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).

Prolog Code P-A.1: *Definition of* *from_to/3*

```
1  from_to(M,N,L) :- (var(L); is_list(L)),                  % clause 0
2                    integer(M),                            %
3                    integer(N),                            %
4                    M =< N,                                %
5                    from_to_acc(M,[N],L), !.               %
6  from_to(H,N,[H|T]) :- last(N,[H|T]), !,                  % clause 1
7                        H =< N.                            %

8  from_to_acc(H,[H|T],[H|T]).                              % clause 2
9  from_to_acc(M,[H|T],L) :- NewHead is H - 1, !,           % clause 3
10                           from_to_acc(M,[NewHead,H|T],L). %
```

The annotated version of the hand computations from Fig. 1.4 is shown in Fig. A.1. The idea suggested by

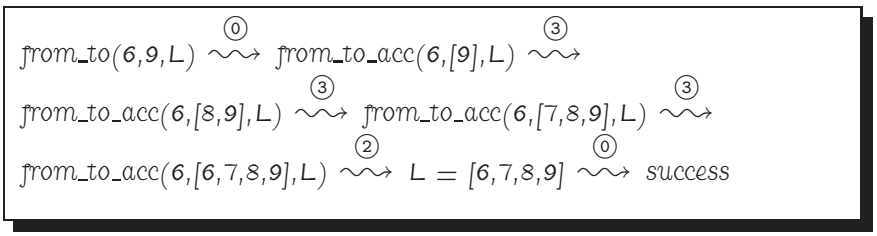

Figure A.1: Annotated Hand Computations for *from_to/3*

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.[1]

```
?- var([_,_,E|_]).
No
```

[1] 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]).
T = [9] ;
T = [_G269, 9] ;
T = [_G269, _G272, 9] ;
...
```

[2]

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 nums([],N,N).                                   % clause 1
4 nums([_],N,N).                                  % clause 2
5 nums([H,E|T],Acc,N) :- not(digit(H)), digit(E), % clause 3
6                        NewAcc is Acc + 1,       %
7                        !, nums([E|T],NewAcc,N). %
8 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.

[2] 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($Atom$)

$Values \leftarrow$ list of ASCII values of characters in $Atom$ (1)
$Acc \leftarrow 0$ (2)
$Switch \leftarrow nodigit$ (3)
while $Values \neq []$
do {
  $[H|T] \leftarrow Values$ (4)
  if H is an encoded digit
  then {
    if $Switch = nodigit$ (5)
    then { $Acc \leftarrow Acc + 1$ (6)
    $Switch \leftarrow digit$ (7)
  else { $Switch \leftarrow nodigit$ (8)
  $Values \leftarrow T$ (9)
$N \leftarrow Acc$ (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 & 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).

$mult(0.2,[5.0,10.5,2.5],L) \overset{0}{\rightsquigarrow} mult(0.2,[5.0,10.5,2.5],[],L) \overset{2}{\rightsquigarrow}$
$mult(0.2,[10.5,2.5],[1.0],L) \overset{2}{\rightsquigarrow} mult(0.2,[2.5],[2.1,1.0],L) \overset{2}{\rightsquigarrow}$
$mult(0.2,[],[0.5,2.1,1.0],L) \overset{1}{\rightsquigarrow} reverse([0.5,2.1,1.0],L) \rightsquigarrow$
$L = [1.0,2.1,0.5] \overset{0}{\rightsquigarrow} success$

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 `dl.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(functor(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[3] 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.)[4]
- 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 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(functor(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)]
```

[3] 'Lists' are understood here to be in terms of the sharp-notation.

[4] 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_dl/2*:

```
flatten_dl(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).[5]

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):[6]

```
?- rev_dl([U|V],W1-W2) = rev_dl(T,L1-[H|L2]).
U = _G384
V = _G385
W1 = _G387
W2 = [_G393|_G394]
T = [_G384|_G385]
```

[5] For the present purposes, the version number (i.e. the suffix '`_3`') is to be ignored.
[6] This is an instance of *self unfolding*.

```
L1 = _G387
H = _G393
L2 = _G394
Yes
```

The unification succeeds and gives rise to

```
rev_dl([H|T],L1-L2) :- 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`

```
colour_dl([],R-R,W-W,B-B).
colour_dl([col(Object,red)|T],
          [col(Object,red)|R1]-R2,W1-W2,B1-B2)   :-
   colour_dl(T,R1-R2,W1-W2,B1-B2).
colour_dl([col(Object,white)|T],
          R1-R2,[col(Object,white)|W1]-W2,B1-B2) :-
   colour_dl(T,R1-R2,W1-W2,B1-B2).
colour_dl([col(Object,blue)|T],
          R1-R2,W1-W2,[col(Object,blue)|B1]-B2)  :-
   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`

```
averages_dl(L1-L2,A1-A2) :- aver_dl([-1,1|L1]-L2,A1-A2), !.

aver_dl([_,0,_|X]-Y,X-Y).
aver_dl(X1-X2,ADL) :- av_rotate_dl(X1-X2,Y1-Y2),
                      aver_dl(Y1-Y2,ADL).

av_rotate_dl([H1,H2|Y]-[Last|Z],[H2|Y]-Z) :- Last is (H1 + H2)/2.
```

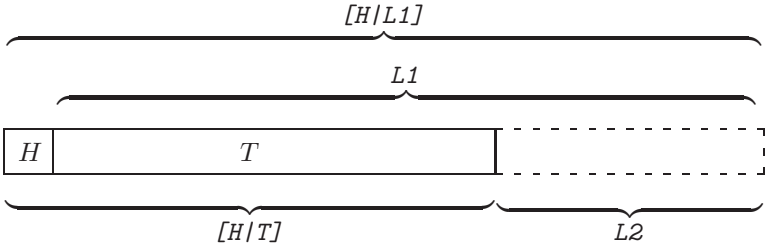

Figure A.3: Illustrating the Second Clause of `dl/2`

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 diplayed 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 invokation 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.)

[7] 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`

```
look_right(Pers) :- look_right(Pers,[Pers|T]),
                    reverse(T,List),
                    write_list(List).

look_right(Pers,[X,Pers]) :- right_to(Pers,X).
look_right(Pers,[X,H|T])  :- right_to(H,X),
                             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

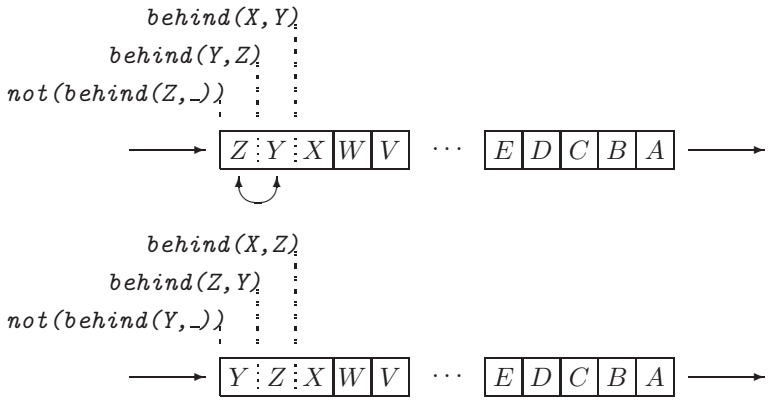

Figure A.4: The Last Two Customers Swap Places

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:

$$Condition \texttt{ -> } Action \texttt{ ; } Alternative\ Action \tag{A.1}$$

with

$$\begin{aligned} Condition &= \forall x(A(x) \rightarrow B(x)) && \text{(A.2)}\\ A(x) &= x \in List && \text{(A.3)}\\ B(x) &= my\_predicate(x, \_) && \text{(A.4)}\\ Action &= write\ to\ file && \text{(A.5)}\\ Alternative\ Action &= display\ error\ message && \text{(A.6)} \end{aligned}$$

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

$$\begin{aligned} Condition &= \neg(\forall x(A(x) \rightarrow B(x))) && \text{(A.7)}\\ Action &= display\ error\ message && \text{(A.8)}\\ Alternative\ Action &= write\ to\ file && \text{(A.9)} \end{aligned}$$

Rewrite now the right-hand side of (A.7) as follows:[8]

$$\begin{aligned} Condition &= \exists x \neg((A(x) \rightarrow B(x))) \\ &= \exists x \neg(B(x) \vee \neg A(x)) \\ &= \exists x (A(x) \wedge \neg B(x)) \end{aligned} \tag{A.10}$$

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:

[8] 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
```

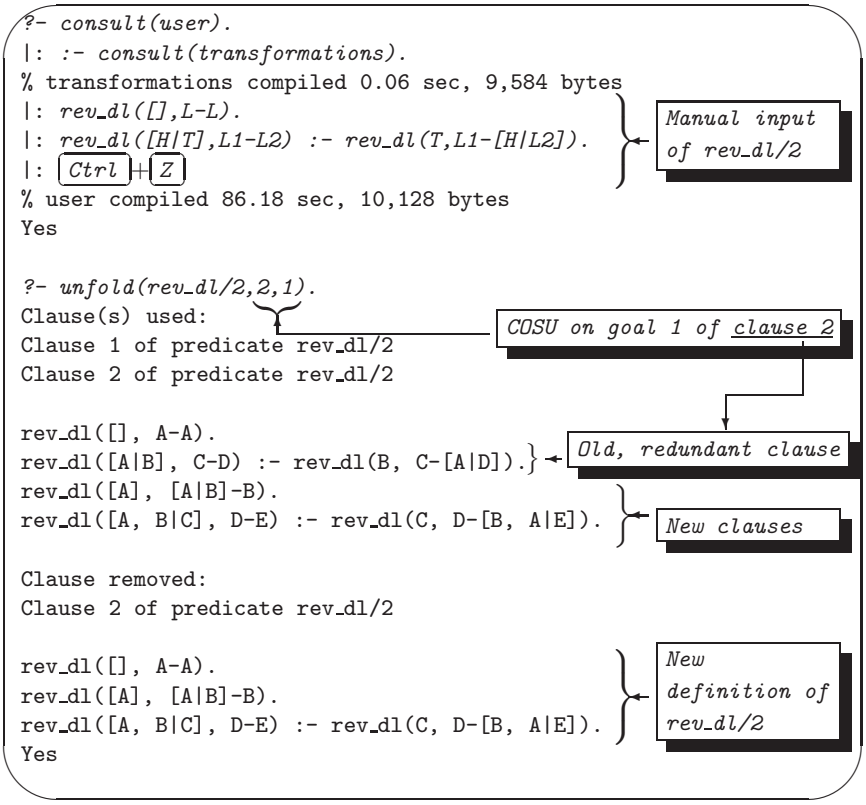

Figure A.5: Automated Solution of Exercise 2.9, Part (c)

```
?- 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:[9]

- Clause 1 is invoked for flattening the empty list.

[9]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).

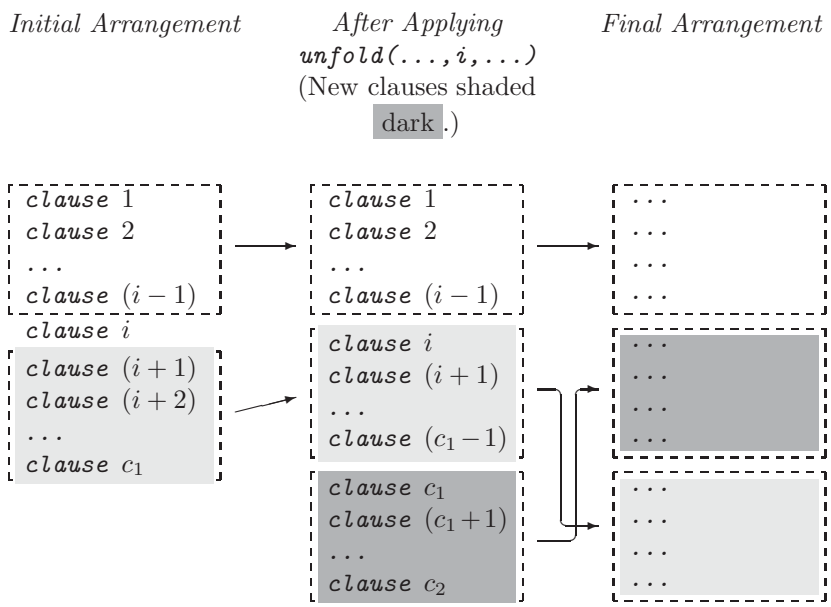

Figure A.6: Database Changes Brought About by *cosu/3*

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

$$L = [1, 2, \ldots, i-1, c_1, c_1+1, \ldots, c_2, i, i+1, \ldots, c_1-1]$$

  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.[10]

```
?- findall(_Time,
           (between(1,7,_J),
            _L is _J * 10 ** 2,
            long_verse(_L),
```

[10]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 inroduce 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

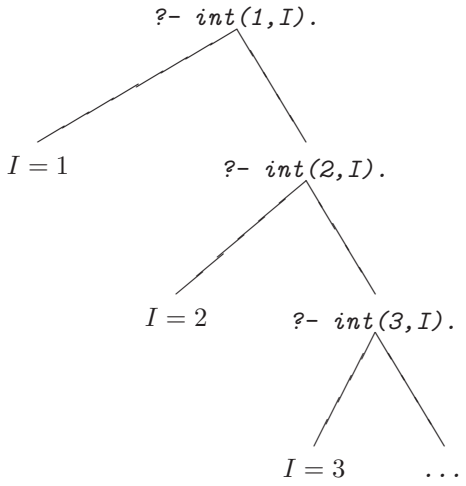

Figure A.7: Search Tree of the Query `?- int(1,I).`

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*)[12] 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]
```

[11] 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.

[12] *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
```

[13] 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($List$)

$Accumulator \leftarrow 0$ (1)
$List \leftarrow$ list of digits, e.g. $[4, 3, 5, 1]$ (2)
while $List \neq []$ (3)
do $\begin{cases} [H|T] \leftarrow List \\ Accumulator \leftarrow 10 * Accumulator \\ Accumulator \leftarrow Accumulator + H \\ List \leftarrow T \end{cases}$
$Number \leftarrow Accumulator$ (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.

```
arrange.pl          party.pl            stamps.pl
committee.pl        people.pl           transformations.pl
dl.pl               queue.pl
```

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 progranmming.

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.