RELATÓRIO DE DESENVOLVIMENTO PRÁTICO -> TRABALHO II

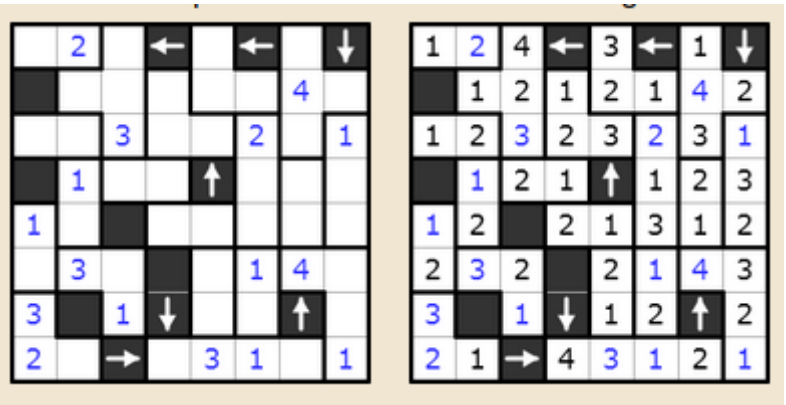
INE5416 - Paradigmas de Programação

Puzzle: Makaro (マカロ)

Alunos: Raquel Behrens, Tainá da Cruz, Vitor Marzarotto

1. Análise do Problema

Makaro (マカロ) é um quebra-cabeça lógico publicado pela primeira vez na revista japonesa de quebra-cabeças Nikoli.



Existem 3 regras para resolver esse puzzle:

1. Inserir um número em cada célula branca do diagrama para que cada região de N células contenha todos os números de 1~N exatamente uma vez.
2. Os mesmos números não devem ser ortogonalmente adjacentes.
3. Uma seta em uma célula preta aponta para a célula adjacente ortogonalmente com o número absolutamente mais alto.

2. Entrada do Usuário

Para resolver uma matriz makaro com o programa é preciso entender como a matriz que representa o tabuleiro é definida no código. Além disso, a entrada da matriz do puzzle é atualizada diretamente no código.

A matriz *makaro* está localizada no arquivo ‘makaro.lisp’. Ela é formada por uma lista de listas, as quais representam as linhas. Cada linha é composta por n células. A célula é uma estrutura criada no arquivo ‘matriz.lisp’, e possui quatro elementos:

- *arrow* (que pode ser T ou NIL - representando se a célula é uma flecha);

- *black* (que pode ser T ou NIL - representando se a célula é um espaço vazio que não pode ser preenchido);
- *region* (que representa a qual região pertence a célula);
- *possibilities* (que representa a lista de possibilides de uma célula).

Se a célula a ser inserida for uma flecha, seu elemento *arrow* terá de ser igual a T, *black* deverá ser NIL, *region* deverá ser 0, e sua lista de possibilidades deve representar para qual lado a flecha está virada. Se a flecha for para a direita, o valor a ser colocado em *possibilities* deve ser 1, se for para baixo, o valor deve ser 2, se for para a esquerda, o valor deve ser 3 e, se for para cima, o valor deve ser 4.

Se for uma célula que está vazia, deve-se deixar sua lista de possibilidades como vazia, *arrow* como NIL, *black* como NIL, e dever-se-á preencher *region* com o identificador da região (número natural).

Se for uma célula com valor já inserido, deve-se preencher *possibilities* com uma lista apenas com seu valor pré-definido, *arrow* deverá ser NIL, *black* deverá ser NIL, e *region* terá o identificador da região (número natural).

Se for uma célula preta (onde não há valores e não se pode inserir valores), *black* deverá ser T, *arrow* deverá ser NIL, *possibilities* deverá ser NIL, e region deverá ser 0.

Para executar o programa, deve-se ter CLISP instalado no computador, compilar o arquivo main com:

- $ clisp -c main.lisp
  Executar com:
- $ clisp main.lisp

3. Estratégia e Solução Aplicada

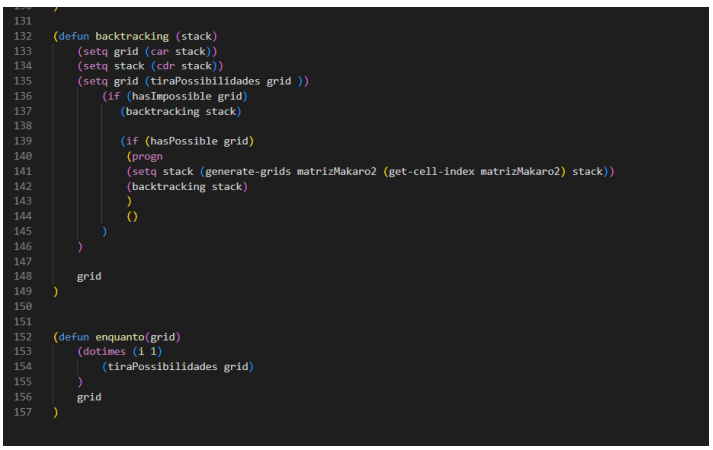
A estratégia aplicada se baseou na técnica de programação de “tentativa e erro” (*backtracking*), utilizando a linguagem Haskell. A função **backtracking** (*Backtracking.hs*) do código funciona analisando o tabuleiro com 2 funções: **hasImpossible,** que retorna verdadeiro quando tabuleiro possui pelo menos 1 casa em que não há nenhum número que possa ser colocado sem quebrar as regras do jogo; e **hasPossible,** que retorna verdadeiro quando tabuleiro possui pelo menos 1 casa que pode receber mais que 1 número). Se nenhuma delas retornar verdadeiro,

significa que o tabuleiro está resolvido, pois não há casas impossíveis e nem com possibilidades, ou seja, todas estão com um único número.

Backtracking recebe uma lista (que vai ser utilizada como pilha) de tabuleiros e retorna um tabuleiro. Ela aplica a função **tiraPossibilidades** sobre o tabuleiro no topo da Pilha, que, a partir das regras do jogo, vai retirando o máximo de possibilidades possível. Após isso, irá checar primeiramente se o tabuleiro fica impossível de ser resolvido (função **hasImpossible** retorna **True**). Nesse caso, ele retira a cabeça da pilha, e chama a si mesma sobre a cauda da pilha. Caso não seja impossível, irá verificar se ainda há possibilidades (função **hasPossible** retorna **True**), e, se houver, ele irá escolher uma das casas com menos possibilidades possível. Então, irá empilhar cópias do tabuleiro na pilha com a casa preenchida com cada possibilidade, e chamará recursivamente a pilha. Se não for possível nem impossível, o tabuleiro está resolvido, então a função backtracking apenas retorna a cabeça da pilha.

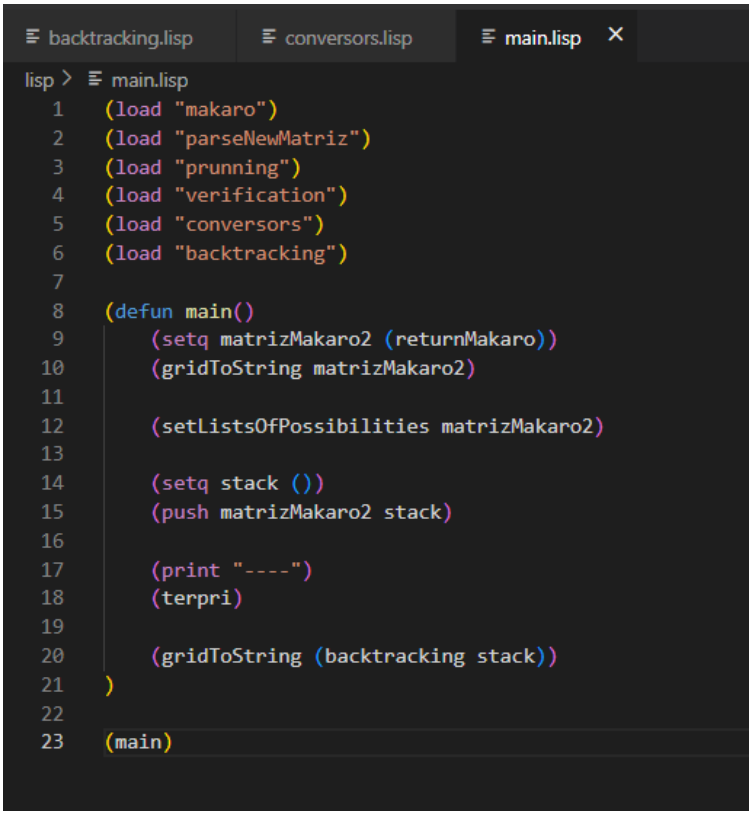
A função **tiraPossibilidadas** (*Backtracking.hs)* primeiramente chama a função **pruning** (*Prunning.hs*), que checa os números que possuem apenas uma possibilidade de cada região, e os retira da lista de possibilidades das células que possuem mais de uma possibilidade daquela região. Após isso, chama a função **verifyOrthogonallyAdjacency** (*Verification.hs*), que poda as listas de possibilidades conforme as regras de adjacência do jogo. A respeito dos módulos, segue abaixo a descrição de cada um:

- *Backtracking* - Agrupa as funções que dizem respeito ao backtracking, a função principal que resolve o problema, a partir da retirada de possibilidades e testes dessas;

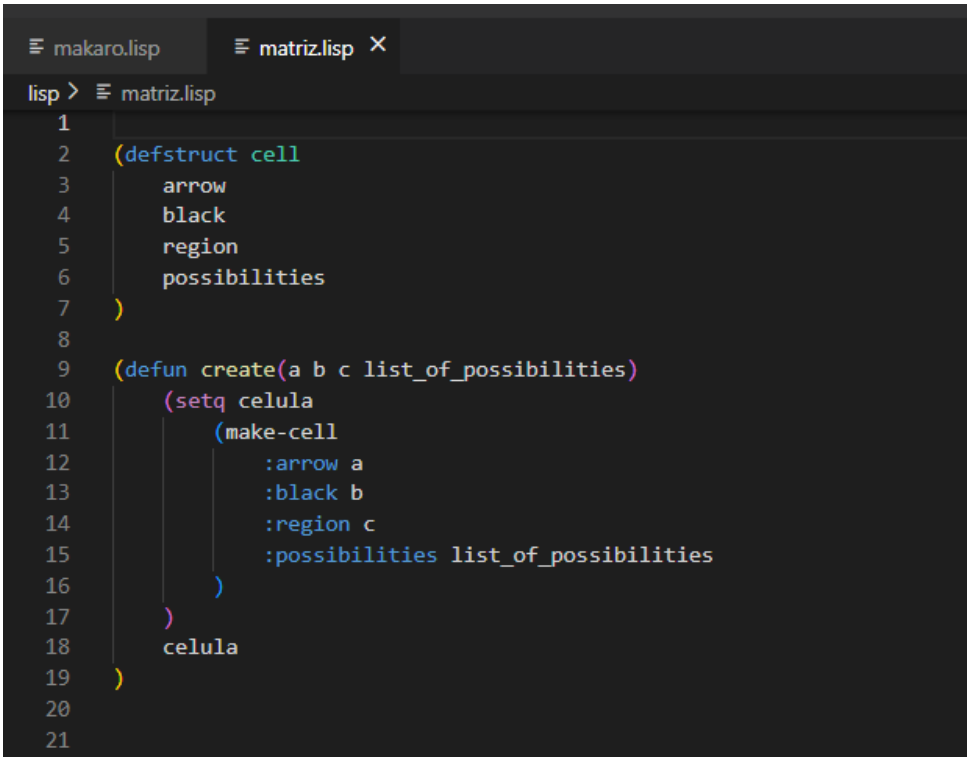


- *Conversors* - Agrupa as funções de conversão de dados, para auxiliar na hora de mostrar valores na saída;

- *Main* - Módulo principal de execução do programa;



- *Matriz* - Agrupa todas as funções e valores referente a construção da matriz a ser analisada;




```
1
2   (defstruct cell
3       arrow
4       black
5       region
6       possibilities
7   )
8
9   (defun create(a b c list_of_possibilities)
10      (setq celula
11          (make-cell
12              :arrow a
13              :black b
14              :region c
15              :possibilities list_of_possibilities
16          )
17      )
18      celula
19  )
20
21
```

- *Makaro* - Constrói a matriz makaro;

```
216     (setq 17c5
217         (create NIL NIL 11 (list 1))
218     )
219     (setq 17c6
220         (create NIL NIL 11 (list ))
221     )
222     (setq 17c7
223         (create T NIL 0 (list 4))
224     )
225
226     (setq matrizMakaro2
227         (list
228             (list 10c0 10c1 10c2 10c3 10c4 10c5 10c6 10c7)
229             (list 11c0 11c1 11c2 11c3 11c4 11c5 11c6 11c7)
230             (list 12c0 12c1 12c2 12c3 12c4 12c5 12c6 12c7)
231             (list 13c0 13c1 13c2 13c3 13c4 13c5 13c6 13c7)
232             (list 14c0 14c1 14c2 14c3 14c4 14c5 14c6 14c7)
233             (list 15c0 15c1 15c2 15c3 15c4 15c5 15c6 15c7)
234             (list 16c0 16c1 16c2 16c3 16c4 16c5 16c6 16c7)
235             (list 17c0 17c1 17c2 17c3 17c4 17c5 17c6 17c7)
236         )
237     )
238
239     matrizMakaro2
240
241 )
```

- *Prunning* - Retira das listas de possibilidades de cada célula, agrupando em regiões, os números de cada região que são Fixed (para todas as regiões);

```
30
31  (defun pruneAndCallFunction(matriz regiao)
32      (deleteFixedValuesOfRegions matriz regiao (getFixedValuesOfRegion matriz regiao))
33      (pruningCellPossibilities matriz (- regiao 1))
34  )
35
36  (defun pruningCellPossibilities(matriz regiao)
37      (if (= regiao 1)
38          (deleteFixedValuesOfRegions matriz regiao (getFixedValuesOfRegion matriz regiao))
39          (pruneAndCallFunction matriz regiao)
40      )
41
42  )
43
44  (defun prunning(matriz)
45      (pruningCellPossibilities matriz (amountOfRegions matriz))
46  )
47  ; end
48
```

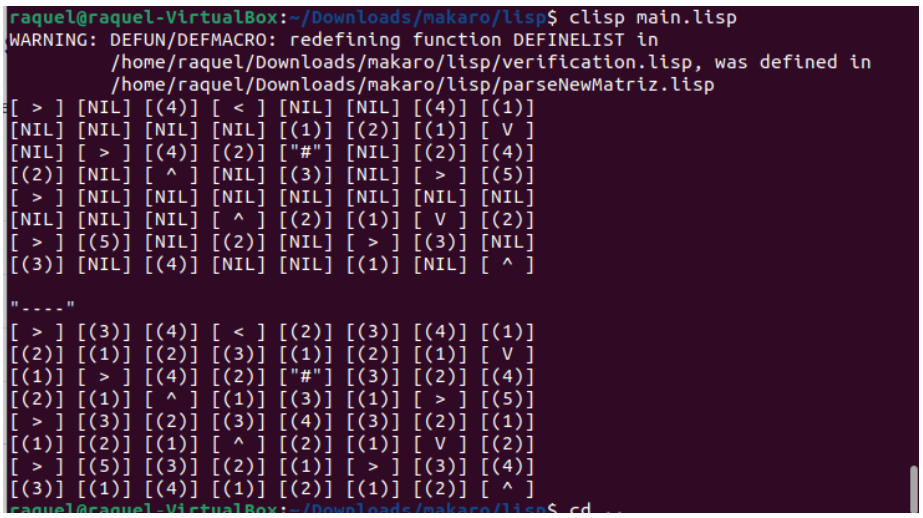
- *Verification* - Varre a matriz e tira da lista de possibilidades das células, com Value Possible, os números que quebram as regras do jogo.

```
179 )
180
181 (defun verifyOrthogonallyAdjacency(matriz)
182     (setq r 0)
183     (setq s 0)
184     (loop for linha in matriz do
185         (loop for celula in linha do
186             (if (isPossible celula)
187                 (setf (cell-possibilities celula) (filtrar (cell-possibilities celula) (getAdjacentValues matriz r s)))
188             )
189             (setf s (+ s 1))
190         )
191         (setf s 0)
192         (setf r (+ r 1))
193     )
194
195 )
```

- *ParseNewMatrix* - Seta as listas de possibilidades das células que não têm valor definido.

```
32      )
33  )
34
35  (defun setListMatrix(matriz listPossibilities)
36      (loop for linha in matriz do
37          (loop for celula in linha do
38              (if (and (not (isArrow celula)) (not (isBlack celula)) (not (isFixed celula)))
39                  (setf (cell-possibilities celula) (defineList 1 (nth (- (getRegion celula) 1) listPossibilities)))
40              )
41          )
42      )
43  )
44
45  (defun setListsOfPossibilities(matriz)
46      (setListMatrix matriz (getListaMaxPossibildadesPorRegiao matriz (amountOfRegions matriz) 1))
47  )
48
```

A saída do programa, ao executar as funções para resolver o problema, é a seguinte:



As setas são representadas pelas string ‘v’, ‘>’, ‘<’ e ‘^’. As células totalmente pretas são representadas pela string ‘#’. Não foi possível realizar a definição das regiões no output, então as mesmas só são conhecidas dentro das 4-tuplas das células.

4. Vantagens e Desvantagens entre Haskell e Lisp

A principal **vantagem** observada na linguagem LISP é a possibilidade de usar a função loop, a fim de percorrer listas, como em:

```
; retorna nova matriz com a lista de possibilidades atualizadas para uma região
(defun deleteFixedValuesOfRegions(matriz regiao fixedValues)
    (loop for linha in matriz do
        (loop for celula in linha do
            (if (and (isPossible celula) (= (getRegion celula) regiao))
                (setf (cell-possibilities celula) (filtrar (cell-possibilities celula) fixedValues))
            )
        )
    )
)
```

Por causa disso, não foi necessário usar recursão para percorrer a matriz, como na linguagem Haskell, e, portanto, criou-se menos funções no código em geral, e a linguagem pareceu ser mais simples de usar para implementar-se programas.

Outra **vantagem** notada na linguagem LISP é que é possível declarar variáveis não apenas na função main, com a função *setq*, diferentemente da linguagem Haskell.

A **desvantagem** observada é a diferença entre *tipo (Haskell)* e *estrutura (Lisp)* nas linguagens. Optou-se por, quando o grupo resolveu Makaro em Haskell, utilizar novos tipos para definir a matriz, a linha e a célula, sendo a construção da matriz mais intuitiva:

```haskell
module Matriz where

    data Value = Fixed Int | Possible [Int] | Arrow Int | Black deriving (Show, Eq)
    type Cell = (Int,Int,Int,Value) -- (Linha, Coluna, Região, Valor)
    type Row  = [Cell]              -- Arrow: 1 - Direita, 2 - Baixo, 3 - Esquerda, 4 - Cima
    type Grid = [Row]
```

Assim, no código em Haskell, uma Grid era definida como uma lista de Linhas, e uma Linha era definida como uma lista de Células, e as Células possuíam quatro valores, três inteiros (para representar linha, coluna, e região) e um outro tipo chamado Value. Value definia se a célula era uma Flecha, ou um Black, ou uma célula que ainda tinha possibilidades, ou uma célula já com número fixo.

Em contrapartida, no código em Lisp, o código ficou menos intuitivo:

```lisp
(defstruct cell
    arrow
    black
    region
    possibilities
)

(defun create(a b c list_of_possibilities)
    (setq celula
        (make-cell
            :arrow a
            :black b
            :region c
            :possibilities list_of_possibilities
        )
    )
    celula
)
```

```lisp
(setq l7c7
    (create T NIL 0 (list 4))
)

(setq matrizMakaro2
    (list
        (list l0c0 l0c1 l0c2 l0c3 l0c4 l0c5 l0c6 l0c7)
        (list l1c0 l1c1 l1c2 l1c3 l1c4 l1c5 l1c6 l1c7)
        (list l2c0 l2c1 l2c2 l2c3 l2c4 l2c5 l2c6 l2c7)
        (list l3c0 l3c1 l3c2 l3c3 l3c4 l3c5 l3c6 l3c7)
        (list l4c0 l4c1 l4c2 l4c3 l4c4 l4c5 l4c6 l4c7)
        (list l5c0 l5c1 l5c2 l5c3 l5c4 l5c5 l5c6 l5c7)
        (list l6c0 l6c1 l6c2 l6c3 l6c4 l6c5 l6c6 l6c7)
        (list l7c0 l7c1 l7c2 l7c3 l7c4 l7c5 l7c6 l7c7)
    )
)
```

Uma matriz é formada por uma lista de listas, onde cada lista dentro da lista principal possui várias células. As células são criadas chamando a função *create*, que instancia a estrutura *cell*, e decide os seus valores arrow, black, region, e possibilities. Portanto, não é possível optar, como em Haskell, se uma célula vai ser do tipo Possible, ou Fixed, ou Black, ou Arrow. Foi necessário criar *arrow* e *black* em *cell*, e, se arrow for NIL, e black for NIL, a célula não é nem uma flecha, nem uma célula black. Além disso, se a lista de possibilidades tiver apenas um valor, assume-se que ela já é um valor fixo.

5. Organização do Grupo

A comunicação do grupo ocorreu por meio de um grupo no aplicativo Telegram. Abaixo, descrição das atividades dos membros:

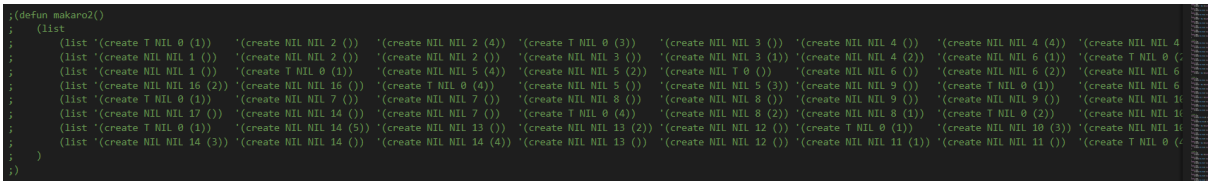
Vitor: Funções do arquivo de *backtracking*.

Raquel: Funções do arquivo *matriz*, funções do arquivo *makaro*, funções do arquivo *prunning*; funções do arquivo *verification*; funções do arquivo *parseNewMatrix*;

Tainá: Funções do arquivo *conversor*.

6. Dificuldades Encontradas

1. A primeira dificuldade foi: criar a matriz makaro. Não foi possível criar diretamente lista de listas em apenas 8 linhas, como no print a seguir:



Sendo necessário definir célula por célula:

```lisp
(defun returnMakaro()
    ;linha 0
    (setq l0c0
        (create T NIL 0 (list 1))
    )
    (setq l0c1
        (create NIL NIL 2 (list ))
    )
    (setq l0c2
        (create NIL NIL 2 (list 4))
    )
    (setq l0c3
        (create T NIL 0 (list 3))
    )
    (setq l0c4
        (create NIL NIL 3 (list ))
    )
    (setq l0c5
        (create NIL NIL 4 (list ))
    )
    (setq l0c6
        (create NIL NIL 4 (list 4))
    )
    (setq l0c7
        (create NIL NIL 4 (list 1))
    )

    ;linha 1
    (setq l1c0
        (create NIL NIL 1 (list ))
    )
```

E juntá-las todas em lista de listas:

```lisp
(setq matrizMakaro2
    (list
        (list l0c0 l0c1 l0c2 l0c3 l0c4 l0c5 l0c6 l0c7)
        (list l1c0 l1c1 l1c2 l1c3 l1c4 l1c5 l1c6 l1c7)
        (list l2c0 l2c1 l2c2 l2c3 l2c4 l2c5 l2c6 l2c7)
        (list l3c0 l3c1 l3c2 l3c3 l3c4 l3c5 l3c6 l3c7)
        (list l4c0 l4c1 l4c2 l4c3 l4c4 l4c5 l4c6 l4c7)
        (list l5c0 l5c1 l5c2 l5c3 l5c4 l5c5 l5c6 l5c7)
        (list l6c0 l6c1 l6c2 l6c3 l6c4 l6c5 l6c6 l6c7)
        (list l7c0 l7c1 l7c2 l7c3 l7c4 l7c5 l7c6 l7c7)
    )
)
```

2. Outra dificuldade foi encontrar como importar as funções de outro arquivo em um arquivo. Após algumas horas de pesquisa, foi possível encontrar que é isso é possível pela função *load*:

main.lisp | matriz.lisp | makaro.lisp | *prunning.lisp* ×

lisp > prunning.lisp

```lisp
(load "matriz")

; Bloco para checar os números fixos de cada região e retirar da lista de possibilidades
```