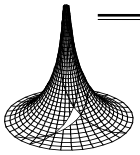

8/1 (2010), 73–87
tmcs@math.klte.hu
http://tmcs.math.klte.hu

Teaching
Mathematics and
Computer Science

Difference lists in Prolog

Attila Csenki

Abstract. Prolog is taught at Bradford University within the two-semester module *Symbolic and Declarative Computing/Artificial Intelligence*. Second year undergraduate students are taught here the basics of the functional and the logic programming paradigms, the latter by using the Linux implementation of SWI Prolog [6]. The topic 'Difference lists' is mentioned in traditional textbooks such as [2] and [5] but it was felt that the available texts do not quite serve our purposes. We present here a lecture handout and a laboratory sheet for the teaching sessions on Difference lists. It is believed that the lectures and lab sessions together with the handouts shown here are a gentle, self-contained and reasoned introduction into the topic. The figures here shown to illustrate the concepts are considered a special feature of the handouts which in this form do not seem to be well known.

Key words and phrases: teaching of Prolog, difference lists, computer science education.

ZDM Subject Classification: P45, P55, Q65, Q85.

Prolog is taught in the School of Computing, Informatics and Media at Bradford University within the two-semester module *Symbolic and Declarative Computing/Artificial Intelligence* (SDC/AI). There are in each semester two lectures and a two hour lab session per week for twelve weeks. The event is for second year undergraduate students of Computer Science and related subjects. Within the module, students are taught the basics of the functional and the logic programming paradigms, the latter by using the Linux implementation of SWI Prolog [6].

'Difference lists' offer a way of concatenating lists in *constant* time, whereas the time required by the usual method with `append` is proportional to the lists' length. The technique relies on the availability of built-in unification in Prolog.

The system appears *at the user level* to be dealing with differences of lists; this is, of course, a mere working hypothesis which, however, turns out to be most useful.

The handouts explain the concepts and are illustrated with figures of the predicates manipulating differences of lists. By the time this topic is addressed, students will have been taught the accumulator technique.[1]

A recent, in-depth treatment of difference lists is in [4], a web resource which is available to students free of charge. The more advanced, research-level application of the technique in [3] reinforces the importance of knowing about it.

Lecture handouts

I describe here various implementations of list concatenation and list reversal based also on difference lists. (This is essentially a record of last week's lectures: week 14.) Starting on page 82, I then describe this week's (week 15) lab tutorials: implementations of flattening lists and some other issues on 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*.

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.

[1] Exercise 4 is intended to interrelate these two topics.

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*:[2]

```
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]
```

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.[3]) 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]
```

[2]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.

[3]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.

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. 1 shows how the three conceptual lists are interrelated.

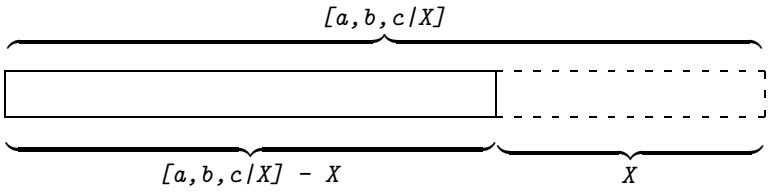

Figure 1. Difference list

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).
```

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. The concatenated list is returned in the last argument of *app_dl4/3* in the form of `[a, b, c, d, e|_G370]-_G370`. (`_G370` being some internally chosen variable name.) It is easily seen that this is accomplished in *one* unification step *irrespective of*

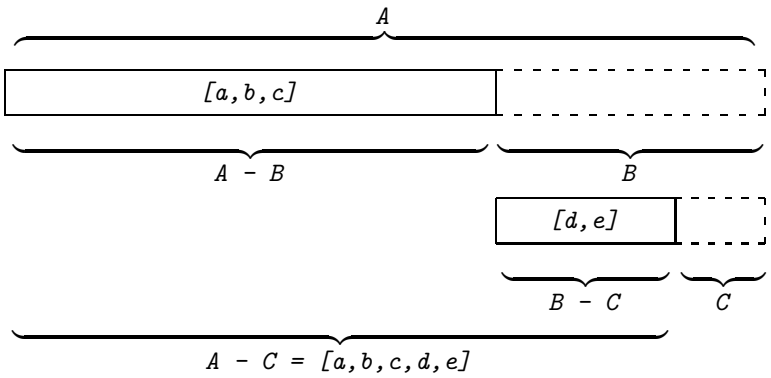

Figure 2. List concatenation by difference lists

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
```

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

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-1) uses *append/2*.

Prolog Code P-1: *First implementation of* reverse/2

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

A declarative reading of clause 2 in (P-1) is suggested in Fig. 3.

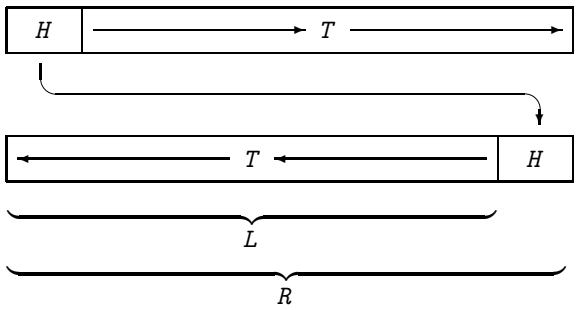

Figure 3. Declarative reading of (P-1)

Another implementation of list reversal, now by the accumulator technique, is by (P-2).

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

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

(P-1), when rewritten in terms of difference lists, results in (P-3).

Prolog Code P-3: *Definition of* `reverse/2` *by difference lists*

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

5 reverse_3(L,R) :- rev_dl(L,R-[]), !.
```

Notice that clause (a2) in (P-3) does not directly correspond to any of the clauses in (P-1); it simply defines the difference list representation of (the reverse of) a list with a single entry.

Concise definitions

We consider several 'improved' versions of the definition of *rev_dl/2* in (P-3).

❶ An alternative to the definition of *rev_dl/2* in (P-3) is the more concise definition (P-4).

Prolog Code P-4: *Concise definition of* `rev_dl/2`

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

An interpretation of clause 2 in (P-4) is shown in Fig. 4.

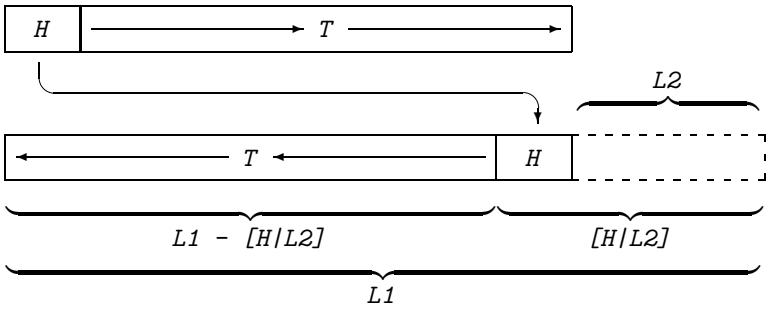

Figure 4. Illustrating clause 2 in (P-4)

It admits the following declarative interpretation:

> 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!)

❷ Fig. 5 shows an improvement version of the earlier definition of *rev_dl/2* by re-arranging the list 'in entries in twos'.

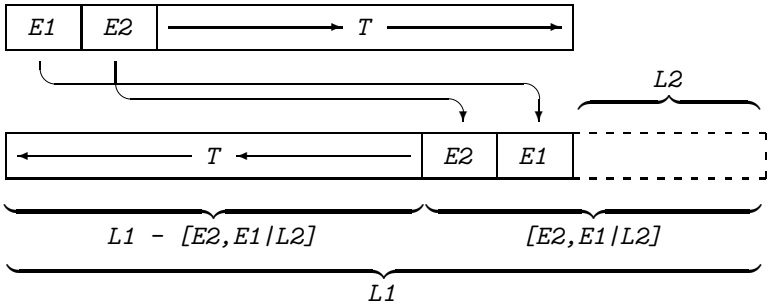

Figure 5. Illustrating an even better version

It admits the following declarative interpretation:

> The difference list *L1-L2* is the reverse of the list *[E1,E2|T]* if the difference list *L1-[E2,E1|L2]* is the reverse of *T*.

The new version based on Fig. 5 is defined in (P-5).

Prolog Code P-5: *Definition of* `reverse_5/2`

```
1 reverse_5(L,R) :- rev_dl_3(L,R-[]).

2 rev_dl_3([],L-L).                                          % clause 0
3 rev_dl_3([X],[X|L]-L).                                     % clause 1
4 rev_dl_3([E1,E2|T],L1-L2) :- rev_dl_3(T,L1-[E2,E1|L2]). % clause 2
```

Noteworthy is in (P-5) that reversal is carried out in 'chunks of twos' resulting in fewer invocations of the auxiliary predicate. There are now two boundary clauses: if the list to be reversed has an even number of entries then clause 0 is used; otherwise, clause 1 applies.

❸ *Further Enhancement.* We modify the last implementation by processing the input list in chunks of threes; this is shown in (P-6).

Prolog Code P-6: *Definition of* `reverse_6/2`

```
reverse_6(L,R) :- rev_dl_4(L,R-[]).

rev_dl_4([],L-L).
rev_dl_4([E1],[E1|L]-L).
rev_dl_4([E1,E2],[E2,E1|L]-L).
rev_dl_4([E1,E2,E3|T],L1-L2) :- rev_dl_4(T,L1-[E3,E2,E1|L2]).
```

It is seen that three base cases are needed now, defining explicitly the reversal of lists with *up to two* entries. The speed can be studied by a query like the one below.

```
?- findall(_N,between(1,100000,_N),_L), time(reverse_6(_L,_R)).
% 33,335 inferences in 0.50 seconds (66670 Lips)
```

❹ *Generalization.* We may provide n base cases catering for the reversal of lists with *up to* $n-1$ entries explicitly and write a recursive clause for reversing lists with *at least* n entries.

Ordinary to difference lists and vice versa

Let the predicate `dl/2` be defined by (P-7).

Prolog Code P-7: *Definition of* `dl/2`

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

`dl/2` 'converts' ordinary lists to difference lists. Here is an interactive session.

```
?- dl([1,2,3,4], DL).
DL = [1, 2, 3, 4|_G255]-_G255
Yes
?- dl([1,2,3,4], L1-L2), dl([a,b,c], L2-L3).
L1 = [1, 2, 3, 4, a, b, c|_G208]
L2 = [a, b, c|_G208]
L3 = _G208
Yes
?- dl([1,2,3,4], L1-L2), dl([a,b,c], L2-L3), L3 = [].
L1 = [1, 2, 3, 4, a, b, c]
L2 = [a, b, c]
L3 = []
Yes
?- dl([1,2,3,4], L1-_L2), dl([a,b,c], _L2-_L3), _L3 = [].
L1 = [1, 2, 3, 4, a, b, c]
Yes
```

Interpretation.

❶ In the first query, I convert `[1,2,3,4]` to a difference list.

❷ In the second query, I do this also for `[a,b,c]` and at the same time I cleverly choose the names of the constituent parts of the difference lists involved such that the concatenation is achieved 'on the sly': Concatenating `L1-L2` and `L2-L3` gives `L1-L3`.[4]

❸ In the third query, I instantiate `L3` to be the empty list whereupon the concatenated list `L1-L3` becomes `L1`, an ordinary list. This then is the result as an ordinary list.[5]

❹ Finally, the fourth query is almost identical to the third one except that intermediate results are not displayed as a result of starting their names with an underscore. (This is because the first line of my Prolog file is the directive `:- set_prolog_flag(toplevel_print_anon, false).`)

Laboratory handouts

Lab Session Week 15 (3/02/2009)

In the first four exercises, I consider various implementations of list flattening. Then, in the exercises five and six, I consider some other questions on difference lists.

Exercise 1

Study the manual's description (using `help/1`) of the built-in predicate `flatten/2`. Devise various examples demonstrating the use of `flatten/2`.

[4]Of course, this is a 'true' statement once we accept our model that we really deal here with differences of lists. Never mind that that is not the case! Difference lists are a mere concept, a figment of the programmer's imagination, a fiction, which, however, affords a workable model. I suspect that this is the case also in other realms of knowledge. Take for example Physics. Do elementary particles *really* exist (for a few nanoseconds (?)) or are they merely a convenient (albeit working) *modelling tool* allowing the observer to predict phenomena correctly? We don't care as long as none of the observations contradicts our model. (Questions concerning 'existence' belong to Philosophy and not Physics.)

[5]Notice that throughout our argument we pretend to be dealing with differences of lists.

Exercise 2

Introductory considerations

Here is Clocksin's implementation of *flatten/2* in [1].

Prolog Code P-8: *Clocksin's definition of* *flatten/2* *in* [1]

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

Test and reflect on this definition. It 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-8) 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([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/2*:

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

The Exercise Proper

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

```
?- flatten([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.

Hint. Consider using the built in predicate *var/1*.

Exercise 3

Define a difference lists based version of *flatten/2* in (P-8).

We have developed several versions of *flatten/2* and now their relative performance should 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 4

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*. (You may wish to draw *hand computations* for *nested/2* in preparation for its definition.)

Exercise 5

Give a pictorial illustration of clause 2 of *dl/2* in (P-7). Based on this illustration, give it a declarative reading.

Exercise 6

Assess the computational speed of the various implementations of list reversal from the lectures interactively. Typically, you will start with the following.

```
?- findall(_N,between(1,2000,_N),_L), time(reverse_1(_L,_R)).
% 2,003,001 inferences in 19.34 seconds (103568 Lips)
?- ...
```

Solutions

Solution of Exercise 2

The improved version is defined in (P-9).

Prolog Code P-9: *Definition of* `flatten_2/2`

```
1 flatten_2(X,[X]) :- var(X), !.              % clause 0
2 flatten_2([],[]).                           % clause 1
3 flatten_2([H|T],L1) :- flatten_2(H,L2),     % clause 2
4                        flatten_2(T,L3),     %
5                        append(L2,L3,L1), !. % cut added here
6 flatten_2(X,[X]).                           % clause 3
```

Clauses 1 to 3 are essentially as in `flatten/2`. (The *cut* in clause 2 has been added to achieve a unique solution.) To rectify the other problem with `flatten/2`, we have to understand why it produces spurious solutions on backtracking. When `flatten/2` arrives at a list entry which is a variable, it will first unify the variable with the empty list and then on further backtracking with `[H|T]` where `H` and `T` are themselves *variables*. Because of the recursive definition, this will then give rise to further such erroneous unifications. To avoid this, we simply 'catch' a variable first argument by clause 0. `flatten_2/2` thus defined behaves as expected:

```
?- flatten_2([a,[Y,[b,X]],c,f(X)],L).
Y = _G339
X = _G345
L = [a, _G339, b, _G345, c, f(_G345)] ;
No
```

Solution of Exercise 3

(P-10) shows a clause-by-clause 'translation' of the definition of `flatten/2` in terms of difference lists.

Prolog Code P-10: *Difference list based definition of* `flatten/2`

```
1 flatten_3(L,F) :- flatten_dl(L,F-[]), !.        % clause 1
2                                                 %
3 flatten_dl([],L-L).                             % clause 2
4 flatten_dl([H|T],L1-L3) :- flatten_dl(H,L1-L2), % clause 3
5                            flatten_dl(T,L2-L3). %
6 flatten_dl(X,[X|Z]-Z).                          % clause 4
```

The *append* goal does not appear in (P-10) as list concatenation is now accomplished by difference lists. *flatten_3/2* will behave identically to *flatten/2* except that its solution is unique because of the *cut* (*!*) in clause 1.

Solution of Exercise 4

We define in (P-11) *nested/2* in terms *nested/4* whose second and third argument are a counter and an accumulator, respectively.

Prolog Code P-11: *Definition of* *nested/2*

```
1 nested(M,L) :- nested(M,1,[1],L), !.

2 nested(M,M,L,L).
3 nested(M,N,Acc,L) :- NewN is N + 1,
4                      nested(M,NewN,[Acc,NewN],L).
```

The versions' relative performance is illustrated below. It is seen in particular that the one based on difference lists is nearly as good as the built-in version.

```
?- nested(8000,_L), time(flatten(_L,_F)).
% 95,999 inferences in 0.44 seconds (218180 Lips)
?- nested(8000,_L), time(flatten_1(_L,_F)).
% 216,004 inferences in 12.96 seconds (16667 Lips)
?- nested(8000,_L), time(flatten_2(_L,_F)).
% 144,007 inferences in 12.79 seconds (11259 Lips)
?- nested(8000,_L), time(flatten(_L,_F)).
% 335,514 inferences in 9.88 seconds (33959 Lips)
ERROR: Out of global stack
?- nested(8000,_L), time(flatten_3(_L,_F)).
% 32,000 inferences in 0.93 seconds (34409 Lips)
```

Furthermore, it is seen that version 3, the implementation using list concatenation with *append/3*, is not practically viable due to stack overflow. (This problem has been experienced even for a nesting depth of 1000.)

Solution of Exercise 5

Fig. 6 admits the following declarative interpretation:

> The difference list version of *[H|T]* is *[H|L1]-L2* if the difference list version of *T* is *L1-L2*.

Solution of Exercise 6

```
?- findall(_N,between(1,2000,_N),_L), time(reverse_1(_L,_R)).
% 2,003,001 inferences in 19.34 seconds (103568 Lips)
```

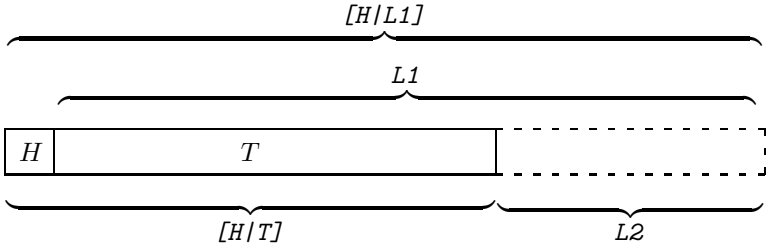

Figure 6. Illustrating the second clause of `dl/2`

```
?- findall(_N,between(1,2000,_N),_L), time(reverse_2(_L,_R)).
% 2,002 inferences in 0.00 seconds (Infinite Lips)
?- findall(_N,between(1,2000,_N),_L), time(reverse_3(_L,_R)).
% 4,000 inferences in 0.06 seconds (66667 Lips)
?- findall(_N,between(1,2000,_N),_L), time(reverse_4(_L,_R)).
% 2,002 inferences in 0.05 seconds (40040 Lips)
```

It is seen that the 'naïve' implementation is far less efficient than either of the other three. Furthermore, version 4 is seen to behave in the same way as the one using accumulators (which is the method used also to implement the built-in version).

References

[1] W. F. Clocksin, *Clause and Effect*, Prolog Programming for the Working Programmer, Springer, London, 1997.

[2] W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, Springer, London, 1994, fourth edition.

[3] A. Csenki, Rotations in the plane and Prolog, *Science of Computer Programming* **66** (2007), 154–161.

[4] A. Csenki, *Prolog Techniques*, Ventus Publishing ApS, Copenhagen, 2009, `http://bookboon.com/uk/student/it/`.

[5] L. Sterling and E. Shapiro, *The Art of Prolog*, Advanced Programming Techniques, MIT Press, Cambridge Ma, London, 1986.

[6] J. Wielemaker, *SWI-Prolog 5.1 Reference Manual*, Amsterdam, 2003.

ATTILA CSENKI
UNIVERSITY OF BRADFORD
BD7 1DP, UNITED KINGDOM

E-mail: `a.csenki@bradford.ac.uk`

(Received May, 2009)