Chapter 4

Exploratory Code Development

Conciseness and accessibility of source code through declarative reading are Prolog's major strengths. It is therefore relatively easy to appreciate the workings of someone else's implementation, while it is much harder independently *to arrive at* one's own solution to the same problem. In this chapter, we illustrate a practical methodology which is intended to overcome this discrepancy: it is a software development style that is interactive, incremental, exploratory and allows Prolog code to be arrived at in a relatively effortless manner.

4.1 A Nursery Rhyme

The task is to write a Prolog predicate `rhyme/0` which displays on the screen the well-known nursery rhyme *This is the House that Jack Built* ([11]):

```
This is the house that Jack built.

This is the malt
That lay in the house that Jack built.

This is the rat
That ate the malt
That lay in the house that Jack built.

This is the cat
That killed the rat
That ate the malt
That lay in the house that Jack built.

This is the dog
That worried the cat
That killed the rat
That ate the malt
That lay in the house that Jack built.

This is the cow with the crumpled horn
That tossed the dog
That worried the cat
That killed the rat
That ate the malt
That lay in the house that Jack built.

This is the maiden all forlorn
That milked the cow with the crumpled horn
That tossed the dog
That worried the cat
That killed the rat
That ate the malt
That lay in the house that Jack built.

This is the man all tattered and torn
That kissed the maiden all forlorn
That milked the cow with the crumpled horn
That tossed the dog
That worried the cat
That killed the rat
That ate the malt
That lay in the house that Jack built.

This is the priest all shaven and shorn
That married the man all tattered and torn
That kissed the maiden all forlorn
That milked the cow with the crumpled horn
That tossed the dog
That worried the cat
That killed the rat
That ate the malt
That lay in the house that Jack built.

This is the cock that crowed in the morn
That waked the priest all shaven and shorn
That married the man all tattered and torn
That kissed the maiden all forlorn
```

That milked the cow with the crumpled horn
That tossed the dog
That worried the cat
That killed the rat
That ate the malt
That lay in the house that Jack built.

This is the farmer sowing his corn
That kept the cock that crowed in the morn
That waked the priest all shaven and shorn
That married the man all tattered and torn
That kissed the maiden all forlorn
That milked the cow with the crumpled horn
That tossed the dog
That worried the cat
That killed the rat
That ate the malt
That lay in the house that Jack built.

In our implementation of `rhyme/0` we want to exploit the rhyme's repetitive structure and the fact that all essential information is contained in its last verse. We record the last verse in the database by `verse/1` as shown in (P-4.1).

Prolog Code P-4.1: *Definition of* `verse/1`

```
verse(['This is the farmer sowing his corn',
       'That kept the cock that crowed in the morn',
       'That waked the priest all shaven and shorn',
       'That married the man all tattered and torn',
       'That kissed the maiden all forlorn',
       'That milked the cow with the crumpled horn',
       'That tossed the dog',
       'That worried the cat',
       'That killed the rat',
       'That ate the malt',
       'That lay in the house that Jack built.']).
```

The rhyme is seen roughly to match the simplified pattern shown in Fig. 4.1.

$$\begin{array}{ccccccc}
\textit{verse 1} & \textit{verse 2} & \textit{verse 3} & \textit{verse 4} & \textit{verse 5} & \textit{verse 6} & \\
\downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \\
 & & & & & & \\
A & B & C & D & E & F & \cdots \\
 & A & B & C & D & E & \cdots \\
 & & A & B & C & D & \cdots \\
 & & & A & B & C & \cdots \\
 & & & & A & B & \cdots \\
 & & & & & A & \cdots
\end{array}$$

Figure 4.1: The Rhyme's Simplified Pattern

Knowing the rhyme's last verse and the above structure will allow (up to some finer detail) the rhyme to be fully reconstructed. With a view to a simplified *preliminary* Prolog implementation, we therefore define the following Prolog fact in the database

```
verse_skeleton(['F','E','D','C','B','A']).
```

The *first task* is now to define a predicate `rhyme_prel/2` which *should* enable us to obtain the skeleton rhyme's structure in the following manner.

```
?- verse_skeleton(_V), rhyme_prel(_V,_R), write_term(_R,[]).
[[A], [B, A], [C, B, A], [D, C, B, A], [E, D, C, B, A],
[F, E, D, C, B, A]]
```

Taking this as an *informal specification* of *rhyme_prel/2*, we want to arrive at its definition by a series of *interactive experiments*.

4.1.1 First Preliminary Implementation

What could be the least ambitious first step in implementing *rhyme_prel/2*? We may for example create a list whose only entry is the last entry of the above list-of-lists. (This will correspond to reproducing the last verse.) This we do by

```
?- verse_skeleton(_V), _R = [_V], write_term(_R,[]).
[[F, E, D, C, B, A]]
```

Still interactively, a list comprising the last two entries of the target list-of-lists may be generated by

```
?- verse_skeleton(_V), _V = [_|_T1], _R = [_T1,_V],
   write_term(_R,[]).
[[E, D, C, B, A], [F, E, D, C, B, A]]
```

Here we unify *_T1* with the tail of *_V* and position it in front of *_V* to form the new list (of lists). How do we now generate the next larger list (comprising the last three entries of the target list-of-lists)? We proceed as before except that we assemble *_R* from the entries *_T2*, *_T1* and *_V* (in that order!) where *_T2* is unified with the tail of *_T1*.

```
?- verse_skeleton(_V), _V = [_|_T1], _T1 = [_|_T2],
   _R = [_T2,_T1,_V], write_term(_R,[]).
[[D, C, B, A], [E, D, C, B, A], [F, E, D, C, B, A]]
```

One more such step should suffice to appreciate the underlying pattern of interactively generating instances of *_R*.

```
?- verse_skeleton(_V), _V = [_|_T1], _T1 = [_|_T2],
   _T2 = [_|_T3], _R = [_T3,_T2,_T1,_V], write_term(_R,[]).
[[C, B, A], [D, C, B, A], [E, D, C, B, A], [F, E, D, C, B, A]]
```

Since our aim is to identify a *recursive* pattern in the above interactive session, we recast the inputs slightly by observing that $[a_1, \cdots, a_{n-1}, a_n]$ and $[a_1|[a_2|[a_3|\cdots|[a_{n-1}|[a_n]]\cdots]]$ are equivalent representations of the same list. Let's have a look at the last two queries again.

```
?- verse_skeleton(_V), _V = [_|_T1], _T1 = [_|_T2],
```

$$\texttt{\_R = }\underbrace{\texttt{[}\underbrace{\texttt{\_T2}}_{Head\_Old}\texttt{ |}\underbrace{\texttt{[\_T1|[\_V]]}}_{Tail\_Old}\texttt{]}}_{Rhyme\_Old}\texttt{, write\_term(\_R,[]).}$$

```
[[D, C, B, A], [E, D, C, B, A], [F, E, D, C, B, A]]
```

```
?- verse_skeleton(_V), _V = [_|_T1], _T1 = [_|_T2],
```

$$\underbrace{\texttt{\_T2}}_{Head\_Old}\texttt{ = [\_|}\underbrace{\texttt{\_T3}}_{Head}\texttt{], \_R = }\underbrace{\texttt{[}\underbrace{\texttt{\_T3}}_{Head}\texttt{|}\underbrace{\texttt{[\_T2|[\_T1|[\_V]]]}}_{Rhyme\_Old}\texttt{]}}_{Rhyme}\texttt{,}$$

```
write_term(_R,[]).

[[C, B, A], [D, C, B, A], [E, D, C, B, A], [F, E, D, C, B, A]]
```

The annotated lists suggest the following *pseudocode* (using Prolog's list-notation) for one single *recursive step.*

$$\begin{aligned} \texttt{Rhyme\_Old} &= \texttt{[Head\_Old|Tail\_Old]} \qquad (4.1)\\ \texttt{Head\_Old} &= \texttt{[\_|Head]}\\ \texttt{Rhyme} &= \texttt{[Head|Rhyme\_Old]} \qquad (4.2) \end{aligned}$$

Notice that by equations (4.1) and (4.2) we may replace the latter by

$$\texttt{Rhyme} = \texttt{[Head|[Head\_Old|Tail\_Old]]}$$

The *base case* for the recursion is given by

$$\texttt{First\_Rhyme} = \texttt{[['F', 'E', 'D', 'C', 'B', 'A']]}$$

A straightforward implementation of the recursive step is by the (auxiliary) predicate *rhyme_aux/3* in (P-4.2).

Prolog Code P-4.2: *First definition of the auxiliary predicate*

```
rhyme_aux(R,1,R).
rhyme_aux([Head_Old|Tail_Old],Counter,R) :-
   Head_Old = [_|Head],
   New_Counter is Counter - 1,
   rhyme_aux([Head|[Head_Old|Tail_Old]],New_Counter,R).
```

In the first argument of *rhyme_aux/3* the most recent version of the rhyme is accumulated; its second argument is a counter which is decremented from an initial value until it reaches unity at which point the third argument is instantiated to the first. It is noteworthy in the definition of *rhyme_aux/3* that, as a consequence of using the accumulator technique, reference to the more complex case in the recursive step is found in the rule's body. (In this sense, as opposed to the familiar situation from imperative programming, progression is from *right to left*.)

We find out by an experiment what the counter should be initialized to.

```
?- verse_skeleton(_V), rhyme_aux([_V],1,_R), write_term(_R,[]).
[[F, E, D, C, B, A]]
?- verse_skeleton(_V), rhyme_aux([_V],2,_R), write_term(_R,[]).
[[E, D, C, B, A], [F, E, D, C, B, A]]
...
?- verse_skeleton(_V), rhyme_aux([_V],6,_R), write_term(_R,[]).
[[A], [B, A], [C, B, A], [D, C, B, A], [E, D, C, B, A],
[F, E, D, C, B, A]]
```

It is seen that the second argument of *rhyme_aux/3* (the counter) will have to be initialized to the length of (what stands for) the last verse. This gives rise to the following *first* version of the predicate *rhyme_prel/2*

```
rhyme_prel_1(V,R) :- length(V,L), rhyme_aux([V],L,R).
```

which then behaves as specified on p. 119.

Even though the solution thus obtained is perfectly acceptable, there is scope for improvement. Counters are commonly used in imperative programming for verifying a stopping criterion. The corresponding task in declarative programming is best achieved by *pattern matching*. There is indeed no need for a counter here since the information for when *not* to apply the recursive step (any more) can be gleaned from the pattern of the first argument of *rhyme_aux/3*: For the recursion to stop, the head of the list-of-lists (in the first argument) should itself be a list with exactly one entry. (The complete rhyme will have been arrived at when the first verse comprises a single line!) This idea gives rise in (P-4.3) to a new, *improved* (and more concise) version of the auxiliary predicate, now called *rhyme_aux/3*.

Prolog Code P-4.3: *Another definition of the auxiliary predicate*

```
rhyme_aux_2([[First]|Rest],[[First]|Rest]).
rhyme_aux_2([Head_Old|Tail_Old],R) :-
   Head_Old = [_|Head],
   rhyme_aux_2([Head|[Head_Old|Tail_Old]],R).
```

rhyme_aux_2/3 behaves as intended:

```
?- verse_skeleton(_V), rhyme_aux_2([_V],_R), write_term(_R,[]).
[[A], [B, A], [C, B, A], [D, C, B, A], [E, D, C, B, A],
[F, E, D, C, B, A]]
```

The definition of a second, improved version of the preliminary rhyme predicate now simplifies to

```
rhyme_prel_2(V,R) :- rhyme_aux_2([V],R).
```

To complete the 'skeleton version' of the rhyme, we display the above by

```
?- verse_skeleton(_V), rhyme_prel_2(_V,_R), show_rhyme(_R).
A

B
A
...

F
E
D
C
B
A
```

with the predicate *show_rhyme/1* defined by

```
show_list([]).
show_list([H|T]) :- write(H), nl, show_list(T).

show_rhyme([]).
show_rhyme([H|T]) :- show_list(H), nl, show_rhyme(T).
```

There is still scope for further improvement leading to an even more concise version of the auxiliary predicate. We may replace in the definition of *rhyme_aux_2/2* all occurrences of `Head_Old` by `[H|T]`, say, accounting for the fact that `Head_Old` will be unified with a list.

```
rhyme_aux_2([[H|T]|Tail_Old],R) :-
   [H|T] = [_|Head],
   rhyme_aux_2([Head|[[H|T]|Tail_Old]],R).
```

But then, by virtue of the first goal in the body of this rule we may replace all occurrences of `Head` by `T`. Subsequently, the first goal may be dropped. Overall, we obtain in (P-4.4) a third, even more concise version of the auxiliary predicate.

Prolog Code P-4.4: *Third definition of the auxiliary predicate*

```
rhyme_aux_3([[First]|Rest],[[First]|Rest]).
rhyme_aux_3([[H|T]|Tail_Old],R) :- rhyme_aux_3([T|[[H|T]|Tail_Old]],R).
```

There is hardly any room for improvement left save perhaps a minor simplification of the first clause. We derive an alternative boundary case by first completing the interactive session from p. 120 and then carrying out one more step:

```
?- verse_skeleton(_V), _V = [_|_T1], _T1 = [_|_T2],
   _T2 = [_|_T3], _T3 = [_|_T4], _T4 = [_|_T5],
   _R = [_T5|[_T4|[_T3|[_T2|[_T1|[_V]]]]]], write_term(_R,[]).
[[A], [B, A], [C, B, A], [D, C, B, A], [E, D, C, B, A],
[F, E, D, C, B, A]]
```

```
?- verse_skeleton(_V), _V = [_|_T1], _T1 = [_|_T2],
   _T2 = [_|_T3], _T3 = [_|_T4], _T4 = [_|_T5], _T5 = [_|_T6],
   _R = [_T6|[_T5|[_T4|[_T3|[_T2|[_T1|[_V]]]]]]],
   write_term(_R,[]).
[[], [A], [B, A], [C, B, A], [D, C, B, A], [E, D, C, B, A],
[F, E, D, C, B, A]]
```

The first query suggests that we are finished if the (partially) completed skeleton rhyme's head is a single-element list; this condition gave rise to the earlier boundary case. On the other hand, in the second query the variable *_R* is unified with a list whose head is empty and whose tail is the *full* skeleton rhyme. This suggests the following *alternative* first clause for *rhyme_aux_3/2*,

```
            rhyme_aux_3([[]|R],R).
```

The disadvantage of this stopping criterion is that it will cause one additional invocation of the recursive step.

Of course, the third version of the auxiliary predicate, *rhyme_aux_3/2*, (with any of the two alternative first clauses) gives rise to yet another version of *rhyme_prel/2*.

```
        rhyme_prel_3(V,R) :- rhyme_aux_3([V],R).
```

4.1.2 Another Preliminary Implementation

With a view to wishing to use the *accumulator technique* (yet again), let us examine the first few steps of an (as yet *imaginary*) interactive session.

```
?-...
[F, E, D, C, B, A], []
?-...
[E, D, C, B, A], [[F, E, D, C, B, A]]
?-...
[D, C, B, A], [[E, D, C, B, A], [F, E, D, C, B, A]]
?-...
[C, B, A], [[D, C, B, A], [E, D, C, B, A], [F, E, D, C, B, A]]
```

Two lists are involved here. The first list serves as a ‘supplier’ for updating the second one in which the skeleton rhyme’s verses are accumulated. We observe that in each step the first list ‘loses’ its head, whereas the second list is augmented by the first one. At the end of this sequence of steps (i.e. when the first list is empty) the second list will contain the full skeleton rhyme. Having established the underlying idea, we now turn to the corresponding interactive session. (This may look tedious but is easily carried out using ‘copy-and-paste’.)

```
?- verse_skeleton(_V), _P1 = (_V,[]), write_term(_P1,[]).
[F, E, D, C, B, A], []
?- verse_skeleton(_V), _P1 = (_V,[]),
   ([_H1|_T1],_Acc1) = _P1, _P2 = (_T1,[[_H1|_T1]|_Acc1]),
   write_term(_P2,[]).
[E, D, C, B, A], [[F, E, D, C, B, A]]
?- verse_skeleton(_V), _P1 = (_V,[]),
   ([_H1|_T1],_Acc1) = _P1, _P2 = (_T1,[[_H1|_T1]|_Acc1]),
   ([_H2|_T2],_Acc2) = _P2, _P3 = (_T2,[[_H2|_T2]|_Acc2]),
   write_term(_P3,[]).
[D, C, B, A], [[E, D, C, B, A], [F, E, D, C, B, A]]
?- ...
```

To see how consecutive steps in the above query are interrelated, we have a look at two goals in the last query in some more detail; this is shown in Fig. 4.2. It is indicated here how the new pair *_P3* is expressed in terms

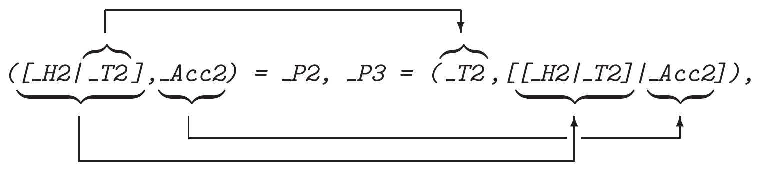

Figure 4.2: Exploring Details of the Rhyme’s Structure

of the old pair *_P2*. This observation gives rise to (P-4.5), a fourth version of *rhyme_prel/2*.

Prolog Code P-4.5: *Fourth version of* *rhyme_prel/2*

```
rhyme_prel_4(V,R) :- rhyme_acc(V,[],R).

rhyme_acc([],R,R).
rhyme_acc([HOld|TOld],AccOld,R) :-
   rhyme_acc(TOld,[[HOld|TOld]|AccOld],R).
```

4.1.3 The Final Version

We may use any of the four versions produced thus far of *rhyme_prel/2* to obtain a rough version of *rhyme/0* by replacing in the query on p. 122, Sect. 4.1.1, the term *verse_skeleton(_V)* by the term *verse(_V)*; for example,

```
?- verse(_V), rhyme_prel_2(_V,_R), show_rhyme(_R).
That lay in the house that Jack built.

That ate the malt
That lay in the house that Jack built.

...

This is the farmer sowing his corn
That kept the cock that crowed in the morn
...
That tossed the dog
That worried the cat
That killed the rat
That ate the malt
That lay in the house that Jack built.
```

We realize that the rhyme thus produced is not quite what we want: the first line of each verse (and not merely that of the last verse) should begin with '`This is ...`'. This effect will be achieved in three steps.

1. Define a predicate *to_first/2* which, when applied to an atom, replaces all its characters up to the first occurrence of the string '`the`' by the string '`This is `'. Example:

```
?- to_first('We find the definite article.',A).
A = 'This is the definite article.'
```

2. Define *change_first/2* in terms of *to_first/2* by

```
change_first([H1|T],[H2|T]) :- to_first(H1,H2).
```

This predicate applies *to_first/2* to the head of a list of atoms while leaving the tail unchanged. Example:

```
?- change_first(['That was the first','Now the second',
   'Now the third'],L).
L = ['This is the first', 'Now the second', 'Now the third']
```

3. Now apply *change_first/2* by means of the built-in predicate *maplist/3* to the first line of each verse of the rhyme's rough version.

Below we show our definition of *to_first/2*.

```
to_first(Old,New) :- atom_chars(Old,Charlist),
                     change(Charlist,Newlist),
                     concat_atom(Newlist,New).
```

Given an atom (in `Old`), it is first converted by means of the built-in predicate *atom_chars/2* into a list of one-character atoms (in `Charlist`).

Built-in Predicate: *atom_chars(?Atom,?CharList)*

It converts an atom into the corresponding list of one–character atoms and vice versa. Example:

```
?- atom_chars('Text',L).
L = ['T', e, x, t]
```

The predicate *change/2* is then used to effect the intended change in the atom's list-of-characters representation; it is defined by[1]

```
change([t,h,e|T],['T',h,i,s,' ',i,s,' ',t,h,e|T]) :- !.
change([_|T],X):- change(T,X).
```

and its behaviour is exemplified by

```
?- change(['F',i,n,d,' ',t,h,e,' ',s,t,r,i,n,g],_L),
   write_term(_L,[]).
[T, h, i, s,  , i, s,  , t, h, e,  ,s, t, r, i, n, g]
```

Finally, the built-in predicate *concat_atom/2* is used to convert the list-of-characters in `Newlist` into an atom (in `New`).[2]

Built-in Predicate: *concat_atom(+List,-Atom)*

Atom is obtained by concatenating the elements of *List*. Example:

```
?- concat_atom([atom1,atom2,atom3],A).
A = atom1atom2atom3
```

Having thus arrived at an implementation of *change_first/2*, we now want to apply this predicate to the head of each of the rough rhyme's verses. Since the latter is available (from *rhyme_prel/2*) as a list, we may use *maplist/3* for a concise definition of *rhyme/0*.

[1]Because of the *cut*, *change/2* will fail on backtracking even for multiple occurrences of the substring '`the`' in its first argument.

[2]For the present purposes where a list of *single character* atoms needs concatenating, we may use *atom_chars/2* as an alternative. The last goal in the definition of *to_first/2* then reads as `atom_chars(New,Newlist)`.

Built-in Predicate: *maplist(+Pred,?List1,?List2)*

The 2–ary predicate *Pred* is applied to each entry of *List1* giving *List2* and vice versa.[3]Example:

```
?- maplist(append([a,b]),[[r,s],[u,v]],L).
L = [[a, b, r, s], [a, b, u, v]]
?- maplist(append([a,b]),L,[[a,b,r,s],[a,b,u,v]]).
L = [[r, s], [u, v]]
```

(Here, *append/3* became a 2–ary predicate by partial application by fixing its first argument to *[a,b]*.)

Now, any of the four versions of *rhyme_prel/2* may be used to define *rhyme/0*; for example,

```
rhyme_2 :- verse(V),
           rhyme_prel_2(V,RTemp),
           maplist(change_first,RTemp,R),
           show_rhyme(R).
```

4.1.4 Other Approaches

All solutions considered thus far were based on (some form of) the accumulator technique. The problem at hand can also be approached by simple recursion, however. To arrive at such a solution, we first show in Table 4.1 the desired rhyme for some last verses of various lengths. We ask ourselves the following question:

Last Verse	*Rhyme*
['A']	[['A']]
['B','A']	[['A'],['B','A']]
['C','B','A']	[['A'],['B','A'],['C','B','A']]
['D','C','B','A']	[['A'],['B','A'],['C','B','A'],['D','C','B','A']]
...	...

Table 4.1: Rhyme Structure

Given a particular rhyme, how can the previous rhyme be expressed in terms of the current one?

A *declarative* reading of the last two lines of Table 4.1 suggest the following: *[H|T]* is the last verse of the current rhyme *C* if *T* is the last verse of the previous rhyme *P* and *C* comes about by appending *[[H|T]]* to

[3]The 'reverse' application of *maplist/3* is possible only if the second argument of *Pred* may be used in the input mode. This is *not* the case for example for *flatten/2* as is shown below.

```
?- maplist(flatten,[[a,[b,[c,d],e]],[[[r,s],t],x,y]],L).
L = [[a, b, c, d, e], [r, s, t, x, y]]
?- maplist(flatten,L,[[a,b,c,d,e],[r,s,t,x,y]]).
No
```

P. And, the boundary case is identified by observing that the one-line verse *[L]* is the last verse of *[[L]]*. The aforesaid is immediately expressed in Prolog by either of the two (*logically* equivalent) definitions (P-4.6) and (P-4.7).[4]

Prolog Code P-4.6: *Fifth version of* rhyme_prel/2

```
rhyme_prel_5([L],[[L]]).
rhyme_prel_5([H|T],C) :- append(P,[[H|T]],C), rhyme_prel_5(T,P).
```

Prolog Code P-4.7: *Sixth version of* rhyme_prel/2

```
rhyme_prel_6([L],[[L]]).
rhyme_prel_6([H|T],C) :- rhyme_prel_6(T,P), append(P,[[H|T]],C).
```

(It is readily confirmed that both versions behave as earlier ones do.) As each of the last two predicates is defined in terms of *append/3* we would expect some improvement in elegance (and performance) by rewriting

[4]The following are alternative first clauses:

```
rhyme_prel_5([],[]).
rhyme_prel_6([],[]).
```

them using *difference lists*. Indeed, both versions give rise to (P-4.8), the same concise, tail recursive implementation using difference lists.

Prolog Code P-4.8: *Seventh version of* `rhyme_prel/2`

```
rhyme_prel_dl([L],[[L]|X]-X).
rhyme_prel_dl([H|T],C1-C2) :- rhyme_prel_dl(T,C1-[[H|T]|C2]).

rhyme_prel_7(V,R) :- rhyme_prel_dl(V,R-[]).
```

Exercise 4.1. We want to make an experimental comparison between the various versions of `rhyme_prel/2` and need therefore a predicate that produces rhymes of any specified length. To be more specific, we will need a predicate `long_verse/1` which removes from the database the current version of `verse/1` and replaces it by something of a repetitive structure and of a specified length as shown in the session below.

```
?- long_verse(3), verse(_V), show_list(_V).
That interacts with the item ...
That interacts with the item ...
That interacts with the item ...
?- rhyme_2.
This is the item ...

This is the item ...
That interacts with the item ...

This is the item ...
That interacts with the item ...
That interacts with the item ...
```

Define the predicate `long_verse/1`.

■

We can now use `long_verse/1` in conjunction with the built-in predicate `time/1` to assess the versions' performance; this is shown for the last three versions in Table 4.2 below.[5] As expected, version seven, the imple-

Version	*length of _V*	100	200	300	400	500
5	*CPU-time [sec]*	1.97	15.77	52.50	125.0	244.1
Version	*length of _V*	1,000	2,000	3,000	4,000	5,000
6	*CPU-time [sec]*	4.51	20.04	45.53	85.63	132.4
Version	*length of _V*	10,000	20,000	30,000	40,000	50,000
7	*CPU-time [sec]*	0.28	0.71	0.55	1.32	1.16

Table 4.2: CPU Times for Versions of the Query `?- rhyme_prel(_V,_R).`

mentation based on difference lists, is by far the most efficient. Furthermore, perhaps surprisingly, version six

[5]The first entry in Table 4.2 for example may be obtained by

```
?- long_verse(100), verse(_V), time(rhyme_prel_5(_V,_R)).
% 176,749 inferences in 1.97 seconds (89720 Lips)
```

turns out to be better than its tail recursive counterpart, version five. We turn to Prolog's tracing facility to find out why this is the case:

```
?- trace([append/3,rhyme_prel_5/2,rhyme_prel_6/2]).
%         append/3: [call, redo, exit, fail]
%         rhyme_prel_5/2: [call, redo, exit, fail]
%         rhyme_prel_6/2: [call, redo, exit, fail]
[debug]  ?- rhyme_prel_5(['B','A'],R).
 T Call: (6) rhyme_prel_5(['B', 'A'], _G418)
 T Call: (7) append(_G506, [['B', 'A']], _G418)
 T Exit: (7) append([], [['B', 'A']], [['B', 'A']])
 T Call: (7) rhyme_prel_5(['A'], [])
 T Call: (8) append(_G512, [['A']], [])
 T Fail: (8) append(_G512, [['A']], [])
 T Fail: (7) rhyme_prel_5(['A'], [])
 T Redo: (7) append(_G506, [['B', 'A']], _G418)
 T Exit: (7) append([_G476], [['B', 'A']], [_G476, ['B', 'A']])
 T Call: (7) rhyme_prel_5(['A'], [_G476])
 T Exit: (7) rhyme_prel_5(['A'], [['A']])
 T Exit: (6) rhyme_prel_5(['B', 'A'], [['A'], ['B', 'A']])
R = [['A'], ['B', 'A']]
[debug]  ?- rhyme_prel_6(['B','A'],R).
 T Call: (6) rhyme_prel_6(['B', 'A'], _G418)
 T Call: (7) rhyme_prel_6(['A'], _G498)
 T Exit: (7) rhyme_prel_6(['A'], [['A']])
 T Call: (7) append([['A']], [['B', 'A']], _G418)
 T Exit: (7) append([['A']], [['B', 'A']], [['A'], ['B', 'A']])
 T Exit: (6) rhyme_prel_6(['B', 'A'], [['A'], ['B', 'A']])
R = [['A'], ['B', 'A']]
```

It is seen that version five causes Prolog to backtrack on the search tree of *append/3* until *append([_G482],[['B','A']]* succeeds. This is quite a contrast to *rhyme_prel_6* which does not cause backtracking but builds up a stack of subgoals all of which eventually are satisfied in turn. It is also easily verified that on backtracking version five will not terminate whereas version six will fail to re-satisfy the goal and returns 'No'.

Exercise 4.2. Modify the definition of *rhyme_prel_5/2* such that it won't loop but fails on backtracking.

■

Exercise 4.3. Define *cputime(+Predname,+Arglist,-Time)* for obtaining the CPU seconds in *Time* for the predicate with name *Predname* and arguments in *Arglist*. Then, for example, the following is an alternative to the query in footnote 5 on p. 129:

```
?- long_verse(100),verse(_V),cputime(rhyme_prel_5,[_V,_R],Time).
Time = 1.97
```

The predicate *cputime/3* will be an improvement on *time/1* since it will then be possible to produce for example the first row of Table 4.2 *in one sweep* interactively as follows.

```
?- findall(_Time,(member(_L,[100,200,300,400,500,600,700]),
                  long_verse(_L),
                  verse(_V),
                  cputime(rhyme_prel_5,[_V,_R],_Time)),
              Times).
Times = [2.03, 15.71, 52.29, 124.51, 242.5, 419.58, 667.78]
```

(Slight variations in the CPU times may be observed even when repeating the same query.) In your definition of *cputime/3* you should use the built-in predicate *statistics/2*.

Built-in Predicate: *statistics(+Key,-Value)*

Unify system statistics determined by *Key* with *Value*. For example, we obtain the CPU seconds and number of inferences accumulated in the present Prolog session by

```
?- statistics(cputime,Time).
Time = 18020.2
?- statistics(inferences,Inf).
Inf = 222054681
```

■

Exercise 4.4. We have created several versions of *rhyme_prel/2* and have indicated the version number by an appropriate suffix attached to the original predicate name. Let us now assume that this is the style for indicating predicates' versions in general. In this exercise, you are asked to define a predicate *cputime/4* which is a generalization of *cputime/3* from Exercise 4.3 in that the former will allow the version number to be specified by an extra (the third) argument. Example:

```
?- long_verse(100),verse(_V),cputime(rhyme_prel,[_V,_R],5,Time).
Time = 1.97
```

The benefit of *cputime/4* is obvious: it will allow the timing of several versions of the same predicate in one sweep, as is illustrated below.

```
?- long_verse(70000), verse(_V),
   maplist(cputime(rhyme_prel,[_V,_R]),[1,2,3,4,7],Times).
Times = [4.28, 3.19, 3.35, 1.54, 3.18]
```

■

Exercise 4.5. Using *cputime/4* from Exercise 4.4, produce all entries of Table 4.2 interactively *by one single query.*

Hint. As a first step, you should revisit the problem of producing interactively a list comprising the *first row* of entries in Table 4.2 (c.f. Exercise 4.3). This is now best achieved by using the built-in predicates *findall/3* and *between/3* and by observing that the last verse's length is expressed in terms of the column number $j = 1, \ldots, 5$ as

$$\mathit{length} = j \times 10^2$$

The general case is dealt with by nesting two such constructs. Version number and length are respectively generated by

$$\mathit{version} = i + 3$$
$$\mathit{length} = j \times 10^i$$

with $i = 2, 3, 4$ and $j = 1, \ldots, 5$.

■

4.2 Project: *'One Man Went to Mow ...'*

Another nursery rhyme with a similar recursive structure is the well-known song *One man went to mow ...* whose three-verse version is as follows.[6]

One man went to mow,
Went to mow a meadow,
One man and his dog,
Went to mow a meadow.

[6]Source: The *BBC* web site
`http://www.bbc.co.uk/cbeebies/tweenies/songtime/`
It is a cornucopia of songs and rhymes for pre-school children.

Two men went to mow,
Went to mow a meadow,
Two men, one man and his dog,
Went to mow a meadow.

Three men went to mow,
Went to mow a meadow,
Three men, two men, one man and his dog,
Went to mow a meadow,
Went to mow a meadow.

We want to outline here the way this rhyme can be produced in Prolog and formulate the stages of the detailed work as exercises.

This song has a very similar recursive structure to that of *This is the house that Jack built* except that there is now no predefined 'last verse' from which we could unravel the entire rhyme. Our aim is to produce a predicate *song/0* returning on the terminal a continuous stream of verses until stopped by the keystrokes $\boxed{Ctrl}$+$\boxed{C}$. The intended behaviour is shown in Fig. 4.3.[7]

```
?- song.
One man went to mow,
Went to mow a meadow,
One man and his dog,
Went to mow a meadow.

Two men went to mow,
Went to mow a meadow,
Two men,
  one man and his dog,
Went to mow a meadow.
...
Seven men went to mow,
Went to mow a meadow,
Seven men,
  six men,
  five men,
  four men,
  three men,
  two men,
  one man and his dog,
Went to mow a meadow.

Action (h for help) ? abort
% Execution Aborted
```

Figure 4.3: Desired Behaviour of *song/0*

[7]Here we deliberately avoid asking for a *fixed* number of verses since otherwise the task would not be dissimilar enough to the one considered in Sect. 4.1: we could then produce a 'last verse' with relative ease and then proceed as before.

The core of the implementation is a predicate *song_skeleton/1* which on backtracking returns the skeleton structure of each verse using numerals.

```
?- song_skeleton(Verse).
Verse = [1] ;
Verse = [2, 1] ;
Verse = [3, 2, 1] ;
...
```

Exercise 4.6. Define the predicate *song_skeleton/1* by *recursion*.
Hint. You may model your definition of *song_skeleton/1* on that of the predicate *int/1*, which on backtracking returns all natural numbers:

```
?- int(N).
N = 1 ;
N = 2 ;
N = 3 ;
...
```

The predicate *int/1* is defined in terms of an auxiliary predicate *int(+Int1,?Int2)* by

```
int(N) :- int(1,N).
```

which on backtracking instantiates *Int2* to all integers starting from *Int1*:

```
?- int(5,I).
I = 5 ;
I = 6 ;
I = 7 ;
...
```

The definition of *int/2* is as follows.

```
int(I,I).
int(Last,I) :- succ(Last,New), int(New,I).
```

Built-in Predicate: *succ(?Int1,?Int2)*

Succeeds if $Int1 = Int2 + 1$. Incrementation by *succ/2* is faster than by the usual arithmetic predicate.

■

There is in Prolog, as an alternative to recursion, the facility of failure driven, and repeat loops for the implementation of code with a repetitive behaviour. We want to illustrate this idea by way of a predicate *nat/1* which has the same specification as the predicate *int/1* from above but is defined in terms of a repeat loop rather than by recursion. Let *nat/1* be defined by

```
            nat(N) :- first_nat, current_nat(N).
            nat(N) :- repeat, update_nat, current_nat(N).
```

with the auxiliary predicates

```
            first_nat :- dynamic(current_nat/1),
                         retractall(current_nat(_)),
                         assert(current_nat(1)).
```

and

```
            update_nat :- current_nat(N),
                          retractall(current_nat(_)),
                          NewN is N + 1,
                          assert(current_nat(NewN)).
```

The predicate *current_nat/1* is used here to hold the current value of the natural number in the database as a fact. *first_nat/0* clears the database of all facts defining *current_nat/1* (possibly originating from earlier invocations of *nat/1*) and writes to the database the first natural number. *update_nat/0* retrieves the previous value, clears the database, and writes back the updated value. The generation of an infinite stream of values by (the second clause of) *nat/1* hinges on the built-in predicate *repeat/0* which always succeeds on backtracking and is best thought of as returning a distinct (albeit invisible) 'solution' each time it is re-invoked. The conjunction of subgoals to the right of `repeat`, i. e.

```
        update_nat, current_nat(N)
```

is re-satisfied on backtracking, resulting in an update of *N*. The database serves here as a 'scratchpad' for intermediate results.

Exercise 4.7. Define a second version of the predicate *song_skeleton/1* by a *repeat loop.* Your solution should be modelled on the definition of *nat/1*.

■

There are of course other possibilities, too, for defining *song_skeleton/1*. Take for example the one suggested by the following query.

```
?- current_prolog_flag(max_integer,_Largest),
   between(1,_Largest,_H), findall(_I,between(1,_H,_I),_R),
   reverse(_R,L).
L = [1] ;
L = [2, 1] ;
L = [3, 2, 1] ;
...
```

The list *L* is constructed here by:

- Getting hold of the largest number *_Largest* which can be represented in SWI–Prolog as an integer.
- Obtaininig the head *_H* of *L* by the built-in predicate *between/3*.
- Creating the reverse *_R* of *L* by the all-solutions predicate *findall/3*.
- And, finally, reversing *_R* to get *L*.

A new *L* is obtained each time the query's second goal is re-satisfied. This solution is neither concise nor is it as elegant as the earlier ones, however.

The remaining steps for the completion of *song/0* are spelt out in the Exercises 4.8 to 4.11 below.

Exercise 4.8. Define a predicate *digits(+Number,-List)* for converting a natural *Number* into the list of its digits in *List*:

```
?- digits(351,L).
L = [3, 5, 1]
```

(As an optional task which, however, is not needed in the present context, you may extend the definition of *digits/2* for the instantiation pattern *digits(-Number,+List)*.)

Now define a predicate *in_words(+Num,-Atom)* for converting a numeral *Num* to its plain English equivalent in *Atom*. (Allow for up to 9,999 in *Num*.) Example:

```
?- in_words(351,A).
A = threehundredfiftyone[8]
```

■

Exercise 4.9. In the definition of the first and third lines of each verse you will need a predicate *capital/2* for converting the first character of an atom to its upper case equivalent:

[8]For reasons of simplicity, the rules of hyphenation and separating spaces are ignored here.

```
?- capital('sixteen men, fifteen men, fourteen men',C).
C = 'Sixteen men, fifteen men, fourteen men'
```

Define *capital/2*.

Note. Use the built-in predicate *atom_chars/2* to disassemble atoms into lists and vice versa; see, inset on p. 126. For a concise solution to converting single letters to upper case you will also need the built-in predicate *char_code/2*.[9]

Built-in Predicate: *char_code(?Char,?ASCII)*

Converts the single-character atom *Char* to its ASCII code in *ASCII* and vice versa. Example:

```
?- char_code(a,ASCII).
ASCII = 97
?- char_code(Char,65).
Char = 'A'
```

■

Exercise 4.10. Define a predicate *line3/2* for generating the third line of each verse; for example, the third verse's third line we get by

```
?- line3([3,2,1],Text), write(Text).
Three men,
  two men,
  one man and his dog,
Text = 'Three men,\n  two men,\n  one man and his dog,'
```

In your work, you may be guided by the following query:

```
?- maplist(in_words,[16,15,14],[H|T]),
   maplist(atom_concat(' men, '),T,L), concat_atom([H|L],A),
   atom_concat(A,' men',A2).
H = sixteen
T = [fifteen, fourteen]
L = [' men, fifteen', ' men, fourteen']
A = 'sixteen men, fifteen men, fourteen'
A2 = 'sixteen men, fifteen men, fourteen men'
```

■

[9]A simpler but more tedious alternative is by using a predicate which is defined by 26 facts – one for each letter in the English alphabet.

Built-in Predicate: *atom_concat(?Atom1,?Atom2,?Atom3)*

Atom3 is the concatenation of *Atom1* and *Atom2*. At least two of the arguments must be instantiated. Alternatively, it suffices if the last argument is instantiated only. Examples:

```
?- atom_concat(atom1,atom2,A).
A = atom1atom2
?- atom_concat(A1,A2,atom3).
A1 = '' A2 = atom3 ;
A1 = a A2 = tom3
Yes
```

Exercise 4.11. Complete the definition of *song/0* by using your predicates from the Exercises 4.6 to 4.10.

■

"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

4.3 Chapter Notes

We have illustrated a practical Prolog development technique based on an incremental, exploratory and interactive working style. It is not dissimilar to the *Incremental Development Model* known from Software Engineering (e. g. [15]) the application of which in the commercial context results in *prototypes* at an early stage for evaluation and feedback. We have identified the following development stages in particular for predicates defined by *recursion*:

- Identify informally a *recursive* structure of the problem.
- Experiment *interactively* to explore and confirm the above.
- Identify a pattern and write *pseudo–code*.
- Write a *preliminary* (and perhaps incomplete) Prolog implementation.
- Refine details to arrive at a *final* Prolog implementation.

The method discussed here won't of course be a substitute for existing formal approaches to logic programming that are rooted in Mathematical Logic (e. g. [5]).