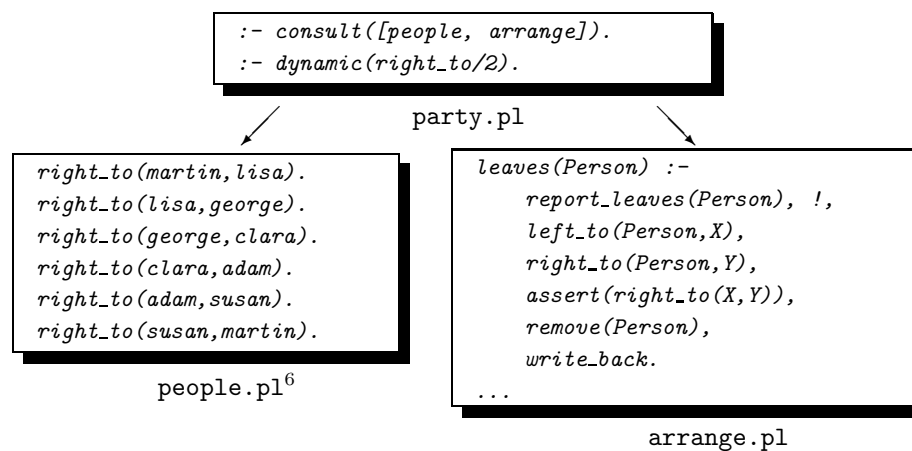


Figure 3.5: File Organization for the Round Table Example

be considered the input data (in *people.pl*). We hasten to add, though, that this separation is not necessary

<sup>6</sup>This is the *initial* state of *people.pl*. By the end of the Prolog session it will have changed to its updated version, Fig. 3.6.

since, as said earlier, Prolog does not distinguish between ‘program’ and ‘data’. Separation of program and data will prove expedient, however, since predicates whose definition is kept separate from the rest of the source code are easier to manipulate. The masterfile `party.pl` comprises a mere two directives: the first one causes the other two files to be *consulted* while the second one indicates that *right-to/2* is a dynamic predicate.

How shall we conclude the interactive session in Sect. 3.1.2 to make the changes in the database also to be mirrored in the file `people.pl`? To do this, we issue the query

```
?- tell('people.pl'), listing(right-to/2), told.  
Yes
```

after which `people.pl` will be as shown in Fig. 3.6.

To understand the above query, we note that

- *listing/1* uses the current output stream.
- At the beginning of an interactive session, the current output stream is the screen.
- The current output stream can be directed to a file by using the built-in predicate *tell(+Filename)*.
- The current output stream can be redirected to the screen by the predicate *told/0*.

```

right_to(martin, lisa).    right_to(clara, adam).    right_to(susan, martin).
right_to(lisa, clara).    right_to(adam, tracy).    right_to(tracy, joe).
right_to(joe, susan).

```

Figure 3.6: The File `people.pl` after the Interactive Session

- If an existing file is used in the argument of `tell/1`, it will be overwritten. Therefore, to avoid accidental loss of Prolog source code, program and dynamic data are best kept in separate files.

### 3.1.4 Updating `right_to/2` and `people.pl`

The work done interactively before (database and file changes), is more conveniently performed by some dedicated predicates `leaves/1` and `joins/3`. Their definition parallels the respective interactive session and can be found in the file `arrange.pl`.

**Exercise 3.6.** `joins/3` in `arrange.pl` does not allow for a guest to join the empty table. Define `join/1` to make this possible. Example:

```

?- guests.
No
?- joins(fred).
fred has joined the table.
Yes
?- guests.
fred
Yes

```

■

### 3.1.5 Automated Saving of Selected Predicates

We may wish to save to a file all (or some) predicate definitions loaded in memory. This is easily accomplished in a piecemeal fashion as indicated in Sect. 3.1.3. Such a ‘manual’ approach is, however, tedious and therefore an automated solution is called for. `save_predicates_to(+Filename, +Choice)`, to be studied below, is designed to do this task.

The collection of all predicates in memory at any given time comprises

- those explicitly loaded by `consult/1` (or by one of its equivalents),
- some built-in predicates depending on prior usage in the same session.

We are interested here in the first group, the user-defined predicates. The predicate `my_predicate(?Functor/?Arity, ?ClauseCount)` will name each of them with the respective number of clauses in `ClauseCount`:

```

?- my_predicate(Pred, ClauseCount).
Pred = my_predicate/2
ClauseCount = 1 ;
Pred = opposites/0

```

```

ClauseCount = 1 ;
Pred = right_to/2
ClauseCount = 6 ;
...

```

*my\_predicate/2* will serve as an auxiliary for *save\_predicates\_to/2* and it is defined in (P-3.7).

**Prolog Code P-3.7: Definition of *my\_predicate/2***

```

1 my_predicate(Fun/Arity,ClauseCount) :-
2     current_predicate(Fun,Head),
3     not(predicate_property(Head,built_in)),
4     not(predicate_property(Head,imported_from(_))),
5     not(predicate_property(Head,foreign)),
6     predicate_property(Head,number_of_clauses(ClauseCount)),
7     functor(Head,Fun,Arity).

```

The built-in predicates *current\_predicate/2*, *predicate\_property/2* and *functor/3* are used in this largely self-documenting definition.<sup>7</sup> The goals 2–4 in the body of *my\_predicate/2* are designed to filter out names of predicates which are not user-defined.

Embedding *my\_predicate/2* into a failure driven loop (see p. 77) gives rise to (P-3.8), the first clause of *save\_predicates\_to/2*.

**Prolog Code P-3.8: First clause of *save\_predicates\_to/2***

```

1 save_predicates_to(Filename,all) :- tell(Filename),
2                                     ((my_predicate(Fun/Arity,_),
3                                         Fun \= 'my_predicate',
4                                         Fun \= 'save_predicates_to',
5                                         listing(Fun/Arity),
6                                         fail); true),
7                                     told.

```

It will write to the specified file *all* user-defined predicates except its own and its auxiliary's definition.<sup>8</sup> Example: After the query

```
?- save_predicates_to('committee.pl',all).
```

the file *committee.pl* will be as indicated in Fig. 3.7. This copy of the database will be inferior to the original source because of

- (1) User-defined (usually *mnemonic*) variable names will be replaced by system-assigned ones (due to *listing/1*), making the code less readable.
- (2) Clause layout may be lost.
- (3) Comments will be lost.
- (4) The order of the predicates may be different.

<sup>7</sup>*functor/3* is known from Sect. 2.2.1. Consult the SWI-manual [18] for detailed information on the other two predicates.

<sup>8</sup>This is a sensible design decision since these two definitions won't usually be relevant to the broader context.



```

opposites :- right_to(A, B),
             opposite_to(A, C),
             write(A),
             write(', '),
             write(C),
             nl,
             fail.

right_to(martin, lisa).
right_to(lisa, george).
...

```

Figure 3.7: The File `committee.pl`

(5) Directives will be lost.

While the first four of these shortcomings could be tolerated, there will be some manual work needed to rectify the last one.

Another clause of `save_predicates_to(+Filename,+Choice)` will define the case when *Choice* unifies with a *list* of entries of the form *Functor/Arity*; for example, upon the query

```
?- save_predicates_to('committee.pl',[remove/1,left_to/2]).
```

the file `committee.pl` should comprise the definitions of the specified predicates `remove/1` and `left_to/2` (Fig. 3.8). We define the second clause of `save_predicates_to/2` in (P-3.9) along the lines of (P-3.8) except

```

remove(A) :- retract(right_to(A, B)),
              retract(right_to(C, A)).

left_to(A, B) :- right_to(B, A).

```

Figure 3.8: The File `committee.pl`

for the additional filtering with the built-in predicate `member/2`.

**Prolog Code P-3.9: Second clause of `save_predicates_to/2`**

```

1 save_predicates_to(Filename,List) :- tell(Filename),
2                                     ((my_predicate(Fun/Arity,_),
3                                         member(Fun/Arity,List),
4                                         listing(Fun/Arity),
5                                         fail); true),
6                                     told.

```

---

**Built-in Predicate:** *member(?Elem,?List)*

Succeeds when *Elem* unifies with one of the elements of *List*. Example:

```
?- member(penguin,[sparrow,stork,magpie]).  
No  
?- member(Bird,[sparrow,stork,magpie]).  
Bird = sparrow ;  
Bird = stork  
Yes
```

---

**Exercise 3.7.** The above version of *save\_predicates\_to/2* will silently skip all entries in the list argument which do not refer to a predicate in the database. An improved version will recognize this and return an error message:

```
?- save_predicates_to('committee.pl',[remove/1,left_to/3]).  
Error: some predicates not in the database  
No
```

(This shows that there is no predicate *left\_to/3* in the database.) Define such an enhanced version of *save\_predicates\_to/2*. It should not write anything to the file unless all list entries refer to existing user-defined predicates. *Hint*. A rather concise solution is possible by using the built-in predicate *->/2*.<sup>9</sup>




---

#### Built-in Predicate: *->/2*

The predicate *->/2* (written in the operator form) is used to define the conditional statement. Syntax: *(+Condition -> +Action ; +Alternative\_Action)*. A property buyer's example:

```
?- member(Capital,[1,4,10]),
   ((member(Mortgage,[1,2,5]),Capital + Mortgage < 9) ->
    (Capital + Mortgage > 4,member(Property,[cottage,house]));
    member(Property,[mansion,villa])).
Capital = 4 Mortgage = 1 Property = cottage ;
Capital = 4 Mortgage = 1 Property = house ;
Capital = 10 Mortgage = _G1170 Property = mansion ;
Capital = 10 Mortgage = _G1170 Property = villa ;
No
```

Notice in particular that

- *->/2* fails if *Condition* succeeds and *Action* fails (*Capital* = 1).
  - Once *Condition* succeeds it won't be re-satisfied on backtracking. (No move from *Mortgage* = 1 to *Mortgage* = 2 when *Capital* = 4.)
  - *->/2* succeeds if *Condition* fails and *Alternative\_Action* can be proved (*Capital* = 10).
- 

### 3.1.6 Miniproject: Modelling a Stamp Collection

The solutions of the exercises in this section are in the source file *stamps.pl* save for Exercise 3.9 which is solved in Appendix A.3.

A stamp collection is modelled by the predicate *album/1* in (P-3.10).

---

<sup>9</sup>This corresponds to the *if-then-else* language construct familiar from imperative programming. (Observe though the Prolog-specific subtleties as exemplified in the inset.)

**Prolog Code P-3.10: Facts defining *album/1***

```

1 album([stamp('Britain','Queen',1965,20),
2         stamp('Britain','Queen',1967,50),
3         stamp('Britain','Queen',1963,120)]).
4 album([stamp('Britain','Poets',1978,19),
5         stamp('Britain','Poets',1979,20),
6         stamp('Britain','Poets',1978,22),
7         stamp('Britain','Poets',1977,40),
8         stamp('Britain','Poets',1978,100)]).
9 album([stamp('Germany','Kaiser',1882,5),
10        stamp('Germany','Kaiser',1879,20),
11        stamp('Germany','Kaiser',1885,50)]).
12 album([stamp('Germany','Castles',1885,10),
13        stamp('Germany','Castles',1879,50),
14        stamp('Germany','Castles',1885,60)]).

```

The arguments in *stamp/4* refer respectively to: country of origin, the set's name, year of issue, denomination. Within a set, the stamps are in ascending order of denomination.

**Exercise 3.8.** (*Pattern matching*) Define a predicate *collection/1* for displaying on the terminal all stamps conforming to a certain criterion. Examples:

- Show all stamps with denomination 50.

```

?- collection(stamp(_,_,_ ,50)).
stamp(Britain, Queen, 1967, 50)
stamp(Germany, Kaiser, 1885, 50)
stamp(Germany, Castles, 1879, 50)
Yes

```

- Show all stamps from the set *Castles*.

```

?- collection(stamp(_,'Castles',_ ,_)).
stamp(Germany, Castles, 1885, 10)
stamp(Germany, Castles, 1879, 50)
stamp(Germany, Castles, 1885, 60)
Yes

```

- Show all stamps issued between 1875 and 1883.

```

?- between(1875,1883,Y), collection(stamp(_,_ ,Y,_)), fail.
stamp(Germany, Kaiser, 1879, 20)
stamp(Germany, Castles, 1879, 50)
stamp(Germany, Kaiser, 1882, 5)
No

```

■

**Exercise 3.9.** Assume that the stamp collector wants to sell the German *Kaiser* set of stamps. Construct a Prolog query to achieve the corresponding database modification *interactively*.

**Exercise 3.10.** (This is a task in preparation for Exercise 3.11.) Define a predicate *remove\_all/3* for removing all entries from a list which match a given pattern. Example:

```
?- remove_all(item(_,5),
               [item(6,9),item(1,5),item(7,1),item(9,5)],L).
L = [item(6, 9), item(7, 1)]
```

(The original order is retained in the third argument of *remove\_all/3*.)

**Exercise 3.11.** Use *remove\_all/3* from Exercise 3.10 to define *sell/1* for removing from the database all stamps conforming to a given criterion. For example, all British stamps from the set *Poets* issued in 1978 may be removed interactively thus

```
?- sell(stamp('Britain','Poets',1978,_)).
Yes
?- collection(stamp(_,'Poets',_,_)).
stamp(Britain, Poets, 1979, 20)
stamp(Britain, Poets, 1977, 40)
Yes
```

**Exercise 3.12.** Define *insert/3* for inserting into a *list* of stamps a new stamp. Requirements:

- The new stamp has to be positioned according to its denomination.
- The new stamp has to fit into the existing set supplied in the second argument of *insert/3*.

(Notice that *insert/3* won't affect the database.) Examples:

```
?- insert(stamp('Britain','Flowers',2001,70),
          [stamp('Britain','Flowers',2000,40),
           stamp('Britain','Flowers',2000,60),
           stamp('Britain','Flowers',1991,100)],L).
L = [stamp('Britain', 'Flowers', 2000, 40),
     stamp('Britain', 'Flowers', 2000, 60),
     stamp('Britain', 'Flowers', 2001, 70),
     stamp('Britain', 'Flowers', 1991, 100)]
Yes

?- insert(stamp('Britain','Sports',2001,70),
          [stamp('Britain','Flowers',2000,40),
           stamp('Britain','Flowers',2000,60),
           stamp('Britain','Flowers',1991,100)],L).
No
```

(A concise recursive solution is sought.)

**Exercise 3.13.** Define *buy/1* for including a new stamp into the database. If the new stamp fits into an existing set, it should be included in there. Otherwise, a new set should be created with just this new stamp in it. For example, the 25 Pence stamp from the 1966 issue of the *Queen* set may be included in the database by

```
?- buy(stamp('Britain','Queen',1966,25)).
```

Yes

```
?- collection(stamp(_,'Queen',_,_)).
```

```
stamp(Britain, Queen, 1965, 20)
```

```
stamp(Britain, Queen, 1966, 25)
```

```
stamp(Britain, Queen, 1967, 50)
```

```
stamp(Britain, Queen, 1963, 120)
```

Yes

And, record the purchase of the 50 Öre stamp from the 1956 issue of the Swedish *Nobel Laureates* set by

```
?- buy(stamp('Sweden','Nobel Laureates',1956,50)).
```

Yes

```
?- collection(stamp('Sweden',_,_,_)).
```

```
stamp(Sweden, Nobel Laureates, 1956, 50)
```

Yes



## 3.2 Case Study: Automated Unfolding

We have introduced in Sect. 2.3.1 the program transformation technique *unfolding* and saw by way of an example that it can enhance a program's performance. There, the transformation was carried out essentially 'manually' though with some assistance (for unification) from the Prolog system. We now want to examine an automated tool for unfolding, written in Prolog. Figs. 3.9–3.10 (pp. 97–98) show an annotated session for solving by this tool the example from Sect. 2.3.1 interactively.

The tool comprises the predicates *elementary\_unfolding/5*, *unfold/3* and *clause\_arrange/2*, the first two of which are implementations of Elementary and Complete One Step Unfolding, respectively. The meaning and use of their arguments is easily gleaned from the sample sessions. The third of these predicates, *clause\_arrange/2*, is used to retain in the Prolog database a specified set of clauses of a predicate as indicated by the clause numbers in the second (list) argument. It thereby allows redundant clauses to be discarded and the others be sorted as deemed necessary.

The steps involved in implementing elementary Unfolding and Complete One Step Unfolding will be demonstrated with reference to the definitions of some predicates *a/5* and *c/2* shown respectively in (P-3.11) and (P-3.12).

**Prolog Code P-3.11: Definition of *a/5***

```

1 a(U,U,U,U,U).
2 a(U,V,U,V,U) :- m(U,V).
3 a(U,V,W,V,U) :- n(U,n(V,W)), b(U,V), e(V,U).
4 a(U,V,W,X,Y) :- b(U,V), c(V,W), d(W,X), e(X,Y).
```

**Prolog Code P-3.12: Definition of *c/2***

```

1 c(A,B) :- f(A), m(A,B).
2 c(A,B) :- A is B + 1.
3 c(A,A) :- f(A), g(A).
```

### 3.2.1 Elementary Unfolding

Let us unfold goal 2 in clause 4 of *a/5* by using clause 3 of *c/2*:

```
?- elementary_unfolding(a/5,4,2,c/2,3).
Yes
```

Thereafter the database will contain an additional clause for *a/5*:

```
?- listing(a/5).
...
a(A, B, B, C, D) :- b(A, B), f(B), g(B), d(B, C), e(C, D).
Yes
```

We show a series of queries in Figs. 3.11–3.14 (pp. 99–101) to illustrate the idea behind the definition of *elementary\_unfolding/5*.

The following observations on these figures are in order.

- **Fig. 3.11:** The query comprises three phases.

1. The built-in predicates *functor/3*, *nth\_clause/3* and *clause/3* are used to split up the fourth clause of *a/5* into its building blocks: in particular, *Body1* is unified with a term which is the conjunction of the clause's goals. (For *nth\_clause/3* and *clause/3*, see inset on p. 102.)
  2. The user-defined predicate *conj/2* then returns the list of conjuncts of *Body1* in *L1*.
  3. Finally, the user-defined predicate *splitlist/5* is used to disassemble the list of conjuncts *L1* around its second entry into three parts. Notice in particular that *Entry1* is unified with the goal to be unfolded later.
- **Fig. 3.12:** Here we disassemble the third clause of *c/2* in a similar manner to steps 1 and 2 above.



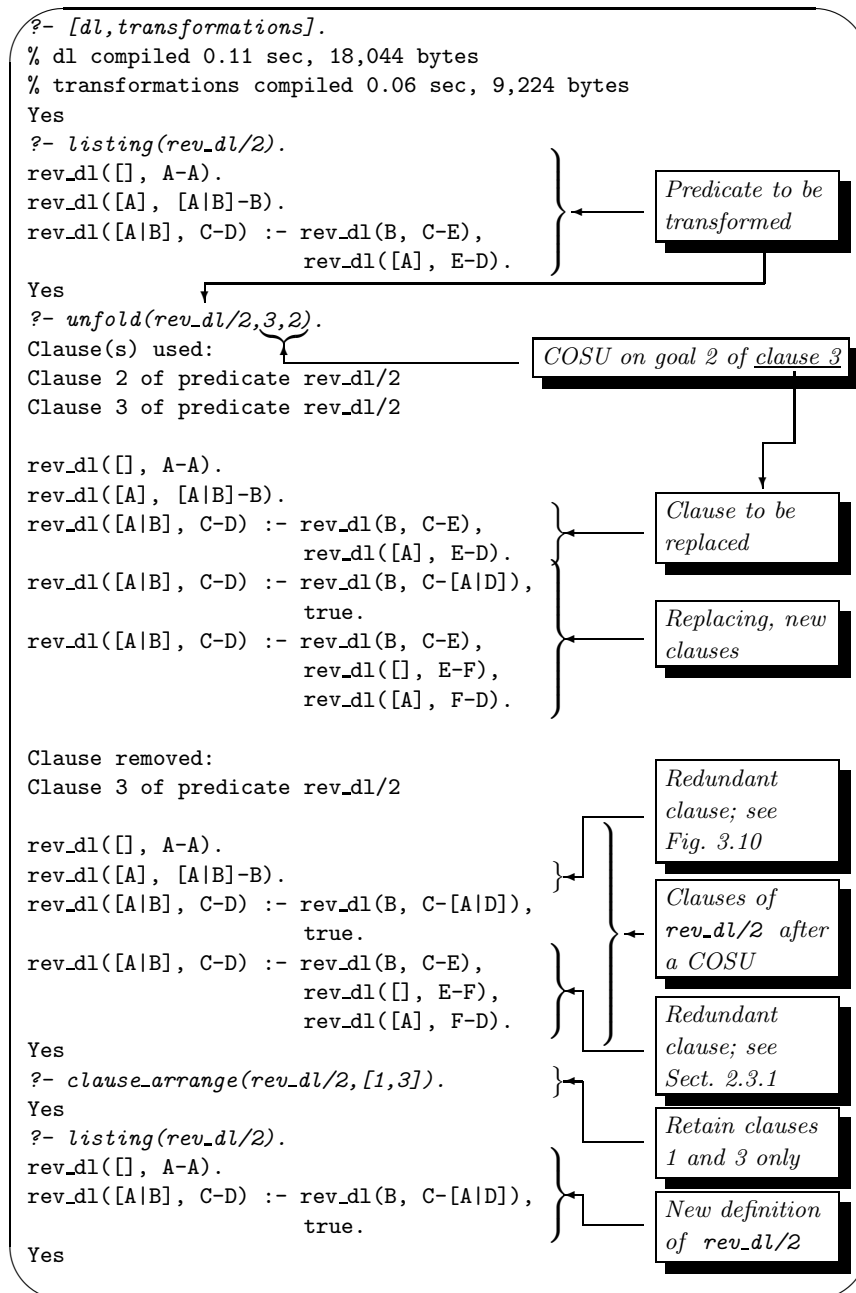


Figure 3.9: Interactive Prolog-Assisted Program Transformation: Session I

```

?- consult(user).
|: :- consult(transformations).
% transformations compiled 0.06 sec, 9,584 bytes
|: rev_dl([],L-L).
|: rev_dl([H/T],L1-L2) :- rev_dl(T,L1-[H/L2]).
|: Ctrl+Z
% user compiled 86.18 sec, 10,128 bytes
Yes

?- elementary_unfolding(rev_dl/2,2,1,rev_dl/2,1).
Yes

?- listing(rev_dl/2).
rev_dl([], A-A).
rev_dl([A|B], C-D) :- rev_dl(B, C-[A|D]).
rev_dl([A], [A|B]-B).
Yes

```

Manual input of `rev_dl/2`

Unfold on goal 1 of clause 2 using clause 1

Old clauses

New clause

Figure 3.10: Interactive Prolog-Assisted Program Transformation: Session II

```
?- functor(Pred1,a,5), nth_clause(Pred1,4,Ref1), clause(Head1,Body1,Ref1), conj(Body1,L1),
   splitlist(2,L1,Front1,Entry1,Behind1).
Pred1 = a(_G1123, _G1124, _G1125, _G1126, _G1127)
Ref1 = 1794731
Head1 = a(_G1132, _G1133, _G1134, _G1135, _G1136)
Body1 = b(_G1132, _G1133), c(_G1133, _G1134), d(_G1134, _G1135), e(_G1135, _G1136)
L1 = [b(_G1132, _G1133), c(_G1133, _G1134), d(_G1134, _G1135), e(_G1135, _G1136)]
Front1 = [b(_G1132, _G1133)]
Entry1 = c(_G1133, _G1134)
Behind1 = [d(_G1134, _G1135), e(_G1135, _G1136)]
Yes
```

Figure 3.11: Unfolding, Experiment 1: Disassembling clause 4 of *a/5*

```
?- functor(Pred2,c,2), nth_clause(Pred2,3,Ref2), clause(Head2,Body2,Ref2), conj(Body2,L2).
Pred2 = c(_G820, _G821)
Ref2 = 1794877
Head2 = c(_G826, _G826)
Body2 = f(_G826), g(_G826)
L2 = [f(_G826), g(_G826)]
Yes
```

Figure 3.12: Unfolding, Experiment 2: Disassembling clause 3 of *c/2*

```

?- functor(Pred1,a,5), ..., functor(Pred2,c,2), ..., Head2 = Entry1.
Pred1 = a(_G1908, _G1909, _G1910, _G1911, _G1912)
Ref1 = 1794731
Head1 = a(_G1917, _G1918, _G1918, _G1920, _G1921)
Body1 = b(_G1917, _G1918), c(_G1918, _G1918), d(_G1918, _G1920), e(_G1920, _G1921)
L1 = [b(_G1917, _G1918), c(_G1918, _G1918), d(_G1918, _G1920), e(_G1920, _G1921)]
Front1 = [b(_G1917, _G1918)]
Entry1 = c(_G1918, _G1918)
Behind1 = [d(_G1918, _G1920), e(_G1920, _G1921)]
Pred2 = c(_G1988, _G1989)
Ref2 = 1794877
Head2 = c(_G1918, _G1918)
Body2 = f(_G1918), g(_G1918)
L2 = [f(_G1918), g(_G1918)]
Yes

```

Figure 3.13: Unfolding, Experiment 3: Experiments 1 &amp; 2 followed by appropriate unification

```

?- functor(Pred1,a,5), ..., functor(Pred2,c,2), ..., Head2 = Entry1, concat3(Front1,L2,Behind1,L),
   conj(NewBody,L), dynamic(a/5), assert(Head1 :- NewBody).
Pred1 = a(_G2534, _G2535, _G2536, _G2537, _G2538)
Ref1 = 1794731
Head1 = a(_G2543, _G2544, _G2544, _G2546, _G2547)
Body1 = b(_G2543, _G2544), c(_G2544, _G2544), d(_G2544, _G2546), e(_G2546, _G2547)
L1 = [b(_G2543, _G2544), c(_G2544, _G2544), d(_G2544, _G2546), e(_G2546, _G2547)]
Front1 = [b(_G2543, _G2544)]
Entry1 = c(_G2544, _G2544)
Behind1 = [d(_G2544, _G2546), e(_G2546, _G2547)]
Pred2 = c(_G2614, _G2615)
Ref2 = 1794877
Head2 = c(_G2544, _G2544)
Body2 = f(_G2544), g(_G2544)
L2 = [f(_G2544), g(_G2544)]
L = [b(_G2543, _G2544), f(_G2544), g(_G2544), d(_G2544, _G2546), e(_G2546, _G2547)]
NewBody = b(_G2543, _G2544), f(_G2544), g(_G2544), d(_G2544, _G2546), e(_G2546, _G2547)
Yes
?- listing(a/5).
...
a(A, B, B, C, D) :- b(A, B), f(B), g(B), d(B, C), e(C, D).
Yes

```

Figure 3.14: Unfolding, Experiment 4: Experiment 3 followed by new clause creation and database update

---

**Built-in Predicates:** *nth\_clause/3* and *clause/3*

*nth\_clause(+Pred,+Index,?Ref)* is used to assign a system-chosen reference to a specific clause of a predicate. This reference may be used subsequently to retrieve head and body of the clause by *clause/3*. Example: Head and body of the second clause of the predicate *c/2*, defined by

```
c(A,B) :- f(A), m(A,B).
c(A,B) :- A is B + 1.
c(A,A) :- f(A), g(A).
```

may be retrieved by

```
?- nth_clause(c(_,_),2,Ref), clause(Head,Body,Ref).
Ref = 1791614
Head = c(_G542, _G543)
Body = _G542 is _G543+1
```

If used with the instantiation pattern *nth\_clause(+Pred,-Index,-Ref)*, on backtracking the references to all clauses of a given predicate are obtained:

```
?- nth_clause(c(_,_),Index,Ref).
Index = 1 Ref = 1791577 ;
Index = 2 Ref = 1791614 ;
Index = 3 Ref = 1791649 ;
No
```

---

- **Fig. 3.13:** The previous two steps are repeated and then *Head2* is unified with *Entry1*, essentially completing the unfolding operation. Notice in particular that the effect of unifying *Head2* with *Entry1* ‘ripples through’ to all other variables: for example, as expected, in *Head1* the second and third arguments become identical while this was not the case before unification (see Fig. 3.11).
- **Fig. 3.14:** Subsequent to the steps above, we first assemble in *L* the list of goals for the new clause; we use here the (fairly straightforward) user-defined predicate *concat3/4*. Then, *conj/2* is used again (now in the ‘reverse’ direction) to create the term *NewBody*, the conjunction of terms in *L*. Finally, the new clause is written to the database, confirmed also by the next query using *listing/1*.

The definition of *elementary\_unfolding/5* in *transformations.pl* follows the query shown in Fig. 3.14. The auxiliary predicates used therein won’t be discussed here; the way conjunctions are composed/decomposed by *conj/2* is noteworthy, however. This is accomplished within *conj/2* via the auxiliaries *conjunction(+List,+Acc,-Term)* and *conjuncts(+Term,+Acc,-List)* whose working is illustrated below.

```
?- conjunction([t(X),u(Y,a),v(b,X)],s(Y),C), conjuncts(C,[],L).
X = _G492
Y = _G497
C = v(b, _G492), u(_G497, a), t(_G492), s(_G497)
L = [s(_G497), t(_G492), u(_G497, a), v(b, _G492)]
```

---

They are defined in (P-3.13) and (P-3.14) by the accumulator technique.<sup>10</sup>

**Prolog Code P-3.13: Definition of *conjunction/3***

```

1 conjunction([], Conj, Conj).
2 conjunction([H/T], Acc, Conj) :- conjunction(T, (H, Acc), Conj).
```

<sup>10</sup>In (P-3.14) we implicitly use the fact that Prolog's conjunction is right-associative. The two queries below thus generate the same response:

```

?- conjuncts((v(b,X), u(Y, a), t(X), s(Y)), [], L).
X = _G409 Y = _G411
L = [s(_G411), t(_G409), u(_G411, a), v(b, _G409)]
?- conjuncts((v(b,X), (u(Y, a), (t(X), s(Y)))), [], L).
X = _G433 Y = _G435
L = [s(_G435), t(_G433), u(_G435, a), v(b, _G433)]
What will Prolog's response be to the query below?
?- conjuncts(((v(b,X), u(Y, a)), t(X)), s(Y)), [], L).
```

**Prolog Code P-3.14: Definition of *conjuncts/3***

```

1 conjuncts(Term, Acc, [Term|Acc]) :- not(funcator(Term, ', ', 2)).
2 conjuncts(Term, Acc, L) :- funcator(Term, ', ', 2),
3                               arg(1, Term, Term1),
4                               arg(2, Term, Term2),
5                               conjuncts(Term2, [Term1|Acc], L).

```

### 3.2.2 Complete One Step Unfolding

Let us now assume that we want to unfold clause 4 of *a/5* on its second goal. We can do this by repeatedly using *elementary\_unfolding/5* in an obvious manner:

```
?- elementary_unfolding(a/5, 4, 2, c/2, K).
```

```
K = 1 ; K = 2 ; K = 3 ;
```

```
No
```

```
?- listing(a/5).
```

```
a(A, A, A, A, A).
```

```
a(A, B, A, B, A) :- m(A, B).
```

```
a(A, B, C, B, A) :- n(A, n(B, C)), b(A, B), e(B, A).
```

```
a(A, B, C, D, E) :- b(A, B), c(B, C), d(C, D), e(D, E).
```

```
a(A, B, C, D, E) :- b(A, B), f(B), m(B, C), d(C, D), e(D, E).
```

```
a(A, B, C, D, E) :- b(A, B), B is C+1, d(C, D), e(D, E).
```

```
a(A, B, B, C, D) :- b(A, B), f(B), g(B), d(B, C), e(C, D).
```

In doing so, the following steps have been carried out:

1. We have visually identified *c(V, W)* as goal 2 in clause 4 of *a/5*.
2. We have attempted (and successfully completed) by backtracking an elementary unfolding operation with each of the clauses of *c/2*.

To complete the task, we would also need to

3. Remove clause 4 of *a/5* from the database.

Step 2 is more concisely implemented by a failure driven loop thus

```
?- elementary_unfolding(a/5, 4, 2, c/2, K), fail.
```

```
No
```

Within the same failure driven loop we may integrate Step 1 by attempting an elementary unfolding operation with *each* predicate in the database. The generation of all predicates may be accomplished by<sup>11</sup>

```

?- current_predicate(Fun, Head),
   not(predicate_property(Head, built_in)),
   not(predicate_property(Head, imported_from(_))),
   not(predicate_property(Head, foreign)),
   functor(Head, Fun, Arity).

```

---

<sup>11</sup>The same functionality (i.e. retrieval from the database of all user-defined predicates) is achieved by the almost identical predicate *my\_predicate/2* from p. 88.



```

Fun = a
Head = a(_G1380, _G1381, _G1382, _G1383, _G1384)
Arity = 5 ;
...
Fun = c
Head = c(_G1380, _G1381)
Arity = 2 ;
...

```

Embedding this within the earlier failure driven loop will essentially implement *unfold/3*:

```

?- current_predicate(Fun,Head),
   not(predicate_property(Head,built_in)),
   not(predicate_property(Head,imported_from(_))),
   not(predicate_property(Head,foreign)),
   functor(Head,Fun,Arity),
   elementary_unfolding(a/5,4,2,Fun/Arity,K), fail.
No

```

For further details on the definition of *unfold/3* the reader is referred to the file `transformations.pl`. (Noteworthy is perhaps the use in Step 3 of the built-in predicate *erase/1*.)

---

#### Built-in Predicate: *erase(+Ref)*

*erase(+Ref)* removes the clause with reference *Ref* from the database. Example:

```

?- dynamic(num/1), ((member(_I,[1,2,3]), assert(num(_I)),
   fail); true), listing(num/1).
num(1).
num(2).
num(3).
?- nth_clause(num_,2,Ref), erase(Ref), listing(num/1).
num(1).
num(3).
Ref = 3904727

```

---

**Exercise 3.14.** Use the predicate *unfold/3* to solve Exercise 2.9, Part (c).

■

### Self-unfolding

There may seem a subtle problem with our implementation of *unfold/3* which we want to address now.

In the definition of *unfold/3* we write (within a failure driven loop) to the database new clauses via *elementary\_unfolding/5* which itself ‘feeds on’ clauses (in its fourth argument) that are retrieved from the database. This construction could conceivably give rise to an infinite loop in the case of what was termed ‘self-unfolding’ in Sect. 2.3.1, p. 52. This cannot happen, however, since a search tree under consideration by Prolog won’t be affected by database changes created by the search itself. The following simple interactive session illustrates this point.

```

?- listing(num/1).
num(1).
Yes
?- num(X), Y is 2 * X, assert(num(Y)), fail.
No
?- listing(num/1).
num(1).
num(2).
Yes

```

Had the search tree been affected by the database changes immediately we would have expected in the database infinitely many clauses of *num/1* like

```
num(1). num(2). num(4). ...
```

The session shown in Fig. 3.9 (involving self-unfolding of the predicate *rev\_dl/2*) confirms indeed that *unfold/3* does not cause looping.

### 3.2.3 Rearranging Clauses

Clauses of a predicate may be rearranged by *clause\_arrange/2* as illustrated in Fig. 3.9. To this end, the following auxiliary predicates have been defined:

- *all\_clauses/2* collects all clauses of a predicate into a list of terms. Example:

```

?- all_clauses(c/2,L).
L = [ (c(_G368, _G369) :- f(_G368), m(_G368, _G369)),
      (c(_G350, _G351) :- _G350 is _G351+1),
      (c(_G331, _G331) :- f(_G331), g(_G331))]

```

*all\_clauses/2* is defined by

```

all_clauses(Fun/Arity,List) :-
    functor(Pred,Fun,Arity),
    findall((Head :- Body),
            (nth_clause(Pred,_,Ref),
             clause(Head,Body,Ref)), List).

```

- *arrange/3* selects (a subset of) the entries of list as specified by a list of integers in the first argument. Example:

```

?- arrange([4,3,5],[a,b,c,d,e,f],L).
L = [d, c, e]

```

*arrange/3* is defined by

```

arrange(IntList,InL,OutL) :-
    findall(E,(member(M,IntList), nth1(M,InL,E)),OutL).

```

---

**Built-in Predicate:** *nth1*(?Index, ?List, ?Elem)

*nth1*/3 is used to select a specified entry from a list. Example:

```
?- nth1(3, [a,b,c,d,e,f], E).  
E = c
```

---

The definition of *clause\_arrange/2* in terms of the two auxiliaries is fairly straightforward; see the file *transformations.pl* for details.

**Exercise 3.15.** Use the predicate *unfold/3* to carry out a Complete One Step Unfolding on an appropriately chosen goal in one of the clauses of *flatten\_dl/2* from Sect. 2.2.3. After some removal and rearranging of clauses via *clause\_arrange/2*, you should arrive at a version of *flatten/2* which is more efficient than the earlier ones. Demonstrate the gain in speed by an experiment akin to the one carried out in Exercise 2.7. ■

**Exercise 3.16.** You will have seen in Exercise 3.15 that *unfold/3* places the new clauses *after* the existing ones. To observe the original order, the new clauses had to be subsequently moved by *clause\_arrange/2* to the position of the clause they were replacing. Write a predicate *cosu/3* which performs a Complete One Step Unfolding and then restores the predicates' order.<sup>12</sup> For example, the suggested solution of Exercise 3.15 (p. 162) could then be achieved simply by

```
?- cosu(flatten_dl/2,2,2).
...
?- listing(flatten_dl/2).
flatten_dl([], A-A).
flatten_dl([A], B-C) :- flatten_dl(A, B-C),
                        true.
flatten_dl([A, B|C], D-E) :- flatten_dl(A, D-F),
                             flatten_dl(B, F-G),
                             flatten_dl(C, G-E).

flatten_dl([A|B], C-D) :- flatten_dl(A, C-[B|D]),
                          true.
flatten_dl(A, [A|B]-B).
```

*Note.* When using *clause\_arrange/2*, you will have to be able to generate integer lists with specified bounds. The built-in predicate *between/3* may be used to achieve this.

■

## 3.3 Dijkstra's Dutch Flag Problem Revisited

### 3.3.1 Problem Generalization and First Solution

Dijkstra's Dutch Flag Problem from Sect. 2.4 may be generalized as follows:

- The items may be of any colour and any number of colours may occur.
- The items are to be grouped to a certain order of colours as specified by the user in some list *Colours*. This list need not include all the items' colours and may include colours not assigned to any of the items. As before, within each colour group the items' original order should be retained.

We call the predicate to be defined *dijkstra(+Colours,+Items,-Grouped)* and illustrate its desired behaviour by an example. Take the list of items

```
new_items([col(soot,black), col(tomato,red), col(nut,brown),
           col(milk,white), col(snow,white), col(coal,black),
           col(bile,green), col(bark,brown), col(ocean,blue),
           col(grass,green), col(apple,red), col(blood,red),
           col(night,black), col(sky,blue)]).
```

<sup>12</sup>To retrieve the number of clauses of a predicate, you should use the built-in predicate *predicate\_property/2* in the form *predicate\_property(+Pred,number\_of\_clauses(-ClauseNumber))*.

and sort it in the order *black, blue, violet, green, red* and *white*. (Notice that *brown* is not one of the colours listed here, nor is there any item whose colour is *violet*.) The expected behaviour of *dijkstra/3* is as follows.<sup>13</sup>

```
?- new_items(_Items),
   dijkstra([black,blue,violet,green,red,white],_Items,Grouped).
Grouped = [col(soot,black), col(coal,black), col(night,black),
           col(ocean,blue), col(sky,blue), col(bile,green),
           col(grass,green), col(tomato,red), col(apple,red),
           col(blood,red), col(milk,white), col(snow,white)]
```

*dijkstra/3* solves the *original* Dutch Flag problem from Sect. 2.4 if its first argument is unified with *[red, white, blue]*.

On inspection of *dijkstra/2* from Sect. 2.4.3 (the version based on difference lists) it is seen that the *current, specific* problem would be solved by *dijkstra/2* if *dijkstra\_dl/2* had been defined by the clause

```
dijkstra_dl(Items,L1-L7) :- colour_dl(black,Items,L1-L2),
                             colour_dl(blue,Items,L2-L3),
                             colour_dl(violet,Items,L3-L4),
                             colour_dl(green,Items,L4-L5),
                             colour_dl(red,Items,L5-L6),
                             colour_dl(white,Items,L6-L7).
```

This suggests introducing a predicate *replace\_dijkstra\_dl(+Colours)* for replacing the existing definition of *dijkstra\_dl/2* in the database by the desired one. Then, *dijkstra/3* may be defined in terms of the old version of *dijkstra/2* thus

```
dijkstra(Colours,Items,List) :- replace_dijkstra_dl(Colours),
                                dijkstra(Items,List).
```

Let us now look at in detail how the change in the database is accomplished.

```
replace_dijkstra_dl(Colours) :-
    dynamic(dijkstra_dl/2),                % goal 1
    retractall(dijkstra_dl(_,_)),          % goal 2
    conjuncts(Items,Colours,L,First,Last), % goal 3
    conj(Body,L),                          % goal 4
    assert(dijkstra_dl(Items,First-Last) :- Body). % goal 5
```

The first two goals are obvious: *dijkstra\_dl/2* is made a dynamic predicate and then its existing definition is removed from the database. The rôle of *conjuncts/5* is best illustrated by a sample query.

```
?- conjuncts(Items,[red,white,green],L,First,Last).
Items = _G399
L = [colour_dl(red, _G399, _G402-_G513),
     colour_dl(white, _G399, _G513-_G514),
     colour_dl(green, _G399, _G514-_G403)]
First = _G402
Last = _G403
```

---

<sup>13</sup>We note in passing that the default maximum number of entries of a list displayed on the terminal by SWI-Prolog is ten. For a *full* display of the twelve-entry list *Grouped*, we issue the prior query

```
?- set_prolog_flag(toplevel_print_options,[max_depth(20)]).
Yes
```

Here, *L* is unified with the list of terms whose conjunction will form the body of the clause for *dijkstra\_dl/2*. The variables *First*, *Last* and *Items* will be used in goal 5 as variables in the head of the clause for *dijkstra\_dl/2*. We won't spell out the definition of *conjuncts/5* here but consider some salient points only. The list of terms in the third argument is created by an auxiliary predicate using the accumulator technique; see the source code for details. Perhaps the most imminent question here is how to get hold of an unspecified number of variable names.<sup>14</sup> This is accomplished by *vars/2*,

```
?- vars(5,V).
V = [_G239, _G240, _G241, _G242, _G243]
```

which may be defined as shown below.<sup>15</sup>

```
vars(N,Vars) :- functor(Term,dummy,N),
               bagof(Var,Arg^arg(Arg,Term,Var),Vars).
```

The requisite number of variables is generated by the built-in predicate *functor/3*, as in

```
?- functor(Term,dummy,5).
Term = dummy(_G313, _G314, _G315, _G316, _G317)
```

subsequent to which *bagof/3* is used to collect the variables in a list. In goal 4, we use *conj/2* (which is known from Sect. 3.2.1, p. 102) to unify with *Body* the conjunction of terms for the body of the clause to be created. Finally, in goal 5 the clause is written to the database.

**Exercise 3.17.** Use *dijkstra/3* to define *dijkstra\_st(+Items,-Grouped)* for returning in *Grouped* the entries of *Items* such that

- All entries of *Items* feature in *Grouped*;
- The colours are *sorted* in alphabetical order;
- And, as before, within each colour group, the items' original order is retained.

Example:

```
?- items(_Items), dijkstra_st(_Items,Grouped).
Grouped = [col(sky, blue), col(ocean, blue), col(tomato, red),
           col(blood, red), col(cherry, red), col(milk, white),
           col(snow, white)]
```

<sup>14</sup>Only at runtime will it be known how many colours *conjuncts/5* holds in its second argument!

<sup>15</sup> There are at least two other alternatives for defining *vars/2*. The simplest is by using the built-in predicate *length/2*:

```
?- length(Vars,5).
Vars = [_G251, _G254, _G257, _G260, _G263]

The second one is based on the built-in predicate =../2 (univ) for assembling and disassembling terms. The idea for this implementation of vars/2 should be clear from the query below.

?- functor(Term,dummy,5), Term =.. [_|Vars].
Term = dummy(_G478, _G479, _G480, _G481, _G482)
Vars = [_G478, _G479, _G480, _G481, _G482]
```

(The predicates *functor/3*, *arg/3* and *univ* will be familiar from Sect. 2.2.1.)

### 3.3.2 Enhanced Implementations

The predicate *dijkstra(+Colours,+Items,-Grouped)* from Sect. 3.3.1 is inefficient inasmuch as it will require as many passes through *Items* as there are entries in *Colours*. We have seen implementations for the *original* Dutch Flag Problem in Exercise 2.10, requiring a *single* pass only through the input list *Items*. In this section, those versions will be enhanced for solving the problem's more general formulation.

As in Sect. 3.3.1 before, we want to glean the plan for solving the *general* problem by considering a *specific* example. Let us assume that *Colours* is the list *[black,white,red,green]*. Then, it is easily seen that the plan for the solution of Exercise 2.10 (pp. 152–153) still applies if *colour\_dl/4* and *dijkstra\_dl/2* are respectively replaced by the predicates *encolour\_dl/5* and *endijkstra\_dl/2* as shown in Fig. 3.15. Clearly,

```

encolour_dl([],B-B,W-W,R-R,G-G).                                } ①

encolour_dl([col(Object,black)|T],
             [col(Object,black)|B1]-B2,W1-W2,R1-R2,G1-G2) :-
    encolour_dl(T,B1-B2,W1-W2,R1-R2,G1-G2).
encolour_dl([col(Object,white)|T],
             B1-B2,[col(Object,white)|W1]-W2,R1-R2,G1-G2) :-
    encolour_dl(T,B1-B2,W1-W2,R1-R2,G1-G2).
encolour_dl([col(Object,red)|T],
             B1-B2,W1-W2,[col(Object,red)|R1]-R2,G1-G2) :-
    encolour_dl(T,B1-B2,W1-W2,R1-R2,G1-G2).
encolour_dl([col(Object,green)|T],
             B1-B2,W1-W2,R1-R2,[col(Object,green)|G1]-G2) :-
    encolour_dl(T,B1-B2,W1-W2,R1-R2,G1-G2).
encolour_dl([col(_,_)|T],B1-B2,W1-W2,R1-R2,G1-G2) :-
    encolour_dl(T,B1-B2,W1-W2,R1-R2,G1-G2).
encolour_dl([col(_,_)|T],B1-B2,W1-W2,R1-R2,G1-G2) :-
    encolour_dl(T,B1-B2,W1-W2,R1-R2,G1-G2).

endijkstra_dl(Items, L1-L7) :-
    encolour_dl(Items,L1-L2,L2-L3,L3-L4,L4-L5,L5-L6,L6-L7).

```

Figure 3.15: Illustrative Example of Intended Database Updates

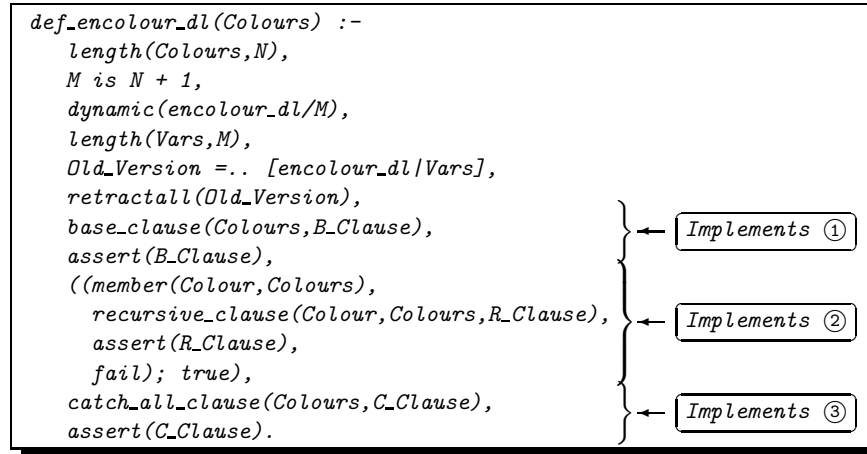
the list of colours in *Colours* will be known at runtime only and thus the predicate definitions indicated in Fig. 3.15 should be accomplished by prior database updates. The predicates *def\_encolour\_dl(+Colours)* and *def\_endijkstra\_dl(+Colours)* shall be responsible for writing to the database clauses like ①–③ and ④, respectively.

The present problem is more complex than that in Sect. 3.3.1 in two respects: both the number of clauses for, and the arity of the predicate *encolour\_dl* will be known at runtime only.

#### Implementing *def\_encolour\_dl/1* and *def\_endijkstra\_dl/1*

The top level definition of *def\_encolour\_dl/1* is shown in Fig. 3.16. The following features are noteworthy:

- The old definition (if present) of *encolour\_dl* (with the same arity as the one to be implemented) is removed from the database.

Figure 3.16: Top Level Definition of *def\_encolour\_dl/1*

- The auxiliary predicate *base\_clause/2* creates the term for the base clause (marked ① in Fig. 3.15), followed by a database update. It is defined by the predicates

```

base_clause(Colours,(Head :- true)) :- length(Colours,N),
                                       base(N,Head).

base(N,Term) :- diffvars1(N,D),
               Term =.. [encolour_dl,[],D].

diffvars1(N,D) :- functor(Term,dummy,N),
                 Term =.. [_|L],
                 diffterms(L,L,D).

```

where *diffvars1/2* produces a list with a given number of differences of pairwise *identical* variables as exemplified by

```

?- diffvars1(3,D).
D = [_G287-_G287, _G288-_G288, _G289-_G289]

```

- The terms for the recursive clauses (marked ② in Fig. 3.15) are created by the auxiliary predicate *recursive\_clause/3* and written to the database within a failure driven loop. *recursive\_clause/3* reads at the top level as

```

recursive_clause(Colour,Colours,(Head :- Body)) :-
    length(Colours,N),
    diffvars2(N,D),
    head(Colour,Colours,T,D,Head),
    body(T,D,Body), !.

```

where



1. *diffvars2/2* produces a list with a given number of differences of pairwise *distinct* variables,
2. *head/5* produces the term for the head of *encolour*,
3. *body/3* produces the term for the body of *encolour*.

*head/5* and *body/3* are respectively defined by

```
head(Colour, Colours, T, D, Head) :-
    comb(Object, Colour, Colours, D, Modified),
    Head =.. [encolour_dl, [col(Object, Colour)|T]|Modified].

body(T, D, Body) :- Body =.. [encolour_dl, T|D].
```

The predicate *comb/5* combines the list of colours with the list of difference terms as exemplified below.

```
?- comb(Object, w, [r, w, g], [R1-R2, W1-W2, G1-G2], M).
Object = _G429
R1 = _G411  R2 = _G412
W1 = _G417  W2 = _G418
G1 = _G423  G2 = _G424
M = [_G411-_G412, [col(_G429, w)|_G417]-_G418, _G423-_G424]
```

In the definition of *comb/5* (not shown here) the accumulator technique is used.

- Finally, the catch-all clause (marked ③ in Fig. 3.15) is created by the auxiliary predicate *catch\_all\_clause/2* along similar lines to *body/3*. (Its definition is not shown here).

The definition of *def\_endijkstra\_dl/1* is broadly analogous to that of *catch\_all\_clause/2* and is not shown here. The full source code for the present version is available in the file *dl.pl*.

**Exercise 3.18.** In the above development, for simplicity, *def\_encolour\_dl/1* was defined such that clause ③ in Fig. 3.15 does not contain any reference to the colours to be omitted; this was accomplished by ③ being the last clause. The resulting definition of *encolour\_dl* will therefore be sensitive to the ordering of its clauses. This is not ideal, however, as it prevents code to be interpreted declaratively.

Redefine *def\_encolour\_dl(+Colours)* such that it writes to the database code which is not sensitive to clause reordering.

*Hints.*

- Aim at excluding the colours not in *Colours* by using the built-in predicate *member/2*. If, for example, *Colours* is unified with *[black, white, red, green]*, then *def\_encolour\_dl/1* writes instead of ③ the following clause to the database

```
encolour_dl([col(_, Clr)|T], B1-B2, W1-W2, R1-R2, G1-G2) :-
    not(member(Clr, [black, white, red, green])),
    encolour_dl(T, B1-B2, W1-W2, R1-R2, G1-G2).
```

- All we need is a new definition of *catch\_all\_clause/2*, used in Fig 3.16. Use *conj/2* (known from Sect. 3.2.1, p. 102) to construct the conjunction of the two goals in the body of the new clause of *encolour\_dl*. Each of the two conjuncts will be obtained by using *=../2*.
- The solution is in *dl.pl*.

■

### Performance Comparison

An experiment confirms that the enhanced version needs a lesser number of inferences than the version from Sect. 3.3.1.

```
?- new_items(_Items),
   _Colours = [black,blue,violet,green,red,white],
   def_encolour_dl(_Colours), def_endijkstra_dl(_Colours),
   time(endijkstra_dl(_Items,Grouped-[])).
% 16 inferences in 0.00 seconds (Infinite Lips)
Grouped = [col(soot, black), col(coal, black), ...]

?- new_items(_Items),
   _Colours = [black,blue,violet,green,red,white],
   replace_dijkstra_dl(_Colours),
   time(dijkstra_dl(_Items,Grouped-[])).
% 91 inferences in 0.00 seconds (Infinite Lips)
Grouped = [col(soot, black), col(coal, black), ...]
```

The earlier version will appear more efficient, however, if we repeat this experiment and take also into account the overhead for creating and writing to the database the versions' definitions. This apparent advantage disappears,

however, as soon as the list of items exceeds a certain length.

### Creating Plain Implementations

**Exercise 3.19.** *def\_encolour\_dl/1* and *def\_endijkstra\_dl/1* gave rise to enhanced implementations which themselves were using difference lists. Write analogues of these two predicates creating *plain* solutions of the Dutch Flag Problem. More precisely, the implementations thus created should themselves be (the augmented) analogues of the solution proposed in Exercise 2.10, p. 60. The interactive session in Fig. 3.17 overleaf illustrates the desired behaviour of *def\_encolour\_pl/1* and *def\_endijkstra\_pl/1*.

■

```

?- listing(encolour_pl).
ERROR: No predicates for 'encolour_pl'
No
?- def_encolour_pl([black,white,red,green]).
Yes
?- listing(encolour_pl).
encolour_pl([], [], [], []).
encolour_pl([col(A, black)|B], [col(A, black)|C], D, E, F) :-
    encolour_pl(B, C, D, E, F).
encolour_pl([col(A, white)|B], C, [col(A, white)|D], E, F) :-
    encolour_pl(B, C, D, E, F).
encolour_pl([col(A, red)|B], C, D, [col(A, red)|E], F) :-
    encolour_pl(B, C, D, E, F).
encolour_pl([col(A, green)|B], C, D, E, [col(A, green)|F]) :-
    encolour_pl(B, C, D, E, F).
encolour_pl([col(A, B)|C], D, E, F, G) :-
    encolour_pl(C, D, E, F, G).
Yes
?- listing(endijkstra_pl).
ERROR: No predicates for 'endijkstra_pl'
No
?- def_endijkstra_pl([black,white,red,green]).
Yes
?- listing(endijkstra_pl).
endijkstra_pl(A, B) :- encolour_pl(A, C, D, E, F),
    flatten([C, D, E, F], B).16
Yes
?- items(_Items)17, endijkstra_pl(_Items, Grouped).
Grouped = [col(milk, white), col(snow, white), col(tomato, red),
    col(blood, red), col(cherry, red)]
Yes

```

Figure 3.17: Example Session for Exercise 3.19

<sup>16</sup>In contrast to the special case in Exercise 2.10, now the number of lists to be concatenated will be known at runtime only. Thus the concatenation is best accomplished by using *flatten/2* and not by (repeated use of) *append/3*.

<sup>17</sup>The predicate *items/1* is as defined in Sect. 2.4, p. 57.

## Chapter 4

# Exploratory Code Development

Conciseness and accessibility of source code through declarative reading are Prolog's major strengths. It is therefore relatively easy to appreciate the workings of someone else's implementation, while it is much harder independently *to arrive at* one's own solution to the same problem. In this chapter, we illustrate a practical methodology which is intended to overcome this discrepancy: it is a software development style that is interactive, incremental, exploratory and allows Prolog code to be arrived at in a relatively effortless manner.

### 4.1 A Nursery Rhyme

The task is to write a Prolog predicate *rhyme/0* which displays on the screen the well-known nursery rhyme *This is the House that Jack Built* ([11]):

This is the house that Jack built.

This is the malt  
That lay in the house that Jack built.

This is the rat  
That ate the malt  
That lay in the house that Jack built.

This is the cat  
That killed the rat  
That ate the malt  
That lay in the house that Jack built.

This is the dog  
That worried the cat  
That killed the rat  
That ate the malt  
That lay in the house that Jack built.

This is the cow with the crumpled horn  
That tossed the dog  
That worried the cat  
That killed the rat  
That ate the malt  
That lay in the house that Jack built.

This is the maiden all forlorn  
That milked the cow with the crumpled horn

That tossed the dog  
That worried the cat  
That killed the rat  
That ate the malt  
That lay in the house that Jack built.

This is the man all tattered and torn  
That kissed the maiden all forlorn  
That milked the cow with the crumpled horn  
That tossed the dog  
That worried the cat  
That killed the rat  
That ate the malt  
That lay in the house that Jack built.

This is the priest all shaven and shorn  
That married the man all tattered and torn  
That kissed the maiden all forlorn  
That milked the cow with the crumpled horn  
That tossed the dog  
That worried the cat  
That killed the rat  
That ate the malt  
That lay in the house that Jack built.

This is the cock that crowed in the morn  
That waked the priest all shaven and shorn  
That married the man all tattered and torn  
That kissed the maiden all forlorn

That milked the cow with the crumpled horn  
 That tossed the dog  
 That worried the cat  
 That killed the rat  
 That ate the malt  
 That lay in the house that Jack built.

This is the farmer sowing his corn  
 That kept the cock that crowed in the morn

That waked the priest all shaven and shorn  
 That married the man all tattered and torn  
 That kissed the maiden all forlorn  
 That milked the cow with the crumpled horn  
 That tossed the dog  
 That worried the cat  
 That killed the rat  
 That ate the malt  
 That lay in the house that Jack built.

In our implementation of *rhyme/0* we want to exploit the rhyme's repetitive structure and the fact that all essential information is contained in its last verse. We record the last verse in the database by *verse/1* as shown in (P-4.1).

**Prolog Code P-4.1: Definition of *verse/1***

```
1 verse(['This is the farmer sowing his corn',
2       'That kept the cock that crowed in the morn',
3       'That waked the priest all shaven and shorn',
4       'That married the man all tattered and torn',
5       'That kissed the maiden all forlorn',
6       'That milked the cow with the crumpled horn',
7       'That tossed the dog',
8       'That worried the cat',
9       'That killed the rat',
10      'That ate the malt',
11      'That lay in the house that Jack built.']).
```

The rhyme is seen roughly to match the simplified pattern shown in Fig. 4.1.

<i>verse 1</i>	<i>verse 2</i>	<i>verse 3</i>	<i>verse 4</i>	<i>verse 5</i>	<i>verse 6</i>	
↓	↓	↓	↓	↓	↓	
A	B	C	D	E	F	...
	A	B	C	D	E	...
		A	B	C	D	...
			A	B	C	...
				A	B	...
					A	...

Figure 4.1: The Rhyme's Simplified Pattern

Knowing the rhyme's last verse and the above structure will allow (up to some finer detail) the rhyme to be fully reconstructed. With a view to a simplified *preliminary* Prolog implementation, we therefore define the following Prolog fact in the database

```
verse_skeleton(['F', 'E', 'D', 'C', 'B', 'A']).
```

The *first task* is now to define a predicate *rhyme\_prel/2* which *should* enable us to obtain the skeleton rhyme's structure in the following manner.

```
?- verse_skeleton(_V), rhyme_prel(_V,_R), write_term(_R, []).
[[A], [B, A], [C, B, A], [D, C, B, A], [E, D, C, B, A],
[F, E, D, C, B, A]]
```

Taking this as an *informal specification* of *rhyme\_prel/2*, we want to arrive at its definition by a series of *interactive experiments*.

#### 4.1.1 First Preliminary Implementation

What could be the least ambitious first step in implementing *rhyme\_prel/2*? We may for example create a list whose only entry is the last entry of the above list-of-lists. (This will correspond to reproducing the last verse.) This we do by

```
?- verse_skeleton(_V), _R = [_V], write_term(_R, []).
[[F, E, D, C, B, A]]
```

Still interactively, a list comprising the last two entries of the target list-of-lists may be generated by

```
?- verse_skeleton(_V), _V = [_/_T1], _R = [_T1,_V],
    write_term(_R, []).
[[E, D, C, B, A], [F, E, D, C, B, A]]
```

Here we unify  $\_T1$  with the tail of  $\_V$  and position it in front of  $\_V$  to form the new list (of lists). How do we now generate the next larger list (comprising the last three entries of the target list-of-lists)? We proceed as before except that we assemble  $\_R$  from the entries  $\_T2$ ,  $\_T1$  and  $\_V$  (in that order!) where  $\_T2$  is unified with the tail of  $\_T1$ .

```
?- verse_skeleton(_V), _V = [_/_T1], _T1 = [_/_T2],
    _R = [_T2,_T1,_V], write_term(_R, []).
[[D, C, B, A], [E, D, C, B, A], [F, E, D, C, B, A]]
```

One more such step should suffice to appreciate the underlying pattern of interactively generating instances of  $\_R$ .

```
?- verse_skeleton(_V), _V = [_/_T1], _T1 = [_/_T2],
    _T2 = [_/_T3], _R = [_T3,_T2,_T1,_V], write_term(_R, []).
[[C, B, A], [D, C, B, A], [E, D, C, B, A], [F, E, D, C, B, A]]
```

Since our aim is to identify a *recursive* pattern in the above interactive session, we recast the inputs slightly by observing that  $[a_1, \dots, a_{n-1}, a_n]$  and  $[a_1|[a_2|[a_3|\dots|[a_{n-1}[[a_n]]\dots]]]$  are equivalent representations of the same list. Let's have a look at the last two queries again.

```
?- verse_skeleton(_V), _V = [_/_T1], _T1 = [_/_T2],
    _R = [
        Head_Old | Tail_Old
        Rhyme_Old
    ], write_term(_R, []).
```

[[D, C, B, A], [E, D, C, B, A], [F, E, D, C, B, A]]

```
?- verse_skeleton(_V), _V = [_/_T1], _T1 = [_/_T2],
    Head_Old = [_/_T3], Head = [_T3],
    Head | Rhyme_Old
    Rhyme
], write_term(_R, []).
```

[[C, B, A], [D, C, B, A], [E, D, C, B, A], [F, E, D, C, B, A]]

The annotated lists suggest the following *pseudocode* (using Prolog's list-notation) for one single *recursive step*.

$$Rhyme\_Old = [Head\_Old/Tail\_Old] \quad (4.1)$$

$$Head\_Old = [_/Head]$$

$$Rhyme = [Head/Rhyme\_Old] \quad (4.2)$$

Notice that by equations (4.1) and (4.2) we may replace the latter by

$$Rhyme = [Head/[Head\_Old/Tail\_Old]]$$

The *base case* for the recursion is given by

$$First\_Rhyme = [['F', 'E', 'D', 'C', 'B', 'A']]$$



A straightforward implementation of the recursive step is by the (auxiliary) predicate *rhyme\_aux*/3 in (P-4.2).

**Prolog Code P-4.2: First definition of the auxiliary predicate**

```

1 rhyme_aux(R,1,R).
2 rhyme_aux([Head_Old|Tail_Old],Counter,R) :-
3     Head_Old = [_|Head],
4     New_Counter is Counter - 1,
5     rhyme_aux([Head|[Head_Old|Tail_Old]],New_Counter,R).

```

In the first argument of *rhyme\_aux*/3 the most recent version of the rhyme is accumulated; its second argument is a counter which is decremented from an initial value until it reaches unity at which point the third argument is instantiated to the first. It is noteworthy in the definition of *rhyme\_aux*/3 that, as a consequence of using the accumulator technique, reference to the more complex case in the recursive step is found in the rule's body. (In this sense, as opposed to the familiar situation from imperative programming, progression is from *right to left*.)

We find out by an experiment what the counter should be initialized to.

```

?- verse_skeleton(_V), rhyme_aux(_V,1,_R), write_term(_R,[]).
[[F, E, D, C, B, A]]
?- verse_skeleton(_V), rhyme_aux(_V,2,_R), write_term(_R,[]).
[[E, D, C, B, A], [F, E, D, C, B, A]]
...
?- verse_skeleton(_V), rhyme_aux(_V,6,_R), write_term(_R,[]).
[[A], [B, A], [C, B, A], [D, C, B, A], [E, D, C, B, A],
[F, E, D, C, B, A]]

```

It is seen that the second argument of *rhyme\_aux*/3 (the counter) will have to be initialized to the length of (what stands for) the last verse. This gives rise to the following *first* version of the predicate *rhyme\_prel*/2

```
rhyme_prel_1(V,R) :- length(V,L), rhyme_aux([V],L,R).
```

which then behaves as specified on p. 119.

Even though the solution thus obtained is perfectly acceptable, there is scope for improvement. Counters are commonly used in imperative programming for verifying a stopping criterion. The corresponding task in declarative programming is best achieved by *pattern matching*. There is indeed no need for a counter here since the information for when *not* to apply the recursive step (any more) can be gleaned from the pattern of the first argument of *rhyme\_aux*/3: For the recursion to stop, the head of the list-of-lists (in the first argument) should itself be a list with exactly one entry. (The complete rhyme will have been arrived at when the first verse comprises a single line!) This idea gives rise in (P-4.3) to a new, *improved* (and more concise) version of the auxiliary predicate, now called *rhyme\_aux*/3.

**Prolog Code P-4.3: Another definition of the auxiliary predicate**

```

1 rhyme_aux_2([[First]|Rest],[[First]|Rest]).
2 rhyme_aux_2([Head_Old|Tail_Old],R) :-
3     Head_Old = [_|Head],
4     rhyme_aux_2([Head|[Head_Old|Tail_Old]],R).

```

*rhyme\_aux\_2*/3 behaves as intended:

```
?- verse_skeleton(_V), rhyme_aux_2([_V],_R), write_term(_R, []).
[[A], [B, A], [C, B, A], [D, C, B, A], [E, D, C, B, A],
[F, E, D, C, B, A]]
```

The definition of a second, improved version of the preliminary rhyme predicate now simplifies to

```
rhyme_prel_2(V,R) :- rhyme_aux_2([V],R).
```

To complete the ‘skeleton version’ of the rhyme, we display the above by

```
?- verse_skeleton(_V), rhyme_prel_2(_V,_R), show_rhyme(_R).
A
B
A
...
F
E
D
C
B
A
```

with the predicate *show\_rhyme/1* defined by

```
show_list([]).
show_list([H|T]) :- write(H), nl, show_list(T).

show_rhyme([]).
show_rhyme([H|T]) :- show_list(H), nl, show_rhyme(T).
```

There is still scope for further improvement leading to an even more concise version of the auxiliary predicate. We may replace in the definition of *rhyme\_aux\_2/2* all occurrences of *Head\_Old* by *[H|T]*, say, accounting for the fact that *Head\_Old* will be unified with a list.

```
rhyme_aux_2([H|T]|Tail_Old,R) :-
    [H|T] = [_|Head],
    rhyme_aux_2([Head|[H|T]|Tail_Old],R).
```

But then, by virtue of the first goal in the body of this rule we may replace all occurrences of *Head* by *T*. Subsequently, the first goal may be dropped. Overall, we obtain in (P-4.4) a third, even more concise version of the auxiliary predicate.

**Prolog Code P-4.4: Third definition of the auxiliary predicate**

```
1 rhyme_aux_3([First|Rest],[First|Rest]).
2 rhyme_aux_3([H|T]|Tail_Old,R) :- rhyme_aux_3([T|[H|T]|Tail_Old],R).
```

There is hardly any room for improvement left save perhaps a minor simplification of the first clause. We derive an alternative boundary case by first completing the interactive session from p. 120 and then carrying out one more step:

```
?- verse_skeleton(_V), _V = [_/_T1], _T1 = [_/_T2],
    _T2 = [_/_T3], _T3 = [_/_T4], _T4 = [_/_T5],
    _R = [_T5|[_T4|[_T3|[_T2|[_T1|[_V]]]]], write_term(_R, []).
[[A], [B, A], [C, B, A], [D, C, B, A], [E, D, C, B, A],
[F, E, D, C, B, A]]

?- verse_skeleton(_V), _V = [_/_T1], _T1 = [_/_T2],
    _T2 = [_/_T3], _T3 = [_/_T4], _T4 = [_/_T5], _T5 = [_/_T6],
    _R = [_T6|[_T5|[_T4|[_T3|[_T2|[_T1|[_V]]]]]],
    write_term(_R, []).
[[], [A], [B, A], [C, B, A], [D, C, B, A], [E, D, C, B, A],
[F, E, D, C, B, A]]
```

The first query suggests that we are finished if the (partially) completed skeleton rhyme's head is a single-element list; this condition gave rise to the earlier boundary case. On the other hand, in the second query the variable `_R` is unified with a list whose head is empty and whose tail is the *full* skeleton rhyme. This suggests the following *alternative* first clause for `rhyme_aux_3/2`,

```
rhyme_aux_3([], [R], R).
```

The disadvantage of this stopping criterion is that it will cause one additional invocation of the recursive step.

Of course, the third version of the auxiliary predicate, *rhyme\_aux\_3/2*, (with any of the two alternative first clauses) gives rise to yet another version of *rhyme\_prel/2*.

```
rhyme_prel_3(V,R) :- rhyme_aux_3([V],R).
```

### 4.1.2 Another Preliminary Implementation

With a view to wishing to use the *accumulator technique* (yet again), let us examine the first few steps of an (as yet *imaginary*) interactive session.

```
?-...
[F, E, D, C, B, A], []
?-...
[E, D, C, B, A], [[F, E, D, C, B, A]]
?-...
[D, C, B, A], [[E, D, C, B, A], [F, E, D, C, B, A]]
?-...
[C, B, A], [[D, C, B, A], [E, D, C, B, A], [F, E, D, C, B, A]]
```

Two lists are involved here. The first list serves as a ‘supplier’ for updating the second one in which the skeleton rhyme’s verses are accumulated. We observe that in each step the first list ‘loses’ its head, whereas the second list is augmented by the first one. At the end of this sequence of steps (i.e. when the first list is empty) the second list will contain the full skeleton rhyme. Having established the underlying idea, we now turn to the corresponding interactive session. (This may look tedious but is easily carried out using ‘copy-and-paste’.)

```
?- verse_skeleton(_V), _P1 = (_V,[]), write_term(_P1,[]).
[F, E, D, C, B, A], []
?- verse_skeleton(_V), _P1 = (_V,[]),
   ([_H1|_T1],_Acc1) = _P1, _P2 = (_T1,[[_H1|_T1]|_Acc1]),
   write_term(_P2,[]).
[E, D, C, B, A], [[F, E, D, C, B, A]]
?- verse_skeleton(_V), _P1 = (_V,[]),
   ([_H1|_T1],_Acc1) = _P1, _P2 = (_T1,[[_H1|_T1]|_Acc1]),
   ([_H2|_T2],_Acc2) = _P2, _P3 = (_T2,[[_H2|_T2]|_Acc2]),
   write_term(_P3,[]).
[D, C, B, A], [[E, D, C, B, A], [F, E, D, C, B, A]]
?- ...
```

To see how consecutive steps in the above query are interrelated, we have a look at two goals in the last query in some more detail; this is shown in Fig. 4.2. It is indicated here how the new pair *\_P3* is expressed in terms

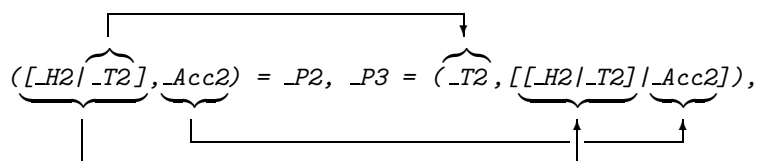


Figure 4.2: Exploring Details of the Rhyme’s Structure

of the old pair *\_P2*. This observation gives rise to (P-4.5), a fourth version of *rhyme\_prel/2*.

**Prolog Code P-4.5: Fourth version of *rhyme\_prel/2***

```

1 rhyme_prel_4(V,R) :- rhyme_acc(V,[],R).
2 rhyme_acc([],R,R).
3 rhyme_acc([H0ld|T0ld],Acc0ld,R) :-
4   rhyme_acc(T0ld,[[H0ld|T0ld]|Acc0ld],R).
```

### 4.1.3 The Final Version

We may use any of the four versions produced thus far of *rhyme\_prel/2* to obtain a rough version of *rhyme/0* by replacing in the query on p. 122, Sect. 4.1.1, the term *verse\_skeleton(\_V)* by the term *verse(\_V)*; for example,

```
?- verse(_V), rhyme_prel_2(_V,_R), show_rhyme(_R).
That lay in the house that Jack built.
```

```
That ate the malt
That lay in the house that Jack built.
```

```
...
```

```
This is the farmer sowing his corn
That kept the cock that crowed in the morn
...
```

```
That tossed the dog
That worried the cat
That killed the rat
That ate the malt
That lay in the house that Jack built.
```

We realize that the rhyme thus produced is not quite what we want: the first line of each verse (and not merely that of the last verse) should begin with ‘This is ...’. This effect will be achieved in three steps.

1. Define a predicate *to\_first/2* which, when applied to an atom, replaces all its characters up to the first occurrence of the string ‘the’ by the string ‘This is ’. Example:

```
?- to_first('We find the definite article.',A).
A = 'This is the definite article.'
```

2. Define *change\_first/2* in terms of *to\_first/2* by

```
change_first([H1|T],[H2|T]) :- to_first(H1,H2).
```

This predicate applies *to\_first/2* to the head of a list of atoms while leaving the tail unchanged. Example:

```
?- change_first(['That was the first','Now the second',
  'Now the third'],L).
L = ['This is the first', 'Now the second', 'Now the third']
```

3. Now apply *change\_first/2* by means of the built-in predicate *maplist/3* to the first line of each verse of the rhyme's rough version.

Below we show our definition of *to\_first/2*.

```
to_first(Old,New) :- atom_chars(Old,Charlist),
                    change(Charlist,Newlist),
                    concat_atom(Newlist,New).
```

Given an atom (in *Old*), it is first converted by means of the built-in predicate *atom\_chars/2* into a list of one-character atoms (in *Charlist*).

---

**Built-in Predicate:** *atom\_chars(?Atom,?CharList)*

It converts an atom into the corresponding list of one-character atoms and vice versa. Example:

```
?- atom_chars('Text',L).
L = ['T', e, x, t]
```

---

The predicate *change/2* is then used to effect the intended change in the atom's list-of-characters representation; it is defined by<sup>1</sup>

```
change([t,h,e|T],[ 'T',h,i,s,' ',i,s,' ',t,h,e|T]) :- !.
change(_|T,X) :- change(T,X).
```

and its behaviour is exemplified by

```
?- change(['F',i,n,d,' ',t,h,e,' ',s,t,r,i,n,g],_L),
   write_term(_L, []).
[T, h, i, s, , i, s, , t, h, e, , s, t, r, i, n, g]
```

Finally, the built-in predicate *concat\_atom/2* is used to convert the list-of-characters in *Newlist* into an atom (in *New*).<sup>2</sup>

---

**Built-in Predicate:** *concat\_atom(+List,-Atom)*

*Atom* is obtained by concatenating the elements of *List*. Example:

```
?- concat_atom([atom1,atom2,atom3],A).
A = atom1atom2atom3
```

---

Having thus arrived at an implementation of *change\_first/2*, we now want to apply this predicate to the head of each of the rough rhyme's verses. Since the latter is available (from *rhyme\_prel/2*) as a list, we may use *maplist/3* for a concise definition of *rhyme/0*.

<sup>1</sup>Because of the *cut*, *change/2* will fail on backtracking even for multiple occurrences of the substring 'the' in its first argument.

<sup>2</sup>For the present purposes where a list of *single character* atoms needs concatenating, we may use *atom\_chars/2* as an alternative. The last goal in the definition of *to\_first/2* then reads as *atom\_chars(New,Newlist)*.

---

**Built-in Predicate:** *maplist*(+Pred,?List1,?List2)

The 2-ary predicate *Pred* is applied to each entry of *List1* giving *List2* and vice versa.<sup>3</sup>Example:

```
?- maplist(append([a,b]),[[r,s],[u,v]],L).
L = [[a, b, r, s], [a, b, u, v]]
?- maplist(append([a,b]),L,[[a,b,r,s],[a,b,u,v]]).
L = [[r, s], [u, v]]
```

(Here, *append*/3 became a 2-ary predicate by partial application by fixing its first argument to *[a,b]*.)

---

Now, any of the four versions of *rhyme\_prel*/2 may be used to define *rhyme*/0; for example,

```
rhyme_2 :- verse(V),
           rhyme_prel_2(V,RTemp),
           maplist(change_first,RTemp,R),
           show_rhyme(R).
```

#### 4.1.4 Other Approaches

All solutions considered thus far were based on (some form of) the accumulator technique. The problem at hand can also be approached by simple recursion, however. To arrive at such a solution, we first show in Table 4.1 the desired rhyme for some last verses of various lengths. We ask ourselves the following question:

<i>Last Verse</i>	<i>Rhyme</i>
[‘A’]	[[‘A’]]
[‘B’, ‘A’]	[[‘A’], [‘B’, ‘A’]]
[‘C’, ‘B’, ‘A’]	[[‘A’], [‘B’, ‘A’], [‘C’, ‘B’, ‘A’]]
[‘D’, ‘C’, ‘B’, ‘A’]	[[‘A’], [‘B’, ‘A’], [‘C’, ‘B’, ‘A’], [‘D’, ‘C’, ‘B’, ‘A’]]
...	...

Table 4.1: Rhyme Structure

*Given a particular rhyme, how can the previous rhyme be expressed in terms of the current one?*

A *declarative* reading of the last two lines of Table 4.1 suggest the following: *[H/T]* is the last verse of the current rhyme *C* if *T* is the last verse of the previous rhyme *P* and *C* comes about by appending *[[H/T]]* to

---

<sup>3</sup>The ‘reverse’ application of *maplist*/3 is possible only if the second argument of *Pred* may be used in the input mode. This is *not* the case for example for *flatten*/2 as is shown below.

```
?- maplist(flatten,[[a,[b,[c,d],e]],[[[r,s],t],x,y]],L).
L = [[a, b, c, d, e], [r, s, t, x, y]]
?- maplist(flatten,L,[[a,b,c,d,e],[r,s,t,x,y]]).
No
```

$P$ . And, the boundary case is identified by observing that the one-line verse  $[L]$  is the last verse of  $[[L]]$ . The aforesaid is immediately expressed in Prolog by either of the two (*logically* equivalent) definitions (P-4.6) and (P-4.7).<sup>4</sup>

**Prolog Code P-4.6: Fifth version of *rhyme\_prel*/2**

```

1 rhyme_prel_5([L],[[L]]).
2 rhyme_prel_5([H|T],C) :- append(P,[[H|T]],C), rhyme_prel_5(T,P).
```

**Prolog Code P-4.7: Sixth version of *rhyme\_prel*/2**

```

1 rhyme_prel_6([L],[[L]]).
2 rhyme_prel_6([H|T],C) :- rhyme_prel_6(T,P), append(P,[[H|T]],C).
```

(It is readily confirmed that both versions behave as earlier ones do.) As each of the last two predicates is defined in terms of *append*/3 we would expect some improvement in elegance (and performance) by rewriting

---

<sup>4</sup>The following are alternative first clauses:

```

rhyme_prel_5([],[]).
rhyme_prel_6([],[]).
```



them using *difference lists*. Indeed, both versions give rise to (P-4.8), the same concise, tail recursive implementation using difference lists.

**Prolog Code P-4.8: Seventh version of *rhyme\_prel/2***

```

1 rhyme_prel_dl([L],[[L]|X]-X).
2 rhyme_prel_dl([H|T],C1-C2) :- rhyme_prel_dl(T,C1-[[H|T]|C2]).
3 rhyme_prel_7(V,R) :- rhyme_prel_dl(V,R-[]).
```

**Exercise 4.1.** We want to make an experimental comparison between the various versions of *rhyme\_prel/2* and need therefore a predicate that produces rhymes of any specified length. To be more specific, we will need a predicate *long\_verse/1* which removes from the database the current version of *verse/1* and replaces it by something of a repetitive structure and of a specified length as shown in the session below.

```

?- long_verse(3), verse(_V), show_list(_V).
That interacts with the item ...
That interacts with the item ...
That interacts with the item ...
?- rhyme_2.
This is the item ...
```

```

This is the item ...
That interacts with the item ...
```

```

This is the item ...
That interacts with the item ...
That interacts with the item ...
```

Define the predicate *long\_verse/1*.

■

We can now use *long\_verse/1* in conjunction with the built-in predicate *time/1* to assess the versions' performance; this is shown for the last three versions in Table 4.2 below.<sup>5</sup> As expected, version seven, the imple-

Version	length of _V	100	200	300	400	500
5	CPU-time [sec]	1.97	15.77	52.50	125.0	244.1
Version	length of _V	1,000	2,000	3,000	4,000	5,000
6	CPU-time [sec]	4.51	20.04	45.53	85.63	132.4
Version	length of _V	10,000	20,000	30,000	40,000	50,000
7	CPU-time [sec]	0.28	0.71	0.55	1.32	1.16

Table 4.2: CPU Times for Versions of the Query *?- rhyme\_prel(\_V,\_R)*.

mentation based on difference lists, is by far the most efficient. Furthermore, perhaps surprisingly, version six

<sup>5</sup>The first entry in Table 4.2 for example may be obtained by  

```

?- long_verse(100), verse(_V), time(rhyme_prel_5(_V,_R)).
% 176,749 inferences in 1.97 seconds (89720 Lips)
```

turns out to be better than its tail recursive counterpart, version five. We turn to Prolog's tracing facility to find out why this is the case:

```
?- trace([append/3,rhyme_prel_5/2,rhyme_prel_6/2]).
%      append/3: [call, redo, exit, fail]
%      rhyme_prel_5/2: [call, redo, exit, fail]
%      rhyme_prel_6/2: [call, redo, exit, fail]
[debug] ?- rhyme_prel_5(['B','A'],R).
T Call: (6) rhyme_prel_5(['B','A'], _G418)
T Call: (7) append(_G506, [['B','A']], _G418)
T Exit: (7) append([], [['B','A']], [['B','A']])
T Call: (7) rhyme_prel_5(['A'], [])
T Call: (8) append(_G512, [['A']], [])
T Fail: (8) append(_G512, [['A']], [])
T Fail: (7) rhyme_prel_5(['A'], [])
T Redo: (7) append(_G506, [['B','A']], _G418)
T Exit: (7) append(_G476, [['B','A']], [_G476, ['B','A']])
T Call: (7) rhyme_prel_5(['A'], [_G476])
T Exit: (7) rhyme_prel_5(['A'], [['A']])
T Exit: (6) rhyme_prel_5(['B','A'], [['A'], ['B','A']])
R = [['A'], ['B','A']]
[debug] ?- rhyme_prel_6(['B','A'],R).
T Call: (6) rhyme_prel_6(['B','A'], _G418)
T Call: (7) rhyme_prel_6(['A'], _G498)
T Exit: (7) rhyme_prel_6(['A'], [['A']])
T Call: (7) append(['A'], [['B','A']], _G418)
T Exit: (7) append(['A'], [['B','A']], [['A'], ['B','A']])
T Exit: (6) rhyme_prel_6(['B','A'], [['A'], ['B','A']])
R = [['A'], ['B','A']]
```

It is seen that version five causes Prolog to backtrack on the search tree of *append/3* until *append(\_G482, [['B','A']])* succeeds. This is quite a contrast to *rhyme\_prel\_6* which does not cause backtracking but builds up a stack of subgoals all of which eventually are satisfied in turn. It is also easily verified that on backtracking version five will not terminate whereas version six will fail to re-satisfy the goal and returns 'No'.

**Exercise 4.2.** Modify the definition of *rhyme\_prel\_5/2* such that it won't loop but fails on backtracking. ■

**Exercise 4.3.** Define *cputime(+Predname,+Arglist,-Time)* for obtaining the CPU seconds in *Time* for the predicate with name *Predname* and arguments in *Arglist*. Then, for example, the following is an alternative to the query in footnote 5 on p. 129:

```
?- long_verse(100),verse(_V),cputime(rhyme_prel_5,[_V,_R],Time).
Time = 1.97
```

The predicate *cputime/3* will be an improvement on *time/1* since it will then be possible to produce for example the first row of Table 4.2 *in one sweep* interactively as follows.

```
?- findall(_Time,(member(_L,[100,200,300,400,500,600,700]),
    long_verse(_L),
    verse(_V),
    cputime(rhyme_prel_5,[_V,_R],_Time)),
```

```
Times).
Times = [2.03, 15.71, 52.29, 124.51, 242.5, 419.58, 667.78]
```

(Slight variations in the CPU times may be observed even when repeating the same query.) In your definition of *cputime/3* you should use the built-in predicate *statistics/2*.

---

**Built-in Predicate:** *statistics(+Key,-Value)*

Unify system statistics determined by *Key* with *Value*. For example, we obtain the CPU seconds and number of inferences accumulated in the present Prolog session by

```
?- statistics(cputime,Time).
Time = 18020.2
?- statistics(inferences,Inf).
Inf = 222054681
```

---

**Exercise 4.4.** We have created several versions of *rhyme\_prel/2* and have indicated the version number by an appropriate suffix attached to the original predicate name. Let us now assume that this is the style for indicating predicates' versions in general. In this exercise, you are asked to define a predicate *cputime/4* which is a generalization of *cputime/3* from Exercise 4.3 in that the former will allow the version number to be specified by an extra (the third) argument. Example:

```
?- long_verse(100), verse(_V), cputime(rhyme_prel, [_V,_R], 5, Time).
Time = 1.97
```

The benefit of *cputime/4* is obvious: it will allow the timing of several versions of the same predicate in one sweep, as is illustrated below.

```
?- long_verse(70000), verse(_V),
   maplist(cputime(rhyme_prel, [_V,_R]), [1,2,3,4,7], Times).
Times = [4.28, 3.19, 3.35, 1.54, 3.18]
```

**Exercise 4.5.** Using *cputime/4* from Exercise 4.4, produce all entries of Table 4.2 interactively *by one single query*.

*Hint.* As a first step, you should revisit the problem of producing interactively a list comprising the *first row* of entries in Table 4.2 (c.f. Exercise 4.3). This is now best achieved by using the built-in predicates *findall/3* and *between/3* and by observing that the last verse's length is expressed in terms of the column number  $j = 1, \dots, 5$  as

$$\text{length} = j \times 10^2$$

The general case is dealt with by nesting two such constructs. Version number and length are respectively generated by

$$\begin{aligned} \text{version} &= i + 3 \\ \text{length} &= j \times 10^i \end{aligned}$$

with  $i = 2, 3, 4$  and  $j = 1, \dots, 5$ .

■

## 4.2 Project: '*One Man Went to Mow ...*'

Another nursery rhyme with a similar recursive structure is the well-known song *One man went to mow ...* whose three-verse version is as follows.<sup>6</sup>

One man went to mow,  
Went to mow a meadow,  
One man and his dog,  
Went to mow a meadow.

---

<sup>6</sup>Source: The *BBC* web site  
<http://www.bbc.co.uk/cbeebies/tweenies/songtime/>  
It is a cornucopia of songs and rhymes for pre-school children.

Two men went to mow,  
 Went to mow a meadow,  
 Two men, one man and his dog,  
 Went to mow a meadow.

Three men went to mow,  
 Went to mow a meadow,  
 Three men, two men, one man and his dog,  
 Went to mow a meadow,  
 Went to mow a meadow.

We want to outline here the way this rhyme can be produced in Prolog and formulate the stages of the detailed work as exercises.

This song has a very similar recursive structure to that of *This is the house that Jack built* except that there is now no predefined ‘last verse’ from which we could unravel the entire rhyme. Our aim is to produce a predicate `song/0` returning on the terminal a continuous stream of verses until stopped by the keystrokes `Ctrl+C`. The intended behaviour is shown in Fig. 4.3.<sup>7</sup>

```
?- song.
One man went to mow,
Went to mow a meadow,
One man and his dog,
Went to mow a meadow.

Two men went to mow,
Went to mow a meadow,
Two men,
  one man and his dog,
Went to mow a meadow.
...
Seven men went to mow,
Went to mow a meadow,
Seven men,
  six men,
  five men,
  four men,
  three men,
  two men,
  one man and his dog,
Went to mow a meadow.

Action (h for help) ? abort
% Execution Aborted
```

Figure 4.3: Desired Behaviour of `song/0`

<sup>7</sup>Here we deliberately avoid asking for a *fixed* number of verses since otherwise the task would not be dissimilar enough to the one considered in Sect. 4.1: we could then produce a ‘last verse’ with relative ease and then proceed as before.

The core of the implementation is a predicate *song\_skeleton/1* which on backtracking returns the skeleton structure of each verse using numerals.

```
?- song_skeleton(Verse).
Verse = [1] ;
Verse = [2, 1] ;
Verse = [3, 2, 1] ;
...
```

**Exercise 4.6.** Define the predicate *song\_skeleton/1* by recursion.

*Hint.* You may model your definition of *song\_skeleton/1* on that of the predicate *int/1*, which on backtracking returns all natural numbers:

```
?- int(N).
N = 1 ;
N = 2 ;
N = 3 ;
...
```

The predicate *int/1* is defined in terms of an auxiliary predicate *int(+Int1, ?Int2)* by

```
int(N) :- int(1, N).
```

which on backtracking instantiates *Int2* to all integers starting from *Int1*:

```
?- int(5, I).
I = 5 ;
I = 6 ;
I = 7 ;
...
```

The definition of *int/2* is as follows.

```
int(I, I).
int(Last, I) :- succ(Last, New), int(New, I).
```

---

**Built-in Predicate:** *succ(?Int1, ?Int2)*

Succeeds if  $Int1 = Int2 + 1$ . Incrementation by *succ/2* is faster than by the usual arithmetic predicate.

---

■

There is in Prolog, as an alternative to recursion, the facility of failure driven, and repeat loops for the implementation of code with a repetitive behaviour. We want to illustrate this idea by way of a predicate *nat/1* which has the same specification as the predicate *int/1* from above but is defined in terms of a repeat loop rather than by recursion. Let *nat/1* be defined by

```
nat(N) :- first_nat, current_nat(N).  
nat(N) :- repeat, update_nat, current_nat(N).
```

with the auxiliary predicates

```
first_nat :- dynamic(current_nat/1),  
            retractall(current_nat(_)),  
            assert(current_nat(1)).
```

and

```
update_nat :- current_nat(N),  
              retractall(current_nat(_)),  
              NewN is N + 1,  
              assert(current_nat(NewN)).
```

The predicate *current\_nat/1* is used here to hold the current value of the natural number in the database as a fact. *first\_nat/0* clears the database of all facts defining *current\_nat/1* (possibly originating from earlier invocations of *nat/1*) and writes to the database the first natural number. *update\_nat/0* retrieves the previous value, clears the database, and writes back the updated value. The generation of an infinite stream of values by (the second clause of) *nat/1* hinges on the built-in predicate *repeat/0* which always succeeds on backtracking and is best thought of as returning a distinct (albeit invisible) ‘solution’ each time it is re-invoked. The conjunction of subgoals to the right of *repeat*, i. e.

```
update_nat, current_nat(N)
```

is re-satisfied on backtracking, resulting in an update of  $N$ . The database serves here as a ‘scratchpad’ for intermediate results.

**Exercise 4.7.** Define a second version of the predicate *song\_skeleton/1* by a *repeat loop*. Your solution should be modelled on the definition of *nat/1*.

■

There are of course other possibilities, too, for defining *song\_skeleton/1*. Take for example the one suggested by the following query.

```
?- current_prolog_flag(max_integer, _Largest),
   between(1, _Largest, _H), findall(_I, between(1, _H, _I), _R),
   reverse(_R, L).
L = [1] ;
L = [2, 1] ;
L = [3, 2, 1] ;
...
```

The list  $L$  is constructed here by:

- Getting hold of the largest number  $\_Largest$  which can be represented in SWI-Prolog as an integer.
- Obtaining the head  $\_H$  of  $L$  by the built-in predicate *between/3*.
- Creating the reverse  $\_R$  of  $L$  by the all-solutions predicate *findall/3*.
- And, finally, reversing  $\_R$  to get  $L$ .

A new  $L$  is obtained each time the query’s second goal is re-satisfied. This solution is neither concise nor is it as elegant as the earlier ones, however.

The remaining steps for the completion of *song/0* are spelt out in the Exercises 4.8 to 4.11 below.

**Exercise 4.8.** Define a predicate *digits(+Number, -List)* for converting a natural *Number* into the list of its digits in *List*:

```
?- digits(351, L).
L = [3, 5, 1]
```

(As an optional task which, however, is not needed in the present context, you may extend the definition of *digits/2* for the instantiation pattern *digits(-Number, +List)*.)

Now define a predicate *in\_words(+Num, -Atom)* for converting a numeral *Num* to its plain English equivalent in *Atom*. (Allow for up to 9,999 in *Num*.) Example:

```
?- in_words(351, A).
A = threehundredfiftyone8
```

■

**Exercise 4.9.** In the definition of the first and third lines of each verse you will need a predicate *capital/2* for converting the first character of an atom to its upper case equivalent:

---

<sup>8</sup>For reasons of simplicity, the rules of hyphenation and separating spaces are ignored here.



```
?- capital('sixteen men, fifteen men, fourteen men',C).
C = 'Sixteen men, fifteen men, fourteen men'
```

Define *capital/2*.

*Note.* Use the built-in predicate *atom\_chars/2* to disassemble atoms into lists and vice versa; see, inset on p. 126. For a concise solution to converting single letters to upper case you will also need the built-in predicate *char\_code/2*.<sup>9</sup>

---

**Built-in Predicate: *char\_code(?Char, ?ASCII)***

Converts the single-character atom *Char* to its ASCII code in *ASCII* and vice versa. Example:

```
?- char_code(a,ASCII).
ASCII = 97
?- char_code(Char,65).
Char = 'A'
```

---

**Exercise 4.10.** Define a predicate *line3/2* for generating the third line of each verse; for example, the third verse's third line we get by

```
?- line3([3,2,1],Text), write(Text).
Three men,
  two men,
  one man and his dog,
Text = 'Three men,\n  two men,\n  one man and his dog,'
```

In your work, you may be guided by the following query:

```
?- maplist(in_words,[16,15,14],[H|T]),
   maplist(atom_concat(' men, '),T,L), concat_atom([H|L],A),
   atom_concat(A,' men',A2).
H = sixteen
T = [fifteen, fourteen]
L = [' men, fifteen', ' men, fourteen']
A = 'sixteen men, fifteen men, fourteen'
A2 = 'sixteen men, fifteen men, fourteen men'
```

---

<sup>9</sup>A simpler but more tedious alternative is by using a predicate which is defined by 26 facts – one for each letter in the English alphabet.

---

**Built-in Predicate:** *atom\_concat(?Atom1,?Atom2,?Atom3)*

*Atom3* is the concatenation of *Atom1* and *Atom2*. At least two of the arguments must be instantiated. Alternatively, it suffices if the last argument is instantiated only. Examples:

```
?- atom_concat(atom1,atom2,A).  
A = atom1atom2  
?- atom_concat(A1,A2,atom3).  
A1 = '' A2 = atom3 ;  
A1 = a A2 = tom3  
Yes
```

---

**Exercise 4.11.** Complete the definition of *song/0* by using your predicates from the Exercises 4.6 to 4.10.



## 4.3 Chapter Notes

We have illustrated a practical Prolog development technique based on an incremental, exploratory and interactive working style. It is not dissimilar to the *Incremental Development Model* known from Software Engineering (e. g. [15]) the application of which in the commercial context results in *prototypes* at an early stage for evaluation and feedback. We have identified the following development stages in particular for predicates defined by *recursion*:

- Identify informally a *recursive* structure of the problem.
- Experiment *interactively* to explore and confirm the above.
- Identify a pattern and write *pseudo-code*.
- Write a *preliminary* (and perhaps incomplete) Prolog implementation.
- Refine details to arrive at a *final* Prolog implementation.

The method discussed here won't of course be a substitute for existing formal approaches to logic programming that are rooted in Mathematical Logic (e. g. [5]).



# Appendix A

## Solutions of Selected Exercises

### A.1 Chapter 1 Exercises

All Prolog source code for Chap. 1 is available in the file `accumulator.pl`.

**Exercise 1.1.** Define *from\_to/3* and its auxiliary *from\_to\_acc/3* by (P-A.1).

<i>Prolog Code P-A.1: Definition of from_to/3</i>	
1	<i>from_to</i> (M,N,L) :- (var(L); is_list(L)), <span style="float: right;">% clause 0</span>
2	integer(M), <span style="float: right;">%</span>
3	integer(N), <span style="float: right;">%</span>
4	M =< N, <span style="float: right;">%</span>
5	from_to_acc(M,[N],L), !. <span style="float: right;">%</span>
6	<i>from_to</i> (H,N,[H/T]) :- last(N,[H/T]), !, <span style="float: right;">% clause 1</span>
7	H =< N. <span style="float: right;">%</span>
8	from_to_acc(H,[H/T],[H/T]). <span style="float: right;">% clause 2</span>
9	from_to_acc(M,[H/T],L) :- NewHead is H - 1, !, <span style="float: right;">% clause 3</span>
10	from_to_acc(M,[NewHead,H/T],L). <span style="float: right;">%</span>

The annotated version of the hand computations from Fig. 1.4 is shown in Fig. A.1. The idea suggested by

$  \begin{array}{l}  \text{from\_to}(6,9,L) \xrightarrow{\textcircled{0}} \text{from\_to\_acc}(6,[9],L) \xrightarrow{\textcircled{3}} \\  \text{from\_to\_acc}(6,[8,9],L) \xrightarrow{\textcircled{3}} \text{from\_to\_acc}(6,[7,8,9],L) \xrightarrow{\textcircled{3}} \\  \text{from\_to\_acc}(6,[6,7,8,9],L) \xrightarrow{\textcircled{2}} L = [6,7,8,9] \xrightarrow{\textcircled{0}} \text{success}  \end{array}  $
--

Figure A.1: Annotated Hand Computations for *from\_to/3*

the hand computations is clearly reflected in the clauses 0, 2 and 3. It is instructive to consider the unexpected consequences of a slight (and perhaps innocent looking) change to clause 0. If we redefine clause 0 as shown here,

```

from_to(M,N,L) :- var(L),                % new clause 0
                  integer(M),             %
                  integer(N),             %
                  M =< N,                  %
                  from_to_acc(M,[N],L), !. %

```

then the predicate's pattern matching functionality will be corrupted:

```

?- from_to(6,9,[_,-,E/_]).
E = 9

```

(The third entry of the list `[6,7,8,9]` is clearly not `9`.) To explain this, we note that Prolog first tries the modified clause 0 which will fail since `[_,-,E/_]` is not a variable but a *compound* term.<sup>1</sup>

```

?- var([_,-,E/_]).
No

```

---

<sup>1</sup>Lists are compound terms with the functor `'.'` (dot). More on this will be found in Sect. 2.2.1.

Next, clause 1 is tried, which then succeeds as indicated by the query below.

```
?- (6,9,[_,_,E/_]) = (H,N,[H/T]), last(N,[H/T]), !, H =< N.
E = 9
H = 6
N = 9
T = [_G269, 9]
```

Why? Well, for the first goal of this query to succeed,  $[H/T]$  has to have at least three entries, requiring  $T$  be of length at least two. The second goal then succeeds with  $T$  as a two-element list (whose first entry is a system chosen internal variable):

```
?- last(9,[6/T]).2
T = [9] ;
T = [_G269, 9] ;
T = [_G269, _G272, 9] ;
...
```

Therefore,  $[H/T]$  will be unified with  $[6, \_G269, 9]$ . Now, the unification  $[\_, \_, E/\_] = [H/T]$  (still in force from the first goal) requires that  $E$  be unified with the third entry of  $[6, \_G269, 9]$ , i.e. with  $9$ .

We note in passing that the predicate *numlist*/3 in SWI-Prolog, Version 5.2.7, has almost the same functionality as our *from\_to*/3. (The instantiation pattern *numlist*(-Low, -High, +List) has not been implemented there.)

**Exercise 1.2.** The new version, *nums*/2, is defined in (P-A.2).

**Prolog Code P-A.2: Definition of *nums*/2**

```
1 nums(Atom,N) :- atom_codes(Atom,Values),           % clause 0
2               nums([47|Values],0,N), !.             %
3
4 nums([],N,N).                                     % clause 1
5 nums([_],N,N).                                    % clause 2
6 nums([H,E|T],Acc,N) :- not(digit(H)), digit(E),    % clause 3
7                       NewAcc is Acc + 1,           %
8                       !, nums([E|T],NewAcc,N).      %
9 nums([_,E|T],Acc,N) :- nums([E|T],Acc,N).          % clause 4
```

- We prefix in clause 0 with the ASCII *Values* with '47', an arbitrary non-digit code, in case the leftmost character was a digit. (Otherwise, the first group of digits will be missed.)
- The first two goals of clause 3 provide the condition for incrementing the accumulator.

**Exercise 1.3.** The pseudocode is shown as Algorithm A.1.1; the correspondence between the pseudocode's statements and the Prolog clauses in Example 1.6 is displayed in Table A.1.

<sup>2</sup>We are using SWI-Prolog, Version 3.4.5 here. In the latest version also available at the time of writing (Version 5.2.7), for some inexplicable reason the order of the arguments of *last*/2 is the other way round.

**Algorithm A.1.1:** NUMBERS(*Atom*)

```

Values ← list of ASCII values of characters in Atom      (1)
Acc ← 0                                                  (2)
Switch ← nodigit                                       (3)
while Values ≠ []
do { [H|T] ← Values                                     (4)
    if H is an encoded digit
    then { if Switch = nodigit                          (5)
           then { Acc ← Acc + 1                        (6)
                 Switch ← digit                        (7)
           else { Switch ← nodigit                     (8)
                 Values ← T                            (9)
    }
    N ← Acc                                             (10)
return (N)

```

Statement	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)
Clause	0	0	0	2, 3, 4	2	2	2, 3	4	2, 3, 4	1

Table A.1: Algorithm A.1.1 &amp; Prolog Clause Correspondence (Example 1.6)

**Exercise 1.4.** A simple tail recursive definition for *mult/3* is by (P-A.3).

**Prolog Code P-A.3:** Definition of *mult/3* by recursion

```

1 mult(_, [], []).
2 mult(C, [H|T], [P|Ps]) :- P is C * H, !,
3                           mult(C, T, Ps).

```

An alternative definition using accumulators is suggested by the hand computations in Fig. A.2, giving rise to (P-A.4).

```

mult(0.2, [5.0, 10.5, 2.5], L)  $\rightsquigarrow^{\textcircled{0}}$  mult(0.2, [5.0, 10.5, 2.5], [], L)  $\rightsquigarrow^{\textcircled{2}}$ 
mult(0.2, [10.5, 2.5], [1.0], L)  $\rightsquigarrow^{\textcircled{2}}$  mult(0.2, [2.5], [2.1, 1.0], L)  $\rightsquigarrow^{\textcircled{2}}$ 
mult(0.2, [], [0.5, 2.1, 1.0], L)  $\rightsquigarrow^{\textcircled{1}}$  reverse([0.5, 2.1, 1.0], L)  $\rightsquigarrow$ 
L = [1.0, 2.1, 0.5]  $\rightsquigarrow^{\textcircled{0}}$  success

```

Figure A.2: Hand Computations for *mult/3*



**Prolog Code P-A.4: *mult/3* by the accumulator technique**

```

1 mult(C,List,L) :- mult(C,List,[],L).           % clause 0
2 mult(_,[],Acc,L) :- reverse(Acc,L).             % clause 1
3 mult(C,[H|T],Acc,L) :- A is C * H, !,          % clause 2
4                        mult(C,T,[A|Acc],L).

```

Timing by *time/1* will show that simple recursion delivers a better performance. *mult/3* is an example of a *mapping* operation where each entry of the input list is mapped by some function to the corresponding entry of the output list. (*add/3* is defined analogously.)

**Exercise 1.5.** Replace clause 1 in (P-1.13), p. 30, (the definition of *pta/2*) by the following two clauses.

```

pta(in(_,-,_,Ws,Acc),out(Ws,I)) :- integer(I),
                                   Acc := I, !.
pta(in(_,-,Ps,Ds,Ws,Acc),out(Ws,I)) :- var(I),
                                   classify_all(Ps,Ws,Ds),
                                   I = Acc, !.

```

If a fixed number of iterations *I* is wanted, the stopping criterion requires that the accumulator be numerically equal to *I*. The alternative stopping criterion is, as before, that all points be correctly classified.

## A.2 Chapter 2 Exercises

All Prolog source code for Chap. 2 is available in the file *d1.pl*.

**Exercise 2.1.** *sharp/2* is defined by recursion in (P-A.5).

**Prolog Code P-A.5: Definition of *sharp/2***

```

1 sharp(E,E)                                :- not(proper_list(E)), !.
2 sharp([], []).
3 sharp([E],#(Term,[]))                    :- sharp(E,Term), !.
4 sharp([H|T],#(Term1,Term2))              :- sharp(H,Term1),
5                                           sharp(T,Term2).

```

Perhaps the order of the two boundary case clauses should be given some thought. As it stands, the *sharp*-notation of a list with a single entry of a free variable is correctly evaluated:

```

?- sharp([E],S).
E = _G210
S = #(_G210, []) ;
No

```

However, on interchanging the first two clauses in (P-A.5), we get an incorrect response:

```

?- sharp([E],S).
E = []
S = #([], []) ;
No

```

**Exercise 2.2.** *lf/2* is defined in (P-A.6).

**Prolog Code P-A.6: Definition of *lf/2***

```
1 lf(Term,Term)      :- var(Term), !.           % clause 1
2 lf(#(Term,_),Term) :- not(funcator(Term,#,2)), % clause 2
3                   Term \= [].                 %
4 lf(#(Term,_),Leaf) :- lf(Term,Leaf).          % clause 3
5 lf(#(_,Term),Leaf) :- lf(Term,Leaf).          % clause 4
```

(P-A.6) admits the following *declarative* reading:

- Clause 1: Variables are leaves.

- Clause 2: *Term* is the left-hand leaf of  $\#(Term, \_)$  if *Term* is not a list<sup>3</sup> of length at least 1 nor is *Term* the empty list. (Notice that in a more precise interpretation of clause 2, the phrase ‘is not’ should be replaced by ‘cannot be unified with’. However, this change in interpretation makes a real difference only if *lf/2* is invoked with an unbound variable in its first argument, a case which will have been caught by clause 1.)<sup>4</sup>
- Clause 3: *Leaf* is a left-hand leaf of  $\#(Term, \_)$  if *Leaf* is a left-hand leaf of its (left-hand) branch *Term*.
- Clause 4: *Leaf* is a left-hand leaf of  $\#(\_, Term)$  if *Leaf* is a left-hand leaf of its (right-hand) branch *Term*.

**Exercise 2.3.** The definition of a first version of *flatten/2* is shown in (P-A.7).

**Prolog Code P-A.7: A first version of *flatten/2***

```
1 flatten_1(L,F) :- sharp(L,S), bagof(Leaf,lf(S,Leaf),F).
```

The discussion on p. 46 shows that the use of the dot-notation for displaying lists can be achieved by the predicate *set\_prolog\_flag/2*. Close scrutiny of the Exercises 2.1 to 2.3 (and their solutions) will in fact reveal that we can implement *flatten/2* also directly, i.e. without recourse to our sharp-notation; such a version is defined in (P-A.8).

**Prolog Code P-A.8: A second version of *flatten/2***

```
1 leaf(Term,Term)      :- var(Term), !.
2 leaf(.(Term,_),Term) :- not(funcator(Term,.,2)),
3                        Term \= [].
4 leaf(.(Term,_),Leaf) :- leaf(Term,Leaf).
5 leaf(._(Term),Leaf)  :- leaf(Term,Leaf).
6 flatten_2(L,F) :- bagof(Leaf,leaf(L,Leaf),F).
```

The above two versions of *flatten/2* behave identically to the built-in one; for example,

```
?- flatten_1([a,[Y,[b,X]],c,f(X)],L).
Y = _G339
X = _G345
L = [a, _G339, b, _G345, c, f(_G345)]

?- flatten_2([a,[Y,[b,X]],c,f(X)],L).
Y = _G339
X = _G345
L = [a, _G339, b, _G345, c, f(_G345)]

?- flatten([a,[Y,[b,X]],c,f(X)],L).
Y = _G330
X = _G336
L = [a, _G330, b, _G336, c, f(_G336)]
```

<sup>3</sup>‘Lists’ are understood here to be in terms of the sharp-notation.

<sup>4</sup>In the absence of clause 1, however, a query like *lf(#(X,[]),Leaf)*. will cause stack overflow since clause 2 will fail and clause 3 will cause looping as can be inferred from

```
?- #(Term,_) = X.
Term = _G219
X = #(_G219, _G220)
```

It is seen in particular that a free variable occurring more than once in the nested list will be unified, as expected, with the *same* internal variable. This would not have been so, however, had we used the built-in predicate *findall/3* (in lieu of *bagof/3*) for collecting the leaves from the list's tree representation:

```
?- findall(Leaf, leaf([a, [Y, [b, X]], c, f(X)], Leaf), Leaves).
Leaf = _G480
Y = _G456
X = _G462
Leaves = [a, _G641, b, _G629, c, f(_G617)]
```

**Exercise 2.4.** The definition of *dot/1* in (P-A.9) follows the suggested route.

**Prolog Code P-A.9: Definition of *dot/1***

```
1 dot(List) :- sharp(List, Term),
2             term_to_atom(Term, A1),
3             atom_chars(A1, L1),
4             sharps_to_dots(L1, L2),
5             concat_atom(L2, A2),
6             write_term(A2, []).
```

The predicate *sharps\_to\_dots/2* is defined by the accumulator technique in (P-A.10).

**Prolog Code P-A.10: Definition of *sharps\_to\_dots/2***

```
1 sharps_to_dots(S, D) :- sharps_to_dots(S, [], R),
2                         reverse(R, D), !.
3 sharps_to_dots([], L, L).
4 sharps_to_dots([#|T], Acc, L) :- sharps_to_dots(T, [.|Acc], L).
5 sharps_to_dots([H|T], Acc, L) :- sharps_to_dots(T, [H|Acc], L).
```

A more concise alternative is offered by the use of the built-in *maplist/3*; this is shown in (P-A.11).

**Prolog Code P-A.11: Alternative definition of *sharps\_to\_dots/2***

```
1 sharps_to_dots(S, D) :- maplist(sharp_to_dot, S, D).
2 sharp_to_dot(_, '._') :- !.
3 sharp_to_dot(C, C).
```

**Exercise 2.5.** The improved version is defined in (P-A.12).

**Prolog Code P-A.12: Definition of *flatten\_4/2***

```
1 flatten_4(X, [X]) :- var(X), !. % clause 0
2 flatten_4([], []). % clause 1
3 flatten_4([H|T], L1) :- flatten_4(H, L2), % clause 2
4                        flatten_4(T, L3), %
5                        append(L2, L3, L1), !. % cut added here
6 flatten_4(X, [X]). % clause 3
```

Clauses 1 to 3 are essentially as in *flatten\_3/2*. (The *cut* in clause 2 has been added to achieve a unique solution.) To rectify the other problem with *flatten\_3/2*, we have to understand why it produces spurious solutions on backtracking. When *flatten\_3/2* arrives at a list entry which is a variable, it will first unify the variable with the empty list and then on further backtracking with *[H/T]* where *H* and *T* are themselves *variables*. Because of the recursive definition, this will then give rise to further such erroneous unifications. To avoid this, we simply ‘catch’ a variable first argument by clause 0. *flatten\_4/2* thus defined behaves as expected:

```
?- flatten_4([a,[Y,[b,X]],c,f(X)],L).
Y = _G339
X = _G345
L = [a, _G339, b, _G345, c, f(_G345)] ;
No
```

**Exercise 2.6.** The following additional clause (an analogue of clause 0 in the definition of *flatten\_4/2*) will become the first clause in *flatten\_d1/2*:

```
flatten_d1(X,[X/T]-T) :- var(X), !.
```

**Exercise 2.7.** We define in (P-A.13) *nested/2* in terms *nested/4* whose second and third argument are a counter and an accumulator, respectively.

**Prolog Code P-A.13: Definition of *nested/2***

```
1 nested(M,L) :- nested(M,1,[1],L), !.
2 nested(M,M,L,L).
3 nested(M,N,Acc,L) :- NewN is N + 1,
4                       nested(M,NewN,[Acc,NewN],L).
```

The versions’ relative performance is illustrated below. It is seen in particular that the one based on difference lists is nearly as good as the built-in version.

```
?- nested(8000,_L), time(flatten(_L,_F)).
% 95,999 inferences in 0.44 seconds (218180 Lips)
?- nested(8000,_L), time(flatten_1(_L,_F)).
% 216,004 inferences in 12.96 seconds (16667 Lips)
?- nested(8000,_L), time(flatten_2(_L,_F)).
% 144,007 inferences in 12.79 seconds (11259 Lips)
?- nested(8000,_L), time(flatten_3(_L,_F)).
% 335,514 inferences in 9.88 seconds (33959 Lips)
ERROR: Out of global stack
?- nested(8000,_L), time(flatten_5(_L,_F)).
% 32,000 inferences in 0.93 seconds (34409 Lips)
```

Furthermore, it is seen that version 3, the implementation using list concatenation with *append/3*, is not practically viable due to stack overflow. (This problem has been experienced even for a nesting depth of 1000.)

**Exercise 2.8.** Your session will typically look like this:

```
?- findall(_N,between(1,2000,_N),_L), time(reverse_1(_L,_R)).
% 2,003,001 inferences in 19.34 seconds (103568 Lips)
?- findall(_N,between(1,2000,_N),_L), time(reverse_2(_L,_R)).
% 2,002 inferences in 0.00 seconds (Infinite Lips)
?- findall(_N,between(1,2000,_N),_L), time(reverse_3(_L,_R)).
% 4,000 inferences in 0.06 seconds (66667 Lips)
?- findall(_N,between(1,2000,_N),_L), time(reverse_4(_L,_R)).
% 2,002 inferences in 0.05 seconds (40040 Lips)
```

It is seen that the ‘naïve’ implementation is far less efficient than either of the other three. Furthermore, version 4 is seen to behave in the same way as the one using accumulators (which is the method used also to implement the built-in version). This is not surprising since these two implementations were shown to be identical in Sect. 2.3.2.

### Exercise 2.9.

#### *Declarative Reading.*

The difference list  $L-X$  is the reverse of the list  $[E1, E2/T]$  if the difference list  $L-[E2, E1/X]$  is the reverse of  $T$ .

**New Version.** This is defined in (P-A.14).

**Prolog Code P-A.14: Definition of *reverse\_5/2***

```

1 reverse_5(L,R) :- rev_dl_3(L,R-[]).
2 rev_dl_3([],L-L).                                % clause 0
3 rev_dl_3([X],[X|L]-L).                            % clause 1
4 rev_dl_3([E1,E2|T],L1-L2) :- rev_dl_3(T,L1-[E2,E1|L2]). % clause 2

```

Noteworthy is in (P-A.14) the fact that reversal is carried out in ‘chunks of twos’ resulting in fewer invocations of the auxiliary predicate. There are now two boundary clauses: if the list to be reversed has an even number of entries then clause 0 is used; otherwise, clause 1 applies.

**Unfolding.** We are going to show here that the clauses 0–2 can be inferred from the clauses (b1)–(b2).<sup>5</sup>

The boundary clause 0 is identical to clause (b1).

We infer clause 1 by an elementary unfolding operation on the only goal in clause (b2): we first rewrite clause (b1) as

```
rev_dl([],L-L) :- true.
```

and then seek to unify its head with the goal in the body of clause (b2):

```

?- rev_dl([],L-L) = rev_dl(T,L1-[H|L2]).
L = [_G360|_G361]
T = []
L1 = [_G360|_G361]
H = _G360
L2 = _G361
Yes

```

The unification succeeds and gives rise to the clause

```
rev_dl([_G360|[]],[_G360|_G361]-_G361) :- true.
```

which is equivalent to clause 1.

To infer now clause 2, we rewrite clause (b2) as

```
rev_dl([U|V],W1-W2) :- rev_dl(V,W1-[U|W2]).
```

and seek to unify the head of this new clause with the goal in clause (b2):<sup>6</sup>

```

?- rev_dl([U|V],W1-W2) = rev_dl(T,L1-[H|L2]).
U = _G384
V = _G385
W1 = _G387
W2 = [_G393|_G394]
T = [_G384|_G385]

```

<sup>5</sup>For the present purposes, the version number (i.e. the suffix ‘\_3’) is to be ignored.

<sup>6</sup>This is an instance of *self unfolding*.

```
L1 = _G387
H = _G393
L2 = _G394
Yes
```

The unification succeeds and gives rise to

$$\text{rev\_dl}([H/T], L1-L2) \text{ :- rev\_dl}(V, W1-[U/W2]).$$

which in terms of Prolog's internal variable names reads as follows.

```
rev_dl([_G393|[_G384|_G385]], _G387-_G394) :-
  rev_dl(_G385, _G387-[_G384|[_G393|_G394]]).
```

The latter clause is readily recognized as clause 2. This second and final elementary unfolding operation concludes a complete one step unfolding, thus making clause (b2) redundant.

**Speed of Execution.** The enhanced version is twice as fast as the previous one:

```
?- findall(_N, between(1, 100000, _N), _L), time(reverse_5(_L, _R)).
% 50,002 inferences in 0.61 seconds (81970 Lips)
?- findall(_N, between(1, 100000, _N), _L), time(reverse_4(_L, _R)).
% 100,002 inferences in 1.92 seconds (52084 Lips)
```

**Further Enhancement.** Modify the implementation by processing the input list in chunks of threes; this is shown in (P-A.15).

**Prolog Code P-A.15: Definition of reverse\_6/2**

```
1 reverse_6(L,R) :- rev_dl_4(L,R-[]).
2 rev_dl_4([],L-L).
3 rev_dl_4([E1], [E1|L]-L).
4 rev_dl_4([E1,E2], [E2,E1|L]-L).
5 rev_dl_4([E1,E2,E3|T], L1-L2) :- rev_dl_4(T, L1-[E3,E2,E1|L2]).
```

It is seen that three base cases are needed now, defining explicitly the reversal of lists with *up to two* entries. The gain in speed is illustrated by the query below.

```
?- findall(_N, between(1, 100000, _N), _L), time(reverse_6(_L, _R)).
% 33,335 inferences in 0.50 seconds (66670 Lips)
```

**Generalization.** Provide  $n$  base cases catering for the reversal of lists with *up to*  $n - 1$  entries and write a recursive clause for reversing lists with *at least*  $n$  entries.

**Exercise 2.10, part (a).** We convert *colour/4* to its difference lists based form by (P-A.16).



**Prolog Code P-A.16: Definition of colour\_dl/4**

```

1 colour_dl([],R-R,W-W,B-B).
2 colour_dl([col(Object,red)|T],
3           [col(Object,red)|R1]-R2,W1-W2,B1-B2) :-
4     colour_dl(T,R1-R2,W1-W2,B1-B2).
5 colour_dl([col(Object,white)|T],
6           R1-R2,[col(Object,white)|W1]-W2,B1-B2) :-
7     colour_dl(T,R1-R2,W1-W2,B1-B2).
8 colour_dl([col(Object,blue)|T],
9           R1-R2,W1-W2,[col(Object,blue)|B1]-B2) :-
10    colour_dl(T,R1-R2,W1-W2,B1-B2).

```

The concatenation of the three output difference lists is accomplished by

```
dijkstra_dl(Items,L1-L4) :- colour_dl(Items,L1-L2,L2-L3,L3-L4).
```

*dijkstra/2* is now defined as in Sect. 2.4.3,

```
dijkstra(Items,Grouped) :- dijkstra_dl(Items,Grouped-[]).
```

Timing by *time/1* will confirm that the difference list based version of each implementation is better (as measured by the number of inferences used) than its plain counterpart. The last version is the best as it uses difference lists and takes a single pass through the input list.

**Exercise 2.10, part (b).** Add the clauses

```

colour([col(_,Colour)|T],R,W,B) :- Colour \= red,
                                   Colour \= white,
                                   Colour \= blue,
                                   colour(T,R,W,B).

```

and

```

colour_dl([col(_,Colour)|T],R1-R2,W1-W2,B1-B2) :-
  Colour \= red,
  Colour \= white,
  Colour \= blue,
  colour_dl(T,R1-R2,W1-W2,B1-B2).

```

to the respective existing definitions.

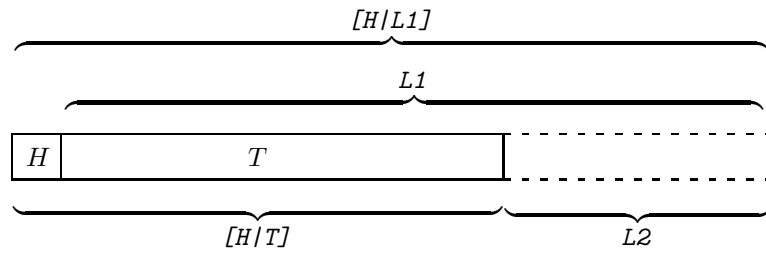
**Exercise 2.11.** Carry out a clause-by-clause ‘translation’ of *averages/2* and allied predicates to get (P-A.17).

**Prolog Code P-A.17: Definition of averages\_dl/2**

```

1 averages_dl(L1-L2,A1-A2) :- aver_dl([-1,1|L1]-L2,A1-A2), !.
2 aver_dl([_,0,_|X]-Y,X-Y).
3 aver_dl(X1-X2,ADL) :- av_rotate_dl(X1-X2,Y1-Y2),
4                       aver_dl(Y1-Y2,ADL).
5 av_rotate_dl([H1,H2|Y]-[Last|Z],[H2|Y]-Z) :- Last is (H1 + H2)/2.

```

Figure A.3: Illustrating the Second Clause of *dl/2*

**Exercise 2.12.** Clause 2 in (P-2.19) is illustrated by Fig. A.3. It admits the following declarative interpretation:

The difference list version of  $[H/T]$  is  $[H/L1]-L2$  if the difference list version of  $T$  is  $L1-L2$ .

**Exercises 2.13 & 2.14.** The first implementation is by (P-A.18).

**Prolog Code P-A.18: Definition of `show_matrix_dl/1`**

```

1 show_matrix_dl(M-[]):- show_matrix(M), nl.      % clause 0
2 show_matrix([]).                                % clause 1
3 show_matrix([H-[]|T]) :- write(H), write(' '), % clause 2
4                               show_matrix(T).    %

```

In clause 0, the argument of `show_matrix_dl` (which expects a difference list of difference lists) is converted to a *proper* list of difference lists. This then is displayed entry-wise by `show_matrix/1`, defined in the clauses 1 and 2. Noteworthy is clause 2 where the matrix head is unified with `H-[]` thereby making `H` a *proper* list which in turn is displayed on the terminal.

Invoking `show_matrix_dl(M1-M2)` with a difference list `M1-M2` will of course unify `M2` with the empty list. This can't be 'undone' later and therefore any subsequent attempt of using `M1-M2` as a genuine difference list will fail. We solve this problem by not displaying the original difference list `M1-M2` but a copy of it which we write to the database prior to the invocation of `show_matrix_dl/2`. The improved version `show_matrix_dl2/2` is defined in (P-A.19).

**Prolog Code P-A.19: Definition of `show_matrix_dl2/1`**

```

1 show_matrix_dl2(DLM):- dynamic(matrix/1),
2                        retractall(matrix(_)),
3                        assert(matrix(DLM)),
4                        matrix(M),
5                        show_matrix_dl(M).

```

It will behave as expected:

```

?- matrix_a(A), dl2(A,_DLA), show_matrix_dl2(_DLA),
   rot_matrix_dl(_DLA,_DLR), show_matrix_dl2(_DLR).
[a11, a12, a13, a14] [a21, a22, a23, a24] [a31, a32, a33, a34]
[a22, a23, a24, a21] [a32, a33, a34, a31] [a12, a13, a14, a11]

```

You will find more on database operations in Sect. 3.1.

In the above approach, a *copy* of the term holding the matrix in difference list form was written to and later retrieved from the database. Subsequently, the new copy (or parts of it) may be unified with some other term without affecting the original. There is a built-in predicate to achieve just that; it is `copy_term/2` (see inset).

**Built-in Predicate: `copy_term(+TermIn,-TermOut)`**

The term in *TermIn* is copied to *TermOut*. Each of the free variables in *TermIn* is given a new (internal) name and subsequently no link is maintained between the two terms. Example:

```

?- copy_term(f(a,X),Y), X = b.
X = b
Y = f(a, _G386)
Yes

```

A new version of `show_matrix_dl/1` is defined in (P-A.20).

**Prolog Code P-A.20: Definition of `show_matrix_dl3/1`**

```
1 show_matrix_dl3(DLM):- copy_term(DLM,M),
2                        show_matrix_dl(M).
```

It will be found to respond exactly as `show_matrix_dl2/1` did.

**Exercise 2.15.** Add to the database the clause

```
g_seidel(in([[First|Rest1]-Rest2|A1]-A2,
            [B|B1]-[B|B2],[_|T1]-[NewX|T2],[S|S1]-[S|S2]),
out(NewAs,B1-B2,T1-T2,S1-S2)) :-
    dot_product_dl(Rest1-Rest2,T1-[NewX|T2],P),7
    NewX is B - P,
    rot_matrix_dl([[First|Rest1]-Rest2|A1]-A2,NewAs).
```

to enable `g_seidel/2` to work also with difference lists. (Notice that this new clause won't interfere with the earlier definition.) No other changes are necessary since `g_seidel/7` will call this modified version of `g_seidel/2` as before:

```
?- a(A), b(B), x0(X), s(S),
   dl2(A,ADL), dl(B,BDL), dl(X,XDL), dl(S,SDL),
   g_seidel(ADL,BDL,XDL,SDL,50,NewX-[],NewS-[]).
...
NewX = [62.5, 62.5, 87.5, 87.5]
NewS = [3, 4, 1, 2]
```

To simplify the query, we may use the new version of `g_seidel/7`, defined in (P-A.21).

**Prolog Code P-A.21: New version of `g_seidel/7`**

```
1 g_seidel_2(A,B,X,S,I,NewX,NewS) :-
2   dl2(A,ADL),
3   dl(B,BDL),
4   dl(X,XDL),
5   dl(S,SDL),
6   g_seidel(ADL,BDL,XDL,SDL,I,NewX-[],NewS-[]), !.
```

(This version uses the same pattern of proper list inputs as `g_seidel/7` but works *internally* with difference lists.)

<sup>7</sup>The dot product of vectors in difference list notation is defined by the accumulator technique as follows

```
dot_product_dl(DL1,DL2,Result) :- dot_product_dl(DL1,DL2,0,Result), !.

dot_product_dl(L-_,_,Acc,Acc) :- var(L).
dot_product_dl([HU|TU1]-TU2,[HV|TV1]-TV2,Acc,Result) :-
    NewAcc is Acc + HU * HV, !,
    dot_product_dl(TU1-TU2,TV1-TV2,NewAcc,Result).
```

Experiments will show that the new implementation always needs a lesser number of inferences. However, for the CPU-time also to show a relative improvement, the problem has to be of a minimum size. (Difference lists carry a certain computational overhead worth paying for problems beyond a certain size only.)

## A.3 Chapter 3 Exercises

Prolog source code: for Sect. 3.1, see `party.pl`, `people.pl`, `arrange.pl` and `queue.pl`; for Sect. 3.2, see `transformations.pl`; for Sect. 3.3, see `dl.pl` and `transformations.pl`.

**Exercise 3.1, part (f).** *facing/3* is recursively defined by

```
facing(X,L,R) :-
    right_to(L,X), right_to(X,R), (L == R, !; true).
facing(X,L,R) :-
    facing(X,Y,Z), right_to(L,Y), right_to(Z,R), (L == R, !; true).
```

The declarative reading of this definition should be straightforward in conjunction with Fig. 3.2. Recursion stops when the last two arguments of *facing/3* are instantiated to identical terms. For an *odd* number of guests, *facing/3* will stop once the second and third arguments are identical to the first:

```
?- listing(right_to/2).
right_to(clara, adam).
right_to(adam, susan).
right_to(susan, clara).

?- facing(adam,Left,Righ).
Left = clara Righ = susan ;
Left = susan Righ = clara ;
Left = adam Righ = adam ;
No
```

Define now *opposite\_to/2* by

```
opposite_to(X,Y) :- facing(X,Y,Y), X \== Y.
```

(The second goal ensures failure for an odd number of guests.)

**Exercise 3.2.** (P-A.22) shows the definition of *opposites/0*; *guests/0* is defined analogously.

### Prolog Code P-A.22: Definition of *opposites/0*

```
1 opposites :- opposite_to(_,_),
2             ((right_to(X,Y),
3               opposite_to(X,Z),
4               write(X), write(', '), write(Z), nl,
5               fail); true).
```

*Observations.* *opposites/0* will succeed iff *opposites\_to/2* does, i.e. if there are an even number of names in the database. From inside a failure driven loop all opposite pairs are displayed and success is enforced by disjunction with *true*.

(P-A.23) defines *look\_right/1* in terms of an auxiliary predicate *look\_right/2*. In the second argument of this predicate the list of names is accumulated until the person's name reappears in the head.

**Prolog Code P-A.23: Definition of *look\_right/1***

```
1 look_right(Pers) :- look_right(Pers,[Pers|T]),  
2                     reverse(T,List),  
3                     write_list(List).  
  
4 look_right(Pers,[X,Pers]) :- right_to(Pers,X).  
5 look_right(Pers,[X,H|T])  :- right_to(H,X),  
6                             look_right(Pers,[H|T]).
```

*write\_list/1* is defined by recursion (not shown here) and displays the entries of a list in a single line.

**Exercise 3.3, part (a).** Don't change the database if one or two people are at the table:

```
swap_neighbours(Pers1,Pers2) :- right_to(Pers1,Pers2),  
                                right_to(Pers2,Pers1).
```

Changes are due if more than two people are at the table:

```

swap_neighbours(Left,Right) :- right_to(Left,Right),
                               right_to(L,Left),
                               right_to(Right,R),
                               retract(right_to(Left,Right)),
                               retract(right_to(L,Left)),
                               retract(right_to(Right,R)),
                               assert(right_to(Right,Left)),
                               assert(right_to(L,Right)),
                               assert(right_to(Left,R)).

```

**Exercise 3.3, part (b).** Use *swap\_neighbours/2* for swapping neighbours:

```

swap(Pers1,Pers2) :- swap_neighbours(Pers1,Pers2).
swap(Pers1,Pers2) :- swap_neighbours(Pers2,Pers1).

```

And, do changes as necessary for swapping people who aren't neighbours:

```

swap(Pers1,Pers2) :- right_to(Pers1,R1),
                     right_to(L1,Pers1),
                     right_to(Pers2,R2),
                     right_to(L2,Pers2),
                     retract(right_to(Pers1,R1)),
                     retract(right_to(L1,Pers1)),
                     retract(right_to(Pers2,R2)),
                     retract(right_to(L2,Pers2)),
                     assert(right_to(Pers1,R2)),
                     assert(right_to(L2,Pers1)),
                     assert(right_to(Pers2,R1)),
                     assert(right_to(L1,Pers2)).

```

**Exercise 3.4, part (a).** Only one of the four cases in Table 3.1 will be discussed here: the last two customers swap places and there are more than two customers in the queue (Fig. A.4). The relations of interest which can

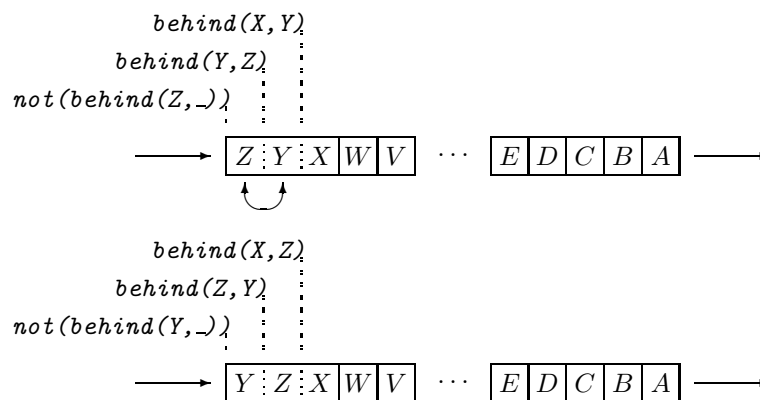


Figure A.4: The Last Two Customers Swap Places

be inferred from the database *before* and *after* the swap are indicated in Fig. A.4. The corresponding clause of *swap\_neighbours/2* is therefore

```

swap_neighbours(Y,Z) :- % swap Y and Z
    not(behind(Z,_)),    % Z is the last in the queue
    behind(Y,Z),        % Z is behind Y in the queue
    behind(X,Y),        % Y is behind X in the queue
    retract(behind(Y,Z)), % remove relation between Y and Z
    retract(behind(X,Y)), % remove relation between X and Y
    assert(behind(X,Z)),  % establish relation between X and Z
    assert(behind(Z,Y)). % establish relation between Z and Y

```

(You should complete the remaining three clauses with reference to Table 3.1 and by using sketches similar to Fig. A.4.)

**Exercise 3.5.** The intended database changes are achieved by a failure driven loop:

```

?- dynamic(lives_in/2),
   ((lives_in(london,_Person), assert(lives_in(york,_Person)),
    fail); true), retractall(lives_in(london,_)).

```

**Exercise 3.6.** The definition of *joins/1* is fairly straightforward: check first that there aren't any facts in the database for *right\_to/2*; then assert the appropriate fact for *right\_to/2*; finally, augment the file *people.pl* and report the job completed.

```

joins(Pers) :- not(right_to(_,_)),
               assert(right_to(Pers,Pers)),
               tell('people.pl'), listing(right_to/2), told,
               write(Pers), write(' has joined the table. '), nl.

```

**Exercise 3.7.** The task is to enhance the definition of the second clause of *save\_predicates\_to(+Filename,+List)*. As a first step, we translate the informal specification as follows:

$$\text{Condition} \rightarrow \text{Action} ; \text{Alternative Action} \quad (\text{A.1})$$

with

$$\text{Condition} = \forall x(A(x) \rightarrow B(x)) \quad (\text{A.2})$$

$$A(x) = x \in \text{List} \quad (\text{A.3})$$

$$B(x) = \text{my\_predicate}(x, -) \quad (\text{A.4})$$

$$\text{Action} = \text{write to file} \quad (\text{A.5})$$

$$\text{Alternative Action} = \text{display error message} \quad (\text{A.6})$$

Since it is more difficult to implement in *standard Prolog* a universally quantified condition than an existentially quantified one, we write (A.1) in terms of the *negation* of (A.2), thereby getting

$$\text{Condition} = \neg(\forall x(A(x) \rightarrow B(x))) \quad (\text{A.7})$$

$$\text{Action} = \text{display error message} \quad (\text{A.8})$$

$$\text{Alternative Action} = \text{write to file} \quad (\text{A.9})$$



Rewrite now the right-hand side of (A.7) as follows:<sup>8</sup>

$$\begin{aligned}
 \text{Condition} &= \exists x \neg (A(x) \rightarrow B(x)) \\
 &= \exists x \neg (B(x) \vee \neg A(x)) \\
 &= \exists x (A(x) \wedge \neg B(x))
 \end{aligned} \tag{A.10}$$

A Prolog implementation of `save_predicates_to(+Filename,+List)` based on (A.1), (A.3)–(A.4) and (A.8)–(A.10) is therefore

```

save_predicates_to(Filename,List) :-
    (member(X,List), not(my_predicate(X,_))) -> (write('...'),
                                                    nl,
                                                    fail);
    write_to_file(Filename,List).

```

where `write_to_file/2` is defined by a failure driven loop:

---

<sup>8</sup>The rules hereby used are from Predicate and Propositional Calculus; they are in turn: a *Quantifier Equivalence Rule*, *Material Implication* and *DeMorgan's Rule*.

```

write_to_file(Filename,List) :- tell(Filename),
                               ((member(Fun/Arity,List),
                                listing(Fun/Arity),
                                fail); true),
                               told.

```

**Alternative Solution of Exercise 3.7.** The built-in *SWI Prolog* predicate *forall(+Condition,+Action)* allows a direct implementation of the *Condition* in (A.2). The resulting alternative definition of *save\_predicates\_to/2* is then

```

save_predicates_to(Filename,List) :-
    (forall(member(X,List),
            my_predicate(X,_)) -> write_to_file(Filename,List));
    write('...'), nl, fail.

```

(Two possibilities are discussed in [16] for defining *forall/2*.)

**Exercise 3.9.** The directive `:- dynamic(album/1).` in the source file will make *album/1* a *dynamic* predicate. Now use the query

```

?- retractall(album([stamp('Germany','Kaiser',-,-)|-])).
Yes

```

to remove the clauses as required.

**Exercise 3.14.** See Fig. A.5.

**Exercise 3.15.** We unfold the second goal in clause two of *flatten\_dl/2*:

```

?- unfold(flatten_dl/2,2,2).
Clause(s) used:
Clause 1 of predicate flatten_dl/2
Clause 2 of predicate flatten_dl/2
Clause 3 of predicate flatten_dl/2
...
Clause removed:
Clause 2 of predicate flatten_dl/2

flatten_dl([], A-A).
flatten_dl(A, [A|B]-B).
flatten_dl([A], B-C) :- flatten_dl(A, B-C),
                        true.
flatten_dl([A, B|C], D-E) :- flatten_dl(A, D-F),
                             flatten_dl(B, F-G),
                             flatten_dl(C, G-E).
flatten_dl([A|B], C-D) :- flatten_dl(A, C-[B|D]),
                          true.

```

As shown above, *flatten\_dl/2* is now defined by five clauses which, however, have to be rearranged to restore the 'original order': clauses 3–5 are a replacement for what was formerly clause 2; thus

```

?- clause_arrange(flatten_dl/2,[1,3,4,5,2]).
Yes

```

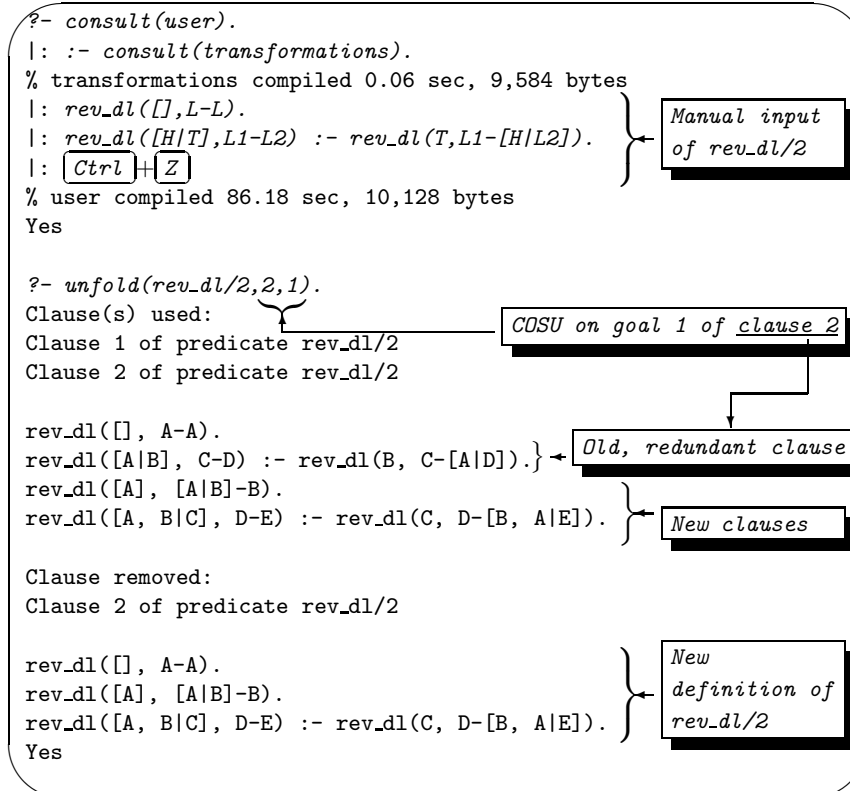


Figure A.5: Automated Solution of Exercise 2.9, Part (c)

```

?- listing(flatten_dl/2).
flatten_dl([], A-A).
flatten_dl([A], B-C) :- flatten_dl(A, B-C),
                        true.
flatten_dl([A, B|C], D-E) :- flatten_dl(A, D-F),
                             flatten_dl(B, F-G),
                             flatten_dl(C, G-E).
flatten_dl([A|B], C-D) :- flatten_dl(A, C-[B|D]),
                        true.
flatten_dl(A, [A|B]-B).

```

The above is equivalent to the initial definition (both logically *and* procedurally). Clause 4 may be removed from the database, however, without affecting the behaviour of `flatten_dl/2` since clause 2 won't ever be made use of:<sup>9</sup>

- Clause 1 is invoked for flattening the empty list.

<sup>9</sup>To be more precise, the *first* solution found by `flatten_dl/2` won't be affected by the removal of this clause; further solutions found on backtracking may differ. They are, however, of no concern here because of the cut used in `flatten_5/2`.

- Clause 2 is invoked for flattening lists with a single entry.
- All other lists are covered by clause 3 which is used for flattening lists with at least two entries.

Remove now the redundant clause:

```
?- clause_arrange(flatten_dl/2, [1,2,3,5]).
Yes
?- listing(flatten_dl/2).
flatten_dl([], A-A).
flatten_dl([A], B-C) :- flatten_dl(A, B-C),
                        true.
flatten_dl([A, B|C], D-E) :- flatten_dl(A, D-F),
                             flatten_dl(B, F-G),
                             flatten_dl(C, G-E).
flatten_dl(A, [A|B]-B).
```

An *experiment* akin to the one in Exercise 2.7 confirms that flattening based on this version is more efficient than the built-in *flatten/2*:

```
?- time(flatten_5([a,[b],[c,[d]], [e,[f],[g,[h]]],
                 [i,[j],[k,[l]], [m,[n],[o,[p]]]]],F)).
% 43 inferences in 0.00 seconds (Infinite Lips)
F = [a, b, c, d, e, f, g, h, i|...]
?- time(flatten([a,[b],[c,[d]], [e,[f],[g,[h]]],
                [i,[j],[k,[l]], [m,[n],[o,[p]]]]],F)).
% 191 inferences in 0.00 seconds (Infinite Lips)
F = [a, b, c, d, e, f, g, h, i|...]
```

Further improvement may be achieved by carrying on unfolding in an analogous manner. Let us unfold goal 3 of clause 3:

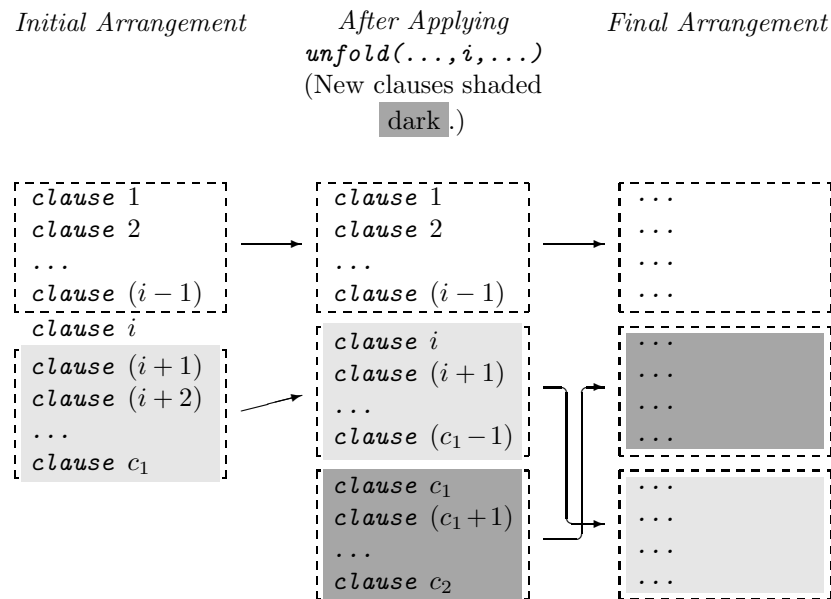
```
?- unfold(flatten_dl/2,3,3).
...
?- clause_arrange(flatten_dl/2, [1,2,4,5,6,3]).
?- listing(flatten_dl/2).
flatten_dl([], A-A).
flatten_dl([A], B-C) :- flatten_dl(A, B-C),
                        true.
flatten_dl([A, B], C-D) :- flatten_dl(A, C-E),
                           flatten_dl(B, E-D),
                           true.
flatten_dl([A, B, C], D-E) :- flatten_dl(A, D-F),
                              flatten_dl(B, F-G),
                              flatten_dl(C, G-E),
                              true.
flatten_dl([A, B, C, D|E], F-G) :- flatten_dl(A, F-H),
                                   flatten_dl(B, H-I),
                                   flatten_dl(C, I-J),
                                   flatten_dl(D, J-K),
                                   flatten_dl(E, K-G).
flatten_dl(A, [A|B]-B).
```

The improvement in performance is gleaned from

```
?- time(flatten_5([a,[b],[c,[d]],[e,[f],[g,[h]]],  
                [i,[j],[k,[l]],[m,[n],[o,[p]]]]],F)).  
% 35 inferences in 0.00 seconds (Infinite Lips)  
F = [a, b, c, d, e, f, g, h, i|...]
```

It is seen that as unfolding is carried further, longer and longer lists will be flattened by rules explicitly referring to their length and less is dealt with by the (penultimate) ‘general rule’.

**Exercise 3.16.** The initial and intended final arrangement of clauses are indicated in Fig. A.6. The predicate *cosu/3* is defined in (P-A.24).

Figure A.6: Database Changes Brought About by *cosu/3*
**Prolog Code P-A.24: Definition of *cosu/3***

```

1  cosu(Fun/Arity,I,J) :-
2      functor(Pred,Fun,Arity),
3      predicate_property(Pred,number_of_clauses(C1)),
4      unfold(Fun/Arity,I,J),
5      predicate_property(Pred,number_of_clauses(C2)),
6      A1 is 1, B1 is I - 1,
7      A2 is I, B2 is C1 - 1,
8      A3 is C1, B3 is C2,
9      from_to(A1,B1,L1),
10     from_to(A2,B2,L2),
11     from_to(A3,B3,L3),
12     concat3(L1,L3,L2,L),
13     clause_arrange(Fun/Arity,L).

```

With reference to Fig. A.6, the steps performed by *cosu/3* are:

- Unify with *C1* the number of clauses in the predicate's *original* definition. The initial arrangement is shown Fig. A.6.
- Unfold by using *unfold/3*. The resulting state of the database is again shown in Fig. A.6.
- Unify with *C2* the number of clauses in the predicate's *new* definition.

- As seen from Fig. A.6, the pattern of intended rearrangement for the clauses is given by the permutation

$$L = [1, 2, \dots, i-1, c_1, c_1+1, \dots, c_2, i, i+1, \dots, c_1-1]$$

This list is then used to rearrange the clauses by *clause\_arrange/2*.

- The predicate *from\_to/3* is used to generate integer lists with specified first and last entries:

```
from_to(Low, High, List) :- bagof(N, between(Low, High, N), List), !.
from_to(_, _, []).
```

(The catch-all clause ensures that *from\_to/3* always succeeds.)

**Exercise 3.17.** Using the built-in predicate *setof/3*, the predicate *colours/2* collects the items' colours in alphabetical order.

```
colours(Items, Colours) :- setof(Colour,
                                Object^(member(col(Object, Colour), Items)),
                                Colours).
```

*dijkstra/3* is then used to obtain the items' list.

```
dijkstra_st(Items, Grouped) :- colours(Items, Colours),
                                dijkstra(Colours, Items, Grouped).
```

**Exercise 3.19.** The definition of *def\_encolour\_pl/1* is not shown here as it is analogous to that of *def\_encolour\_dl/1*. (The source code is found in the file *dl.pl*.) The predicate *def\_endijkstra\_pl/1* is defined in (P-A.25).

**Prolog Code P-A.25: Definition of *def\_endijkstra\_pl/1***

```
1 def_endijkstra_pl(Colours) :- dynamic(endijkstra_pl/2),
2                               retractall(endijkstra_pl(_, _)),
3                               length(Colours, N),
4                               length(Vars, N),
5                               Head = endijkstra_pl(Items, Grouped),
6                               Goal1 =.. [encolour_pl, Items|Vars],
7                               Goal2 =.. [flatten, Vars, Grouped],
8                               Body = (Goal1, Goal2),
9                               assert((Head :- Body)).
```

*length/1* is used here to create a list of the requisite number of unbound variables which then serve as arguments to both *encolour\_pl* and *flatten/2*. (The former receives them as individual arguments whereas to the latter they are passed as a list.)

## A.4 Chapter 4 Exercises

All Prolog source code for Chap. 4 is available in the file *rhyme\_demo.pl*.

**Exercise 4.1.** A predicate *n\_times/3* will be needed which returns in a list a specified number of copies of any term:

```
?- n_times(3,any(term),L).
L = [any(term), any(term), any(term)]
```

This we define by the accumulator technique as follows.

```
n_times_acc(0,_,L,L).
n_times_acc(N,X,L1,L2) :- N1 is N - 1,
                           n_times_acc(N1,X,[X|L1],L2).

n_times(N,X,L) :- n_times_acc(N,X,[],L), !.
```

Now, we define *long\_verse/1* by

```
long_verse(N) :- n_times(N,'That interacts with the item ...',L),
                 dynamic(verse/1),
                 retract(verse(_)),
                 assert(verse(L)).
```

**Exercise 4.2.** The second clause in the definition of *rhyme\_prel\_5/* (p. 128) should be augmented by a *cut*:

```
rhyme_prel_5([H|T],C) :- append(P,[H|T],C),
                          rhyme_prel_5(T,P), !.
```

**Exercise 4.3.** Let us examine interactively, for example, how the query



```
?- cputime(rhyme_prel_5,['B','A'],R),Time).
```

could be dealt with. Obviously, we will want *rhyme\_prel\_5/2* to be invoked by *call/1* and therefore we will have to create first a *term* which will serve as the argument of *call/1*. To achieve this, we use the built-in predicate *univ*.

```
?- T =.. [rhyme_prel_5,['B','A'],R].
T = rhyme_prel_5(['B','A'],_G345)
R = _G345
Yes
```

We now submit *T* to *call/1*, the latter sandwiched between two invocations of *statistics/2*:

```
?- T =.. [rhyme_prel_5,['B','A'],R],
   statistics(cputime,Before), call(T),
   statistics(cputime,After), Time is Before - After.
T = rhyme_prel_5(['B','A'],[['A'], ['B','A']])
R = [['A'], ['B','A']]
Before = 15124
After = 15124
Time = 0
Yes
```

(The CPU time for the above happens to be negligible hence the zero response.) This gives rise to the following definition.

```
cputime(Predname,Arglist,Time) :- T =.. [Predname|Arglist],
                                   statistics(cputime,Before),
                                   call(T),
                                   statistics(cputime,After), !,
                                   Time is After - Before.
```

As a consequence of the *cut* in the above definition, *cputime/3* will find one solution only even if the underlying query could be re-satisfied on backtracking. Furthermore, and perhaps more importantly in our context, if the query has a solution but would be caught in an infinite loop on trying to re-satisfy the goal, *cputime/3* will still deliver this unique solution and respond with failure subsequently. This property of *cputime/3* is essential when timing the same predicate with several sets of arguments using *findall/3*, as seen on p. 131 for *rhyme\_prel\_5/2*.

**Exercise 4.4.** Prior to applying *cputime/3* from Exercise 4.3, we construct the predicate's name by using *concat\_atom/2* (see, inset on p. 126):

```
cputime(Predname,Arglist,Version,Time) :- concat_atom([Predname,'_',Version],Pred),
                                           cputime(Pred,Arglist,Time).
```

**Exercise 4.5.** We first show how the first row of Table 4.2 is produced interactively.<sup>10</sup>

```
?- findall(_Time,
   (between(1,7,_J),
    _L is _J * 10 ** 2,
    long_verse(_L),
```

---

<sup>10</sup>The Java/C-style code layout is of course not the actual one but is employed here for better readability only.

```

        verse(_V),
        cputime(rhyme_prel, [_V, _R], 5, _Time)
    ),
    Row
).
Row = [1.98, 15.71, 52.23, 124.19, 241.95, 418.81, 666.96]

```

Now, after some modifications (involving the introduction of the variables `_I` and `_Version`), we embed this query into another `findall` to collect all the rows of Table 4.2 in the variable `_Rows` which, as a list (of lists), we then display by using `show_list/1`:

```

?- findall(_Row,
    (between(2, 4, _I),
     findall(_Time,
        (between(1, 7, _J),
         _Version is _I + 3,
         _L is _J * 10 ** _I,
         long_verse(_L),
         verse(_V),
         cputime(rhyme_prel, [_V, _R], _Version, _Time)
        ),
        _Row
     ),
    _Rows
),
show_list(_Rows).
[1.97, 15.77, 52.35, 124.51, 242.6, 419.9, 666.41]
[4.23, 19.99, 45.59, 85.74, 135.45, 194.44, 276.88]
[0.11, 0.44, 0.99, 1.2, 1.37, 1.48, 1.76]

```

*Alternative Solution.* For a perhaps simpler solution by using a *single instance* of `bagof/3`, we revisit the first query above with `findall` replaced by `bagof`.

```

?- bagof(_Time,
    _J^_L^_V^_R^(between(1, 7, _J),
        _L is _J * 10 ** 2,
        long_verse(_L),
        verse(_V),
        cputime(rhyme_prel, [_V, _R], 5, _Time)
    ),
    Row).
Row = [1.98, 15.76, 52.24, 124.29, 242.11, 419.08, 666.96]

```

How should the above be augmented to display on backtracking *all three* rows of Table 4.2? We introduce new variables `_Version` and `_I` as before but *won't* prefix the goal inside `bagof` by `_Version^` thus allowing Prolog to find solutions corresponding to each particular value of `_Version`. Finally, backtracking is accomplished by a failure-driven loop.

```

?- bagof(Time,
    I^J^L^V^R^(between(2, 4, I),
        between(1, 7, J),

```

```

        Version is I + 3,
        L is J * 10 ** I,
        long_verse(L),
        verse(V),
        cputime(rhyme_prel,
                [V,R],
                Version,
                Time
        )
    ),
    Row
),
write(Version),
write(' - '),
write(Row),
nl,
fail.
5 - [1.98, 15.76, 52.29, 124.46, 242.67, 419.58, 667.4]
6 - [4.28, 20.05, 45.65, 85.79, 135.62, 194.5, 278.85]
```

7 - [0.11, 0.44, 0.77, 1.21, 1.43, 1.48, 1.7]

No

**Exercise 4.6.** The definition of *song\_skeleton/1* is fairly obvious if we use *int/1* and *int/2* as ‘templates’:

```
song_skeleton(L) :- song_skeleton([1],L).

song_skeleton(L,L).
song_skeleton([H|T],L) :- succ(H,N),
                           song_skeleton([N|H|T],L).
```

A more interesting question is perhaps how the definition of *int/2* (p. 134) came about in the first place. To examine this, we first consider the following partial implementation of *int/2*

```
int(I,I).                % clause 1
int(1,I) :- int(2,I).    % clause 2
```

The query `?- int(1,I).` will be first satisfied by virtue of clause 1 with  $I = 1$  and on backtracking re-satisfied by clause 2 which succeeds with  $I = 2$  since its only subgoal (i.e. `int(2,I)`) unifies with clause 1. If we now take also the clause

```
int(2,I) :- int(3,I).    % clause 3
```

aboard, everything said thus far still applies; moreover, the body of clause 2 now succeeds also by clause 3 with  $I = 3$  since the body of the latter unifies with clause 1. Clearly, any number of new clauses could be added in this manner to the database. (The resulting search tree is shown in Fig. A.7 below.) Now, the second clause

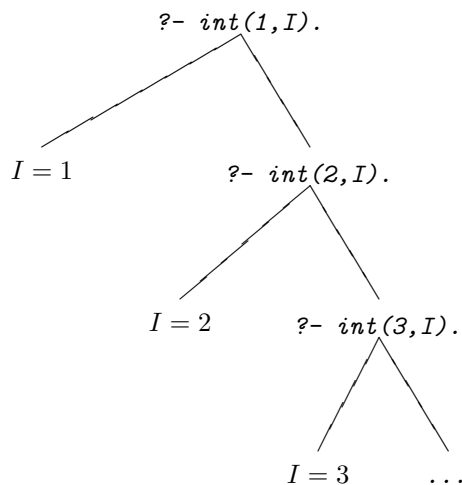


Figure A.7: Search Tree of the Query `?- int(1, I).`

in the definition of *int/2* on p. 134 can be considered a subsumption of all possible such augmentations of the database.

It is also instructive to observe that *int/1* is defined by solving another problem (the definition of *int/2*) of which the original problem is a special case. This approach is often successful in Prolog programming.

**Exercise 4.7.** Our definition of *song\_skeleton/1* very closely models that of *nat/1*:

```
song_skeleton(L) :- first_verse, current_verse(L).
song_skeleton(L) :- repeat, update_verse, current_verse(L).
```

with the predicates *first\_verse/0* and *update\_verse/0* defined by

```
first_verse :- dynamic(current_verse/1),11
              retractall(current_verse(_)),
              assert(current_verse([1])).

update_verse :- current_verse([H|T]),
                retractall(current_verse(_)),
                NewH is H + 1,
                assert(current_verse([NewH,H|T])).
```

**Exercise 4.8.** We calculate the digits of a natural number by applying the built-in arithmetic functions *mod* (the *modulo*)<sup>12</sup> and *//* (the *integer division*) in an alternate fashion; the digits of 351, for example, may be obtained by

```
?- _N0 is 351,
   D1 is _N0 mod 10, _N1 is _N0 // 10,
   D2 is _N1 mod 10, _N2 is _N1 // 10,
   D3 is _N2 mod 10.
D1 = 1
D2 = 5
D3 = 3
```

suggesting a predicate *digits/3* with

```
digits(N,Acc,[N|Acc]) :- N < 10, !.13
digits(N,Acc,D)       :- H is N mod 10, NewN is N // 10,
                        digits(NewN,[H|Acc],D).
```

which then behaves as expected:

```
?- digits(351,[],D).
D = [3, 5, 1]
```

---

<sup>11</sup>As an alternative, the predicate *current\_verse/1* may be declared *dynamic* also by the directive

```
:- dynamic(current_verse/1).
```

This is usually placed at the head of the source file.

<sup>12</sup>*mod* computes the *remainder* of an integer division. It is not to be confused with Prolog's built-in arithmetic function *rem* which returns the fractional part of a quotient:

```
?- Frac is 3896 rem 100.
Frac = 0.96
```

<sup>13</sup>Without this *cut* some spurious solutions are returned on backtracking:

```
?- digits(98,[],L).
L = [9, 8] ;
L = [0, 9, 8]
Yes
```

We define the predicate *digits(+Number, -List)* thus by

```
digits(N,D) :- integer(N), digits(N,[],D).
```

(This definition works for the instantiation pattern *digits(+Number, +List)*, too.)

With a view to the instantiation pattern *digits(-Number, +List)*, we observe that any number can be written in terms of its digits as in

$$4351 = 10 \times (10 \times (10 \times (10 \times 0 + 4) + 3) + 5) + 1$$

suggesting Algorithm A.4.1.

**Algorithm A.4.1:** VALUE(*List*)

```
Accumulator ← 0 (1)
```

```
List ← list of digits, e.g. [4, 3, 5, 1] (2)
```

```
while List ≠ [] (3)
```

```
do { [H|T] ← List
    Accumulator ← 10 * Accumulator
    Accumulator ← Accumulator + H
    List ← T
```

```
Number ← Accumulator (4)
```

```
return (Number)
```

We implement (3)–(4) by *value/3*,

```
value([],N,N).
value([H|T],Acc,N) :- integer(H), H < 10,
                      AccNew is H + 10 * Acc,
                      value(T,AccNew,N).
```

while (1) and (2) will take effect when *value/3* is invoked:

```
?- value([4,3,5,1],0,V).
V = 4351
```

The definition of *digits(-Number, +List)* is now straightforward:

```
digits(N,D) :- var(N), value(D,0,N).
```

The predicate *in\_words/2* finally is defined by

```
in_words(N,Text) :- digits(N,D), number(D,Text).
```

with a predicate *number/2* which assembles from a list of digits the corresponding number in plain English:

```
?- number([3,5,1],Text).
Text = threehundredfiftyone
```

We won't spell out here the definition of *number/2*. The idea for a first rough version can be gleaned, however, from the following query:

```
?- maplist(units, [4,3,5,1], [_Th,_H,_T,_U]),
   concat_atom([_Th,thousand,_H,hundred,_T,ten,_U],Text).
Text = fourthousandthreehundredfivetenone
```

where *units/2* is defined by a collection of facts in the database:

```
units(0,'').      units(1,one).    units(2,two).
units(3,three).   units(4,four).   units(5,five).
...
```

**Exercise 4.9.** The definition of *capital/2* in (P-A.26) is self-explanatory.

**Prolog Code P-A.26: Definition of *capital/2***

```
1 capital(Atom1,Atom2) :-
2     atom_chars(Atom1,[H|T]),      % disassemble Atom
3     to_upper(H,Upper),            % convert H to upper case
4     atom_chars(Atom2,[Upper|T]).  % re-assemble Atom

5 to_upper(Lower,Upper) :- char_code(Lower,L),
6                           U is L - 32,
7                           char_code(Upper,U).
```

**Exercise 4.10.** The following definition of *line3/2* is derived from the sample query on p. 137.

```
line3(Numbers,Text) :- maplist(in_words,Numbers,[H|T]),
                       maplist(atom_concat(' men,\n '),T,L1),
                       capital(H,C),
                       concat_atom([C|L1],Text1),
                       atom_concat(Text1,' man and his dog,',Text).
```

Notice the *partial application* of *atom\_concat/3* here in that its first argument is fixed, thereby becoming a predicate of two arguments, ready to be used by *maplist/3*.

**Exercise 4.11.** The top level predicate *song/0* is finally defined by a failure driven loop thus

```
song :- song_skeleton([H|T]),
        line1(H,L1),
        line2(L2),
        line3([H|T],L3),
        line4(L4), nl,
        write(L1), nl,
        write(L2), nl,
        write(L3), nl,
        write(L4), nl, fail.
```

The only building block of *song/0* perhaps in need of some comment is *line1/2* which is expected to behave as follows.

```
?- line1(1,L).
L = 'One man went to mow,'
?- line1(351,L).
L = 'Threehundredfiftyone men went to mow,'
```

We use the predicates *in\_words/2* and *capital/2* (from Exercise 4.8 and (P-A.26) in Exercise 4.9, respectively) to define *line1/2*:

```
line1(N,Text) :- in_words(N,HowMany),
                 capital(HowMany,C),
                 ((N == 1, atom_concat(C,' man went to mow,',Text));
                  (N > 1, atom_concat(C,' men went to mow,',Text))).
```

A simpler alternative definition is as follows.

```
line1(1,'One man went to mow,') :- !.
line1(N,Text) :- in_words(N,HowMany),
                 capital(HowMany,C),
                 atom_concat(C,' men went to mow,',Text).
```

This is the preferred version as it does not involve any arithmetic operations nor a choice of case by the disjunction operator; it uses Prolog's search and unification mechanisms instead.



# Appendix B

## Software

Below are listed the Prolog source files referenced in the various chapters. They are available on the Ventus website.

**Referred to in Chap. 1.**

`accumulator.pl`

**Referred to in Chap. 2.**

`dl.pl`

**Referred to in Chap. 3.**

<code>arrange.pl</code>	<code>party.pl</code>	<code>stamps.pl</code>
<code>committee.pl</code>	<code>people.pl</code>	<code>transformations.pl</code>
<code>dl.pl</code>	<code>queue.pl</code>	

**Referred to in Chap. 4.**

`rhyme_demo.pl`



# Appendix C

## Glossary

*Note.* You will find a more complete collection of Prolog terms defined in the SWI-Prolog manual [18].

**Accumulator.** An auxiliary argument whose final value is calculated by repeated updating. It plays the rôle of an accumulator variable in a loop in imperative programming.

**Anonymous variable.** It is a variable with no user-defined name and it is denoted by the underscore (`_`). It is used to replace singleton variables (i.e. variables occurring once only in a clause). Several anonymous variables in the same clause will be unrelated, i.e. their system-chosen names will be different.

**Argument.** One of the positions of a predicate if this has arity at least one.

**Argument pattern.** This is a way of describing the modes in which a predicate can be called. The name of an *input* argument is prefixed by a plus sign (+); the name of an *output* argument is prefixed by a minus sign (−); and, the name of an argument which can be used in *both* modes is prefixed by a question mark (?). *Example.* The inset for *between/3* (p. 41) says that the first two arguments of *between/3* are for input only while the third one can be used for input or output (depending on whether the predicate is used to *test* or to *generate* values thereof).

**Arity.** The number of arguments of a predicate, or more generally, of a compound term. *Example.* The term *parents\_of(F, M, joe)* has arity 3.

**Atom.** A constant value which is assigned to a variable. *Example.* Strings starting with a lower case character such as *joe*.

**Backtracking.** A way of finding values of the variables in a predicate such that this succeeds. This is accomplished by traversing the associated search tree using Depth First search.

**Binding.** Assignment of a term as a value to a variable.

**Body of a clause.** The conjunction of the goals which have to be satisfied for the head of the clause to be 'true'.

**Bound variable.** A variable which has been assigned a value.

**Clause.** A fact or a rule in the database.

**Closed World Assumption.** Any goal that cannot be inferred from the database is assumed 'false'. Therefore, the negation of such a goal will succeed.

**Cut (!).** A built-in predicate for 'freezing' the assignment of values to variables in goals to the left of the cut. Variables in goals to the right will be assigned new values on backtracking.

**Database.** The collection of all facts and rules loaded in memory.

**Declarative reading.** A program (a predicate) is viewed as a collection of declarative assertions about the problem to be solved.

**Difference list.** A way of representing a list as a 'difference' of two lists. Implicitly, its use involves unification and is equivalent to the accumulator technique.

**Fact.** A clause with no body. More precisely, a clause whose body is assumed **true**.

**Failure.** A predicate is said to fail if its truth value inferred from the database is 'false'.

**Free variable.** A variable with no value assigned to it.

**Functor.** The name of a predicate, or more generally, the name of a compound term. *Example.* In *parents\_of(george, susan, joe)* the functor is *parents\_of*.

**Goal.** An atom or a compound term which will be assigned a truth value by the Prolog system.

**Ground term.** A term with no free variables in it, i.e. a one where all variables are bound.

**Head of a clause.** The part of a clause which follows from the conjunction of the other goals of the clause, the body.

**Head of a list.** The first entry if we use the square bracket notation. The first argument if we use the dot (.) functor to denote lists.

**Higher order predicate.** A predicate which *uses* another predicate by expecting in one of its arguments the name of this predicate; or, which *defines* or *modifies* another predicate. *Example.* The built-in predicate *bagof/3* is a higher order predicate of the former kind as it uses the predicate named in its second argument. *unfold/3* (see Fig. 3.9, p. 97) is a higher order predicate of the latter kind as it modifies the definition of the predicate named in its first argument.

**Instantiation.** The assignment of a value to a variable.

**List.** It is a recursively defined built-in binary predicate with the dot functor (.). Its second argument is either the empty list or a list. The user friendly notation uses square brackets to denote lists.

**Predicate.** A Prolog structure for representing an  $n$ -ary relation. *Example.* The ternary relation *parents\_of/3* is a relation on (i.e. a subset of) the Cartesian product  $C = People \times People \times People$ . A triplet in  $C$  which can be inferred to satisfy the relation *parents\_of/3* is said to succeed; otherwise it is said to fail.

**Predicate Calculus.** PC is a system for formalizing arguments with a view to establishing their validity. It is an extension of Propositional Calculus using predicates, constants and variables which are universally or existentially

quantified.

**Predicate.** The collection of clauses whose heads have the same functor.

**Propositional Calculus.** PC is the simplest system for formalizing arguments with a view to establishing their validity. Its smallest units are the sentence letters that are assigned the values 'true' or 'false'. These then are strung together with connectives according to certain rules to form well-formed formulae. Finally, the latter are built up to argument forms; PC is concerned with establishing the validity of these.

**Recursion.** Defining a predicate in terms of itself.

**Rule.** An assertion that a certain goal, the head of the clause, is 'true' provided that all the goals in its body are 'true'.

**Success.** A predicate is said to succeed if it can be inferred from the database.

**Switch.** A predicate argument which can take two values only. Used as a programming tool.

**Tail.** The latter part of a list: the list comprising all entries except its first entry.

**Tail recursion.** A tail recursive clause defines a predicate in terms of itself where the predicate is called as the *last* goal in the body.

**Term.** The most general data object in Prolog. It can be one of the following: a constant, a variable, or a compound term.

**Unification.** A pattern matching algorithm returning a set of values assigned to the variables of two terms such that these become equal. The assignment is most general in that any other such assignment can be obtained by specialization of the variables *after* unification.

**Variable.** A named location in the memory which may be assigned a value.

# References

- [1] W. F. Clocksin. *Clause and Effect – Prolog Programming for the Working Programmer*. Springer, London, 1997.
- [2] W. F. Clocksin and C. S. Mellish. *Programming in Prolog*. Springer, London, fourth edition, 1994.
- [3] M. A. Covington, D. Nute, and A. Vellino. *Prolog Programming in Depth*. Prentice Hall, Upper Saddle River, NJ, 1997.
- [4] A. Csenki. Rotations in the plane and Prolog. *Science of Computer Programming*, 66:154–161, 2007.
- [5] Y. Deville. *Logic Programming – Systematic Program Development*. Addison–Wesley, Wokingham, 1990.
- [6] DIN Deutsches Institut für Normung e.V., Berlin. *DIN 66 261 : Nassi–Shneiderman–Diagramm, eine Entwurfsmethode für die strukturierte Programmierung*, 1985.
- [7] A. Hoffmann. *Paradigms of Artificial Intelligence – A Methodological & Computational Analysis*. Springer, Singapore, 1998.
- [8] C. J. Hogger. *Introduction to Logic Programming*. Academic Press, London, 1984.
- [9] C. J. Hogger. *Essentials of Logic Programming*. Clarendon Press, Oxford, 1990.
- [10] E. Kreyszig. *Advanced Engineering Mathematics*. Wiley, New York, eighth edition, 1998.
- [11] J. Mulherin. *Popular Nursery Rhymes*. Grosset & Dunlap, New York, eighth edition, 1983.
- [12] I. Nassi and B. Shneiderman. Flowchart Techniques for Structured Programming. *SIGPLAN Notices*, 8, August 1973.
- [13] M. Negnevitsky. *Artificial Intelligence – A Guide to Intelligent Systems*. Addison–Wesley, Harlow and London and New York, 2002.
- [14] N. J. Nilsson and P. Norvig. *Artificial Intelligence – A Modern Approach*. Prentice Hall, Upper Saddle River, NJ, 1995.
- [15] I. Sommerville. *Software Engineering*. Addison–Wesley, Harlow and London and New York, sixth edition, 2001.
- [16] L. Sterling and E. Shapiro. *The Art of Prolog – Advanced Programming Techniques*. MIT Press, Cambridge Ma, London, 1986.

- 
- [17] S. Todd. *Basic Numerical Mathematics*, volume 2. Academic Press, Harlow and London and New York, 1978. Basic Numerical Algebra.
  - [18] J. Wielemaker. *SWI-Prolog 5.1 Reference Manual*. Amsterdam, 2003.  
<http://www.wsi-prolog.org>.



# Index

- >/2, 91
- ./2, 43
- //, 173
- =./2, 43
- \=/2, 58
- accumulators, 13–36
  - difference lists as acc's, 57
- arg*/3, 43
- assert*/1, 80
- asserta*/1, 81
- atom\_chars*/2, 126
- atom\_codes*/2, 19
- atom\_concat*/3, 138
- bagof*/3, 41
- between*/3, 41
- char\_code*/2, 137
- clause*/3, 102
- Closed World Assumption, 54
- concat\_atom*/2, 126
- copy\_term*/2, 156
- current\_predicate*/2, 88
- current\_prolog\_flag*/2, 16, 46, 136
- DeMorgan's Rule, 161
- difference lists, 37–73, 97, 98, 107–115, 129
- Dijkstra's Dutch Flag Problem, 57–60, 108–116
- directive, 38
- dynamic*/1, 79
- erase*/1, 105
- fail*/0, 77
- failure driven loop, 77, 104, 105, 161
- findall*/3, 56
- flatten*/2, 42–49
- folding, 54
- forall*/2, 162
- functor*/3, 43
- Gauss–Seidel Method, 69–73
- hand computations, 14–23
- Implication Introduction Rule, 56
- integer*/1, 17
- is\_list*/1, 17
- last*/2, 17, 143
- listing*/1, 86
- maplist*/3
  - definition of, 127
- Material Implication, 161
- member*/2, 90
- mod*, 173
- nth*1/3, 107
- nth\_clause*/3, 102
- numlist*/3, 143
- op*/3, 38
- operator, 38
- partial application, 127, 175
- pattern matching, 121
- Perceptron Training Algorithm, 27–36, 64–65
- predicate\_property*/2, 88, 108
- proper\_list*/1, 45, 145
- pseudocodes, 23–26
- Quantifier Equivalence Rule, 161
- rem*, 173
- repeat loop, 134–136, 173

---

*repeat/0*, *see* repeat loop  
repeat loop, 173  
*retract/1*, 79  
*retractall/1*, 80  
*reverse/2*, 50–57  
rotation  
    list rotation, 61–64  
    planar rotation, 65–69  
  
self-unfolding, 52, 105–106  
*set\_prolog\_flag/2*, 33, 46, 109, 147  
*setof/3*, 41  
*statistics/2*, 131  
*succ/2*, 134  
switch, 21  
  
tail recursion, 13  
*tell/1*, 86  
*term\_to\_atom/2*, 47  
*time/1*, 41  
*told/0*, 86  
torus, 66  
*trace/1*, 130  
*true/0*, 77  
  
unfolding, 52–53, 95–108  
*univ*, *see* *=./2*  
  
*var/1*, 17  
  
*write\_term/2*, 46