UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO (CTC)
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA (INE)
CURSO DE CIÊNCIA DA COMPUTAÇÃO
DISCIPLINA INE5416 - PARADIGMAS DE PROGRAMAÇÃO

ESTUDANTE: EDUARDO DANI PEROTTONI (21204003)

TRABALHO PRÁTICO DE PROGRAMAÇÃO FUNCIONAL: RESOLUÇÃO DO KOJUN EM PROLOG

1. INTRODUÇÃO E APRESENTAÇÃO DO PROBLEMA

O Kojun é um jogo de quebra-cabeça japonês parecido com o famoso Sudoku. Nele, o tabuleiro é dividido em regiões formadas por células adjacentes. Suas regras são as seguintes:

- Cada região de n células deve conter os números de 1 a n
- Número de uma célula não pode ser igual ao número das células adjacentes ortogonalmente (células imediatamente acima, abaixo, à esquerda e à direita)
- Se duas células são adjacentes verticalmente na mesma região, o número na célula superior deve ser maior que o número na célula inferior

2. ESTRATÉGIA DE RESOLUÇÃO

Uma vez que o Prolog internamente já resolve combinações a partir de regras por meio de *backtracking*, a estratégia utilizada para a resolução do *puzzle* foi impor regras para o preenchimento de cada uma das células e deixar que o Prolog calculasse o valor mais apropriado para cada uma delas. Logo, para cada célula em branco, determinou-se as regras para ela, de acordo com as regras do Kojun. Com isso, o Prolog conseguiu determinar a combinação para cada uma das instâncias de resolução.

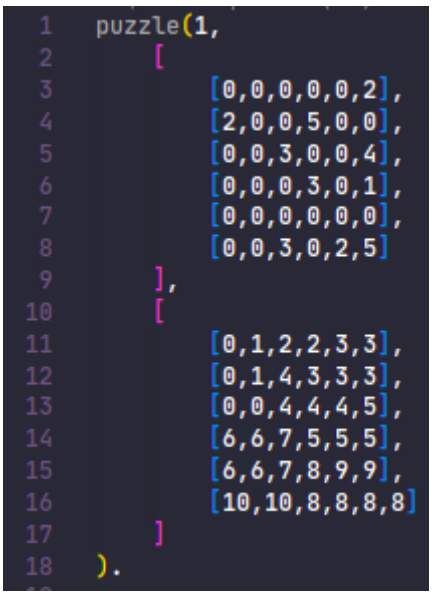
3. ORGANIZAÇÃO DO CÓDIGO

O código foi organizado em dois módulos. Os módulos desenvolvidos foram:

- **puzzles.pl:** Módulo para geração de exemplos de *puzzles* a serem resolvidos.
- **kojun.pl:** Módulo com todos os predicados usados para a resolução do *puzzle*.

4. ENTRADA E SAÍDA

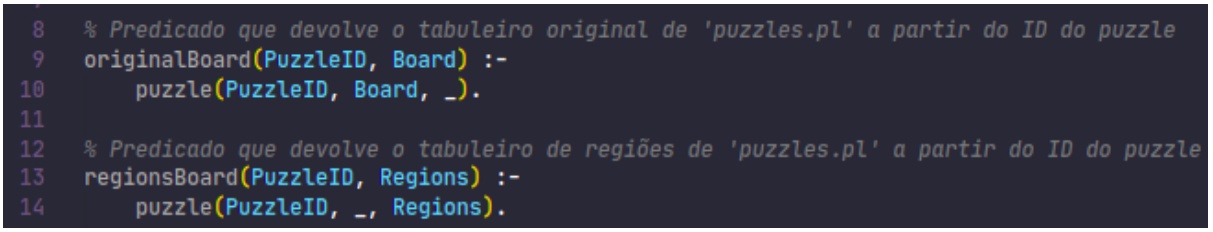
O módulo *puzzles.pl* contém um predicado que gera exemplos de *puzzles* a serem resolvidos a partir de um ID. Cada *puzzles* de exemplo (todos retirados do site de referência do enunciado do trabalho) possui um identificador. A Figura 1 mostra o predicado *puzzle*, formado por um ID e duas listas de listas. A primeira, é o tabuleiro inicial com as dicas e a segunda é o tabuleiro de regiões, onde cada região é identificada por um número.



```
puzzle(1,
    [
        [0,0,0,0,0,2],
        [2,0,0,5,0,0],
        [0,0,3,0,0,4],
        [0,0,0,3,0,1],
        [0,0,0,0,0,0],
        [0,0,3,0,2,5]
    ],
    [
        [0,1,2,2,3,3],
        [0,1,4,3,3,3],
        [0,0,4,4,4,5],
        [6,6,7,5,5,5],
        [6,6,7,8,9,9],
        [10,10,8,8,8,8]
    ]
).
```

Figura 1. Módulo puzzle.pl contendo o predicado puzzle.

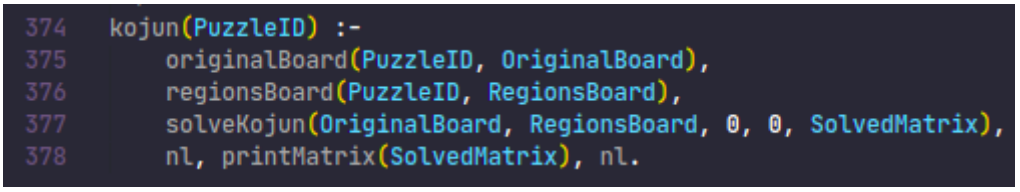
O predicado *puzzle* é utilizado nos predicados *originalPuzzle* e *regionsBoard*, que por sua vez são utilizados no predicado inicial da solução (*kojun*).



```
% Predicado que devolve o tabuleiro original de 'puzzles.pl' a partir do ID do puzzle
originalBoard(PuzzleID, Board) :-
    puzzle(PuzzleID, Board, _).

% Predicado que devolve o tabuleiro de regiões de 'puzzles.pl' a partir do ID do puzzle
regionsBoard(PuzzleID, Regions) :-
    puzzle(PuzzleID, _, Regions).
```

Figura 2. Predicados que definem o tabuleiro original e o tabuleiro de regiões.



```
kojun(PuzzleID) :-
    originalBoard(PuzzleID, OriginalBoard),
    regionsBoard(PuzzleID, RegionsBoard),
    solveKojun(OriginalBoard, RegionsBoard, 0, 0, SolvedMatrix),
    nl, printMatrix(SolvedMatrix), nl.
```

Figura 3. Predicado de entrada da solução. Define os tabuleiros utilizando os predicados da Figura 2 e chama o predicado de resolução do puzzle.

Nota-se que a saída é dada pelo predicado *printMatrix*, que mostra o tabuleiro resolvido para o usuário:

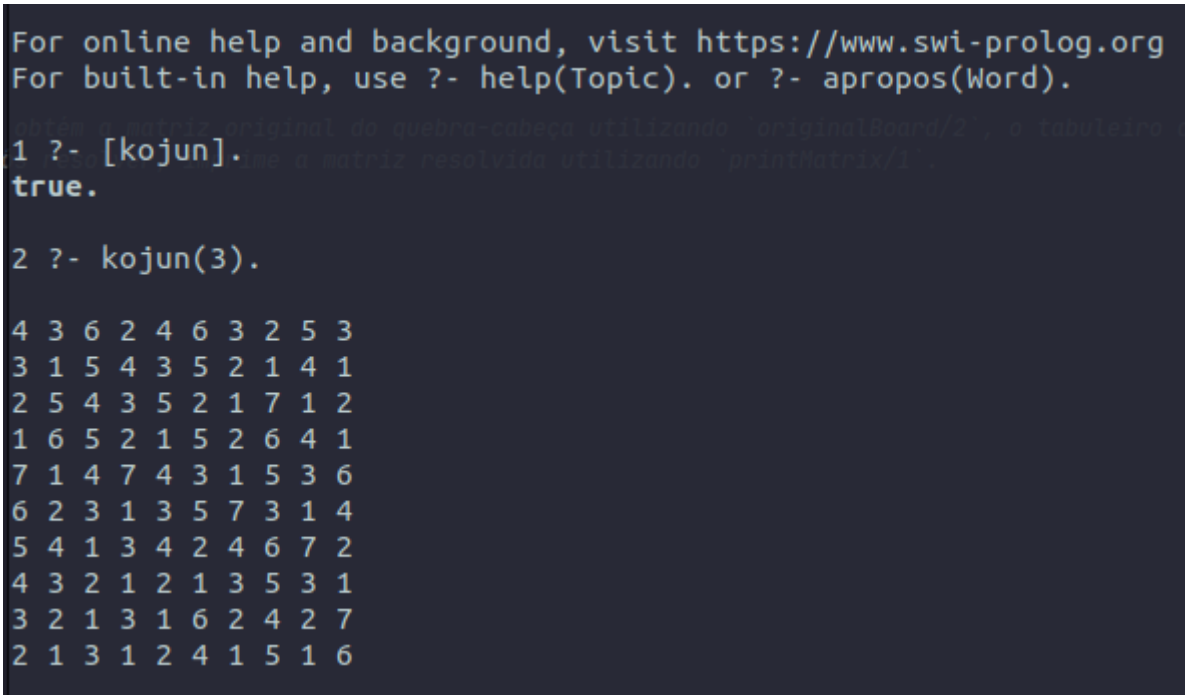


Figura 4. Exemplo de saída. Após carregar o programa no *swipl* e chamar o predicado *kojun* informando o ID do *puzzle* a ser resolvido, o tabuleiro preenchido é mostrado para o usuário.

5. DESENVOLVIMENTO DO TRABALHO

5.1. Modelagem dos elementos do problema

Para modelar o tabuleiro, definiu-se uma estrutura bidimensional (lista de listas). Para modelar as regiões, também se utilizou-se a mesma estrutura, sendo que cada região foi definida com um identificador único sendo um inteiro. Essas estruturas já estão ilustradas na Figura 1.

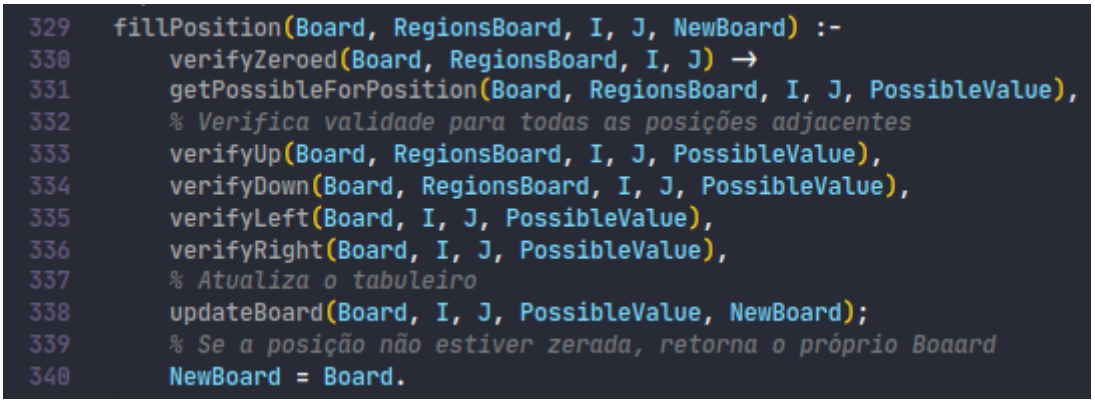
5.2. Predicados iniciais da solução

O predicado inicial, mostrado na Figura 3, gera a instância do *puzzle* a ser resolvido a partir do ID informado e chama o segundo predicado, *solveKojun*. Este, é ilustrado na Figura 5.

```
357  solveKojun(OriginalBoard, RegionsBoard, I, J, SolvedBoard) :-
358      length(OriginalBoard, N),
359      (I ≥ N → SolvedBoard = OriginalBoard; % All positions are filled
360      (J ≥ N → NI is I + 1, NJ is 0, solveKojun(OriginalBoard, RegionsBoard, NI, NJ, SolvedBoard);  % Move to next row
361      (fillPosition(OriginalBoard, RegionsBoard, I, J, UpdatedBoard),
362      NJ is J + 1, % Goes to next col
363      solveKojun(UpdatedBoard, RegionsBoard, I, NJ, SolvedBoard)))).  % Move to next column
```

Figura 5. Predicado solveKojun.

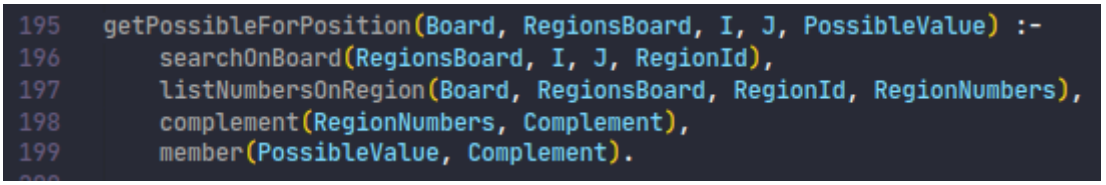
Nota-se que, *solveKojun* apenas chama o predicado *fillPosition* para cada uma das células do tabuleiro. *fillPosition* pode ser considerado o predicado principal da solução. Ele que chama os predicados de verificação e atualização do tabuleiro. *fillPosition* é ilustrado na Figura 6.



```
fillPosition(Board, RegionsBoard, I, J, NewBoard) :-
    verifyZeroed(Board, RegionsBoard, I, J) →
    getPossibleForPosition(Board, RegionsBoard, I, J, PossibleValue),
    % Verifica validade para todas as posições adjacentes
    verifyUp(Board, RegionsBoard, I, J, PossibleValue),
    verifyDown(Board, RegionsBoard, I, J, PossibleValue),
    verifyLeft(Board, I, J, PossibleValue),
    verifyRight(Board, I, J, PossibleValue),
    % Atualiza o tabuleiro
    updateBoard(Board, I, J, PossibleValue, NewBoard);
    % Se a posição não estiver zerada, retorna o próprio Boaard
    NewBoard = Board.
```

Figura 6. Predicado *fillPosition*.

Nota-se que o predicado acima testa se a célula já foi preenchida (se está ou não zerada). Caso esteja preenchida, o próprio tabuleiro é retornado para a posição I, J. Caso contrário, geram-se as possibilidades com o predicado *getPossibleForPosition* e faz-se a verificação das regras do Kojun com os predicados *verifyUp*, *verifyDown*, *verifyLeft* e *verifyRight* para o valor possível retornado. Ou seja, o valor retornado deve obedecer as regras e caso isso não ocorra, um novo valor será testado pelo próprio PROLOG, pois mais de um valor é possível para a maioria das células. A Figura 7 ilustra o predicado *getPossibleForPosition*.



```
getPossibleForPosition(Board, RegionsBoard, I, J, PossibleValue) :-
    searchOnBoard(RegionsBoard, I, J, RegionId),
    listNumbersOnRegion(Board, RegionsBoard, RegionId, RegionNumbers),
    complement(RegionNumbers, Complement),
    member(PossibleValue, Complement).
```

Figura 7. Predicado *getPossibleForPosition*. Dada uma posição I, J, o tabuleiro de regiões e o tabuleiro normal, retorna um possível valor para preenchimento da célula.

6. DIFICULDADES ENCONTRADAS

As principais dificuldades residiram na falta de experiência com PROLOG e com programação lógica. É difícil sair das “amarras” do paradigma imperativo e ir para o declarativo, apenas especificand regras para que a própria linguagem “pense e resolva”. Apesar disso, gerou-se uma implementação funcional, apesar de não ser tão otimizada.