

Resolução e Elaboração de jogos KenKen implementado em Prolog

Diogo Pinto ei11120 and Wilson Oliveira ei11085

FEUP-PLOG, Turma 3MIEIC06, Grupo 67
www.fe.up.pt

Abstract. O objectivo do trabalho é o desenvolvimento de um programa em prolog com a capacidade de resolver problemas do conhecido jogo de puzzle numerico *KenKen*, e também a elaboração de puzzles deste jogo. Para a resolução deste problema será utilizado técnicas de programação com restrições em prolog.

Keywords: KenKen, Prolog, Restrições, Resolução, Geração

1 Introdução

Este trabalho tinha como objectivo a utilização de programação com restrições em prolog, para a resolução de problemas de optimização ou decisão, verificando assim a eficácia deste método de programação em relação a métodos de tentativa e erro. Para a realização deste objectivo decidimos implementar um programa capaz de resolver puzzle numéricos do conhecido jogo KenKen. Neste artigo é possível analisar os métodos utilizados para a resolução do problema em prolog, assim como excertos mais importantes do código implementado.

2 Descrição do problema

A implementação de um programa capaz de resolver puzzles numéricos do típico jogo KenKen, um jogo similar a Sudoku, em que temos um tabuleiro NxN com números diferentes em cada linha e coluna. Este tabuleiro é subdividido em campos, para os quais é indicado uma operação matemática e um resultado. Estes campos impõem restrições no preenchimento do tabuleiro, uma vez que os números utilizados em cada campo, após a aplicação da operação matemática indicada, devem originar o resultado pedido.

Para a resolução deste problema foram utilizados métodos de programação com restrições em prolog, impondo restrições nos números que podem ser utilizados, verificando que não existem números repetidos por linha e coluna, e verificando que as operações matemáticas indicadas se evidenciam.

3 Estruturas de dados

O puzzle que modela um dado problema é abstracionado sobre duas estruturas de dados elementares. Os valores do tabuleiro são guardados numa matriz de comprimento e largura iguais. Esta matriz é preenchida com elementos do tipo

```
cell(FieldID, Value)
```

, onde o primeiro campo guarda uma referencia para a segunda estrutura de dados utilizada, que é uma lista de itens

```
field(FieldID, Operation, Result)
```

. Aqui são mapeados a operação e o resultado a que corresponde uma determinada região no puzzle. Tal traduz-se numa estrutura como a seguinte:

```
testBoard([[cell(1, _), cell(2, _), cell(2, _), cell(3, _)],
[cell(1, _), cell(4, _), cell(5, _), cell(3, _)],
[cell(6, _), cell(4, _), cell(5, _), cell(5, _)],
[cell(6, _), cell(7, _), cell(7, _), cell(8, _)]]).
```

```
testFields([field(1, '-', 1),
```

```

field(2, '/', 2),
field(3, '-', 3),
field(4, '-', 2),
field(5, '*', 24),
field(6, '-', 1),
field(7, '+', 7),
field(8, '=', 2)]).

```

4 Variáveis de Decisão

Para a resolução deste problema são necessárias quatro variáveis de decisão, primeiro o domínio dos números a serem utilizados no tabuleiro. Só podem ser utilizados no tabuleiro números de valor igual ou inferior ao tamanho do tabuleiro, por exemplo, se o tabuleiro for 4x4, só poderão ser utilizados números de um a quatro. A segunda e terceira restrição são similares, deve ser verificado que não existem números repetidos, por coluna e linha. A última restrição deve verificar que as operações matemáticas pedidas e respectivo resultado se verificam. Fazendo a verificação destas quatro restrições estaremos a reduzir eficazmente o domínio de pesquisa de possíveis soluções.

5 Restrições

A implementação da restrição do domínio, é feita criando uma lista com as variáveis de domínio que irão ser preenchidas, e impondo a restrição do valor máximo a ser utilizado (*SupLim*).

```
imposeDomainConstrainInRow([cell(_, Value) | Rs], SupLim) :-
Value in 1..SupLim,      imposeDomainConstrainInRow(Rs, SupLim).
```

A implementação da restrição de números diferentes numa linha é feita através da construção de uma lista com as variáveis de domínio, e utilizando a função `.....`”, para adicionar esta restrição. A restrição para as colunas é feita utilizando o mesmo predicado, mas é utilizado a função *transpose* para inverter a ordem das listas que formam o tabuleiro.

```
imposeRowConstrain([], List) :- all_distinct(List).
```

```
imposeRowConstrain([cell(_, Value) | Rs], List) :-
append([Value], List, NewList),
imposeRowConstrain(Rs, NewList).
```

A implementação da restrição das operações matemáticas é feita construindo uma lista com as células que compoem um campo, e mediante a operação especificada nesse campo, é imposta a restrição.

```
imposeFieldConstrain(Board, [field(FID, Op, Res) | Fs]) :-
getFieldCells(FID, Board, L),
applyOpConstrain(L, Op, Res),
imposeFieldConstrain(Board, Fs).
```

6 Estratégia de Pesquisa

Na resolução de um tabuleiro, após a imposição das restrições é executado o *labelling* com os valores por defeito, para obter a melhor solução possível para o tabuleiro apresentado.

Na geração de um tabuleiro, após a imposição das restrições nas variáveis de domínio, o *labelling* é feito um número aleatório de vezes, com o objectivo de obter sempre tabuleiros diferentes e válidos para o mesmo tamanho pedido.

7 Visualização da solução

Os predicados de visualização são chamados automaticamente após a resolução de um tabuleiro. Os predicados de impressão são bastante simples, fazendo uso da recursividade de prolog e de iteradores para fazer a construção do tabuleiro, e numerar as colunas e linhas, para facilitar a identificação dos campos internos do tabuleiro. Como os campos são aleatórios, fizemos uso de caracteres ascii para salientar a diferença entre campos e ao mesmo tempo tornar o tabuleiro mais apelativo. Também é impressa uma lista com os detalhes dos campos (operação matemática, e resultado final) e fazendo uso da numeração do tabuleiro, a coluna e linha onde se inicia o campo, sendo facilmente identificável as restantes células desse campo.

Predicado responsável por salientar a mudança de campo no tabuleiro, entre células adjacentes na mesma linha.

```
printRow([C1]) :-
  cell(_, Value1) = C1,
  printNumber(Value1),
  write('|\n').

printRow([C1, C2 | Cs]) :-
  cell(FieldID1, Value1) = C1,
  cell(FieldID2, _Value2) = C2,
  FieldID1 = FieldID2,
  printNumber(Value1),
  write('|'),
  printRow([C2 | Cs]).

printRow([C1 | Cs]) :-
  cell(_, Value1) = C1,
  printNumber(Value1),
  write('||'),
  printRow(Cs).
```

Predicado responsável por salientar a mudança de campo no tabuleiro, entre células adjacentes na mesma coluna.

```

printHorizMidBorder([R1 | R1s], [R2 | R2s]) :-
  cell(FieldID1, _) = R1,
  cell(FieldID2, _) = R2,
  FieldID1 = FieldID2,
  write('---='),
  printHorizMidBorder(R1s, R2s).

printHorizMidBorder([_R1 | R1s], [_R2 | R2s]) :-
  write('===='),
  printHorizMidBorder(R1s, R2s).

```

1	3		2		1		4
2	4		3		2		1
3	2		1		4		3
4	1		4		3		2
	1		2		3		4

Field: 1 Operation: - Result: 1 Row: 1 Column: 1
 Field: 2 Operation: / Result: 2 Row: 1 Column: 2
 Field: 3 Operation: - Result: 3 Row: 1 Column: 4
 Field: 4 Operation: - Result: 2 Row: 2 Column: 2
 Field: 5 Operation: * Result: 24 Row: 2 Column: 3
 Field: 6 Operation: - Result: 1 Row: 3 Column: 1
 Field: 7 Operation: + Result: 7 Row: 4 Column: 2
 Field: 8 Operation: = Result: 2 Row: 4 Column: 4

Fig. 1. Picture of a solved board.

8 Resultados

Após a realização de vários testes, com geração e resolução de tabuleiros KenKen, o programa demorou sempre menos de 1 segundo a gerar e resolver o tabuleiro. Foi constatado para além disso que para tabuleiros com lado maior ou igual que dez verificam-se casos em que a obtenção de solução não ocorrera em tempo útil, crescendo exponencialmente a probabilidade de tal acontecer.

Board Size	Creating Time	Solving Time
4	0.010	0.000
6	0.000	0.000
8	0.000	0.000
10	0.010	222.720

Fig. 2. Results of different tests.

9 Conclusões e perspectivas de desenvolvimento

Para terminar, é importante salientar que foi necessário a reestruturação do programa, uma vez que a variada complexidade dos puzzles levava a quebra do sistema. Após análise dos resultados obtidos e diferentes testes, podemos concluir que os métodos de programação com restrições se tornam mais eficazes que métodos de tentativa e erro, uma vez que limitam à priori o nível de possibilidades para a resolução do problema tornando mais eficaz a procura de uma solução válida.

10 Bibliografia

References

1. Sicstus Documentation, <http://sicstus.sics.se/sicstus/docs/latest/html/sicstus.html/>
2. Swi-Prolog Documentation, <http://www.swi-prolog.org/pldoc/refman/>

A kenken.pl

```

1 :- use_module(library(clpfd)).
2 :- use_module(library(lists)).
3 :- use_module(library(random)).
4 :- use_module(library(between)).
5 :- use_module(library(system)).
6
7 testBoard([[cell(1, -), cell(2, -), cell(2, -), cell(3, -
8           )],
9           [cell(1, -), cell(4, -), cell(5, -), cell(3, -)],
10          [cell(6, -), cell(4, -), cell(5, -), cell(5, -)],
11          [cell(6, -), cell(7, -), cell(7, -), cell(8, -)
12          ]]).
13
14 testBoardPrint([[cell(1, 5), cell(2, 5), cell(2, 555),
15                  cell(3, 5)],
16                 [cell(1, 55), cell(4, 5), cell(5, 5), cell(3, 5)
17                 ],
18                 [cell(6, 5), cell(4, 55), cell(5, 5), cell(5, 5)
19                 ],
20                 [cell(6, 5), cell(7, 5), cell(7, 5), cell(8, 55)
21                 ]]).
22
23 testFields([field(1, '-', 1),
24             field(2, '/', 2),
25             field(3, '-', 3),
26             field(4, '-', 2),
27             field(5, '*', 24),
28             field(6, '-', 1),
29             field(7, '+', 7),
30             field(8, '=', 2)]).
31
32 % for testing the domain constrains
33 testBoard2([[cell(1, -), cell(1, -)],
34             [cell(1, -), cell(1, -)]]).
35
36 testFields2([field(1, +, 6)]).
37
38 cell(_FieldID, _Value).
39 field(_FieldID, _Op, _FinalValue).
40
41 solveBoard :- testBoard(Board),
42               testFields(Fields),
43               length(Board, Size),

```



```

38 imposeDomainConstrain(Board, Size),
39 imposeRowConstrain(Board),
40 imposeColumnConstrain(Board),
41 imposeFieldConstrain(Board, Fields),
42 getValsList(Board, List),
43 labeling([], List),
44 printBoard(Board),
45 write('\n\n'),
46 printFieldTable(Board, Fields).
47
48 solveBoard(Board, Fields) :- statistics(runtime, [T0, _])
49 ,
49 length(Board, Size),
50 imposeDomainConstrain(Board, Size),
51 imposeRowConstrain(Board),
52 imposeColumnConstrain(Board),
53 imposeFieldConstrain(Board, Fields),
54 getValsList(Board, List),
55 labeling([], List),
56 statistics(runtime, [T1, _]),
57 printBoard(Board),
58 write('\n\n'),
59 printFieldTable(Board, Fields),
60 T is T1 - T0,
61 format('~n~nSolving took ~3d~sec.~n', [T]),
62 fd_statistics.
63
64
65 createBoard(Size) :- statistics(runtime, [T0, _]),
66 length(Board, Size),
67 now(Now),
68 setrand(Now),
69 initBoard(Size, Board),
70 imposeDomainConstrain(Board, Size),
71 imposeRowConstrain(Board),
72 imposeColumnConstrain(Board),
73 generateRandomVals(Board, Size),
74 generateFields(Board, Fields),
75 statistics(runtime, [T1, _]),
76 removeBoardFilling(Board, EmptyBoard),
77 printBoard(EmptyBoard),
78 write('\n\n'),
79 printFieldTable(EmptyBoard, Fields),
80 solveBoard(EmptyBoard, Fields),
81 T is T1 - T0,

```

```

82 format('Creating took ~3d~sec~n', [T]).
83
84
85
86 /*
      *****
87  *
88  *  Generations
89  *
90  *****
      */
91
92
93 removeBoardFilling([], []).
94 removeBoardFilling([B | Bs], [EB | EBs]) :-
95     removeBoardFillingInRow(B, EB),
96     removeBoardFilling(Bs, EBs).
97
98 removeBoardFillingInRow([], []).
99 removeBoardFillingInRow([cell(FID, _Val) | Rs], [cell(
100     FID, _) | ERs]) :- removeBoardFillingInRow(Rs, ERs).
101
102
103 initBoard(_Size, []).
104 initBoard(Size, [B | Bs]) :- length(B, Size),
105     initBoardRow(B),
106     initBoard(Size, Bs).
107
108 initBoardRow([]).
109 initBoardRow([cell(_FieldID, _Val) | Rs]) :-
110     initBoardRow(Rs).
111
112
113 generateRandomVals(Board, Size) :- getValsList(Board, L)
114     ,
115     random(1, Size, X),
116     retractall(val(X)),
117     random(1, 3, Dir),
118     asserta(val(X)),
119     genBoard(Dir, L).
120
121
122 generateRandomVals(Board, _Size) :- getValsList(Board, L)
123     ,
124     labeling([bisect], L).
125
126
127 genBoard(1, L) :- !, labeling([bisect, up], L),

```

```

119   val(X),
120   retract(val(X)),
121   X1 is X - 1,
122   asserta(val(X1)),
123   X1 = 0.
124
125   genBoard(2, L) :- !, labeling([bisect, down], L),
126   val(X),
127   retract(val(X)),
128   X1 is X - 1,
129   asserta(val(X1)),
130   X1 = 0.
131
132   not(X) :- X, !, fail.
133   not(_X).
134
135   generateFields(Board, Fields) :- generateFields(Board,
        Fields, 1).
136
137   generateFields(Board, [F | Fs], FieldIt) :- not(
        isBoardFilled(Board)),
138   iterBoard(Board, F, FieldIt),
139   FieldIt1 is FieldIt + 1,
140   !,
141   generateFields(Board, Fs, FieldIt1).
142
143   generateFields(_Board, [], _FieldIt).
144
145   isBoardFilled([]).
146   isBoardFilled([B | Bs]) :- isBoardFilledIterRow(B),
147   isBoardFilled(Bs).
148
149   isBoardFilledIterRow([]).
150   isBoardFilledIterRow([cell(FID, _) | Rs]) :- nonvar(FID)
        ,
151   isBoardFilledIterRow(Rs).
152
153   iterBoard(Board, F, FieldIt) :- field(FieldIt, _Op, _Res
        ) = F,
154   repeat,
155   random(1, 6, OpIt),
156   length(Board, BoardSize),
157   initField(OpIt, F, BoardSize, FieldSize),
158   getFstAvailCell(Board, Row, Col),
159   getCell(Board, Row, Col, cell(FieldIt, Val)),

```

```

160 FieldSize1 is FieldSize - 1,
161 prepMakeField(Board, Row, Col, F, FieldSize1, Val).
162
163 prepMakeField(Board, Row, Col, F, FieldSize, Val) :-
164     FieldSize > 0,
165     getNextCellPos(Board, Row, Col, NewRow, NewCol),
166     !,
167     makeField(Board, NewRow, NewCol, F, FieldSize, Val).
168
169 prepMakeField(_Board, _Row, _Col, field(_, '=', Val),
170     _FieldSize1, Val).
171
172 initField(1, field(_FieldIt, '+', _Res), BoardSize, Size
173     ) :- random(2, BoardSize, Size).
174 initField(2, field(_FieldIt, '-', _Res), _BoardSize, 2).
175 initField(3, field(_FieldIt, '*', _Res), BoardSize, Size
176     ) :- random(2, BoardSize, Size).
177 initField(4, field(_FieldIt, '/', _Res), _BoardSize, 2).
178 initField(5, field(_FieldIt, '=', _Res), _BoardSize, 1).
179
180 makeField(Board, Row, Col, field(FieldIt, Op, NewAcum),
181     1, Acum) :- getCell(Board, Row, Col, cell(FieldIt,
182     Val)),
183     getNewAcum(Acum, Op, Val, NewAcum), !.
184
185 makeField(Board, Row, Col, field(FieldIt, Op, Res),
186     FieldSize, Acum) :- getCell(Board, Row, Col, cell(
187     FieldIt, Val)),
188     getNewAcum(Acum, Op, Val, NewAcum),
189     getNextCellPos(Board, Row, Col, NewRow, NewCol),
190     getCell(Board, NewRow, NewCol, cell(FID, -)),
191     FID = FieldIt, %se falhar, backtracking para new random
192     FieldSize1 is FieldSize - 1,
193     makeField(Board, NewRow, NewCol, field(FieldIt, Op, Res)
194     , FieldSize1, NewAcum).
195
196 getNextCellPos(Board, Row, Col, NewRow, NewCol) :-
197     getNextCellPos(Board, Row, Col, NewRow, NewCol, 9).
198 getNextCellPos(_Board, _Row, _Col, _NewRow, _NewCol, 0)
199     :- !, fail.
200 getNextCellPos(Board, Row, Col, NewRow, NewCol, _It) :-
201     random(1, 4, X),
202     isCellAvailable(Board, Row, Col, X, NewRow, NewCol).
203 getNextCellPos(Board, Row, Col, NewRow, NewCol, It) :-
204     NewIt is It - 1,

```

```

192 getNextCellPos(Board, Row, Col, NewRow, NewCol, NewIt).
193
194 isCellAvailable(Board, Row, Col, X, NewRow, NewCol) :-
    getTestRow(X, Row, Col, NewRow, NewCol),
195 getCell(Board, NewRow, NewCol, cell(FID, _)),
196 var(FID).
197
198 getTestRow(1, Row, Col, Row, NewCol) :- NewCol is Col +
    1.
199 getTestRow(2, Row, Col, NewRow, Col) :- NewRow is Row +
    1.
200 getTestRow(3, Row, Col, Row, NewCol) :- NewCol is Col -
    1,
201 NewCol > 0.
202
203 getCell([B | _Bs], 1, Col, Cell) :- getCellInRow(B, Col,
    Cell).
204 getCell([_B | Bs], Row, Col, Cell) :- Row1 is Row - 1,
205 getCell(Bs, Row1, Col, Cell).
206
207 getCellInRow([C | _Cs], 1, C).
208 getCellInRow([_C | Cs], Col, Cell) :- Col1 is Col - 1,
209 getCellInRow(Cs, Col1, Cell).
210
211 getNewAcum(Acum, '+', Val, NewAcum) :- NewAcum is Acum +
    Val.
212 getNewAcum(Acum, '-', Val, NewAcum) :- min_member(Min, [
    Acum, Val]),
213 max_member(Max, [Acum, Val]),
214 NewAcum is Max - Min.
215 getNewAcum(Acum, '*', Val, NewAcum) :- NewAcum is Acum *
    Val.
216 getNewAcum(Acum, '/', Val, NewAcum) :- min_member(Min, [
    Acum, Val]),
217 max_member(Max, [Acum, Val]),
218 X is Max mod Min,
219 X = 0,
220 NewAcum is Max // Min.
221 getNewAcum(_Acum, '=', Val, Val).
222
223 getFstAvailCell(Board, RetRow, RetCol) :-
    getFstAvailCell(Board, 1, RetRow, RetCol).
224
225 getFstAvailCell([], _RowIt, _RetRow, _RetCol) :- fail.

```

```

226 getFstAvailCell([B | _Bs], RowIt, RowIt, RetCol) :-
    getFstAvailCellInRow(B, 1, RetCol).
227 getFstAvailCell([_B | Bs], RowIt, RetRow, RetCol) :-
    RowIt1 is RowIt + 1,
228 getFstAvailCell(Bs, RowIt1, RetRow, RetCol).
229
230 getFstAvailCellInRow([], -, -) :- fail.
231 getFstAvailCellInRow([cell(FID, _) | _Rs], ColIt, ColIt)
    :- var(FID).
232 getFstAvailCellInRow([_R | Rs], ColIt, RetCol) :- ColIt1
    is ColIt + 1,
233 getFstAvailCellInRow(Rs, ColIt1, RetCol).
234
235
236
237 /*
    *****
238 *
239 * Getters
240 *
241 *****
    */
242
243 getFieldCells(_, [], []).
244 getFieldCells(FieldID, [B | Bs], RetList) :-
    getFieldCellsInRow(FieldID, B, RetList1),
245 getFieldCells(FieldID, Bs, RetList2),
246 append(RetList1, RetList2, RetList).
247
248 getFieldCellsInRow(_FieldID, [], []).
249 getFieldCellsInRow(FieldID, [cell(FieldID, Val) | Rs],
    RetList) :- getFieldCellsInRow(FieldID, Rs, RetList1)
    ,
250 append([Val], RetList1, RetList).
251
252 getFieldCellsInRow(FieldID, [_R | Rs], RetList) :-
    getFieldCellsInRow(FieldID, Rs, RetList).
253
254 getValsList([], []).
255 getValsList([B | Bs], L) :- getValsListInRow(B, L1),
256 getValsList(Bs, L2),
257 append(L1, L2, L).
258
259 getValsListInRow([], []).

```

```

260   getValsListInRow ([ cell (_, Val) | Rs], L) :-
261       getValsListInRow (Rs, L1),
262       append ([ Val], L1, L).
263
264   /*
265       *
266       * Restrictions
267       *
268       *****/
269
270   imposeDomainConstrain ([], _).
271   imposeDomainConstrain ([B | Bs], SupLim) :-
272       imposeDomainConstrainInRow (B, SupLim),
273       imposeDomainConstrain (Bs, SupLim).
274
275   imposeDomainConstrainInRow ([], _).
276   imposeDomainConstrainInRow ([ cell (_, Value) | Rs], SupLim
277       ) :- Value in 1..SupLim,
278       imposeDomainConstrainInRow (Rs, SupLim).
279
280   imposeRowConstrain ([]).
281   imposeRowConstrain ([B | Bs]) :- imposeRowConstrain (B,
282       []),
283       imposeRowConstrain (Bs).
284
285   imposeRowConstrain ([], List) :- all_distinct (List).
286
287   imposeRowConstrain ([ cell (_, Value) | Rs], List) :-
288       append ([ Value], List, NewList),
289       imposeRowConstrain (Rs, NewList).
290
291   imposeColumnConstrain (Board) :- transpose (Board, TBoard)
292       ,
293       imposeRowConstrain (TBoard).
294
295   imposeFieldConstrain (_Board, []).
296   imposeFieldConstrain (Board, [ field (FID, Op, Res) | Fs])
297       :- getFieldCells (FID, Board, L),
298       applyOpConstrain (L, Op, Res),
299       imposeFieldConstrain (Board, Fs).

```

```

295 applyOpConstrain([], _Op, _Res) :- fail.
296
297 applyOpConstrain(L, '+', Res) :- sum(L, #=, Res).
298
299 applyOpConstrain(L, '-', Res) :- L = [_ , _],
300 maximum(Max, L),
301 minimum(Min, L),
302 Res #= Max - Min.
303
304 applyOpConstrain(L, '/', Res) :- L = [_ , _],
305 maximum(Max, L),
306 minimum(Min, L),
307 Res #= Max / Min.
308
309 applyOpConstrain([L], '=', Res) :- L #= Res.
310
311 applyOpConstrain([L | Ls], '*', Res) :- applyOpConstrain
    (Ls, '*', L, Res).
312
313 applyOpConstrain([], '*', Acum, Res) :- Acum #= Res.
314
315 applyOpConstrain([L | Ls], '*', Acum, Res) :- Acum1 #= L
    * Acum,
316 applyOpConstrain(Ls, '*', Acum1, Res).
317
318
319 /*
    *****
320 *
321 * Print
322 *
323 *****
    */
324
325 printBoard :- testBoardPrint(Board),
326 printBoard(Board).
327
328 printBoard(Board) :- printTopBorder(Board),
329 length(Board, Size),
330 printBoard(Board, Size, 1),
331 printBottomBorder(Board).
332
333 printTopBorder(Board) :- length(Board, Size), asserta(
    size(Size)),

```



```

334 write('---'), write('||'),
335 printHorizTopBorder(Size).
336
337 printBoard([], -, -).
338 printBoard([B1, B2 | Bs], Size, RowIt) :- printNumber(
      RowIt), write('||'),
339 printRow(B1),
340 write('---||'),
341 printHorizMidBorder(B1, B2),
342 RowIt1 is RowIt + 1,
343 printBoard([B2 | Bs], Size, RowIt1).
344
345 printBoard([B1 | Bs], Size, RowIt) :- printNumber(RowIt)
      , write('||'),
346 printRow(B1),
347 RowIt1 is RowIt + 1,
348 printBoard(Bs, Size, RowIt1).
349
350 printRow([C1]) :- cell(_, Value1) = C1,
351 printNumber(Value1),
352 write('||\n').
353
354 printRow([C1, C2 | Cs]) :- cell(FieldID1, Value1) = C1,
355 cell(FieldID2, _Value2) = C2,
356 FieldID1 = FieldID2,
357 printNumber(Value1),
358 write('|'),
359 printRow([C2 | Cs]).
360
361 printRow([C1 | Cs]) :- cell(_, Value1) = C1,
362 printNumber(Value1),
363 write('||'),
364 printRow(Cs).
365
366 printBottomBorder(Board) :- length(Board, Size), write('
      ---||'), printHorizBotBorder(Size), write(' '),
      printBottomNumbers(Size).
367
368 printHorizTopBorder(1) :- write('===||'), write('\n').
369
370 printHorizTopBorder(Size) :- write('===='), Size1 is
      Size - 1, printHorizTopBorder(Size1).
371
372 printHorizBotBorder(1) :- write('===||'), write('\n').
373

```

```

374 | printHorizBotBorder(Size) :- write('===='), Size1 is
    | Size - 1, printHorizBotBorder(Size1).
375 |
376 | printBottomNumbers(Size) :- write('___'),
    | printBottomNumbers(Size, 1).
377 |
378 | printBottomNumbers(Size, Size) :- printNumber(Size).
379 |
380 | printBottomNumbers(Size, Number) :- printNumber(Number),
    | write('_'), Number1 is Number + 1,
    | printBottomNumbers(Size, Number1).
381 |
382 |
383 | printHorizMidBorder([R1], [R2]) :- cell(FieldID1, _) =
    | R1,
384 | cell(FieldID2, _) = R2,
385 | FieldID1 = FieldID2,
386 | write('---||\n').
387 |
388 | printHorizMidBorder([_R1], [_R2]) :- write('==||\n').
389 |
390 | printHorizMidBorder([R1 | R1s], [R2 | R2s]) :- cell(
    | FieldID1, _) = R1,
391 | cell(FieldID2, _) = R2,
392 | FieldID1 = FieldID2,
393 | write('===='),
394 | printHorizMidBorder(R1s, R2s).
395 |
396 | printHorizMidBorder([_R1 | R1s], [_R2 | R2s]) :- write('
    | ===='), printHorizMidBorder(R1s, R2s).
397 |
398 | printNumber(Number) :- var(Number), write('___').
399 | printNumber(Number) :- Number > 99, write(Number).
400 | printNumber(Number) :- Number > 9, write('_'), write(
    | Number).
401 | printNumber(Number) :- write('_'), write(Number), write(
    | '_').
402 |
403 | printFieldTable(_, []).
404 | printFieldTable(Board, [F | Fs]) :- RowIt = 1,
    | printFieldTableAux(Board, F, RowIt), printFieldTable(
    | Board, Fs).
405 |
406 |
407 | printFieldTableAux([], _, _).

```

```

408 | printFieldTableAux([B | _Bs], F, RowIt) :- ColIt = 1,
      | findFieldInCell(B, F, ColIt, Col), field(FID, Op, Val
      | ) = F,
409 | printField(FID, Op, Val, RowIt, Col).
410 |
411 | printFieldTableAux([_B | Bs], F, RowIt) :- RowIt1 is
      | RowIt + 1, printFieldTableAux(Bs, F, RowIt1).
412 |
413 | findFieldInCell([], _, _, _) :- fail.
414 | findFieldInCell([cell(FID, _) | _Bs], field(FID, _, _),
      | ColIt, ColIt).
415 | findFieldInCell([_B | Bs], F, ColIt, Col) :- ColIt1 is
      | ColIt + 1, findFieldInCell(Bs, F, ColIt1, Col).
416 |
417 | printField(FID, Op, Val, Row, Col) :- write('Field:_'),
      | write(FID), write(' '),
418 | write('Operation:_'), write(Op), write(' '),
419 | write('Result:_'), write(Val), write(' '),
420 | write('Row:_'), write(Row), write(' '),
421 | write('Column:_'), write(Col), write(' '),
422 | write('\\n').

```