

Chapter 3

Program Manipulations

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

3.1 Simple Database Operations

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

The Round Table

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

Prolog Code P-3.1: Initial definition of right_to/2

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

Exercise 3.1. Write queries to answer the following questions:

- Who is seated to the right of Adam?
- To whom is Clara the right neighbour?
- Who are the neighbours of George?

Define Prolog rules for

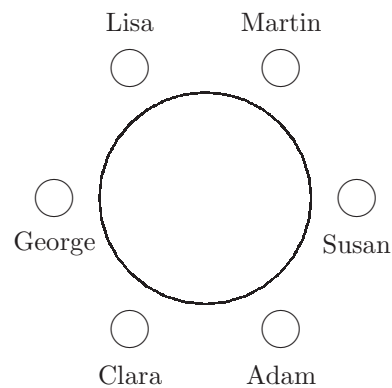


Figure 3.1: The Initial Seating Arrangement

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

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

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

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

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

■

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

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

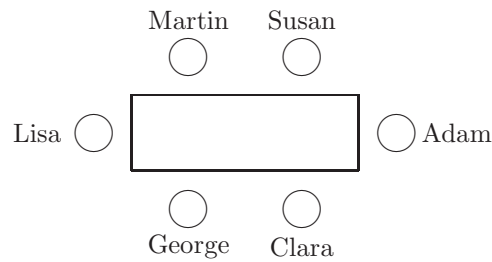


Figure 3.2: Rectangular Table

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

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

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

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

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

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

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

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

```

george clara adam susan martin lisa
Yes

```



Departures and Arrivals

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

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

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

.....Alcatel-Lucent 

www.alcatel-lucent.com/careers

What if you could build your future and create the future?

One generation's transformation is the next's status quo. In the near future, people may soon think it's strange that devices ever had to be "plugged in." To obtain that status, there needs to be "The Shift".



Prolog Code P-3.2: New facts for *right_to/2*

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

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

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

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

3.1.1 Basic Database Manipulation

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

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

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

or interactively by

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

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

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

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

```

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

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

Adds to the database the clause in *Term*. Example: a possible (reasonable) definition by *assert/2* of a predicate *near/2* for the Round Table example may be achieved by²

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

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

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

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

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

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

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

3.1.2 Changing the Database

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

- George leaves the table (Fig. 3.3).

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

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

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

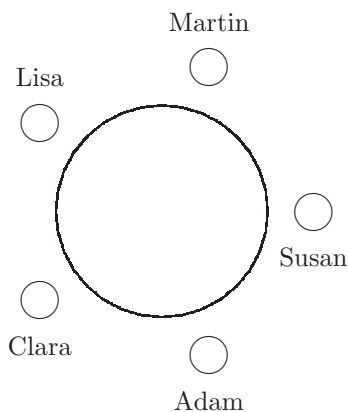


Figure 3.3: After George's Departure

Prolog Code P-3.3: Updated definition of *right_to/2*

```

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

```

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

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

```

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

```

```

X = susan
Yes

```

Notice that due to the previous query the predicate *right_to/2* is now dynamic. It is now defined in the database by the facts in (P-3.4).³

Prolog Code P-3.4: Final definition of *right_to/2*

```

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

```

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

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

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

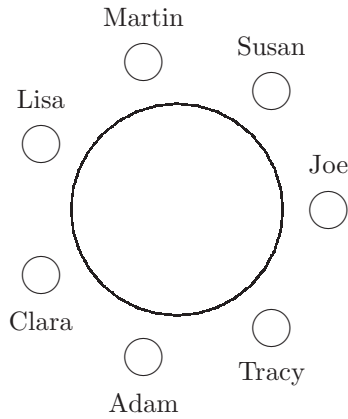


Figure 3.4: After Tracy’s and Joe’s arrival

Maastricht University *Leading in Learning!*

Join the best at the Maastricht University School of Business and Economics!

Top master's programmes

- 33rd place Financial Times worldwide ranking: MSc International Business
- 1st place: MSc International Business
- 1st place: MSc Financial Economics
- 2nd place: MSc Management of Learning
- 2nd place: MSc Economics
- 2nd place: MSc Econometrics and Operations Research
- 2nd place: MSc Global Supply Chain Management and Change

Sources: Keuzegids Master ranking 2013; Elsevier 'Beste Studies' ranking 2012; Financial Times Global Masters in Management ranking 2012

Visit us and find out why we are the best!
Master's Open Day: 22 February 2014

Maastricht University is the best specialist university in the Netherlands (Elsevier)

www.mastersopenday.nl



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

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

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

After this, the database will look as follows.

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

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

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

upon which the database is as shown below.

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

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

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

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

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

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

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

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

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

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

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

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

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

Table 3.1: Cases for *swap_neighbours/2*

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

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

After this query, the database will look as follows.

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

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

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

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

■

Exercise 3.5. The predicate *lives_in/2* is defined by (P-3.5).

Prolog Code P-3.5: Initial definition of *lives_in/2*

```

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

```

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

Prolog Code P-3.6: Final definition of *lives_in/2*

```

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

```

3.1.3 File Modifications

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

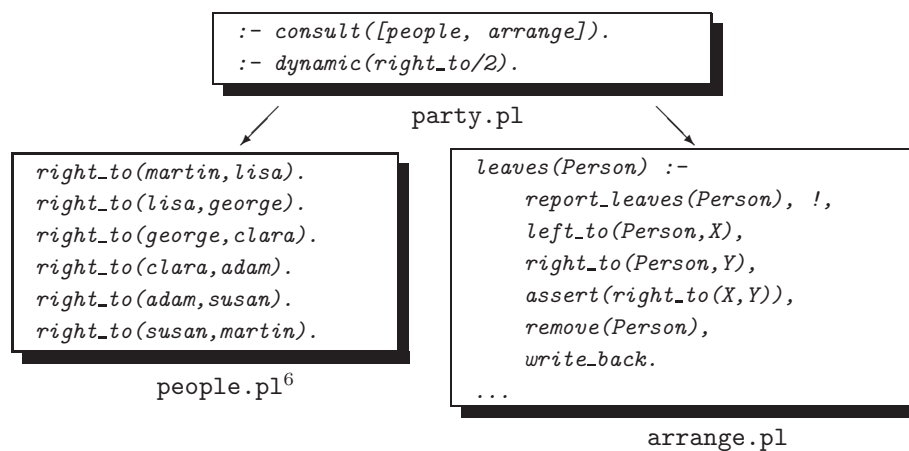


Figure 3.5: File Organization for the Round Table Example

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

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

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

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

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

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

To understand the above query, we note that

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



agency: cdfg - © Photonstop

> Apply now

REDEFINE YOUR FUTURE
**AXA GLOBAL GRADUATE
PROGRAM 2015**

redefining / standards 

```

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

```

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

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

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

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

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

```

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

```

■

3.1.5 Automated Saving of Selected Predicates

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

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

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

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

```

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

```

```

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

```

my_predicate/2 will serve as an auxiliary for *save_predicates_to/2* and it is defined in (P-3.7).

Prolog Code P-3.7: Definition of *my_predicate/2*

```

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

```

The built-in predicates *current_predicate/2*, *predicate_property/2* and *functor/3* are used in this largely self-documenting definition.⁷ The goals 2–4 in the body of *my_predicate/2* are designed to filter out names of predicates which are not user-defined.

Embedding *my_predicate/2* into a failure driven loop (see p. 77) gives rise to (P-3.8), the first clause of *save_predicates_to/2*.

Prolog Code P-3.8: First clause of *save_predicates_to/2*

```

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

```

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

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

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

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

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

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

```

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

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

```

Figure 3.7: The File `committee.pl`

(5) Directives will be lost.

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

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

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

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

```

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

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

```

Figure 3.8: The File `committee.pl`

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

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

```

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

```


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

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

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

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

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



Empowering People. Improving Business.

BI Norwegian Business School is one of Europe's largest business schools welcoming more than 20,000 students. Our programmes provide a stimulating and multi-cultural learning environment with an international outlook ultimately providing students with professional skills to meet the increasing needs of businesses.

BI offers four different two-year, full-time Master of Science (MSc) programmes that are taught entirely in English and have been designed to provide professional skills to meet the increasing need of businesses. The MSc programmes provide a stimulating and multi-cultural learning environment to give you the best platform to launch into your career.

- MSc in Business
- MSc in Financial Economics
- MSc in Strategic Marketing Management
- MSc in Leadership and Organisational Psychology

www.bi.edu/master

BI NORWEGIAN BUSINESS SCHOOL

EFMD EQUIS ACCREDITED



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

■

Built-in Predicate: *->/2*

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

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

Notice in particular that

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

3.1.6 Miniproject: Modelling a Stamp Collection

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

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

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

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

```

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

```

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

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

- Show all stamps with denomination 50.

```

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

```

- Show all stamps from the set *Castles*.

```

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

```

- Show all stamps issued between 1875 and 1883.

```

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

```

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

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

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

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

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

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

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

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

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

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

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

(A concise recursive solution is sought.)

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

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

Yes

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

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

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

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

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

Yes

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

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

Yes

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

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

Yes



Need help with your dissertation?

Get in-depth feedback & advice from experts in your topic area. Find out what you can do to improve the quality of your dissertation!

Get Help Now



Go to www.helpmyassignment.co.uk for more info



Helpmyassignment



3.2 Case Study: Automated Unfolding

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

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

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

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

```

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

```

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

```

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

```

3.2.1 Elementary Unfolding

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

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

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

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

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

The following observations on these figures are in order.

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

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



Brain power

By 2020, wind could provide one-tenth of our planet's electricity needs. Already today, SKF's innovative know-how is crucial to running a large proportion of the world's wind turbines.

Up to 25 % of the generating costs relate to maintenance. These can be reduced dramatically thanks to our systems for on-line condition monitoring and automatic lubrication. We help make it more economical to create cleaner, cheaper energy out of thin air.

By sharing our experience, expertise, and creativity, industries can boost performance beyond expectations. Therefore we need the best employees who can meet this challenge!

The Power of Knowledge Engineering

Plug into The Power of Knowledge Engineering.
Visit us at www.skf.com/knowledge

SKF



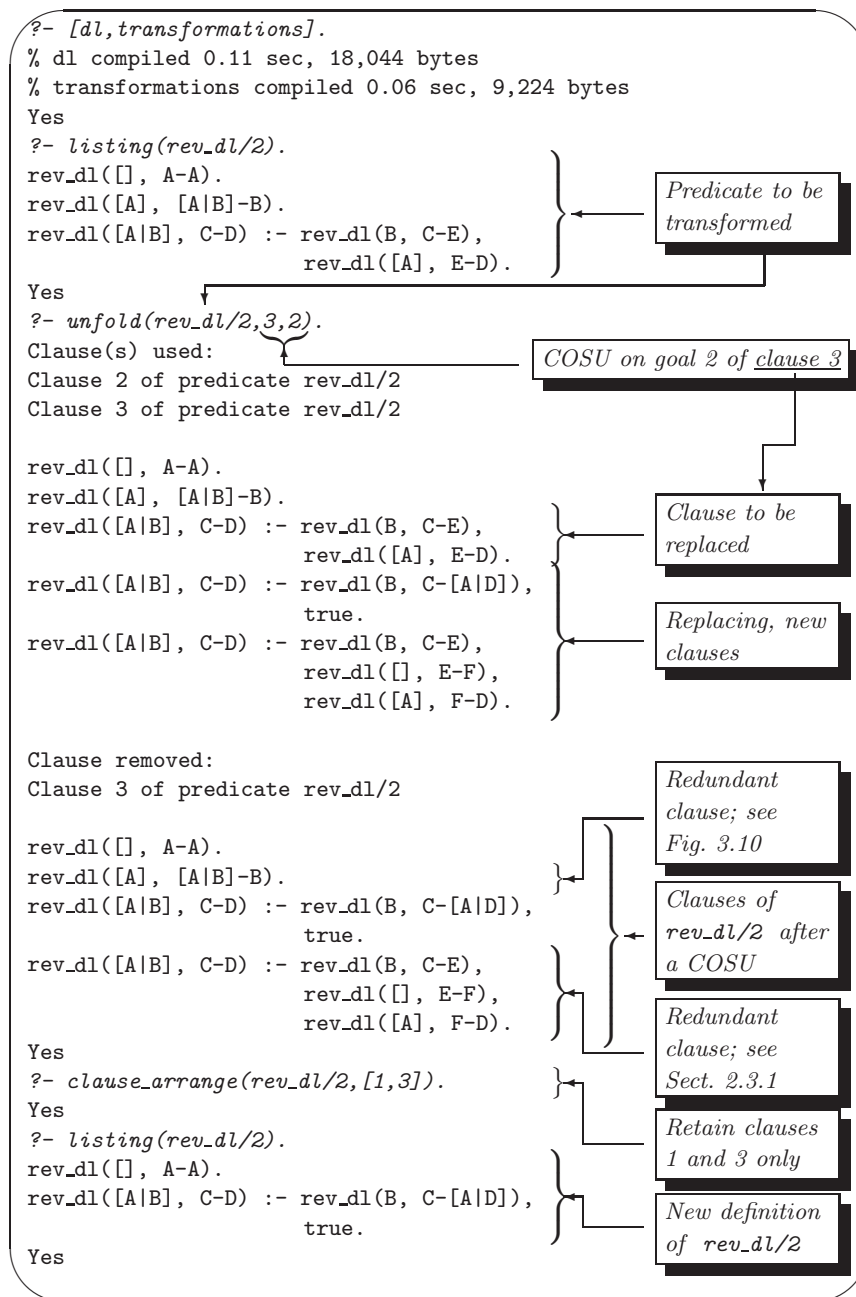


Figure 3.9: Interactive Prolog-Assisted Program Transformation: Session I


```

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

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

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

Manual input
of rev_dl/2

Unfold on
goal 1 of
clause 2 using
clause 1

Old clauses

New clause

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

“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





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

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

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

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

```

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

```

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

```

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

```

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

Built-in Predicates: *nth_clause/3* and *clause/3*

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

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

may be retrieved by

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

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

```
?- nth_clause(c(.,.),Index,Ref).
Index = 1 Ref = 1791577 ;
Index = 2 Ref = 1791614 ;
Index = 3 Ref = 1791649 ;
No
```

- **Fig. 3.13:** The previous two steps are repeated and then *Head2* is unified with *Entry1*, essentially completing the unfolding operation. Notice in particular that the effect of unifying *Head2* with *Entry1* ‘ripples through’ to all other variables: for example, as expected, in *Head1* the second and third arguments become identical while this was not the case before unification (see Fig. 3.11).
- **Fig. 3.14:** Subsequent to the steps above, we first assemble in *L* the list of goals for the new clause; we use here the (fairly straightforward) user-defined predicate *concat3/4*. Then, *conj/2* is used again (now in the ‘reverse’ direction) to create the term *NewBody*, the conjunction of terms in *L*. Finally, the new clause is written to the database, confirmed also by the next query using *listing/1*.

The definition of *elementary_unfolding/5* in *transformations.pl* follows the query shown in Fig. 3.14. The auxiliary predicates used therein won’t be discussed here; the way conjunctions are composed/decomposed by *conj/2* is noteworthy, however. This is accomplished within *conj/2* via the auxiliaries *conjunction(+List,+Acc,-Term)* and *conjuncts(+Term,+Acc,-List)* whose working is illustrated below.

```
?- conjunction([t(X),u(Y,a),v(b,X)],s(Y),C), conjuncts(C,[],L).
X = _G492
Y = _G497
C = v(b, _G492), u(_G497, a), t(_G492), s(_G497)
L = [s(_G497), t(_G492), u(_G497, a), v(b, _G492)]
```

They are defined in (P-3.13) and (P-3.14) by the accumulator technique.¹⁰

Prolog Code P-3.13: Definition of conjunction/3

```

1 conjunction([], Conj, Conj).
2 conjunction([H|T], Acc, Conj) :- conjunction(T, (H, Acc), Conj).
    
```

¹⁰In (P-3.14) we implicitly use the fact that Prolog's conjunction is right-associative. The two queries below thus generate the same response:

```

?- conjuncts((v(b,X), u(Y, a), t(X), s(Y)), [], L).
X = _G409 Y = _G411
L = [s(_G411), t(_G409), u(_G411, a), v(b, _G409)]
?- conjuncts((v(b,X), (u(Y, a), (t(X), s(Y))))), [], L).
X = _G433 Y = _G435
L = [s(_G435), t(_G433), u(_G435, a), v(b, _G433)]
    
```

What will Prolog's response be to the query below?

```

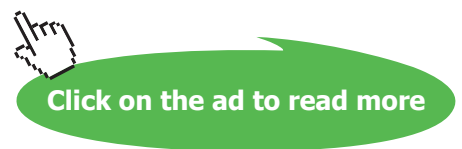
?- conjuncts(((v(b,X), u(Y, a)), t(X)), s(Y)), [], L).
    
```

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



Prolog Code P-3.14: Definition of *conjuncts/3*

```

1 conjuncts(Term,Acc,[Term|Acc]) :- not(funcator(Term,',',2)).
2 conjuncts(Term,Acc,L) :- funcator(Term,',',2),
3                       arg(1,Term,Term1),
4                       arg(2,Term,Term2),
5                       conjuncts(Term2,[Term1|Acc],L).

```

3.2.2 Complete One Step Unfolding

Let us now assume that we want to unfold clause 4 of *a/5* on its second goal. We can do this by repeatedly using *elementary_unfolding/5* in an obvious manner:

```

?- elementary_unfolding(a/5,4,2,c/2,K).
K = 1 ; K = 2 ; K = 3 ;
No

```

```

?- listing(a/5).
a(A, A, A, A, A).
a(A, B, A, B, A) :- m(A, B).
a(A, B, C, B, A) :- n(A, n(B, C)), b(A, B), e(B, A).
a(A, B, C, D, E) :- b(A, B), c(B, C), d(C, D), e(D, E).
a(A, B, C, D, E) :- b(A, B), f(B), m(B, C), d(C, D), e(D, E).
a(A, B, C, D, E) :- b(A, B), B is C+1, d(C, D), e(D, E).
a(A, B, B, C, D) :- b(A, B), f(B), g(B), d(B, C), e(C, D).

```

In doing so, the following steps have been carried out:

1. We have visually identified *c(V,W)* as goal 2 in clause 4 of *a/5*.
2. We have attempted (and successfully completed) by backtracking an elementary unfolding operation with each of the clauses of *c/2*.

To complete the task, we would also need to

3. Remove clause 4 of *a/5* from the database.

Step 2 is more concisely implemented by a failure driven loop thus

```

?- elementary_unfolding(a/5,4,2,c/2,K), fail.
No

```

Within the same failure driven loop we may integrate Step 1 by attempting an elementary unfolding operation with *each* predicate in the database. The generation of all predicates may be accomplished by¹¹

```

?- current_predicate(Fun,Head),
   not(predicate_property(Head,built_in)),
   not(predicate_property(Head,imported_from(_))),
   not(predicate_property(Head,foreign)),
   functor(Head,Fun,Arity).

```

¹¹The same functionality (i.e. retrieval from the database of all user-defined predicates) is achieved by the almost identical predicate *my_predicate/2* from p. 88.

```

Fun = a
Head = a(_G1380, _G1381, _G1382, _G1383, _G1384)
Arity = 5 ;
...
Fun = c
Head = c(_G1380, _G1381)
Arity = 2 ;
...

```

Embedding this within the earlier failure driven loop will essentially implement *unfold/3*:

```

?- current_predicate(Fun,Head),
   not(predicate_property(Head,built_in)),
   not(predicate_property(Head,imported_from(_))),
   not(predicate_property(Head,foreign)),
   functor(Head,Fun,Arity),
   elementary_unfolding(a/5,4,2,Fun/Arity,K), fail.

```

No

For further details on the definition of *unfold/3* the reader is referred to the file `transformations.pl`. (Noteworthy is perhaps the use in Step 3 of the built-in predicate *erase/1*.)

Built-in Predicate: *erase(+Ref)*

erase(+Ref) removes the clause with reference *Ref* from the database. Example:

```

?- dynamic(num/1), ((member(_I,[1,2,3]), assert(num(_I)),
   fail); true), listing(num/1).
num(1).
num(2).
num(3).
?- nth_clause(num(_),2,Ref), erase(Ref), listing(num/1).
num(1).
num(3).
Ref = 3904727

```

Exercise 3.14. Use the predicate *unfold/3* to solve Exercise 2.9, Part (c). ■

Self-unfolding

There may seem a subtle problem with our implementation of *unfold/3* which we want to address now.

In the definition of *unfold/3* we write (within a failure driven loop) to the database new clauses via *elementary_unfolding/5* which itself ‘feeds on’ clauses (in its fourth argument) that are retrieved from the database. This construction could conceivably give rise to an infinite loop in the case of what was termed ‘self-unfolding’ in Sect. 2.3.1, p. 52. This cannot happen, however, since a search tree under consideration by Prolog won’t be affected by database changes created by the search itself. The following simple interactive session illustrates this point.

```

?- listing(num/1).
num(1).
Yes
?- num(X), Y is 2 * X, assert(num(Y)), fail.
No
?- listing(num/1).
num(1).
num(2).
Yes

```

Had the search tree been affected by the database changes immediately we would have expected in the database infinitely many clauses of *num/1* like

```
num(1). num(2). num(4). ...
```

The session shown in Fig. 3.9 (involving self-unfolding of the predicate *rev_dl/2*) confirms indeed that *unfold/3* does not cause looping.

3.2.3 Rearranging Clauses

Clauses of a predicate may be rearranged by *clause_arrange/2* as illustrated in Fig. 3.9. To this end, the following auxiliary predicates have been defined:

- *all_clauses/2* collects all clauses of a predicate into a list of terms. Example:

```

?- all_clauses(c/2,L).
L = [ (c(_G368, _G369) :- f(_G368), m(_G368, _G369)),
      (c(_G350, _G351) :- _G350 is _G351+1),
      (c(_G331, _G331) :- f(_G331), g(_G331))]

```

all_clauses/2 is defined by

```

all_clauses(Fun/Arity,List) :-
    functor(Pred,Fun,Arity),
    findall((Head :- Body),
           (nth_clause(Pred,_,Ref),
            clause(Head,Body,Ref)), List).

```

- *arrange/3* selects (a subset of) the entries of list as specified by a list of integers in the first argument. Example:

```

?- arrange([4,3,5],[a,b,c,d,e,f],L).
L = [d, c, e]

```

arrange/3 is defined by

```

arrange(IntList,InL,OutL) :-
    findall(E,(member(M,IntList), nth1(M,InL,E)),OutL).

```

Built-in Predicate: *nth1(?Index, ?List, ?Elem)*

nth1/3 is used to select a specified entry from a list. Example:

```
?- nth1(3, [a, b, c, d, e, f], E).
E = c
```

The definition of *clause_arrange/2* in terms of the two auxiliaries is fairly straightforward; see the file *transformations.pl* for details.

Exercise 3.15. Use the predicate *unfold/3* to carry out a Complete One Step Unfolding on an appropriately chosen goal in one of the clauses of *flatten_dl/2* from Sect. 2.2.3. After some removal and rearranging of clauses via *clause_arrange/2*, you should arrive at a version of *flatten/2* which is more efficient than the earlier ones. Demonstrate the gain in speed by an experiment akin to the one carried out in Exercise 2.7.

gaiTEYE[®]
Challenge the way we run

EXPERIENCE THE POWER OF
FULL ENGAGEMENT...

.....

RUN FASTER.
RUN LONGER..
RUN EASIER...

READ MORE & PRE-ORDER TODAY
WWW.GAITEYE.COM

Exercise 3.16. You will have seen in Exercise 3.15 that *unfold/3* places the new clauses *after* the existing ones. To observe the original order, the new clauses had to be subsequently moved by *clause_arrange/2* to the position of the clause they were replacing. Write a predicate *cosu/3* which performs a Complete One Step Unfolding and then restores the predicates' order.¹² For example, the suggested solution of Exercise 3.15 (p. 162) could then be achieved simply by

```
?- cosu(flatten_dl/2,2,2).
...
?- listing(flatten_dl/2).
flatten_dl([], A-A).
flatten_dl([A], B-C) :- flatten_dl(A, B-C),
                       true.
flatten_dl([A, B|C], D-E) :- flatten_dl(A, D-F),
                             flatten_dl(B, F-G),
                             flatten_dl(C, G-E).

flatten_dl([A|B], C-D) :- flatten_dl(A, C-[B|D]),
                          true.
flatten_dl(A, [A|B]-B).
```

Note. When using *clause_arrange/2*, you will have to be able to generate integer lists with specified bounds. The built-in predicate *between/3* may be used to achieve this.

■

3.3 Dijkstra's Dutch Flag Problem Revisited

3.3.1 Problem Generalization and First Solution

Dijkstra's Dutch Flag Problem from Sect. 2.4 may be generalized as follows:

- The items may be of any colour and any number of colours may occur.
- The items are to be grouped to a certain order of colours as specified by the user in some list *Colours*. This list need not include all the items' colours and may include colours not assigned to any of the items. As before, within each colour group the items' original order should be retained.

We call the predicate to be defined *dijkstra(+Colours,+Items,-Grouped)* and illustrate its desired behaviour by an example. Take the list of items

```
new_items([col(soot,black), col(tomato,red), col(nut,brown),
           col(milk,white), col(snow,white), col(coal,black),
           col(bile,green), col(bark,brown), col(ocean,blue),
           col(grass,green), col(apple,red), col(blood,red),
           col(night,black), col(sky,blue)]).
```

¹²To retrieve the number of clauses of a predicate, you should use the built-in predicate *predicate_property/2* in the form *predicate_property(+Pred,number_of_clauses(-ClauseNumber))*.

and sort it in the order *black, blue, violet, green, red* and *white*. (Notice that *brown* is not one of the colours listed here, nor is there any item whose colour is *violet*.) The expected behaviour of *dijkstra/3* is as follows.¹³

```
?- new_items(_Items),
   dijkstra([black,blue,violet,green,red,white],_Items,Grouped).
Grouped = [col(soot,black), col(coal,black), col(night,black),
           col(ocean,blue), col(sky,blue), col(bile,green),
           col(grass,green), col(tomato,red), col(apple,red),
           col(blood,red), col(milk,white), col(snow,white)]
```

dijkstra/3 solves the *original* Dutch Flag problem from Sect. 2.4 if its first argument is unified with *[red, white, blue]*.

On inspection of *dijkstra/2* from Sect. 2.4.3 (the version based on difference lists) it is seen that the *current, specific* problem would be solved by *dijkstra/2* if *dijkstra_dl/2* had been defined by the clause

```
dijkstra_dl(Items,L1-L7) :- colour_dl(black,Items,L1-L2),
                           colour_dl(blue,Items,L2-L3),
                           colour_dl(violet,Items,L3-L4),
                           colour_dl(green,Items,L4-L5),
                           colour_dl(red,Items,L5-L6),
                           colour_dl(white,Items,L6-L7).
```

This suggests introducing a predicate *replace_dijkstra_dl(+Colours)* for replacing the existing definition of *dijkstra_dl/2* in the database by the desired one. Then, *dijkstra/3* may be defined in terms of the old version of *dijkstra/2* thus

```
dijkstra(Colours,Items,List) :- replace_dijkstra_dl(Colours),
                               dijkstra(Items,List).
```

Let us now look at in detail how the change in the database is accomplished.

```
replace_dijkstra_dl(Colours) :-
    dynamic(dijkstra_dl/2),                % goal 1
    retractall(dijkstra_dl(_, _)),        % goal 2
    conjuncts(Items,Colours,L,First,Last), % goal 3
    conj(Body,L),                          % goal 4
    assert(dijkstra_dl(Items,First-Last) :- Body). % goal 5
```

The first two goals are obvious: *dijkstra_dl/2* is made a dynamic predicate and then its existing definition is removed from the database. The rôle of *conjuncts/5* is best illustrated by a sample query.

```
?- conjuncts(Items,[red,white,green],L,First,Last).
Items = _G399
L = [colour_dl(red, _G399, _G402-_G513),
     colour_dl(white, _G399, _G513-_G514),
     colour_dl(green, _G399, _G514-_G403)]
First = _G402
Last = _G403
```

¹³We note in passing that the default maximum number of entries of a list displayed on the terminal by SWI-Prolog is ten. For a *full* display of the twelve-entry list *Grouped*, we issue the prior query

```
?- set_prolog_flag(toplevel_print_options,[max_depth(20)]).
Yes
```

Here, L is unified with the list of terms whose conjunction will form the body of the clause for *dijkstra_dl/2*. The variables *First*, *Last* and *Items* will be used in goal 5 as variables in the head of the clause for *dijkstra_dl/2*. We won't spell out the definition of *conjunctions/5* here but consider some salient points only. The list of terms in the third argument is created by an auxiliary predicate using the accumulator technique; see the source code for details. Perhaps the most imminent question here is how to get hold of an unspecified number of variable names.¹⁴ This is accomplished by *vars/2*,

```
?- vars(5, V).
V = [_G239, _G240, _G241, _G242, _G243]
```

which may be defined as shown below.¹⁵

```
vars(N, Vars) :- functor(Term, dummy, N),
                bagof(Var, Arg^arg(Arg, Term, Var), Vars).
```

The requisite number of variables is generated by the built-in predicate *functor/3*, as in

```
?- functor(Term, dummy, 5).
Term = dummy(_G313, _G314, _G315, _G316, _G317)
```

subsequent to which *bagof/3* is used to collect the variables in a list. In goal 4, we use *conj/2* (which is known from Sect. 3.2.1, p. 102) to unify with *Body* the conjunction of terms for the body of the clause to be created. Finally, in goal 5 the clause is written to the database.

Exercise 3.17. Use *dijkstra/3* to define *dijkstra_st(+Items, -Grouped)* for returning in *Grouped* the entries of *Items* such that

- All entries of *Items* feature in *Grouped*;
- The colours are *sorted* in alphabetical order;
- And, as before, within each colour group, the items' original order is retained.

Example:

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

¹⁴Only at runtime will it be known how many colours *conjunctions/5* holds in its second argument!

¹⁵There are at least two other alternatives for defining *vars/2*. The simplest is by using the built-in predicate *length/2*:

```
?- length(Vars, 5).
Vars = [_G251, _G254, _G257, _G260, _G263]
```

The second one is based on the built-in predicate *=./2* (*univ*) for assembling and disassembling terms. The idea for this implementation of *vars/2* should be clear from the query below.

```
?- functor(Term, dummy, 5), Term =.. [_|Vars].
Term = dummy(_G478, _G479, _G480, _G481, _G482)
Vars = [_G478, _G479, _G480, _G481, _G482]
```

(The predicates *functor/3*, *arg/3* and *univ* will be familiar from Sect. 2.2.1.)

3.3.2 Enhanced Implementations

The predicate *dijkstra(+Colours,+Items,-Grouped)* from Sect. 3.3.1 is inefficient inasmuch as it will require as many passes through *Items* as there are entries in *Colours*. We have seen implementations for the *original* Dutch Flag Problem in Exercise 2.10, requiring a *single* pass only through the input list *Items*. In this section, those versions will be enhanced for solving the problem’s more general formulation.

As in Sect. 3.3.1 before, we want to glean the plan for solving the *general* problem by considering a *specific* example. Let us assume that *Colours* is the list *[black,white,red,green]*. Then, it is easily seen that the plan for the solution of Exercise 2.10 (pp. 152–153) still applies if *colour_dl/4* and *dijkstra_dl/2* are respectively replaced by the predicates *encolour_dl/5* and *endijkstra_dl/2* as shown in Fig. 3.15. Clearly,

```

encolour_dl([],B-B,W-W,R-R,G-G). } ①

encolour_dl([col(Object,black)|T],
            [col(Object,black)|B1]-B2,W1-W2,R1-R2,G1-G2) :-
    encolour_dl(T,B1-B2,W1-W2,R1-R2,G1-G2).
encolour_dl([col(Object,white)|T],
            B1-B2,[col(Object,white)|W1]-W2,R1-R2,G1-G2) :-
    encolour_dl(T,B1-B2,W1-W2,R1-R2,G1-G2).
encolour_dl([col(Object,red)|T],
            B1-B2,W1-W2,[col(Object,red)|R1]-R2,G1-G2) :-
    encolour_dl(T,B1-B2,W1-W2,R1-R2,G1-G2).
encolour_dl([col(Object,green)|T],
            B1-B2,W1-W2,R1-R2,[col(Object,green)|G1]-G2) :-
    encolour_dl(T,B1-B2,W1-W2,R1-R2,G1-G2). } ②

encolour_dl([col(.,_)|T],B1-B2,W1-W2,R1-R2,G1-G2) :-
    encolour_dl(T,B1-B2,W1-W2,R1-R2,G1-G2). } ③

endijkstra_dl(Items,L1-L7) :-
    encolour_dl(Items,L1-L2,L2-L3,L3-L4,L4-L5,L5-L6,L6-L7). } ④
    
```

Figure 3.15: Illustrative Example of Intended Database Updates

the list of colours in *Colours* will be known at runtime only and thus the predicate definitions indicated in Fig. 3.15 should be accomplished by prior database updates. The predicates *def_encolour_dl(+Colours)* and *def_endijkstra_dl(+Colours)* shall be responsible for writing to the database clauses like ①–③ and ④, respectively.

The present problem is more complex than that in Sect. 3.3.1 in two respects: both the number of clauses for, and the arity of the predicate *encolour_dl* will be known at runtime only.

Implementing *def_encolour_dl/1* and *def_endijkstra_dl/1*

The top level definition of *def_encolour_dl/1* is shown in Fig. 3.16. The following features are noteworthy:

- The old definition (if present) of *encolour_dl* (with the same arity as the one to be implemented) is removed from the database.

```

def_encolour_dl(Colours) :-
    length(Colours,N),
    M is N + 1,
    dynamic(encolour_dl/M),
    length(Vars,M),
    Old_Version =.. [encolour_dl/Vars],
    retractall(Old_Version),
    base_clause(Colours,B-Clause),
    assert(B-Clause),
    ((member(Colour,Colours),
     recursive_clause(Colour,Colours,R-Clause),
     assert(R-Clause),
     fail); true),
    catch_all_clause(Colours,C-Clause),
    assert(C-Clause).

```

Figure 3.16: Top Level Definition of *def_encolour_dl/1*

- The auxiliary predicate *base_clause/2* creates the term for the base clause (marked ① in Fig. 3.15), followed by a database update. It is defined by the predicates

```

base_clause(Colours,(Head :- true)) :- length(Colours,N),
                                       base(N,Head).

base(N,Term) :- diffvars1(N,D),
               Term =.. [encolour_dl,[],D].

diffvars1(N,D) :- functor(Term,dummy,N),
                 Term =.. [_|L],
                 diffterms(L,L,D).

```

where *diffvars1/2* produces a list with a given number of differences of pairwise *identical* variables as exemplified by

```

?- diffvars1(3,D).
D = [_G287-_G287, _G288-_G288, _G289-_G289]

```

- The terms for the recursive clauses (marked ② in Fig. 3.15) are created by the auxiliary predicate *recursive_clause/3* and written to the database within a failure driven loop. *recursive_clause/3* reads at the top level as

```

recursive_clause(Colour,Colours,(Head :- Body)) :-
    length(Colours,N),
    diffvars2(N,D),
    head(Colour,Colours,T,D,Head),
    body(T,D,Body), !.

```

where

1. *diffvars2/2* produces a list with a given number of differences of pairwise *distinct* variables,
2. *head/5* produces the term for the head of *encolour*,
3. *body/3* produces the term for the body of *encolour*.

head/5 and *body/3* are respectively defined by

```
head(Colour, Colours, T, D, Head) :-
    comb(Object, Colour, Colours, D, Modified),
    Head =.. [encolour_d1, [col(Object, Colour)|T]|Modified].

body(T, D, Body) :- Body =.. [encolour_d1, T|D].
```

The predicate *comb/5* combines the list of colours with the list of difference terms as exemplified below.

```
?- comb(Object, w, [r, w, g], [R1-R2, W1-W2, G1-G2], M).
Object = _G429
R1 = _G411  R2 = _G412
W1 = _G417  W2 = _G418
G1 = _G423  G2 = _G424
M = [_G411-_G412, [col(_G429, w)|_G417]-_G418, _G423-_G424]
```

In the definition of *comb/5* (not shown here) the accumulator technique is used.

- Finally, the catch-all clause (marked ③ in Fig. 3.15) is created by the auxiliary predicate *catch_all_clause/2* along similar lines to *body/3*. (Its definition is not shown here).

The definition of *def_endijkstra_d1/1* is broadly analogous to that of *catch_all_clause/2* and is not shown here. The full source code for the present version is available in the file *d1.pl*.

Exercise 3.18. In the above development, for simplicity, *def_encolour_d1/1* was defined such that clause ③ in Fig. 3.15 does not contain any reference to the colours to be omitted; this was accomplished by ③ being the last clause. The resulting definition of *encolour_d1* will therefore be sensitive to the ordering of its clauses. This is not ideal, however, as it prevents code to be interpreted declaratively.

Redefine *def_encolour_d1(+Colours)* such that it writes to the database code which is not sensitive to clause reordering.

Hints.

- Aim at excluding the colours not in *Colours* by using the built-in predicate *member/2*. If, for example, *Colours* is unified with *[black, white, red, green]*, then *def_encolour_d1/1* writes instead of ③ the following clause to the database

```
encolour_d1([col(_, Clr)|T], B1-B2, W1-W2, R1-R2, G1-G2) :-
    not(member(Clr, [black, white, red, green])),
    encolour_d1(T, B1-B2, W1-W2, R1-R2, G1-G2).
```

- All we need is a new definition of *catch_all_clause/2*, used in Fig 3.16. Use *conj/2* (known from Sect. 3.2.1, p. 102) to construct the conjunction of the two goals in the body of the new clause of *encolour_d1*. Each of the two conjuncts will be obtained by using *=../2*.
- The solution is in *d1.pl*.

■

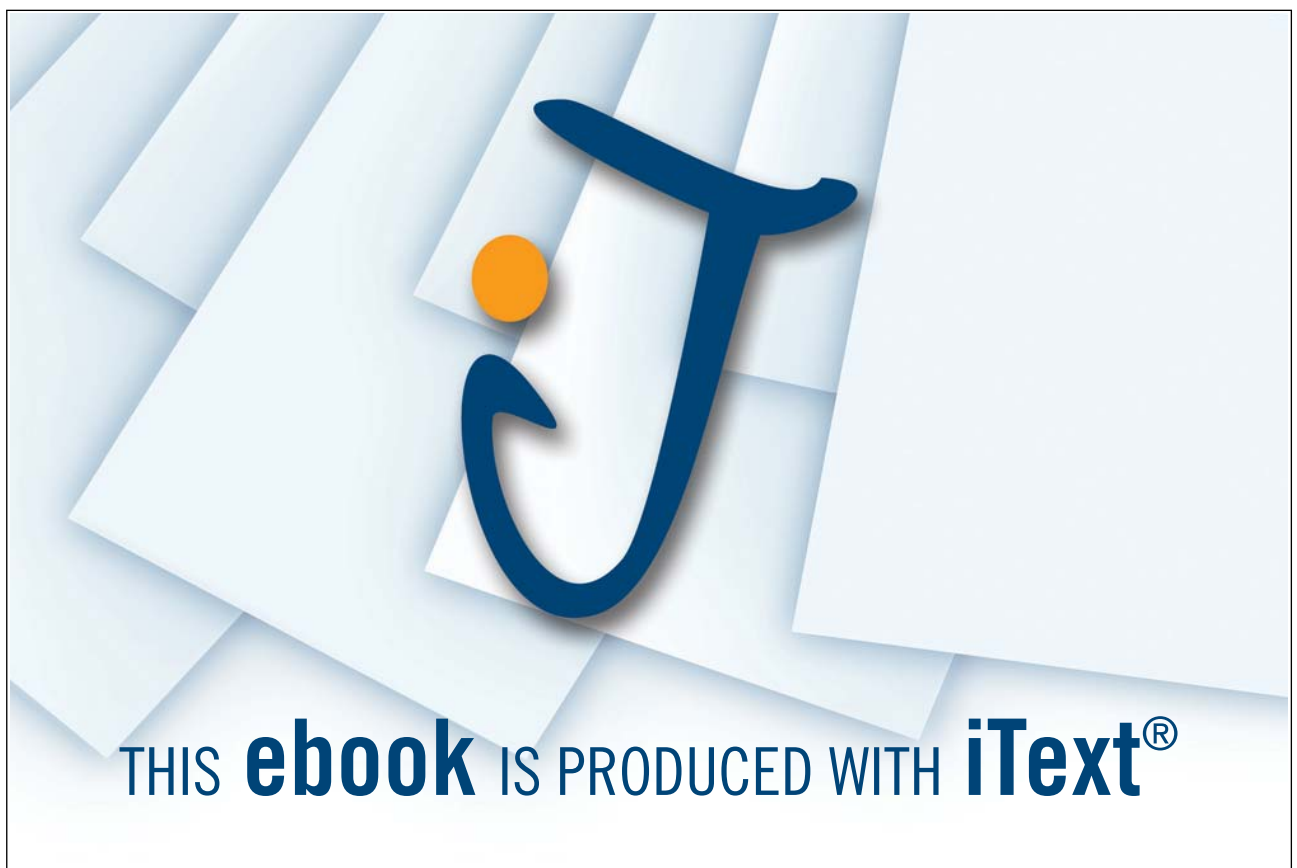
Performance Comparison

An experiment confirms that the enhanced version needs a lesser number of inferences than the version from Sect. 3.3.1.

```
?- new_items(_Items),
   _Colours = [black,blue,violet,green,red,white],
   def_encolour_dl(_Colours), def_endijkstra_dl(_Colours),
   time(endijkstra_dl(_Items,Grouped-[])).
% 16 inferences in 0.00 seconds (Infinite Lips)
Grouped = [col(soot, black), col(coal, black), ...]

?- new_items(_Items),
   _Colours = [black,blue,violet,green,red,white],
   replace_dijkstra_dl(_Colours),
   time(dijkstra_dl(_Items,Grouped-[])).
% 91 inferences in 0.00 seconds (Infinite Lips)
Grouped = [col(soot, black), col(coal, black), ...]
```

The earlier version will appear more efficient, however, if we repeat this experiment and take also into account the overhead for creating and writing to the database the versions' definitions. This apparent advantage disappears,



however, as soon as the list of items exceeds a certain length.

Creating Plain Implementations

Exercise 3.19. *def_encolour_dl/1* and *def_endijkstra_dl/1* gave rise to enhanced implementations which themselves were using difference lists. Write analogues of these two predicates creating *plain* solutions of the Dutch Flag Problem. More precisely, the implementations thus created should themselves be (the augmented) analogues of the solution proposed in Exercise 2.10, p. 60. The interactive session in Fig. 3.17 overleaf illustrates the desired behaviour of *def_encolour_pl/1* and *def_endijkstra_pl/1*.

■

```

?- listing(encolour_pl).
ERROR: No predicates for 'encolour_pl'
No
?- def_encolour_pl([black,white,red,green]).
Yes
?- listing(encolour_pl).
encolour_pl([], [], [], []).
encolour_pl([col(A, black)|B], [col(A, black)|C], D, E, F) :-
    encolour_pl(B, C, D, E, F).
encolour_pl([col(A, white)|B], C, [col(A, white)|D], E, F) :-
    encolour_pl(B, C, D, E, F).
encolour_pl([col(A, red)|B], C, D, [col(A, red)|E], F) :-
    encolour_pl(B, C, D, E, F).
encolour_pl([col(A, green)|B], C, D, E, [col(A, green)|F]) :-
    encolour_pl(B, C, D, E, F).
encolour_pl([col(A, B)|C], D, E, F, G) :-
    encolour_pl(C, D, E, F, G).
Yes
?- listing(endijkstra_pl).
ERROR: No predicates for 'endijkstra_pl'
No
?- def_endijkstra_pl([black,white,red,green]).
Yes
?- listing(endijkstra_pl).
endijkstra_pl(A, B) :- encolour_pl(A, C, D, E, F),
                    flatten([C, D, E, F], B).16
Yes
?- items(_Items)17, endijkstra_pl(_Items, Grouped).
Grouped = [col(milk, white), col(snow, white), col(tomato, red),
           col(blood, red), col(cherry, red)]
Yes

```

Figure 3.17: Example Session for Exercise 3.19

¹⁶In contrast to the special case in Exercise 2.10, now the number of lists to be concatenated will be known at runtime only. Thus the concatenation is best accomplished by using *flatten/2* and not by (repeated use of) *append/3*.

¹⁷The predicate *items/1* is as defined in Sect. 2.4, p. 57.