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***:[1]

```
          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]
```

[1]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.[2]) 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).
```

[2]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.

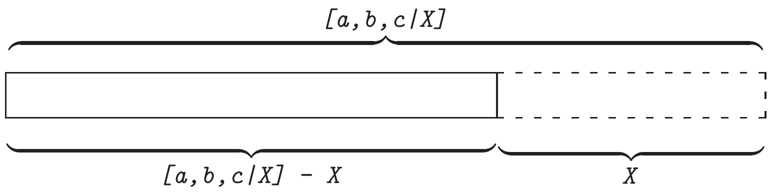


Figure 2.1: Difference List

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`.

What do you want to do?
No matter what you want out of your future career, an employer with a broad range of operations in a load of countries will always be the ticket. Working within the Volvo Group means more than 100,000 friends and colleagues in more than 185 countries all over the world. We offer graduates great career opportunities – check out the Career section at our web site www.volvogroup.com. We look forward to getting to know you!
VOLVO
AB Volvo (publ)
www.volvogroup.com
Volvo Trucks | Renault Trucks | Mack Trucks | Volvo Buses | Volvo Construction Equipment | Volvo Penta | Volvo Aero | Volvo IT
Volvo Financial Services | Volvo 3P | Volvo Powertrain | Volvo Parts | Volvo Technology | Volvo Logistics | Business Area Asia

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

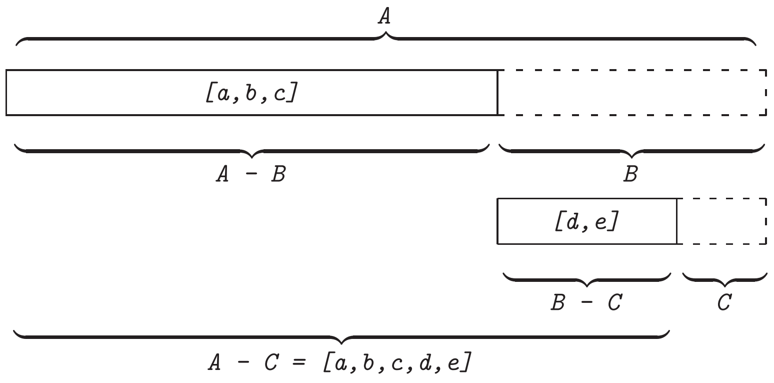


Figure 2.2: List Concatenation by Difference Lists

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 `=../2`[3]. 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.[4]

[3]This is an infix predicate and is called *univ*.

[4]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.

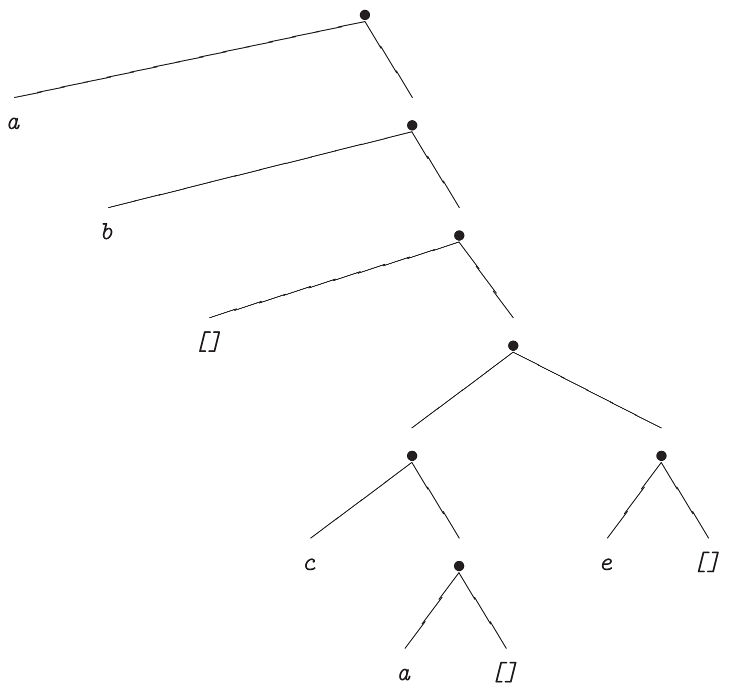
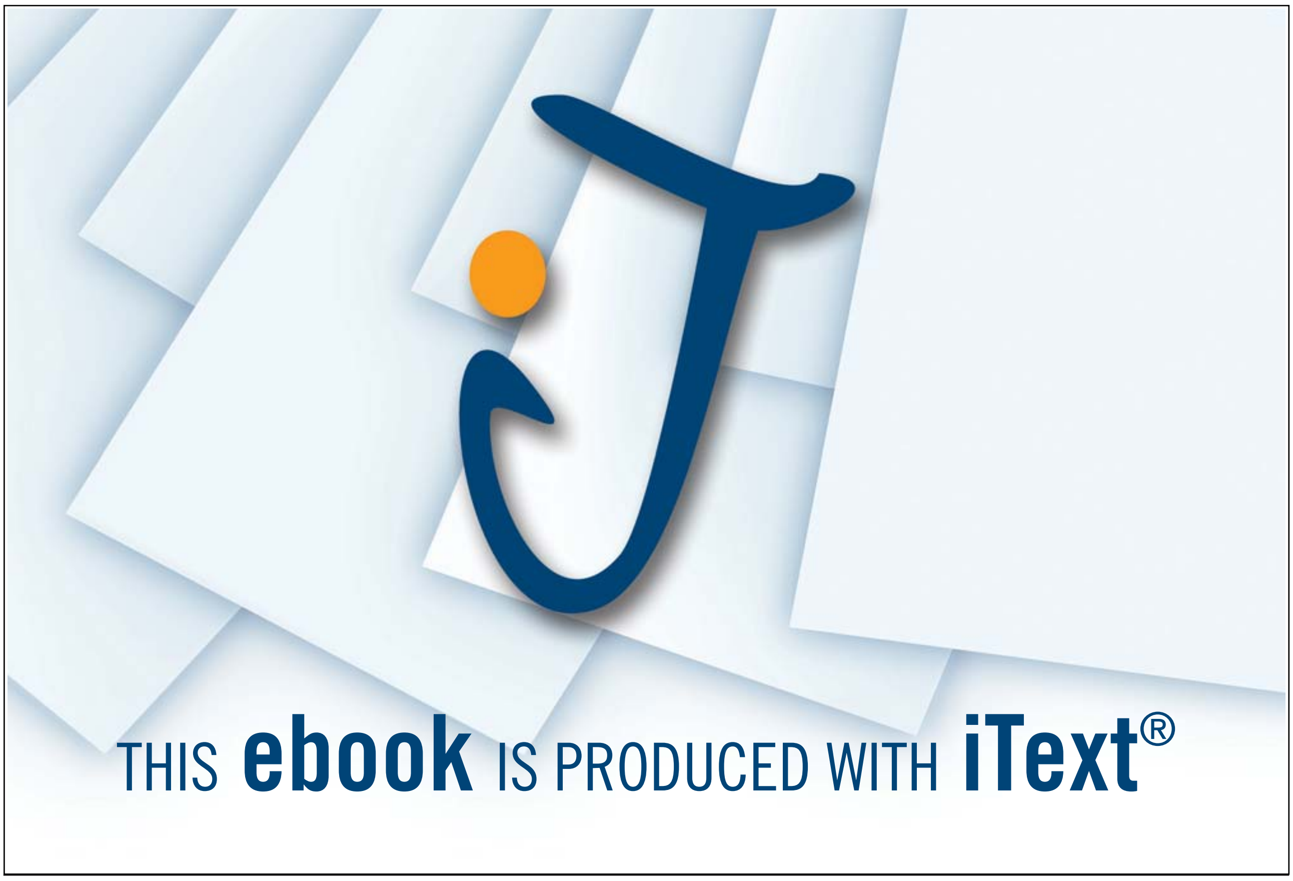


Figure 2.3: Tree Representation of `[a,[b,[],[c,a],e]]`



```
?- 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,#(#(_G315,#(#(b,#(_G321,[])),[])),#(c,#(f(_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.[5] Example:

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

[5]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[6] 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*

```
flatten_3([],[]).                               % clause 1
flatten_3([H|T],L1) :- flatten_3(H,L2),         % clause 2
                       flatten_3(T,L3),         %
                       append(L2,L3,L1).        %
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:

[6] 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*

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

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

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

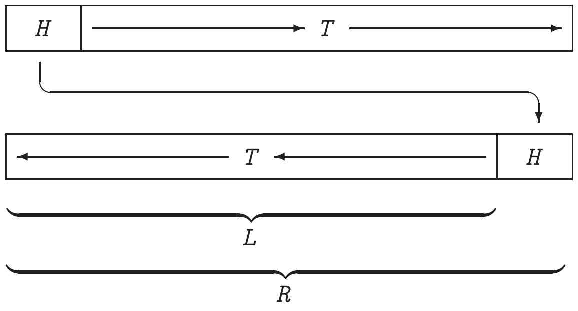


Figure 2.4: Declarative Reading of (P-2.3)

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

```
reverse([],R,R).                              % clause 1
reverse([H|T],Acc,R) :- reverse(T,[H|Acc],R). % clause 2
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*

```
rev_dl([],L-L).                          % clause (a1)
rev_dl([X],[X|L]-L).                     % clause (a2)
rev_dl([H|T],L1-L3) :- rev_dl(T,L1-L2),  % clause (a3)
                       rev_dl([H],L2-L3). %

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`

```
rev_dl([],L-L).                               % clause (b1)
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:

$$A \quad :- \quad B_1, ..., B_m, C, B_{m+1}, ..., B_n. \tag{2.1}$$
$$C \quad :- \quad D_1, ..., D_k. \tag{2.2}$$

Then the clause

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

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 `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)*

```
rev_dl([],L-L) :- true.
rev_dl([X],[X|L]-L) :- true.
rev_dl([U|V],W1-W3) :- rev_dl(V,W1-W2),
                       rev_dl([U],W2-W3).
```

and then seek to unify in turn the head of each with the term `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).[7]

Prolog Code P-2.8: *Partially transformed clauses*

```
rev_dl([],L-L).                                % clause (a1)
rev_dl([X],[X|L]-L).                           % clause (a2)
rev_dl([H|T],L1-L2) :- rev_dl(T,L1-[H|L2]).  % clause (a3.1)
rev_dl([H|T],L1-L3) :- rev_dl(T,L1-L2),      % clause (a3.2)
                       rev_dl([],L2-W),      %
                       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).[8] 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*.

[7]Notice that some of the variables are renamed when writing down (P-2.8).
[8]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

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

$$D \quad :- \quad C. \tag{2.5}$$

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.[9] We can therefore augment the database by the clause

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

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

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

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).[10]

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
```

[9]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.

[10]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

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

  or, in a more readable format,

  ```
  rev_dl([],W1-[E|W2]) :- rev_dl([E],W1-W2). % clause (e)
  ```

- Finally, we use clause (e) to obtain clause (c2) by unfolding on the call *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):

```
rev_dl(T,L1-L2), rev_dl([],L2-W), rev_dl([H],W-L3).
```

We infer by clause (c1) that

```
L2 = W.
```

and therefore

```
rev_dl([H],L2-L3).
```

from which by clause (c2)

```
L2 = [H|L3].
```

and therefore

```
rev_dl(T,L1-[H|L3]) :- true.
```

Unfold now clause (a3.1) to get

```
rev_dl([H|T],L1-L3).
```

which is indeed the head of clause (a3.2).[11]

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

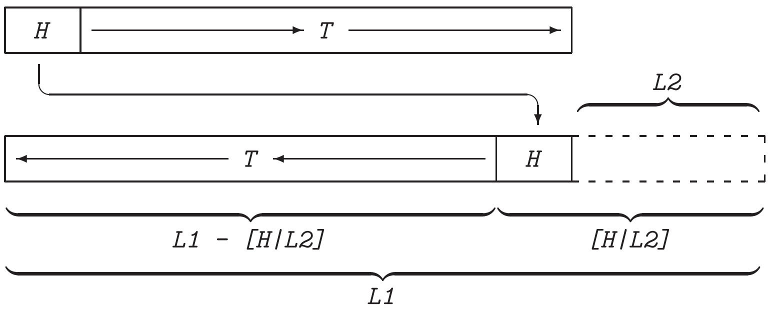


Figure 2.5: Illustrating Clause (b2) in (P-2.6)

> 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`.[12]

■

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.

(a) Give a declarative reading of Fig. 2.6.

(b) Define a new version of `reverse/2` based on Fig. 2.6.

(c) Obtain your new version also by unfolding clause (b2).

[11]The foregoing reasoning is an instance of the application of the *Implication Introduction Rule* in Propositional Calculus.

[12]`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?

■

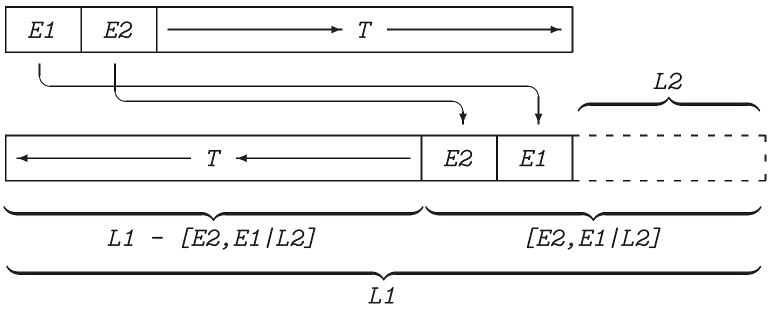


Figure 2.6: Illustrating Exercise 2.9

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*

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

```
dijkstra(Items,Grouped) :- reds(Items,R),
                           whites(Items,W),
                           blues(Items,B),
                           append(R,W,RandW),
                           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*

```
colour(_,[],[]).
colour(Clr,[col(Object,Clr)|T],[col(Object,Clr)|L]) :- colour(Clr,T,L).
colour(Clr,[col(_,Colour)|T],L) :- Colour \= Clr,
                                   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***

```
dijkstra(Items,Grouped) :- colour(red,Items,R),
                           colour(white,Items,W),
                           colour(blue,Items,B),
                           append(R,W,RandW),
                           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.

I joined MITAS because
I wanted **real responsibility**
The Graduate Programme
for Engineers and Geoscientists
www.discovermitas.com
Month 16
I was a construction supervisor in the North Sea advising and helping foremen solve problems
Real work
International opportunities
Three work placements
MAERSK

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

Prolog Code P-2.13: *Definition of* colour_dl/3

```
colour_dl(_,[],L-L).
colour_dl(Clr,[col(Object,Clr)|T],[col(Object,Clr)|L1]-L2) :-
   colour_dl(Clr,T,L1-L2).
colour_dl(Clr,[col(_,Colour)|T],L1-L2) :-
   Colour \= Clr,
   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

```
dijkstra_dl(Items,L1-L4) :- colour_dl(red,Items,L1-L2),
                            colour_dl(white,Items,L2-L3),
                            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*

```
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

```
colour([],[],[],[]).
colour([col(Object,red)|T],[col(Object,red)|R],W,B)     :- colour(T,R,W,B).
colour([col(Object,white)|T],R,[col(Object,white)|W],B) :- colour(T,R,W,B).
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

```
dijkstra(Items,Grouped) :- colour(Items,R,W,B),
                           append(R,W,RandW),
                           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.)

(a) 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*.

(b) 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 $[\mathbf{p}_1, \mathbf{p}_2, \cdots, \mathbf{p}_N]$, say, then *transform/2* will return in *NewPs* the 'rotated' list $[\mathbf{p}_2, \cdots, \mathbf{p}_N, \mathbf{p}_1]$. (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.

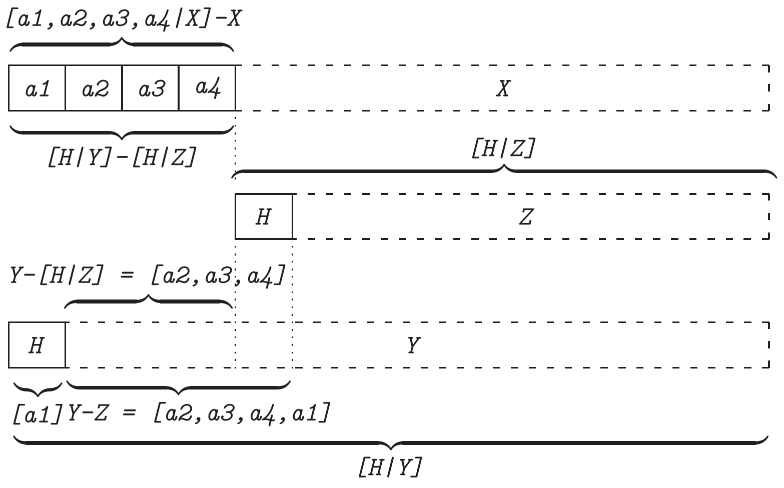


Figure 2.7: Rotating by Difference Lists

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`

```
averages(L,A) :- aver([-1,1|L],A), !.              % clause 1

aver([_,0,_|T],T).                                 % clause 2
aver(X,Result) :- av_rotate(X,Y),                  % clause 3
                  aver(Y,Result).                  %

av_rotate([H1,H2|Y],L) :- Last is (H1 + H2)/2,     % clause 4
                          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.

$$
\begin{array}{l}
averages([4,8,16,32],\ A) \overset{①}{\rightsquigarrow} \\
aver([-1,1,4,8,16,32],\ A) \overset{③}{\rightsquigarrow} aver([1,4,8,16,32,0],\ A) \overset{③}{\rightsquigarrow} \\
aver([4,8,16,32,0,2.5],\ A) \overset{③}{\rightsquigarrow} aver([8,16,32,0,2.5,6],\ A) \overset{③}{\rightsquigarrow} \\
aver([16,32,0,2.5,6,12],\ A) \overset{③}{\rightsquigarrow} aver([32,0,2.5,6,12,24],\ A) \rightsquigarrow \\
aver([32,0,2.5\,|\,[6,12,24]],\ A) \overset{②}{\rightsquigarrow} A = [6,12,24]) \overset{①}{\rightsquigarrow} success
\end{array}
$$

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*

```
dl([],L-L).                             % clause 1
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*

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

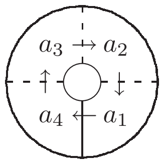


Figure 2.9: Rotating a List with Four Entries

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).[13]

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

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

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

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

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

[13]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.[14] (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:[15]

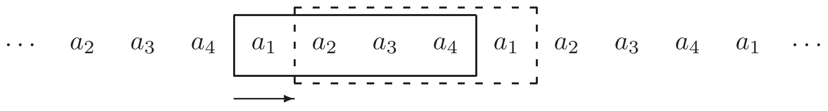
```
?- ws(Ws), ps(_Ps), ds(_Ds), time(pta(0.25,_Ps,_Ds,Ws,W,801)).
% 41,335 inferences in 0.38 seconds (108776 Lips)
Ws = [-0.51, -0.35, 0.13] W = [3.018, 4.1935, -39.87]
?- ws(Ws), ps(_Ps), ds(_Ds), dl(_Ps,_PsDL), dl(_Ds,_DsDL),
   time(pta(0.25,_PsDL,_DsDL,Ws,W,801)).
% 28,519 inferences in 0.28 seconds (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[16]

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.



$\cdots$ a_2 a_3 a_4 a_1 a_2 a_3 a_4 a_1 a_2 a_3 a_4 a_1 $\cdots$

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

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

We tile the entire plane with copies of **A** and shift a three by four frame from **A** to South–East to obtain the rotated matrix

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

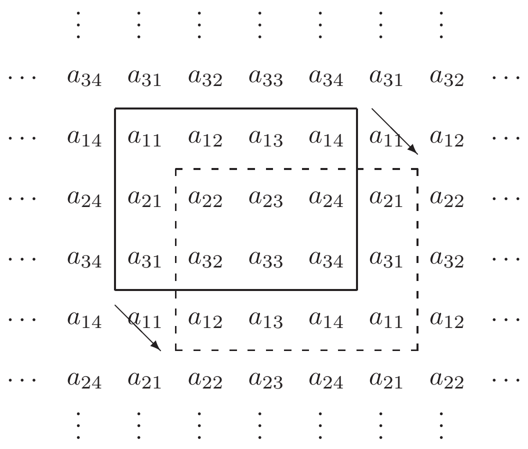


Figure 2.11: The Original Matrix **A** and its Rotated Image $\mathbf{A}^{(rot)}$

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 **A** in (2.7) may be defined by (P-2.23).

Prolog Code P-2.23: *Definition of* `matrix_a/1`

```
matrix_a([[ a11, a12, a13, a14],
          [ a21, a22, a23, a24],
          [ 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).

[14]All code pertinent to the Perceptron Training Algorithm is replicated in the file `dl.pl`.
[15]A similar result applies when calling `pta/6` with a *variable* in its last argument.
[16]This section and the next are based on [4]. The author thankfully acknowledges the permission by Elsevier to republish this material here.

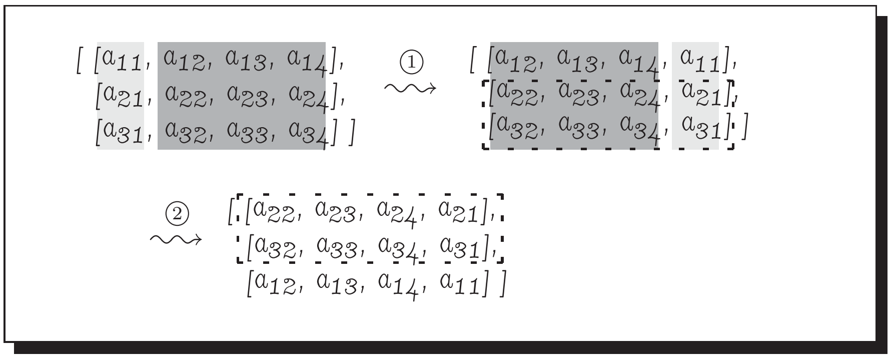


Figure 2.12: Hand Computations for Rotation in the Plane

"I studied English for 16 years but...
...I finally learned to speak it in just six lessons"
Jane, Chinese architect
ENGLISH OUT THERE
Click to hear me talking before and after my unique course download

Prolog Code P-2.24: *Definition of* `rot_rows/2`

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

```
rot_matrix(M,R) :- rot_rows(M,[H|T]), % clause 1
                   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`

```
dl2([],L-L).
dl2([H|T],[HDL|L1]-L2) :- dl(H,HDL), !,
                          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`

```
rot_rows_dl(L-_,Y-Y) :- var(L).
rot_rows_dl([[H|T1]-[H|T2]|Ls1]-Ls2,[T1-T2|R1]-R2) :-
   rot_rows_dl(Ls1-Ls2,R1-R2).
```

Prolog Code P-2.28: *Definition of* `rot_matrix_dl/2`

```
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

$$\begin{aligned} u + \alpha v + \beta w &= r && (2.8)\\ \gamma u + v + \delta w &= s && (2.9)\\ \lambda u + \rho v + w &= t && (2.10) \end{aligned}$$

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

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

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

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

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

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

We have thus completed one cycle of the iteration scheme known as the *Gauss–Seidel Method*[17] (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

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

  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.*[18]

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

$$x_1 = b_1 - a_{12}x_2 - \ldots - a_{1n}x_n \tag{2.15}$$

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.)

[17]The special feature of this iteration scheme is that updated values are used as soon as they become available.
[18]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 $Iterations \neq 0$
   do { Update (the first entry of) $\mathbf{x}$ by (2.15)
        $\mathbf{A} \leftarrow$ ROTATEMATRIX($\mathbf{A}$)
        $\mathbf{b} \leftarrow$ ROTATELIST($\mathbf{b}$)
        $\mathbf{x} \leftarrow$ ROTATELIST($\mathbf{x}$)
        $Subscripts \leftarrow$ ROTATELIST($Subscripts$)
        $Iterations \leftarrow Iterations - 1$
 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*

```
g_seidel(in([[First|Rest]|OtherRows],
             [B|OtherBs],[_|OtherXs],[S|OtherSs]),
         out(NewAs,NewBs,NewXs,NewSs)) :-
   dot_product(Rest,OtherXs,P),
   NewX is B - P,
   rot_matrix([[First|Rest]|OtherRows],NewAs),
   append(OtherBs,[B],NewBs),
   append(OtherXs,[NewX],NewXs),
   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*

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

Example 2.1.[19] 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 $x_1^{(0)} = \ldots = x_4^{(0)} = 100$ 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

[19]Source: [10].

Excellent Economics and Business programmes at:
university of groningen
"The perfect start of a successful, international career."
AACSB ACCREDITED
CLICK HERE
to discover why both socially and academically the University of Groningen is one of the best places for a student to be
www.rug.nl/feb/education

```
?- 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).

■