Chapter 1

Accumulator Technique

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

1.1 A Simple Example

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

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

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

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

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

[1]

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

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

[2]

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

[1] For a definition of the predicate *from_to(+Low,+High,-List)*, see Exercise 1.1, p. 17 and the solution of Exercise 3.16, p. 167. *from_to/3* returns in *List* the list of integers between the bounds *Low* and *High*.

[2] The query below shows that stack overflow is caused here by *sum/2* and *not* by *from_to/3*.

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

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

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

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

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

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

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

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

We may define `new_sum/2` by

Prolog Code P-1.3: *Definition of* `new_sum/2`

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

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

1.2 Hand Computations

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

1.3 Further Examples

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

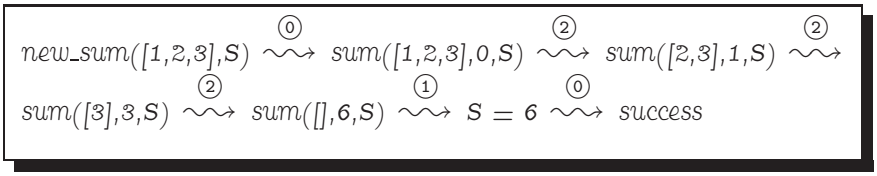

Figure 1.1: Hand Computations for *new_sum/2*

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

[3]In Sect. 1.4, *pseudocodes* will be introduced for describing algorithms.

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

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

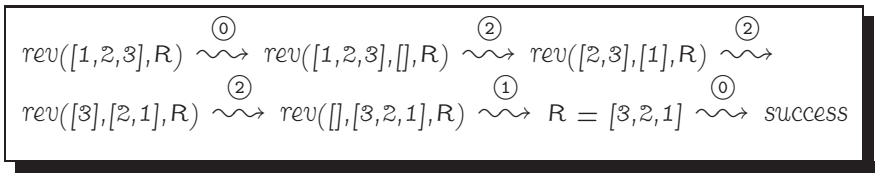

Figure 1.2: Hand Computations for *rev/2*

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

```
1 rev(L,R) :- rev(L,[],R).              % clause 0

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

■

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

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

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

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

From the hand computations in Fig. 1.3 it is seen that there should be two recursive clauses: in the case marked ②, the head of the list is smaller than the current value of the accumulator and thus it will be replaced by the former; in the case marked ③, this condition does not apply and therefore the old accumulator value is retained. Fig. 1.3 also shows that the initial value of the accumulator in *min/3* is the head of the input list (step ⓪).[4]

[4]There is an alternative to this. Use the built-in predicate *current_prolog_flag/2* to find the largest integer Prolog can represent and initialize the accumulator to this value:

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

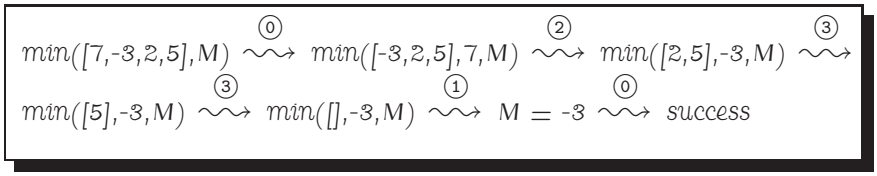

Figure 1.3: Hand Computations for *min/2*

Prolog Code P-1.5: *Definition of* *min/2*

```
1 min([H|T],M) :- min(T,H,M).                  % clause 0

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

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

■

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

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

Some suggested hand computations are shown in Fig. 1.4.

■

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

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

[5] Use the built-in predicates *var/1* and *integer/1* when implementing this functionality.

[6] Here we pick out the third entry of the result list by *pattern matching*. The built-in predicate *is_list/1* should be used to allow for such behaviour.

[7] To implement this functionality, you will need the built-in predicate *last/2*.

[8] The outside quotes mark the atom. Inside the atom, the characters '' stand for the quote.

$$from\_to(6,9,L) \rightsquigarrow from\_to\_acc(6,[9],L) \rightsquigarrow$$

$$from\_to\_acc(6,[8,9],L) \rightsquigarrow from\_to\_acc(6,[7,8,9],L) \rightsquigarrow$$

$$from\_to\_acc(6,[6,7,8,9],L) \rightsquigarrow L = [6,7,8,9] \rightsquigarrow success$$

Figure 1.4: Suggested Hand Computations for *from_to/3*

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

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

Built-in Predicate: *atom_codes(?Atom,?List)*

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

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

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

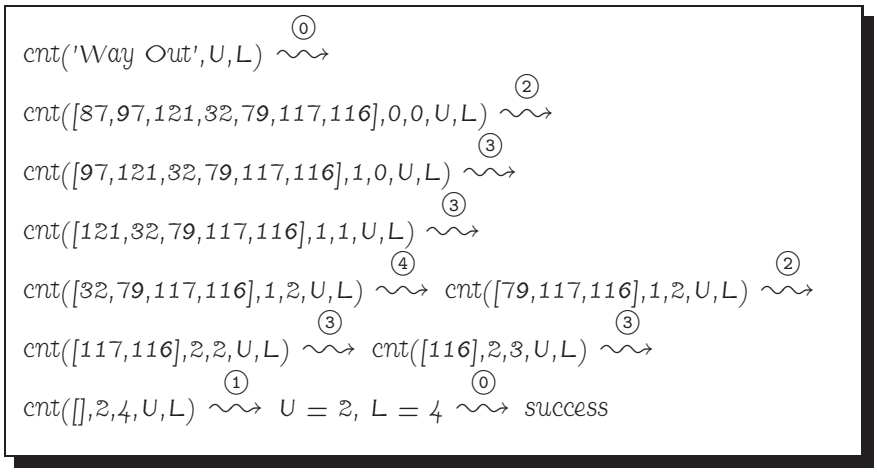

Figure 1.5: Hand Computations for *cnt/3*

mirrors the hand computations.

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

```
1  cnt(Atom,U,L) :- atom_codes(Atom,Values),              % clause 0
2                   cnt(Values,0,0,U,L), !.               %

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

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

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

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

■

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

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

```
1  count(Atom,cases(U,L)) :- atom_codes(Atom,Values),            % clause 0
2                            count(Values,acc(0,0),acc(U,L)), !. %

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

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

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

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

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

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

■

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

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

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

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

```
1 palin(L) :- palin(L,[]).              % clause 0

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

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

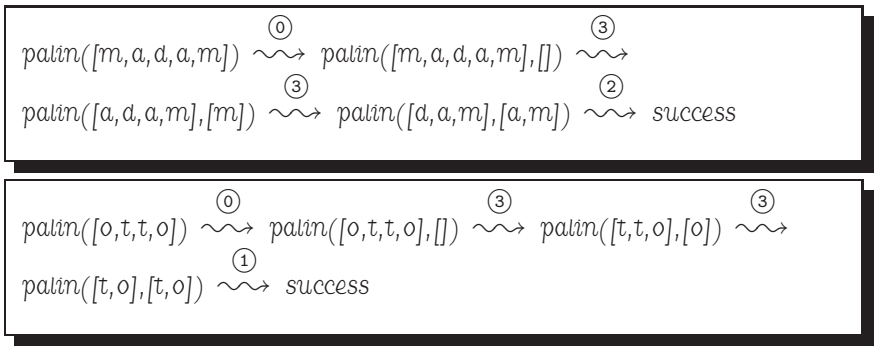

Figure 1.6: Hand Computations for ***palin/1*** — *success*

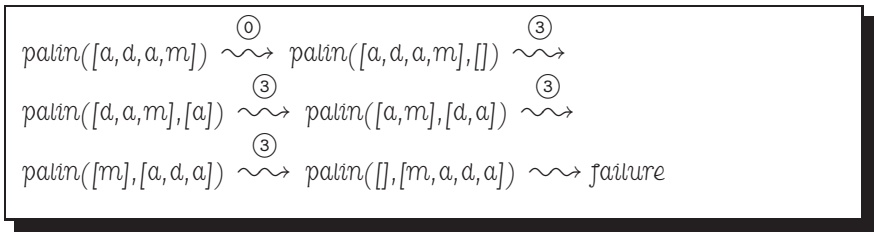

Figure 1.7: Hand Computations for ***palin/1*** — *failure*

■

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

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

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

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

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

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

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

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

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

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

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

```
1 numbers(Atom,N) :- atom_codes(Atom,Values),                % clause 0
2                    numbers(Values,nodigit,0,N), !.         %

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

■

Exercise 1.2. Solve the problem from Example 1.6 as before (i.e. by a single pass through the data), now without using a switch.[10]

■

1.4 Pseudocodes

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

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

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

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

[9] Reference will be made to (P-1.11) in Exercise 1.3, p. 26.

[10] *Hint.* Employ a 'look ahead' strategy to see what (encoded) character will be read *after* the present one. (This plan allows a concise implementation to be achieved.)

Algorithm 1.4.1: REVERSE(*List*)

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

Statement	(1)	(2)	(3)	(4)	(5)
Hand Computation Step	⓪	②	②	②	①

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

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

These examples illustrate the following points.

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

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

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

Algorithm 1.4.2: COUNT(*Atom*)

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

Statement	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)
Hand Comp^{n} Step	⓪	⓪	⓪	② ③ ④	②	③	② ③ ④	①	①

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

Algorithm 1.4.3: PALINDROME(*List*)

Accumulator ← [] (1)
while *List* ≠ []
do
  if *List* = *Accumulator** (2)
  then {**return** (*success*)
  else
    [*H*|*T*] ← *List* (3)
    if *T* = *Accumulator** (4)
    then {**return** (*success*)
  List ← *T* (5)
  Accumulator ← [*H*|*Accumulator*] (6)
return (*failure*)

Statement	(1)	(2)	(3)	(4)	(5)	(6)
Hand Computation Step	⓪	①	② ③	②	③	③

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

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

■

1.5 Generalization

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

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

[12] For *palin/2* from Example 1.5 also to fit this mould, the *Result* argument is to be ignored.
[13] If necessary, apply first the technique from Example 1.4 to reduce the arity of *predicate* to 3.

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

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

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

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

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

Figure 1.9: Generalized Clause Structures

1.6 Case Study: The Perceptron Training Algorithm

1.6.1 Classification Problem

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

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

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

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

1.6.2 Algorithm

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

$$d(x_1, x_2) = \begin{cases} +1 & \text{if } w_1 x_1 + w_2 x_2 \geq t, \\ -1 & \text{if } w_1 x_1 + w_2 x_2 < t, \end{cases} \tag{1.1}$$

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

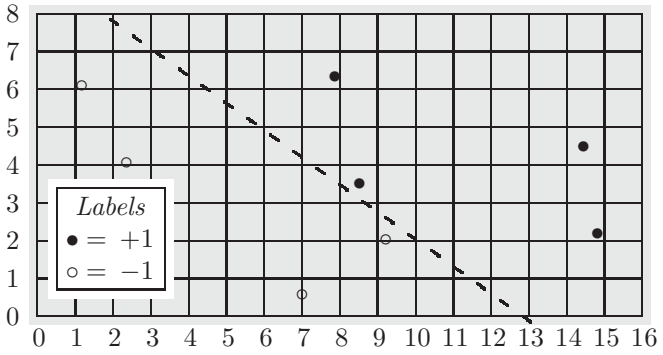

Figure 1.10: A Linearly Separable Data Set

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

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

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

Figure 1.11: Classifying a Point

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

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

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

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

1.6.3 Implementation

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

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

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

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

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

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

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

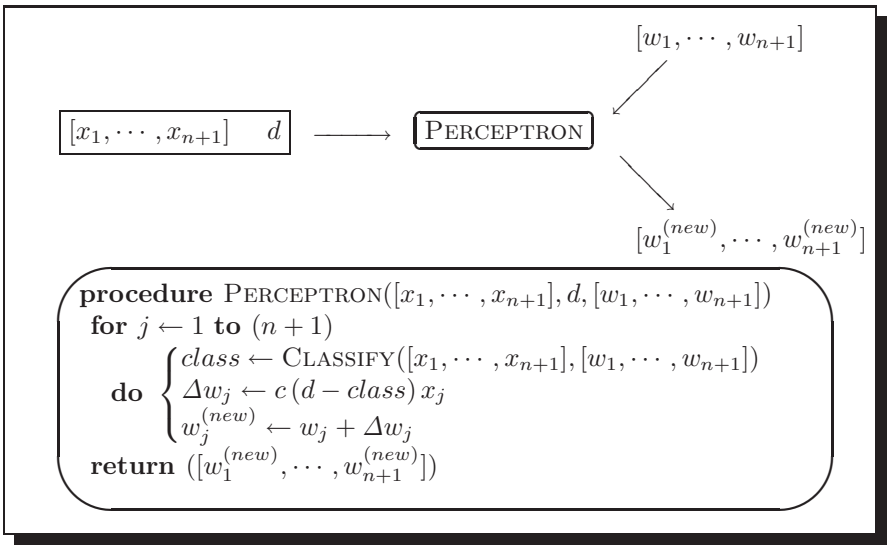

Figure 1.12: A Single Updating Step

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

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

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

With reference to Fig. 1.9 it is seen that

[14] Equation (1.2) thereby subsumes (1.1) by putting $n = 2$ and $t = -w_{n+1}$.
[15] The symbol ++ stands for list concatenation.

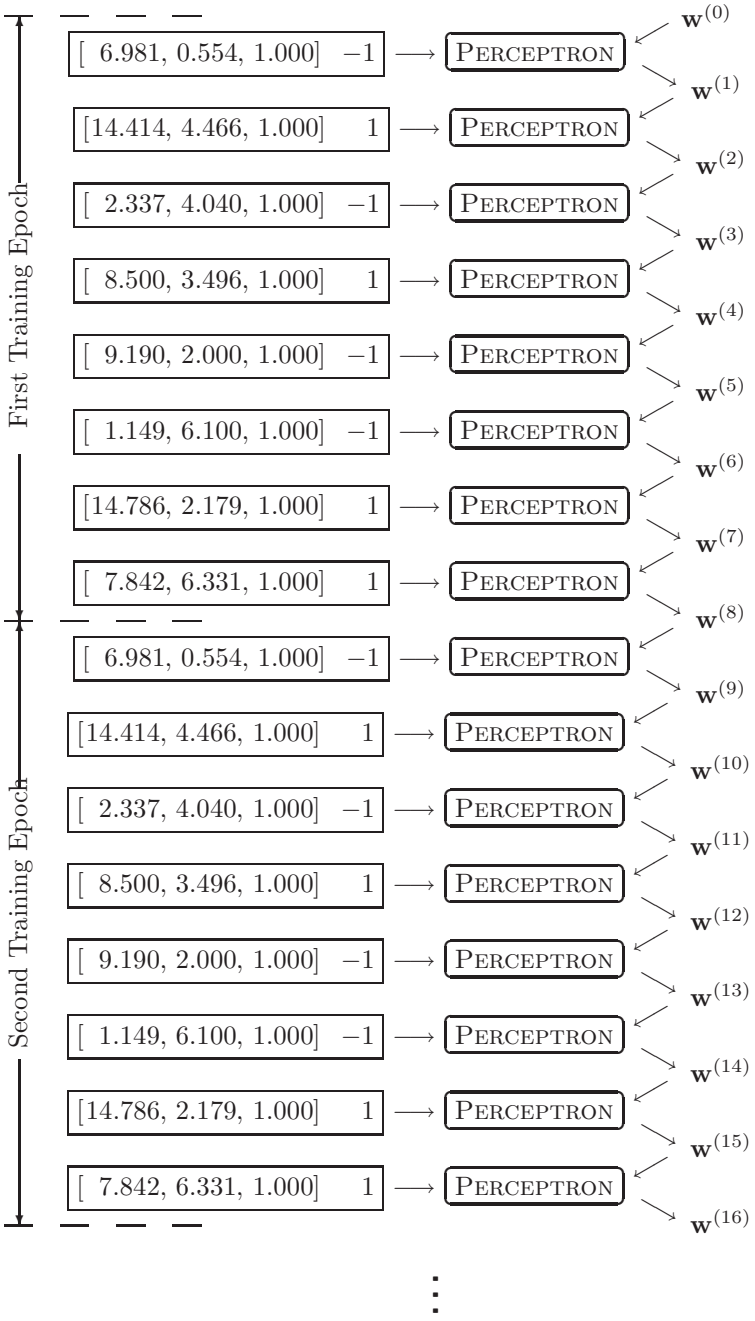

Figure 1.13: Applying the Perceptron Training Algorithm

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

comment: Perceptron Training Algorithm.
Iterate until all points are correctly classified.

procedure ClassifyAll$([\mathbf{x}^{(1)}, \cdots, \mathbf{x}^{(N)}], \mathbf{w})$
 for $i \leftarrow 1$ **to** N
  do $\{c_i \leftarrow$ Classify$(\mathbf{x}^{(i)}, \mathbf{w})$
 return $([c_1, \cdots, c_N])$

main
 $Weights \leftarrow \mathbf{w}$
 $Points \leftarrow [\mathbf{x}^{(1)}, \cdots, \mathbf{x}^{(N)}]$
 $DesiredOutputs \leftarrow [d_1, \cdots, d_N]$
 $ActualOutputs \leftarrow$ ClassifyAll$(Points, Weights)$
 $Iterations \leftarrow 0$
 while $ActualOutputs \neq DesiredOutputs$
  do $\begin{cases} [P|OtherPs] \leftarrow Points \\ [D|OtherDs] \leftarrow DesiredOutputs \\ Weights \leftarrow \text{Perceptron}(c, P, D, Weights) \\ Points \leftarrow OtherPs +\!\!+^{15} [P] \\ DesiredOutputs \leftarrow OtherDs +\!\!+ [D] \\ ActualOutputs \leftarrow \text{ClassifyAll}(Points, Weights) \\ Iterations \leftarrow Iterations + 1 \end{cases}$
 output $(Iterations, Weights)$

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

Prolog Code P-1.14: *Definition of* classify_all/3

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

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

- In clause 1, *extract-info-from* is realized by unification of the last two arguments of *in/5* with those of *out/2*.

- In clause 2, the *carry-on-condition* is implicitly defined by failure of the predicate *classify_all/3* in clause 1.

- Finally, the predicate *transform/2* is defined by (P-1.15).

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

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

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

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

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

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

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

[16]

While the initial weights give rise to some incorrect classifications,

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

the new weights define a correct classifier,

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

(The corresponding separating straight line

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

is shown in Fig. 1.10.)

[16] In the version of SWI–Prolog used here (version 3.4.5), variables whose name starts with an underscore (such as `_Ps`) won't be displayed. Issue the query

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

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

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

■

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

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

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

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

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

■

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

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

main
$Weights \leftarrow \mathbf{w}$
$Points \leftarrow [\mathbf{x}^{(1)}, \cdots, \mathbf{x}^{(N)}]$
$DesiredOutputs \leftarrow [d_1, \cdots, d_N]$
$Iterations \leftarrow m$
repeat
  $[P|OtherPs] \leftarrow Points$
  $[D|OtherDs] \leftarrow DesiredOutputs$
  $Weights \leftarrow$ PERCEPTRON$(c, P, D, Weights)$
  $Points \leftarrow OtherPs +\!\!+ [P]$
  $DesiredOutputs \leftarrow OtherDs +\!\!+ [D]$
  $Iterations \leftarrow Iterations - 1$
until $Iterations = 0$
output $(Weights)$

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