Chapter 1

Enigma 1225: Rows are Columns[1]

1.1 A Puzzle

A regular feature in the *New Scientist* magazine is *Enigma*, a weekly puzzle entry which readers are invited to solve. In the 8 February 2003 issue [1] the following puzzle was published.

> First, draw a chessboard. Now number the horizontal rows $1, 2, ..., 8$, from top to bottom and number the vertical columns $1, 2, ..., 8$, from left to right.You have to put a whole number in each of the sixty-four squares, subject to the following:
>
> 1. No two rows are exactly the same.
> 2. Each row is equal to one of the columns, but not to the column with the same number as the row.
> 3. If N is the largest number you write on the chessboard then you must also write $1, 2, ..., N - 1$ on the chessboard.
>
> The sum of the sixty-four numbers you write on the chessboard is called your total. What is the largest total you can obtain?

We are going to solve this puzzle here using Prolog. The solution to be described will illustrate two techniques: *unification* and *generate-and-test*.

Unification is a built-in pattern matching mechanism in Prolog which has been used in [9]; for example, the difference list technique essentially depended on it. For our approach here, unification will again be crucial in that the proposed method of solution hinges on the availability of built-in unification. It will be used as a kind of concise symbolic pattern generating facility without which the current approach wouldn't be viable.

Generate-and-test is easily implemented in Prolog. Prolog's backtracking mechanism is used to *generate* candidate solutions to the problem which then are *tested* to see whether certain of the problem-specific constraints are satisfied.

1.2 First Thoughts

Fig. 1.1 shows a board arrangement with all required constraints satisfied. It is seen that the first requirement

[1] This chapter is based on [7]. The author thankfully acknowledges the permission by Elsevier to republish the material here.

Download free eBooks at bookboon.com

1	3	1	3	6	6	6	6	6
2	3	3	1	6	6	6	6	6
3	1	3	3	6	6	6	6	6
4	6	6	6	4	5	2	5	4
5	6	6	6	4	4	5	2	5
6	6	6	6	5	4	4	5	2
7	6	6	6	2	5	4	4	5
8	6	6	6	5	2	5	4	4
	1	*2*	*3*	*4*	*5*	*6*	*7*	*8*

Figure 1.1: A Feasible Solution

is satisfied since the rows are all distinct. The second condition is also seen to hold whereby rows and columns are interrelated in the following fashion:

Column	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
Row	*2*	*3*	*1*	*5*	*6*	*7*	*8*	*4*

We use the permutation

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 2 & 3 & 1 & 5 & 6 & 7 & 8 & 4 \end{pmatrix} \tag{1.1}$$

to denote the corresponding column–to–row transformation. The board also satisfies the latter part of the second condition since no row is mapped to a column in the same position. In terms of permutations, this requirement implies that no entry remains fixed; these are those permutations which in our context are *permissible.* [2] The third condition is obviously also satisfied with $N = 6$. The board's total is 301, not the maximum, which, as we shall see later, is 544.

1.3 Symbolic Solutions

The solution scheme described below in i–v is based on first generating all feasible solutions (an example of which was seen in Sect. 1.2) and then choosing a one with the maximum total.

i. Take an admissible permutation, such as π in (1.1).

ii. Find an 8×8 matrix with *symbolic* entries whose rows and columns are interrelated by the permutation

[2]Such permutations are called *derangements* ([3], p. 73).

in i. As an example, let us consider for the permutation π two such matrices, $\mathbf{M}_1$ and $\mathbf{M}_2$, with

$$\mathbf{M}_1 = \begin{bmatrix} X_3 & X_1 & X_3 & X_6 & X_6 & X_6 & X_6 & X_6 \\ X_3 & X_3 & X_1 & X_6 & X_6 & X_6 & X_6 & X_6 \\ X_1 & X_3 & X_3 & X_6 & X_6 & X_6 & X_6 & X_6 \\ X_6 & X_6 & X_6 & X_4 & X_5 & X_2 & X_5 & X_4 \\ X_6 & X_6 & X_6 & X_4 & X_4 & X_5 & X_2 & X_5 \\ X_6 & X_6 & X_6 & X_5 & X_4 & X_4 & X_5 & X_2 \\ X_6 & X_6 & X_6 & X_2 & X_5 & X_4 & X_4 & X_5 \\ X_6 & X_6 & X_6 & X_5 & X_2 & X_5 & X_4 & X_4 \end{bmatrix}$$

$$\mathbf{M}_2 = \begin{bmatrix} Y_3 & Y_1 & Y_3 & Y_1 & Y_1 & Y_1 & Y_1 & Y_1 \\ Y_3 & Y_3 & Y_1 & Y_1 & Y_1 & Y_1 & Y_1 & Y_1 \\ Y_1 & Y_3 & Y_3 & Y_1 & Y_1 & Y_1 & Y_1 & Y_1 \\ Y_1 & Y_1 & Y_1 & Y_4 & Y_5 & Y_2 & Y_5 & Y_4 \\ Y_1 & Y_1 & Y_1 & Y_4 & Y_4 & Y_5 & Y_2 & Y_5 \\ Y_1 & Y_1 & Y_1 & Y_5 & Y_4 & Y_4 & Y_5 & Y_2 \\ Y_1 & Y_1 & Y_1 & Y_2 & Y_5 & Y_4 & Y_4 & Y_5 \\ Y_1 & Y_1 & Y_1 & Y_5 & Y_2 & Y_5 & Y_4 & Y_4 \end{bmatrix}$$

$\mathbf{M}_1$ and $\mathbf{M}_2$ both satisfy conditions 1 and 2. We also observe that the pattern of $\mathbf{M}_2$ may be obtained from that of $\mathbf{M}_1$ by specialization (by matching the variables X_1 and X_6). Thus, any total achievable for $\mathbf{M}_2$ is also achievable for $\mathbf{M}_1$. For any given permissible permutation, we can therefore concentrate on the *most general* pattern of variables, $\mathbf{M}$. (We term a pattern of variables *most general* if it cannot be obtained by specialization from a more general one.) All this is reminiscent of 'unification' and the 'most general unifier', and we will indeed be using Prolog's unification mechanism in this step.

iii. Verify condition 1 for the symbolic matrix $\mathbf{M}$. [3] Once this test is passed, we are sure that also the latter part of condition 2 is satisfied. [4]

iv. We now *evaluate* the pattern $\mathbf{M}$. If N symbols have been used in $\mathbf{M}$, assign the values $1, ..., N$ to them

[3] This test is necessary since at this stage a matrix may have been generated failing to satisfy condition 1 as is illustrated by the (admissible) permutation

$$\rho = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 2 & 3 & 1 & 5 & 4 & 7 & 8 & 6 \end{pmatrix} \tag{1.2}$$

and the corresponding most general matrix $\mathbf{M}_3$:

$$\mathbf{M}_3 = \begin{bmatrix} Z_4 & Z_1 & Z_4 & Z_9 & Z_9 & Z_5 & Z_6 & Z_7 \\ Z_4 & Z_4 & Z_1 & Z_9 & Z_9 & Z_7 & Z_5 & Z_6 \\ Z_1 & Z_4 & Z_4 & Z_9 & Z_9 & Z_6 & Z_7 & Z_5 \\ Z_9 & Z_9 & Z_9 & Z_3 & Z_3 & Z_{10} & Z_{10} & Z_{10} \\ Z_9 & Z_9 & Z_9 & Z_3 & Z_3 & Z_{10} & Z_{10} & Z_{10} \\ Z_7 & Z_6 & Z_5 & Z_{10} & Z_{10} & Z_8 & Z_2 & Z_8 \\ Z_5 & Z_7 & Z_6 & Z_{10} & Z_{10} & Z_8 & Z_8 & Z_2 \\ Z_6 & Z_5 & Z_7 & Z_{10} & Z_{10} & Z_2 & Z_8 & Z_8 \end{bmatrix}$$

[4] Were it not so, there would exist a row and a column with the same index such that the two were identical. However, this row will be identical (by way of the *admissible* permutation) to some other column too. Hence two columns and therefore also two rows would be identical, thus failing the test.

Download free eBooks at bookboon.com

in reverse order by first assigning N to the most frequently occurring symbol, $N-1$ to the second most frequently occurring symbol etc. The total thus achieved will be a maximum for the given pattern $\mathbf{M}$.

v. The problem is finally solved by generating and evaluating all patterns according to i–iv and selecting a one with the maximum total.

1.4 Implementation Details

1.4.1 Design Decisions

The original formulation from the *New Scientist* uses a chessboard but the problem can be equally set with a square board of any size. In our implementation, we shall allow for any board size since this will allow the limitations of the method employed to be explored.

We write matrices in Prolog as lists of their rows which themselves are lists. Permutations will be represented by the list of the bottom entries of their two-line representation; thus, `[2, 3, 1, 5, 6, 7, 8, 4]` stands for π in (1.1).

Need help with your dissertation?

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

Get Help Now

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

Click on the ad to read more

1.4.2 Admissible Permutations

First, we want to generate all permutations of a list. Let us assume that we want to do this by the predicate *permute(+List,-Perm)* and let us see how *List = [1, 2, 3, 4]* might be permuted. A permuted list, *Perm = [3, 4, 1, 2]* say, may be obtained by

- Removing from *List* the entry *E = 3*, leaving the reduced list
  R = [1, 2, 4]
- Permuting the reduced list *R* to get *P = [4, 1, 2]*
- Assembling the permuted list as *[E|P] = [3, 4, 1, 2]*.

Lists with a single entry are left unchanged. This gives rise to the definition

```
permute([X],[X]).
permute(L,[E|P]) :- remove_one(L,E,R), permute(R,P).
```

with the predicate *remove_one(+List,?Entry,?Reduced)* defined by

```
remove_one([H|T],H,T).
remove_one([H|T],E,[H|L]) :- remove_one(T,E,L).
```

(Here we remove either the head or an entry from the tail.) For a permutation to be admissible, all entries must have changed position. We implement this by

```
admissible(L,P) :- permute(L,P), all_changed(L,P).

all_changed([X],[Y]) :- X ~ Y.
all_changed([H1|T1],[H2|T2]) :- H1 ~ H2, all_changed(T1,T2).
```

Exercise 1.1. Provide an alternative definition of *remove_one/3* by using one clause and *append/3*. ■

1.4.3 Generating Symbolic Matrices

To generate a list of *N* unbound variables, *L*, we use *var_list(+N,-L)* which is defined in terms of *length(-L,+N)* by

```
var_list(N,L) :- length(L,N).
```

(See [9, p. 110, footnote 15].) Matrices with distinct symbolic entries may now be produced by mapping; for example, a 3×2 matrix is obtained by

```
?- maplist(var_list,[2,2,2],M).
M = [[_G370, _G373], [_G379, _G382], [_G388, _G391]]
```

Exercise 1.2. Use the above idea to define *var_matrix(+Size,-M)* for generating a square symbolic matrix of any size. ■

Download free eBooks at bookboon.com

1.4.4 Permuting Rows

This is accomplished by *list_permute(+Perm,+L,-P)* as indicated below.

```
?- var_matrix(3,_M), list_permute([3,1,2],_M,_P),
   write_matrix(_M), nl, write_matrix(_P).
[_G779, _G782, _G785]
[_G791, _G794, _G797]
[_G803, _G806, _G809]

[_G803, _G806, _G809]
[_G779, _G782, _G785]
[_G791, _G794, _G797]
```

(The permutation *Perm* establishes a correspondence between the entries of *P* and those of *L*.)

Exercise 1.3. Define the predicate *list_permute/3* by recursion, using *nth1/3* from [9, p. 107]. ■

1.4.5 Transposing

This will be accomplished by *transpose(+M,-T)*.

```
?- maplist(var_list,[2,2,2],_M), transpose(_M,_T),
   write_matrix(_M), nl, write_matrix(_T).
[_G779, _G782]
[_G788, _G791]
[_G797, _G800]

[_G779, _G788, _G797]
[_G782, _G791, _G800]
```

Exercise 1.4. Use *maplist/3* to define *transpose/2*. Allow for any not necessarily square matrix as indicated above.

Hint. First define a predicate *col(+Matrix,+N,-Column)* for returning the *Nth* column of a matrix. ■

1.4.6 Most General Patterned Symbolic Matrices

It is now that Prolog shows its true strength: we use *unification* to generate symbolic square matrices with certain patterns.[5] For example, we may produce a 3×3 symmetric matrix thus

```
?- var_matrix(3,_M), transpose(_M,_M), write_matrix(_M).
[_G535, _G538, _G541]
[_G538, _G550, _G553]
[_G541, _G553, _G565]
```

[5] Trying to produce the results in this section by a programming language without built-in unification will be a much more involved exercise.

Download free eBooks at bookboon.com

More importantly, we are now in a position to produce symbolic matrices with prescribed patterns. For example, below we generate the most general 3×3 matrix whose rows and columns are interrelated by the permutation

$$\begin{pmatrix} 1 & 2 & 3 \\ 3 & 1 & 2 \end{pmatrix}$$

```
?- var_matrix(3,_M), list_permute([3,1,2],_M,_P),
   transpose(_P,_M), write_matrix(_M).
[_G748, _G748, _G754]
[_G754, _G748, _G748]
[_G748, _G754, _G748]
```

Unification is again seen to play a crucial rôle here as _M is *declared* to be the transpose of _P:

- `transpose/2` receives in its first argument the Prolog term for _P.
- The term for the transpose of _P is returned in the second argument of `transpose/2`.
- This then is unified with the term for _M thereby producing the intended pattern.

Download free eBooks at bookboon.com

Click on the ad to read more

1.4.7 Distinct Rows

We want to test whether all rows of a matrix with symbolic entries are distinct. Matrices are lists, we therefore need to test for distinctness of list entries which are Prolog *terms*. The matrix `[[A, B], [C, D]]` should pass the test, whereas `[[A, B], [A, B]]` should not. The negation of the unification operator (`\=/2`) cannot tell apart the rows of the first matrix; we need here a 'stronger' (i. e. more specialized) notion of equality as defined by the *term equivalence* operator `==/2` and its negation, `\==/2`. (See inset overleaf.) Thus, using `\==/2` will allow the rows of the former matrix to be recognized as different, whereas those of the latter are verified identical.

```
?- [A, B] \== [C, D].
A = _G240
B = _G243
C = _G246
D = _G249
Yes
?- [A, B] \== [A, B].
No
```

Built-in Predicates: `==/2` and `\==/2`

These two predicates are used to test for term 'equivalence' and its negation, respectively. Two terms are equivalent if there exists a term to which both of them have been bound *prior to* the invocation of `==/2`. For example, the query

```
?- X = u, g(X,V) = Y, f(h(g(u,V)),Y) == f(h(Y),g(X,V)).
X = u
V = _G448
Y = g(u, _G448)
Yes
```

succeeds since both sides have been bound (by prior unification) to the term `f(h(g(u,V)),g(u,V))`. However, the query

```
?- f(h(g(u,V)),Y) == f(h(Y),g(X,V)).
No
```

fails even though the two terms are unifiable:

```
?- f(h(g(u,V)),Y) = f(h(Y),g(X,V)).
V = _G325
Y = g(u, _G325)
X = u
Yes
```

Exercise 1.5. Use `\==/2` to define a predicate `distinct/1` for testing the distinctness of entries of a list as discussed above. ■

Download free eBooks at bookboon.com

1.4.8 Evaluating Patterns

Given a patterned symbolic matrix, we want to sort the list of its entries according to their frequencies of occurrence and assign the rank order to each. For example, in the matrix *_M* from the second query in Sect. 1.4.6, p. 22, the entry *_G748* occurs six times while *_G754* occurs thrice. Therefore, as shown below, *_G754* and *_G748* will be assigned the values 1 and 2 respectively.

```
?- var_matrix(3,_M), list_permute([3,1,2],_M,_P),
   transpose(_P,_M), eval_matrix(_M,Freq), write_matrix(_M).
[2, 2, 1]
[1, 2, 2]
[2, 1, 2]

Freq = [ (3, 1),  (6, 2)]
```

This shall be accomplished by the predicate `eval_matrix(?M,-Freq)`; it expects a symbolic matrix `M` in its first argument which then is unified with an integer matrix whose each entry will be the rank order of the frequency of the corresponding symbolic entry. The second argument `Freq` is unified with the list of frequencies for each number in the matrix as indicated above.

The hand computations in Fig. 1.2 on p. 27 indicate the steps involved in implementing `eval_matrix/2`.

① Produce the list of matrix entries by `flatten(+Matrix,-Entries)`.

② Discard multiple occurrences by `setof(E,member(E,+Entries),-Set)`.

③ Use `maplist(count_var(+Entries),+Set,-Multiplicities)` to count how many times each variable occurs in the matrix.

Exercise 1.6. Define the predicate `count_var(+VarList,+Var,-Num)`. It will behave as follows.

```
?- count_var([_A,_B,_A,_C,_B,_A],_B,N).
N = 2
```

④ Use `zip(+Multiplicities,+Set,-Frequencies)` to obtain the list of matrix entry frequencies by *zipping* the lists produced in ② and ③.

Exercise 1.7. Define the predicate `zip/3`. It should behave as follows.

```
?- zip([1,2,3],[a,b,c],L).
L = [ (1, a),  (2, b),  (3, c)]
```

⑤ Use `sort(+Frequencies,-FreqSorted)` (Prolog's built-in `sort/2`) to sort the pairs from ④. Tuples with less frequent matrix entries will precede those with more frequent ones.

⑥ Use `maplist(snd,+FreqSorted,-VarsSorted)` to retain the tuples' second entries only. We get a complete list of matrix entries, with no multiple copies, featuring in the rank order of their frequencies. `snd/2` extracts the second entry of a 2–tuple and is defined by

```
snd((_,X),X).
```

⑦ Use *length(+VarsSorted,-NVars)* to count the number of distinct matrix entries.

⑧ Use *from_to/3* to generate the list of integers *[1, ..., NVars]*. (The predicate *from_to/3* is known from [9, p. 17].)

⑨ Unify each variable in *VarsSorted* with the rank order of its frequency. A single call to *from_to(1,+NVars,?VarsSorted)* will accomplish both steps, ⑧ and ⑨. The effect of this call will also be that

- The initial (input) matrix will be bound to the integer matrix of frequency ranks. This will form the first output of *eval_matrix/2*.
- *FreqSorted* will be bound to the list of frequency pairs, forming the second output of *eval_matrix/2*.

The complete definition of *eval_matrix/2*, now a mere sequencing of clauses from ①–⑨, will be found in the source file `enigma.pl`.

"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

Download free eBooks at bookboon.com

Click on the ad to read more

The predicate *eval_matrix/2* has been defined in a style reminiscent of that used in *functional programming*. (The predicates *maplist/3*, *zip/3* and *snd/2* have indeed direct analogues in Haskell [30].) In [24], Parker espouses the virtues of this style for Prolog and calls it the 'stream data analysis paradigm'. Fig. 1.2 corresponds to what is called in [24] a 'dataflow diagram' or 'Henderson diagram'.

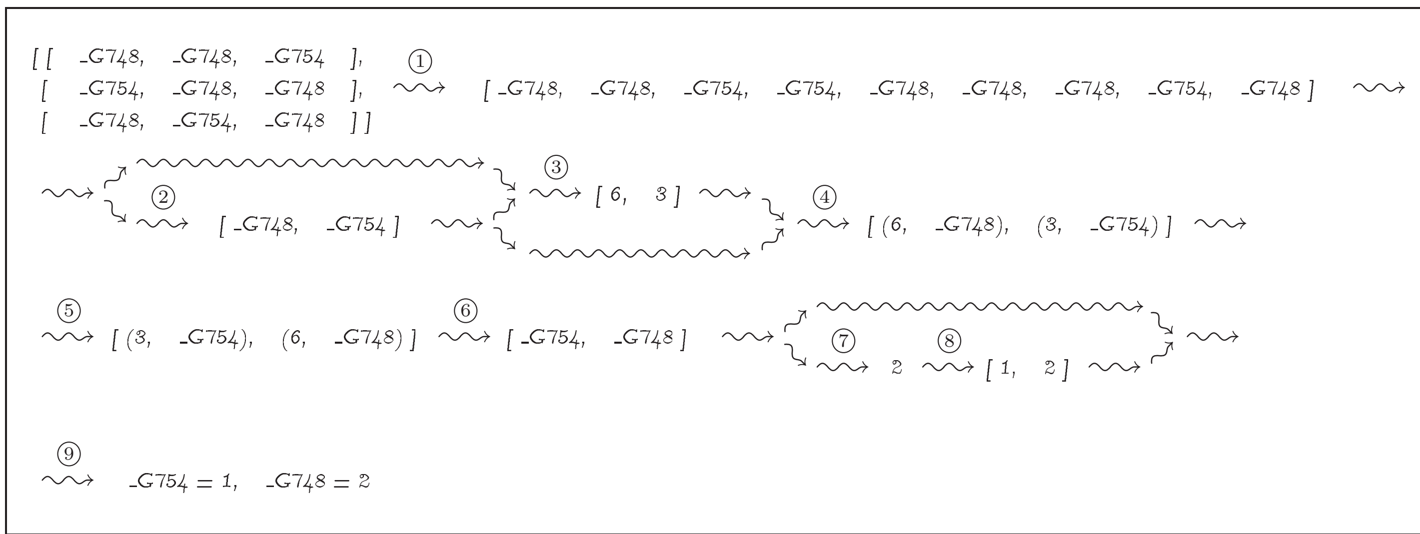


Figure 1.2: Hand Computations for Pattern Evaluation

total([(1,10),(2,100),(3,1000)], Total) ⇝

total([(1,10),(2,100),(3,1000)], 0, Total) ⇝

total([(2,100),(3,1000)], 10, Total) ⇝

total([(3,1000)], 210, Total) ⇝ total([], 3210, Total) ⇝

Total = 3210 ⇝ success

Figure 1.3: Suggested Hand Computations for *total/2*

1.4.9 Computing Totals

Exercise 1.8. For the computation of the matrix total we shall need a predicate *total(+IntPairs,-Total)* which should sum the product of paired entries as exemplified below.

```
?- total([(1,10),(2,100),(3,1000)],Total).
Total = 3210
```

Define *total/2* by the accumulator technique along the hand computations shown in Fig. 1.3. ■

1.4.10 Complete Implementation

In (P-1.1), we show the definition of *square/5* which has been assembled from the predicates in Sects. 1.4.2–1.4.9.

Prolog Code P-1.1: *Definition of square/5*

```
1 square(Size,M,Total,Freq,Perm) :- var_matrix(Size,M),
2                                   from_to(1,Size,One_to_Size),
3                                   admissible(One_to_Size,Perm),
4                                   list_permute(Perm,M,P),
5                                   transpose(P,M),
6                                   distinct(M),
7                                   eval_matrix(M,Freq),
8                                   total(Freq,Total).
```

square/5 may be used to search for feasible solutions as shown by the query in Fig. 1.4 for a 4×4 board. We know that all boards with the maximum total will be amongst those generated by the current process. Therefore, the largest of all totals thus generated will be the maximum total. We use *setof/3* to obtain the sorted list of all totals generated (without duplicates) and select the maximum value by the built-in predicate *last/2*:[6]

[6]There is some inconsistency between versions of SWI–Prolog here. Version 3.4.5 is used in the query below, but, the order of the arguments in *last/2* will have to be reversed if using version 5.2.7.

Download free eBooks at bookboon.com

```
?- square(4,_M,Total,Freq,Perm), write_matrix(_M).
[1, 1, 2, 3]
[1, 1, 3, 2]
[3, 2, 4, 4]
[2, 3, 4, 4]

Total = 40
Freq = [ (4, 1), (4, 2), (4, 3), (4, 4)]
Perm = [2, 1, 4, 3] ;
[1, 2, 2, 1]
[1, 1, 2, 2]
[2, 1, 1, 2]
[2, 2, 1, 1]

Total = 24
Freq = [ (8, 1), (8, 2)]
Perm = [2, 3, 4, 1] ;
...
```

Figure 1.4: Generating Feasible Solutions by *square/5*

Download free eBooks at bookboon.com

Click on the ad to read more

```
?- setof(_Tot,_M^_Freq^_Perm^square(8,_M,_Tot,_Freq,_Perm),Tots),
   last(Max,Tots).
Tots = [160, 244, 288, 301, 400, 544]
Max = 544
```

We now know that the maximum total is 544 and may find a board with that total (and the corresponding permutation) by

```
?- square(8,_M,544,_,Perm), write_imatrix(_M).
[   1   1   2   3   4   5   6   7]
[   1   1   3   2   5   4   7   6]
[   3   2   8   8   9  10  11  12]
[   2   3   8   8  10   9  12  11]
[   5   4  10   9  13  13  14  15]
[   4   5   9  10  13  13  15  14]
[   7   6  12  11  15  14  16  16]
[   6   7  11  12  14  15  16  16]

Perm = [2, 1, 4, 3, 6, 5, 8, 7]
```

Exercise 1.9. Define the predicate *write_imatrix/1* for displaying on the terminal an *integer* matrix with non-negative entries, right justified. In your definition, you should use *writef(+Format,+Arguments)* (Prolog's formatted *write*); see inset. The built-in predicates *concat_atom/2* [9, p. 126] and *int_to_atom/2* (see inset) may be used to construct *writef*'s first argument. ■

Built-in Predicate: *writef(+Format,+Arguments)*

This is one of Prolog's predicates for formatted *write*. *Arguments* is a list whose entries are displayed on the terminal according to the atom *Format*. Example:

```
?- writef('[%8r%8r%8r]',[12, 345, 6789]).
[      12     345    6789]
```

displays the list *[12, 345, 6789]* with its entries right justified, each occupying up to eight digits. Consult the manual [33] for the options available for *Format*.

Built-in Predicate: *int_to_atom(+Int,-Atom)*

Unifies *Atom* with the ASCII representation of *Int*. Example:

```
?- int_to_atom(1953,A).
A = '1953'
```

Download free eBooks at bookboon.com

Size	3	4	5	6	7	8	9
CPU Seconds	0.00	0.06	0.11	2.03	15.37	209.59	3,334.14

Table 1.1: CPU times for Various Board Sizes

1.5 Enhanced Implementation

1.5.1 What is Wrong with the Present Implementation?

The implementation obtained in Sect. 1.4.10 has serious limitations. Table 1.1 shows the CPU times needed for solving the puzzle for up to size 9 on a 300 MHz PC. The size of the original puzzle seems to be the practical limit of what can be solved by this method.[7] Table 1.1 indicates that the computing time increases roughly with the factorial of *Size*. This means for the original puzzle that $8! = 40,320$ permutations have to be generated of which $14,833$ will be admissible.[8] Each of these will give rise to a patterned symbolic matrix, each to be tested by ***distinct/1***. The number of patterned matrices passing this test is $13,713$.[9] All of them are then evaluated, resulting in a list with $13,713$ entries. After removing duplicates with ***setof/3***, we end up with a list of just six values!

There is obviously a great deal of duplication of effort here.

To reduce the number of permutations to be considered, we are going to introduce in the next section a *partitioning* of the set of all permutations into subsets, called *types*, such that permutations of the same type will *share* certain pertinent properties. More precisely, each of the following properties will be such that permutations *of the same type* either all have it or none has it.[10]

- Being admissible,
- For admissible permutations, the corresponding most general symbolic pattern having distinct rows.

Furthermore,

- For permutations of the same type, the corresponding most general symbolic pattern will evaluate to the same maximum total.

---

[7] There is another problem for larger sizes which could be overcome, however. For sizes exceeding 9, insufficient memory will be available for using *setof/3* to collect the values of total. To remedy the situation, we could instead calculate the maximum total in an incremental fashion by using, for instance, *assert/1* to save in the database the most recent maximum value of total.

[8] The number of admissible permutations can be found by the query

```
?- bagof(_A,admissible([1,2,3,4,5,6,7,8],_A),_As), length(_As,L).
L = 14833
```

Alternatively, the number of admissible permutations of $\{1,\ldots,n\}$, a_n, may be calculated by the recurrence relation

$$a_n = n! - (f_{1n} + f_{2n} + \ldots + f_{(n-1)n} + 1)$$

where

$$f_{in} = \binom{n}{i} a_{n-i}$$

denotes the number of permutations of $\{1,\ldots,n\}$ which leave *exactly* i entries fixed. Start with $a_1 = 0$. Other ways of calculating a_n may be found in [3, p. 73].

[9] We find this by the query

```
?- bagof(_Tot,_M^_Freq^_Perm^square(8,_M,_Tot,_Freq,_Perm),_Tots), length(_Tots,L).
L = 13713
```

The matrix $\mathbf{M}_3$ in footnote 3, p. 19, is an example for a pattern which will be tested by *distinct/1* and fail.

[10] We may call them therefore *type-properties*.

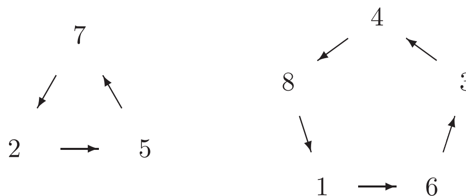


Figure 1.5: The Cycles τ_1 and τ_2

It will therefore suffice to concentrate on a *representative permutation from each type* (Sect. 1.5.3). Before elaborating on this idea, however, we first review some results from the Theory of Permutations [3].

1.5.2 Some Results from the Theory of Permutations

The Cycle Notation for Permutations

Let us look at the permutation

$$\tau = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 6 & 5 & 4 & 8 & 7 & 3 & 2 & 1 \end{pmatrix}$$

It can be thought of as the composition of two *cycles* τ_1 and τ_2 with

$$\tau_1 = \begin{pmatrix} 2 & 5 & 7 \\ 5 & 7 & 2 \end{pmatrix}, \ \tau_2 = \begin{pmatrix} 1 & 3 & 4 & 6 & 8 \\ 6 & 4 & 8 & 3 & 1 \end{pmatrix}$$

It is seen from Fig. 1.5 that both cycles (as the name implies) effect a *cyclical* interchange on a subset of $\{1, \ldots, 8\}$; these subsets form a *partition* of $\{1, \ldots, 8\} = \{2, 5, 7\} \cup \{1, 3, 4, 6, 8\}$. We may use the *cycle notation* to denote cycles: $\tau_1 = (5\ 7\ 2)$, $\tau_2 = (6\ 3\ 4\ 8\ 1)$. The permutation τ is said to be the *product* of the cycles τ_1 and τ_2,

$$\tau = (5\ 7\ 2)(6\ 3\ 4\ 8\ 1) \tag{1.3}$$

As the individual cycles of a product operate on disjoint sets, the order in which the cycles are listed is immaterial, though shorter cycles are usually written before longer ones. Thus $\tau = (6\ 3\ 4\ 8\ 1)(5\ 7\ 2)$. The entries of a cycle in the cycle notation may be *rotated* [9]; for example, $(3\ 4\ 8\ 1\ 6)$ still refers to the cycle τ_2.

Another example of a permutation in the cycle notation is

$$\rho = (4\ 5)(1\ 2\ 3)(6\ 7\ 8) \tag{1.4}$$

from (1.2) on p. 19; it is the product of three cycles.

Finally, permissible permutations (so-called *derangements*) are now easily recognized as those without a 1–cycle.

Download free eBooks at bookboon.com

Types

The permutation τ in (1.3) is the product of two cycles, τ_1 and τ_2, of length 3 and 5, respectively. Therefore, τ is said to be of *type* $[3^1 5^1]$.[11] π in (1.1) is another permutation of the same type, since

$$\pi = (3\ 1\ 2)(7\ 8\ 4\ 5\ 6) \tag{1.5}$$

On the other hand, ρ in (1.4) is seen to be of type $[2^1 3^2]$.

We note in passing that each type corresponds to a *partition* of the number of elements permuted. A partition of a positive whole number is its representation as the sum of some positive whole numbers. For example, the above types define the partitions $8 = 3 + 5$ and $8 = 2 + 3 + 3$.

Types in our context become significant by the following

> **Observation**. Column–to–row transformations *of the same type* give rise to most general patterned symbolic matrices which are essentially the same in that they can be transformed into each other by appropriate row–to–row and column–to–column rearrangements.

We won't prove this result here but illustrate it by an example. To determine the most general symbolic matrix for τ from that of π, proceed as follows.

1. Write the permutations π and τ in cycle notation (as in (1.5) and (1.3)) and place them above each other as shown below.

$$\begin{array}{rcccccccc} \pi = & (3 & 1 & 2) & (7 & 8 & 4 & 5 & 6) \\ & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\ \tau = & (5 & 7 & 2) & (6 & 3 & 4 & 8 & 1) \end{array}$$

   Shorter cycles should precede longer ones.

2. Read off the rearrangement as

$$\begin{pmatrix} 3 & 1 & 2 & 7 & 8 & 4 & 5 & 6 \\ 5 & 7 & 2 & 6 & 3 & 4 & 8 & 1 \end{pmatrix}$$

   or, written in the usual way, as

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 7 & 2 & 5 & 4 & 8 & 1 & 6 & 3 \end{pmatrix} \tag{1.6}$$

3. Produce the most general patterned symbolic matrix for π by

```
?- var_matrix(8,_M), list_permute([2,3,1,5,6,7,8,4],_M,_P),
   transpose(_P,_M), write_matrix(_M).
[_G868, _G871, _G868, _G877, _G877, _G877, _G877, _G877]
[_G868, _G868, _G871, _G877, _G877, _G877, _G877, _G877]
[_G871, _G868, _G868, _G877, _G877, _G877, _G877, _G877]
[_G877, _G877, _G877, _G958, _G961, _G964, _G961, _G958]
[_G877, _G877, _G877, _G958, _G958, _G961, _G964, _G961]
[_G877, _G877, _G877, _G961, _G958, _G958, _G961, _G964]
[_G877, _G877, _G877, _G964, _G961, _G958, _G958, _G961]
[_G877, _G877, _G877, _G961, _G964, _G961, _G958, _G958]
```

[11] In this notation for types (see [3]), the superscripts stand for the number of times cycles of a particular length occur. The square brackets have nothing to do with Prolog's list notation.

Download free eBooks at bookboon.com

Rename the variables as necessary to see that the above is $\mathbf{M}_1$ (p. 19).

4. Rearrange the *columns* of $\mathbf{M}_1$ according to (1.6) to get

$$\begin{bmatrix} X_6 & X_1 & X_6 & X_6 & X_3 & X_6 & X_3 & X_6 \\ X_6 & X_3 & X_6 & X_6 & X_1 & X_6 & X_3 & X_6 \\ X_6 & X_3 & X_6 & X_6 & X_3 & X_6 & X_1 & X_6 \\ X_2 & X_6 & X_4 & X_4 & X_6 & X_5 & X_6 & X_5 \\ X_5 & X_6 & X_5 & X_4 & X_6 & X_2 & X_6 & X_4 \\ X_4 & X_6 & X_2 & X_5 & X_6 & X_5 & X_6 & X_4 \\ X_4 & X_6 & X_5 & X_2 & X_6 & X_4 & X_6 & X_5 \\ X_5 & X_6 & X_4 & X_5 & X_6 & X_4 & X_6 & X_2 \end{bmatrix}$$

5. Now, using (1.6) again, rearrange the *rows* of the matrix from the previous step.

gaiteye®
Challenge the way we run
EXPERIENCE THE POWER OF FULL ENGAGEMENT...
RUN FASTER.
RUN LONGER..
RUN EASIER...
READ MORE & PRE-ORDER TODAY
WWW.GAITEYE.COM
Download free eBooks at bookboon.com
Click on the ad to read more

$$\begin{bmatrix} X_4 & X_6 & X_2 & X_5 & X_6 & X_5 & X_6 & X_4 \\ X_6 & X_3 & X_6 & X_6 & X_1 & X_6 & X_3 & X_6 \\ X_5 & X_6 & X_4 & X_5 & X_6 & X_4 & X_6 & X_2 \\ X_2 & X_6 & X_4 & X_4 & X_6 & X_5 & X_6 & X_5 \\ X_6 & X_3 & X_6 & X_6 & X_3 & X_6 & X_1 & X_6 \\ X_4 & X_6 & X_5 & X_2 & X_6 & X_4 & X_6 & X_5 \\ X_6 & X_1 & X_6 & X_6 & X_3 & X_6 & X_3 & X_6 \\ X_5 & X_6 & X_5 & X_4 & X_6 & X_2 & X_6 & X_4 \end{bmatrix}$$

This is the most general patterned symbolic matrix for τ as is confirmed by the query below.

```
?- var_matrix(8,_M), list_permute([6,5,4,8,7,3,2,1],_M,_P),
   transpose(_P,_M), write_matrix(_M).
[_G868, _G871, _G874, _G877, _G871, _G877, _G871, _G868]
[_G871, _G898, _G871, _G871, _G907, _G871, _G898, _G871]
[_G877, _G871, _G868, _G877, _G871, _G868, _G871, _G874]
[_G874, _G871, _G868, _G868, _G871, _G877, _G871, _G877]
[_G871, _G898, _G871, _G871, _G898, _G871, _G907, _G871]
[_G868, _G871, _G877, _G874, _G871, _G868, _G871, _G877]
[_G871, _G907, _G871, _G871, _G898, _G871, _G898, _G871]
[_G877, _G871, _G877, _G868, _G871, _G874, _G871, _G868]
```

Row–to–row and column–to–column rearrangements obviously retain the total of a numerical matrix. Therefore, most general patterned symbolic matrices belonging to permutations of the same type will evaluate to the same maximum total. This confirms the last of the three results announced in Sect. 1.5.1. The other two are more straightforward. Admissibility (i.e. not having any 1–cycle) is clearly a type-property. Finally, a matrix with distinct rows will be transformed to a such by a row–to–row or column–to–column rearrangement. Therefore, row-distinctness is also a type-property.

1.5.3 Generating Representative Permutations

Generating Permutation Types

The following algorithm, which is from [3, p. 440], is for obtaining all partitions of a number. It will serve as a basis for generating all permutation types for a given problem size. (As mentioned earlier, there is a one–to–one correspondence between partitions of a number and permutation types.)

> The following rule is the basis for a method of listing all partitions of n in lexicographic order.[12] The first partition is $[n]$. Suppose the current partition λ has parts $\lambda_1 \geqslant \lambda_2 \geqslant \ldots \geqslant \lambda_r$. Then the next partition is found as follows:
>
> (i) if $\lambda_r \neq 1$, then the parts of the next partition are $\lambda_1, \lambda_2, \ldots, \lambda_{r-1}$, $\lambda_r - 1, 1$;

[12]The following is an appropriate ordering. For two partitions of n, $p = [1^{\alpha_1} 2^{\alpha_2} \ldots n^{\alpha_n}]$ and $r = [1^{\beta_1} 2^{\beta_2} \ldots n^{\beta_n}]$, we say that p *comes before* r (denoted by $p \prec_n r$) if for some $k \in \{1, \ldots, n\}$, $\alpha_k > \beta_k$ and $\alpha_i = \beta_i$ for all $i \in \{k+1, \ldots, n\}$. For example, $[1^1 3^1 4^1] \prec_8 [1^4 4^1]$ since, more explicitly, $[1^1 2^0 3^1 \; 4^1 5^0 6^0 7^0 8^0 \;] \prec_8 [1^4 2^0 3^0 \; 4^1 5^0 6^0 7^0 8^0 \;]$. (Longest possible identical tail sections are shaded.) In the ascending chain of successors produced by the algorithm, *every* partition of n appears since $\prec_n$ is a *total* ordering on the partitions of n.

Download free eBooks at bookboon.com

Current Partition	*(1)* ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ⊗ ⊗ ⊗ ⊗ $[1^2 2^1 4^2 5^2]$	*(3)* ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ⊗ ⊗ ⊗ ⊗ ⊗ ⊗ ⊗ ⊗ $[1^4 4^2 5^2]$	*(5)* ○ ⊗ ⊗ $[2^1 3^2 4^1 5^2]$
Next Partition	*(2)* ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ⊗ ⊗ ⊗ ⊗ $[1^4 4^2 5^2]$	*(4)* ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ○ ⊗ ⊗ ⊗ ⊗ ⊗ ⊗ ⊗ ⊗ $[2^1 3^2 4^1 5^2]$	*(6)* ○ ⊗ ⊗ $[1^2 3^2 4^1 5^2]$
Step Used	(ii)	(ii)	(i)

Table 1.2: A Ferrers Diagram

(ii) if $\lambda_r = \lambda_{r-1} = \cdots = \lambda_{r-s+1} = 1$ but $\lambda_{r-s} = x \neq 1$, then the parts of the next partition are obtained by replacing $\lambda_{r-s}, \lambda_{r-s+1}, \ldots, \lambda_r$ by $x-1, x-1, x-1, \ldots, x-1, y$, where $1 \leqslant y \leqslant x-1$ and the number of parts $x-1$ is chosen so that the result is a partition of n.

To make the recursive step of this algorithm more accessible, we show in Table 1.2 some typical instances for generating partitions of $n = 22$. *Ferrers Diagrams* ([3]) are used in Table 1.2 to illustrate partitions. Tokens involved in the recursive step are marked ($\times$).

We paraphrase the algorithm in plain English as it may look rather cryptic at first sight. We lay out n tokens to represent the current partition as a Ferrers diagram. The initial pattern will be just a single row of n tokens, denoting the partition $[n]$. All subsequent diagrams will have several rows and (as a rule) longer rows are placed above shorter ones. To decide which of the recursive steps (i) or (ii) applies, we inspect the bottom row. If it contains more than one token, we then remove its last (i.e. rightmost) token and start a new row by placing it below what was hitherto the bottom row. This completes step (i). On the other hand, if the bottom row consists of a single token, we then scan the diagram from bottom to top. There are now two possibilities. We may find that all rows are single-token rows in which case we have found the last partition, $[1^n]$, and stop.

Download free eBooks at bookboon.com

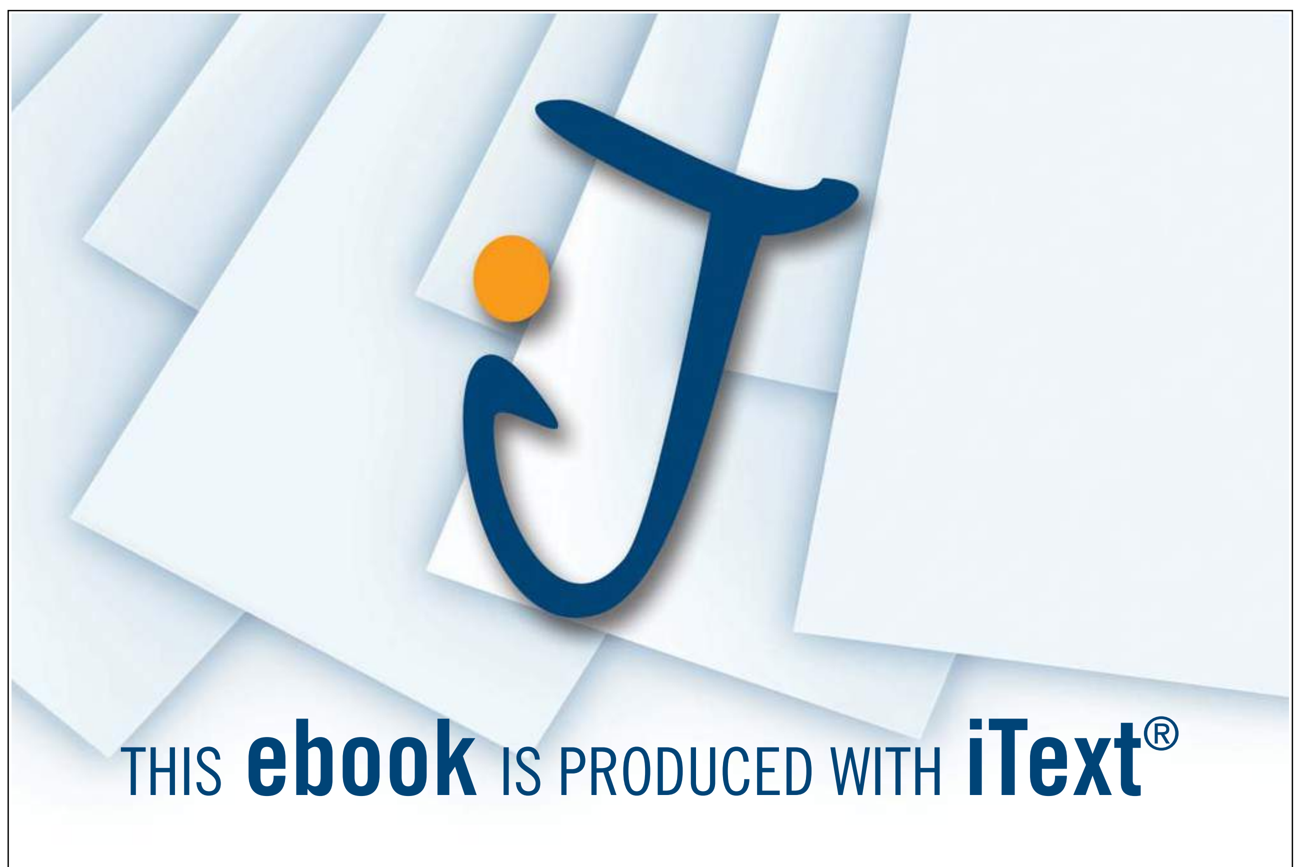
(This has been omitted in the algorithm.) The other possibility is that there is a row containing more than one token. In this case, we remove from the diagram all single-token rows as well as the bottom non-single-token row which has $x(\geqslant 2)$ tokens, say. (These tokens have been marked in Table 1.2, parts (1) and (3).) The tokens thus removed are now used to build up as many new rows of length $x-1$ as possible; we place them below the other (undisturbed) tokens. All the remaining tokens, less than $x-1$, if any, are placed below all the other tokens. This completes step (ii).

Partitions will be represented in our Prolog implementation by lists of pairs; for example, *[(2,1), (3,2), (4,1), (5,2)]* stands for $[2^1 3^2 4^1 5^2]$.

As a first step towards implementing a type *generator*, we define *next_partition(+Current,-Next)* which for a *Current* partition returns the *Next* partition; for example,

```
?- next_partition([(2,1), (3,2), (4,1), (5,2)], Next).
Next = [ (1, 2),  (3, 2),  (4, 1),  (5, 2)]
```

In (P-1.2) we define those three clauses of *next_partition/2* which are typified by the cases in Table 1.2; the definition of the remaining clauses is asked for in Exercise 1.10.



Download free eBooks at bookboon.com

Click on the ad to read more

Current Partition	$[2^3 4^1 6^2]$	...	$[4^3 5^2]$	...
Next Partition	...	$[1^1 3^1 6^3]$	...	$[1^5 2^3 3^1 4^2]$
Step Used	...	...	...	...

Current Partition	...	$[1^3 5^1 7^2]$	...
Next Partition	$[2^1 4^2 6^2]$	...	$[3^3 4^2 5^1]$
Step Used	...	...	...

Table 1.3: Suggested Examples for Exercise 1.10

Prolog Code P-1.2: *Three clauses of the predicate* `next_partition/2`

```
1  next_partition([(1,Alpha),(2,1)|T],       % Cases (1)-(2) in Table 1.2
2                 [(1,NewAlpha)|T]) :-       %
3     NewAlpha is Alpha + 2.                 %

4  next_partition([(1,Alpha1),(L,AlphaL)|T], % Cases (3)-(4) in Table 1.2
5                 [(Rest,1),                 %
6                  (NewL,Ratio),             %
7                  (L,NewAlphaL)|T]) :-      %
8     L > 2,                                 %
9     AlphaL > 1,                            %
10    NewL is L - 1,                         %
11    Rest is (Alpha1 + L) mod NewL,         %
12    Rest > 0,                              %
13    Ratio is (Alpha1 + L) // NewL,         %
14    NewAlphaL is AlphaL - 1.               %

15 next_partition([(2,1)|T],[(1,2)|T]).      % Cases (5)-(6) in Table 1.2
```

Exercise 1.10. The complete definition of *next_partition/2* comprises ten clauses three of which have been defined already. Typical examples covered by each of the remaining seven clauses are partially shown in Table 1.3. Complete Table 1.3 and then define the missing clauses of *next_partition/2*. (It may be helpful to devise the corresponding Ferrers diagrams by using coins.) ■

The predicate *next_partition/2* returns for a given partition its *successor*. We want, however, a *generator* (also called *enumerator*) of partitions, i.e. a predicate which on backtracking will eventually return *all* partitions. The more general question is as follows: How do we 'convert' a successor predicate into a generator? The key to answering this question is by recognizing that this type of problem has been met before. In Exercise 4.6, [9, p. 134], the following definition of *int(+N,?NextN)* was considered,

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

This definition can be used as a *template* for defining another generator: replace *succ* and *int* respectively by *next_partition* and *part* thus giving,

Download free eBooks at bookboon.com

```
part(P, P).
part(Last, Next) :- next_partition(Last, New), part(New, Next).
```

This will result in an acceptable solution,

```
?- part([(1,2),(2,1),(4,2),(5,2)], P).
P = [ (1, 2),  (2, 1),  (4, 2),  (5, 2)] ;
P = [ (1, 4),  (4, 2),  (5, 2)] ;
P = [ (2, 1),  (3, 2),  (4, 1),  (5, 2)] ;
...
```

A better idea still is to write a *higher order* predicate, *generator/3*, say, to accomplish the same task for *any* successor predicate. We then have, for example,

```
?- generator(next_partition,[(1,2),(2,1),(4,2),(5,2)], P).
P = [ (1, 2),  (2, 1),  (4, 2),  (5, 2)] ;
P = [ (1, 4),  (4, 2),  (5, 2)] ;
P = [ (2, 1),  (3, 2),  (4, 1),  (5, 2)] ;
...
```

and

```
?- generator(succ,7,I).
I = 7 ;
I = 8 ;
I = 9 ;
...
```

We define *generator(+Pred,+Init,?Element)* in (P-1.3) by

Prolog Code P-1.3: *Definition of generator/3*

```
1 generator(Pred,From,Element) :-
2   retractall(temp(_,_)),
3   assert(temp(First,First)),
4   assert(temp(Last,E) :- (call(Pred,Last,New), temp(New,E))),
5   temp(From,Element).
```

(P-1.3) shows that

- The temporary generator to be defined in the database is named *temp/2*. Possible earlier definitions are removed first.
- Following our template, two clauses of *temp/2* are written to the database. For instance, after running the above example, the database may be inspected thus

  ```
  ?- listing(temp).
  temp(A, A).
  temp(A, B) :- call(succ, A, C), temp(C, B).
  ```

  As the predicate name is open at this stage, *call/3* is used to invoke the predicate in *Pred*. (See inset.)
- Finally, *temp/2*, just written to the database, is invoked and backtracking is used to produce the sequence.

Download free eBooks at bookboon.com

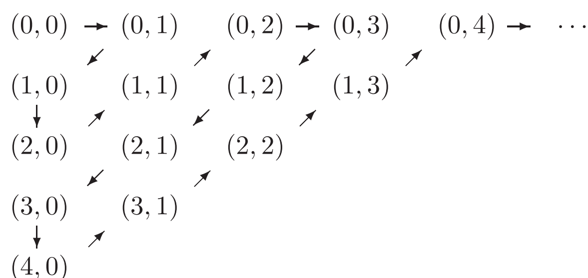


Figure 1.6: Enumeration Scheme for $\{(m, n) : m, n = 0, 1, 2, \ldots\}$. (See Exercise 1.12.)

Built-in Predicate: *call/n, n = 1, 2, 3, ...*

call(+Goal) invokes *Goal*. Combine *call/1* with *=../2*, the built in predicate *univ* ([9] or [33]), if the arity of the predicate in *Goal* is known at run time only. Example:

```
?- _Functor = append, _Args = [[1,2],[3],L],
   Goal =.. [_Functor|_Args], call(Goal).
L = [1, 2, 3]
Goal = append([1, 2], [3], [1, 2, 3])
```

Use *call(+Predicate, +Arg1, +Arg2, ...)* to invoke a *Predicate* whose arity is known at compile time. Examples:

```
?- Pred = append, call(Pred,[1,2],[3],L).
Pred = append
L = [1, 2, 3]
?- Pred = append([1,2]), call(Pred,[3],L).
Pred = append([1, 2])
L = [1, 2, 3]
```

call/n is a *higher order* predicate.

Exercise 1.11. Define a predicate *next_int(+Upper,+I,-NextI)* for unifying *NextI* with the value of *I* incremented by 1. The predicate should fail if *Upper* does not exceed *I*. Use *next_int/3* in conjunction with *generator/3* to generate all integers between 3 and 9. ■

Exercise 1.12. Fig. 1.6 indicates an enumeration scheme for all pairs of non-negative integers (the *Cartesian* product). Define *next_pair/2* for returning the *successor* of any given pair. Then use *next_pair/2* in conjunction with *generator/3* for defining an enumerator for the said Cartesian product. ■

Exercise 1.13. (*An improved generator*) The predicate *pairs/1*, defined by

```
pairs((I,J)) :- int(0,Sum), between(0,Sum,I), J is Sum - I.
```

Download free eBooks at bookboon.com

enumerates the pairs of non-negative integers as shown in Fig. 1.7.[13] It will return on backtracking all pairs starting from $(0, 0)$.

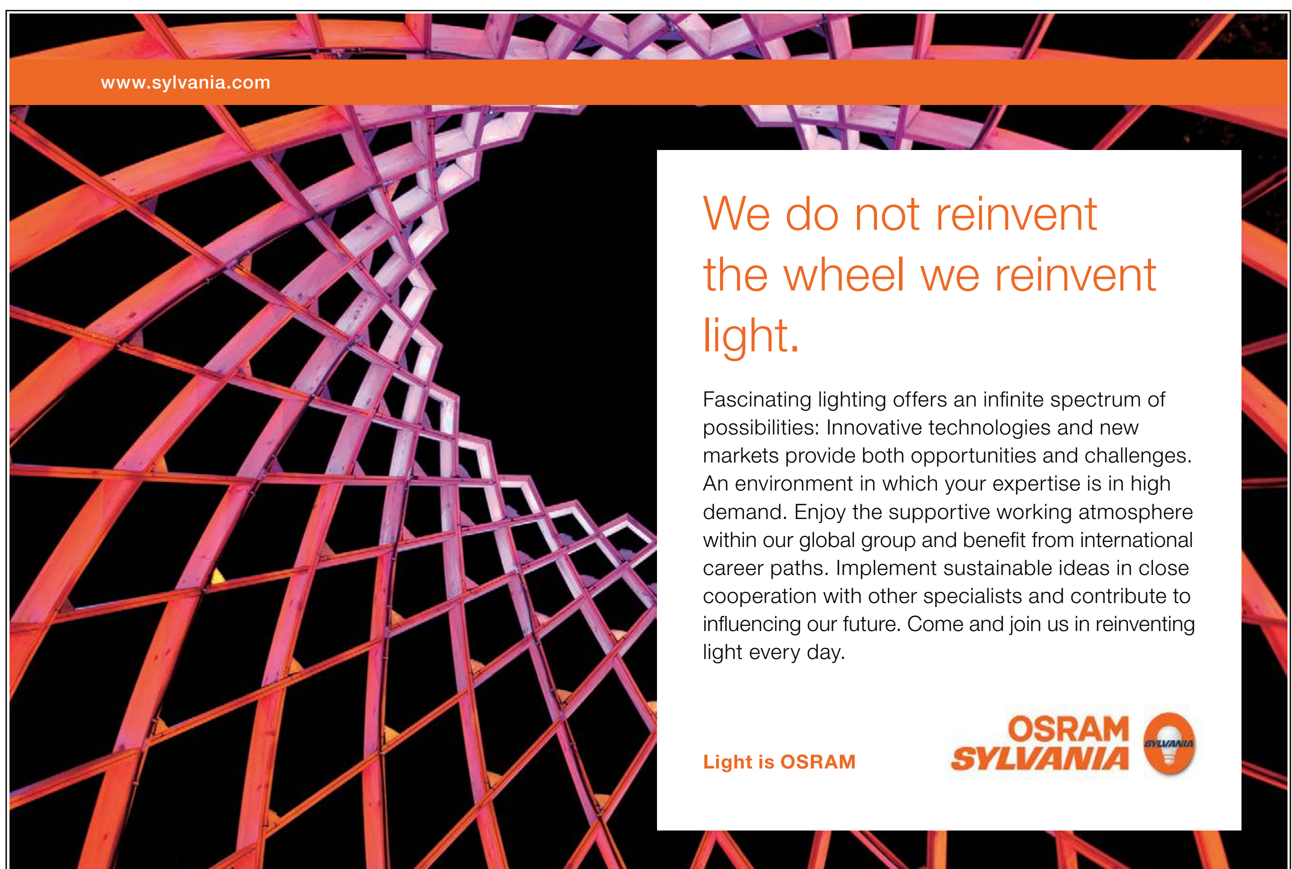
```
?- pairs(P).
P = 0, 0 ;
P = 0, 1 ;
P = 1, 0 ;
P = 0, 2 ;
P = 1, 1 ;
...
```

An *alternative* implementation of `pairs/1` may conceivably be obtained by replacing in its definition the predicates `int/2` and `between/3` by their respective definitions using `generator/3`:

```
pairs_alt((I,J)) :- generator(succ,0,Sum),
                    generator(next_int(Sum),0,I),
                    J is Sum - I.
```

Testing will reveal, however, that this implementation is flawed. The problem is due to the use by `generator/3` of the same name `temp` for predicates written to the database.

[13]The built-in predicate `between/3` is described in [9, p. 41].



Download free eBooks at bookboon.com
Click on the ad to read more

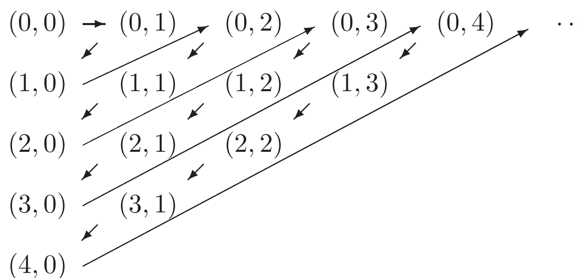


Figure 1.7: Enumeration Scheme for $\{(m, n) : m, n = 0, 1, 2, \ldots\}$. (See Exercise 1.13.)

(The call to *generator(next_int(Sum),0,I)* will interfere with that of *generator(succ,0,Sum)*.) The problem could be avoided though if *generator/3* created temporary predicates with a *different* and *unique* name every time it is invoked.

Define such an improved version of *generator/3*.

Hint. It is suggested that the temporary predicates be named *temp_0*, *temp_1*, etc. You should use the built-in predicate *current_predicate/2* (described in the SWI manual [33]) for finding out whether a proposed new predicate name is available. Use *concat_atom/2* [9, p. 126] for constructing new predicate names. ■

Admissible Representative Permutations

How many permutation types will have to be considered for the original 8×8 problem? This is easily found out by a query,

```
?- bagof(_P,generator(next_partition,[(8,1)],_P),_Ps),
   length(_Ps,NTypes).
NTypes = 22
```

The number 22 is further reduced by concentrating on *admissible* permutations, i.e. on those without a 1–cycle; the types of these we obtain by[14]

```
?- bagof(_P,_I^_A^_T^(generator(next_partition,[(8,1)],_P),
   _P = [(_I,_A)|_T], _I > 1), Ps).
Ps = [[(8, 1)], [(2, 1),(6, 1)], [(3, 1),(5, 1)], [(4, 2)],
      [(2, 2),(4, 1)], [(2, 1),(3, 2)], [(2, 4)]]
```

We therefore have to consider here a mere 7 types. (Contrast this with the 14,833 admissible permutations considered earlier!) All we have to do now is to create for each admissible type a representative permutation.

Suppose we want to construct a representative permutation for the type $[2^1 3^3 5^1]$, a partition of 16. An example permutation of this type in the cycle notation is obtained by simply grouping the elements of $\{1, \ldots, 16\}$ according to the length of the cycles needed:

$$(1\ \ 2)(3\ \ 4\ \ 5)(6\ \ 7\ \ 8)(9\ \ 10\ \ 11)(12\ \ 13\ \ 14\ \ 15\ \ 16) \tag{1.7}$$

[14]This query gives rise to *ad_partition(+N,?P)*, a predicate for generating (and testing) admissible partitions of *N*:

```
ad_partition(N,[(I,A)|T]) :- generator(next_partition,[(N,1)],[(I,A)|T]),
                             I > 1.
```

Download free eBooks at bookboon.com

Using the two-line notation, we rewrite this as

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\ 2 & 1 & 4 & 5 & 3 & 7 & 8 & 6 & 10 & 11 & 9 & 13 & 14 & 15 & 16 & 12 \end{pmatrix} \quad (1.8)$$

which then in the Prolog implementation will be denoted by

[2,1,4,5,3,7,8,6,10,11,9,13,14,15,16,12] (1.9)

The Prolog implementation of (1.7)–(1.9) is in three steps:

(a) A predicate *split(+N,+Type,-S)* is used for partitioning $[1, \ldots, 16]$ into a list of sublists *S* according to *Type*:[15]

```
?- split(16,[(2,1),(3,3),(5,1)],_S), write_term(_S,[]).
[[1,2], [3,4,5], [6,7,8], [9,10,11], [12,13,14,15,16]]
```

split/3 is defined below in terms of an auxiliary predicate *split/4* which itself uses the accumulator technique.

```
split(N,Type,S) :- from_to(1,N,L), split(L,Type,[],S).
```

Exercise 1.14. Define *split/4*. (Some suggested hand computations are shown in Fig. 1.8, p. 45.) ■

(b) *maplist/3* is applied to rotate each sublist in the above list–of–lists.[16]

```
?- split(16,[(2,1),(3,3),(5,1)],_S), maplist(rotate,_S,_R),
   write_term(_R,[]).
[[2,1], [4,5,3], [7,8,6], [10,11,9], [13,14,15,16,12]]
```

(c) Finally, *flatten/2* is used to obtain the list in (1.9).

(a)–(c) give rise to *rep_perm(+N,+Type,-Perm)*, a predicate for finding a representative permutation of a given type.

```
rep_perm(N,Type,Perm) :- split(N,Type,S),
                         maplist(rotate,S,R),
                         flatten(R,Perm).
```

1.5.4 Finishing Touches

Based on the ideas in Sect. 1.5.3, we are now in a position to define a new version of the predicate *square/5*, defined in (P-1.1); the new definition is shown in (P-1.4). Now the queries from Sect. 1.4.10 may be completed as before and with a much reduced computing time. For example, for a 14×14 board we find by a near instantaneous response that the maximum total is 4900. (The earlier version won't solve this problem due to memory shortage and excessive computing time.)

[15] The first argument of *split/3* is redundant as it can be computed from *Type*. Not having to recompute it, however, will save computing time.

[16] Prolog implementations of list rotation are discussed in [5], [8] and [9].

Download free eBooks at bookboon.com

Prolog Code P-1.4: *Definition of* `square2/5`

```
square2(Size,M,Total,Freq,Perm) :- var_matrix(Size,M),
                                   ad_partition(Size,Type),
                                   rep_perm(Size,Type,Perm),
                                   list_permute(Perm,M,P),
                                   transpose(P,M),
                                   distinct(M),
                                   eval_matrix(M,Freq),
                                   total(Freq,Total).
```

Deloitte.
Discover the truth at www.deloitte.ca/careers
© Deloitte & Touche LLP and affiliated entities.

Click on the ad to read more

split([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16], [(2,1),(3,3),(5,1)], [], S) ⇝

split([3,4,5,6,7,8,9,10,11,12,13,14,15,16], [(2,0),(3,3),(5,1)], [[1,2]], S) ⇝

split([3,4,5,6,7,8,9,10,11,12,13,14,15,16], [(3,3),(5,1)], [[1,2]], S) ⇝

split([6,7,8,9,10,11,12,13,14,15,16], [(3,2),(5,1)], [[3,4,5], [1,2]], S) ⇝

split([9,10,11,12,13,14,15,16], [(3,1),(5,1)], [[6,7,8], [3,4,5], [1,2]], S) ⇝

split([12,13,14,15,16], [(3,0),(5,1)], [[9,10,11], [6,7,8], [3,4,5], [1,2]], S) ⇝

split([12,13,14,15,16], [(5,1)], [[9,10,11], [6,7,8], [3,4,5], [1,2]], S) ⇝

split([], [(5,0)], [[12,13,14,15,16], [9,10,11], [6,7,8], [3,4,5], [1,2]], S) ⇝

S = [[1,2], [3,4,5], [6,7,8], [9,10,11], [12,13,14,15,16]] ⇝ *success*

Figure 1.8: Suggested Hand Computations for `split/4`

Download free eBooks at bookboon.com