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 LISP

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

A técnica de *backtracking* foi utilizada para a resolução do quebra-cabeça. Com essa técnica, a estratégia geral de resolução foi, para cada célula em branco, atribuir a ela o número que respeita todas as regras do Kojun.

A estratégia de resolução adotada foi igual ao trabalho anterior, que resolveu o mesmo *puzzle*, usando as mesmas estruturas criadas. Porém, no presente trabalho não utilizou-se definição de tipos.

Como otimizações, criou-se uma estrutura de *possibilities*, que guarda as possibilidades de preenchimento para cada célula, o que otimiza o tempo de execução, uma vez que, no geral, diminui a quantidade de números a serem testados para as células inicialmente vazias.

3. ORGANIZAÇÃO DO CÓDIGO

O código foi organizado em módulos, sendo que cada um implementou funcionalidades específicas para partes da solução. Os módulos desenvolvidos foram:

- **board.lisp:** Módulo com função para geração de tabuleiros de exemplo.
- **printer.lisp:** Funções relacionadas com o *print* de um tabuleiro.
- **solver.lisp:** Funções para solução do quebra-cabeça.
- **main.lisp:** Módulo que invoca as funções dos demais módulos (gera um exemplo de tabuleiro, estrutura de regiões e possibilidades e chama a função principal de resolução do *puzzle*).

4. ENTRADA E SAÍDA

O módulo *Board* contém uma função que gera tabuleiros de acordo com o tamanho especificado. Dessa forma, basta mudar o parâmetro da função *generate-kojun* e um novo tabuleiro de Kojun será gerado. Caso se queira mais exemplos, pode-se modificar a função de geração. A Figura 1 mostra a utilização da função de geração do tabuleiro.

```
  (:import-from :board :generate-kojun)
  (:import-from :solver :define-regions-struct :get-region-from-position :get-adjac

(in-package :main)

(defun main()
    (let* ((size 6)
           (kojun-data (generate-kojun size))
           (board (first kojun-data))
           (regions (second kojun-data))
           (regions-struct (define-regions-struct regions))
           (possibilities (initialize-possibilities board regions regions-struct)))

    (print (solve board regions possibilities regions-struct)))
```

Figura 1. Módulo main.lisp contendo a função de entrada *main.*

Nota-se que a saída é dada pela função *print-board*, do módulo *Printer*, que mostra o tabuleiro resolvido para o usuário:

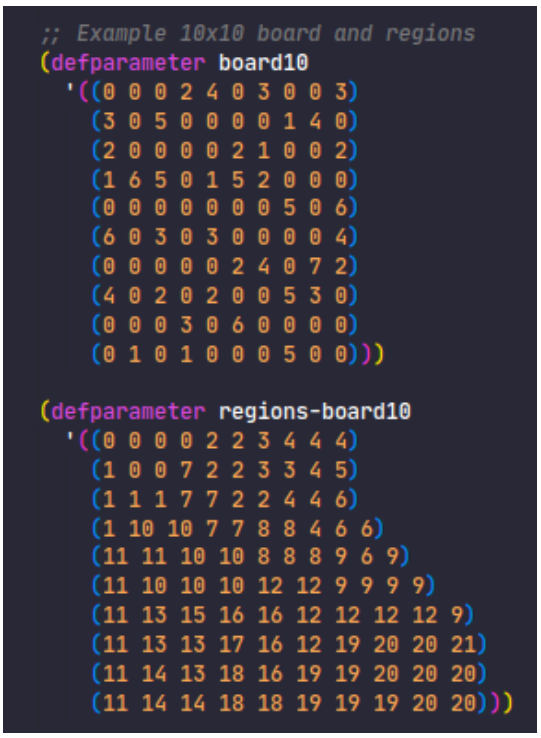
```
No empty positions found, puzzle solved!
4 3 6 2 4 6 3 2 5 3
3 1 5 4 3 5 2 1 4 1
2 5 4 3 5 2 1 7 1 2
1 6 5 2 1 5 2 6 4 1
7 1 4 7 4 3 1 5 3 6
6 2 3 1 3 5 7 3 1 4
5 4 1 3 4 2 4 6 7 2
4 3 2 1 2 1 3 5 3 1
3 2 1 3 1 6 2 4 2 7
2 1 3 1 2 4 1 5 1 6
```

Figura 2. Exemplo de saída.

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. A Figura 3 ilustra essas estruturas.

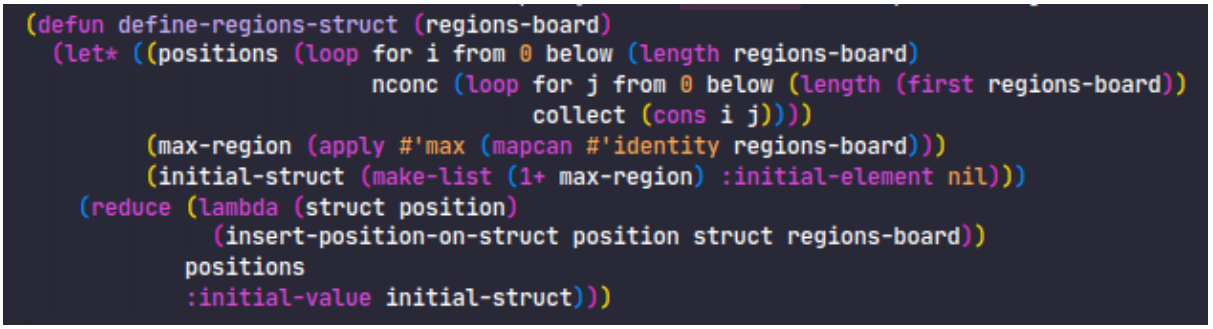


```
;; Example 10x10 board and regions
(defparameter board10
  '((0 0 0 2 4 0 3 0 0 3)
    (3 0 5 0 0 0 0 1 4 0)
    (2 0 0 0 0 2 1 0 0 2)
    (1 6 5 0 1 5 2 0 0 0)
    (0 0 0 0 0 0 0 5 0 6)
    (6 0 3 0 3 0 0 0 0 4)
    (0 0 0 0 0 2 4 0 7 2)
    (4 0 2 0 2 0 0 5 3 0)
    (0 0 0 3 0 6 0 0 0 0)
    (0 1 0 1 0 0 0 5 0 0)))

(defparameter regions-board10
  '((0 0 0 0 2 2 3 4 4 4)
    (1 0 0 7 2 2 3 3 4 5)
    (1 1 1 7 7 2 2 4 4 6)
    (1 10 10 7 7 8 8 4 6 6)
    (11 11 10 10 8 8 8 9 6 9)
    (11 10 10 10 12 12 9 9 9 9)
    (11 13 15 16 16 12 12 12 12 9)
    (11 13 13 17 16 12 19 20 20 21)
    (11 14 13 18 16 19 19 20 20 20)
    (11 14 14 18 18 19 19 19 20 20)))
```

Figura 3. Exemplo das estruturas de tabuleiro e tabuleiro de regiões.

Além desses, também foi definida uma estrutura de pesquisa indexada por região, sendo ela uma estrutura bidimensional auxiliar de pesquisa, em que cada índice i dessa estrutura contém as posições que estão na região *i* do tabuleiro. Essa estrutura é inicializada (Figura 4) no início da resolução e é passada como parâmetro para a função que resolve o quebra-cabeça, como é mostrado na Figura 1.



```
(defun define-regions-struct (regions-board)
  (let* ((positions (loop for i from 0 below (length regions-board)
                          nconc (loop for j from 0 below (length (first regions-board))
                                      collect (cons i j))))
         (max-region (apply #'max (mapcan #'identity regions-board)))
         (initial-struct (make-list (1+ max-region) :initial-element nil)))
    (reduce (lambda (struct position)
              (insert-position-on-struct position struct regions-board))
            positions
            :initial-value initial-struct)))
```

Figura 4. Função de inicialização da estrutura de pesquisa indexada por região.

A estrutura de *possibilities* é uma estrutura tridimensional com n x n listas, cada uma indicando as possibilidades para cada uma das células do tabuleiro. Essa estrutura é inicializada no início do algoritmo (Figura 1) e mantida até o final. Durante o *backtracking*, o algoritmo testa os números de acordo com as possibilidades correspondentes à célula. A Figura 5 mostra a inicialização da estrutura *possibilities*.

```
(defun initialize-possibilities (board regions-board regions-struct)
  (loop for i below (length board)
        collect
        (loop for j below (length (first board))
              collect (initialize-cell (cons i j) (nth j (nth i board)) board regions-board regions-struct))))
```

Figura 6. Inicialização da estrutura *Possibilities.*

5.2. Função principal da solução

A Figura 6 ilustra a principal função utilizada. A função recursiva *solve* recebe o tabuleiro, o tabuleiro de regiões e as estruturas de possibilidades e de busca de regiões.

A primeira ação é buscar por células vazias. Ao achar, invoca-se a função *try-numbers*. Esta, por sua vez, testa números para a posição. Esse teste considera as três regras do *Kojun* para definir se o número é válido para a posição. Sendo válido, o tabuleiro é atualizado e chama-se recursivamente a função *solve* com o novo tabuleiro. Se *solve* retornar que o *puzzle* está resolvido, retorna-se verdadeiro e o tabuleiro resolvido. Caso contrário, continua-se tentando para os demais valores possíveis para aquela célula. Caso o número for inválido em *try-numbers*, segue-se testando os demais valores possíveis para a célula, chamando recursivamente *try-numbers*.

```
(defun solve (board regions-board possibilities regions-struct)
  (let ((empty (find-empty board)))
    (if (null empty)
        (progn
          (format t "~%No empty positions found, puzzle solved!~%")
          (list t board)) ; Return a list with two elements
        (let* ((row (car empty))
               (col (cdr empty))
               (current-region (get-region-from-position (cons row col) regions-board))
               (row-possibilities (nth row possibilities))
               (empty-possibilities (if (listp row-possibilities) (nth col row-possibilities) nil)))
          (if (not (listp empty-possibilities))
              (progn
                (format t "Error: Possibilities for position (~a, ~a) are not a list: ~a~%" row col empty-possibilities)
                (list nil board)) ; Return a list with two elements
              (try-numbers (cons row col) empty-possibilities board regions-struct regions-board possibilities))))))
```

Figura 6. Função *solve*.

Uma das funções mais importantes para a solução é a função que valida se o número é válido para a posição, considerando as regras do Kojun. Ela é utilizada pela função *try-numbers*.

```
(defun is-number-valid-for-the-position (num position board regions-struct regions-board before-run)
  (let* ((current-region (get-region-from-position position regions-board))
         (region-positions (nth current-region regions-struct))
         (is-unique-in-region (every (lambda (pos)
                                       (/= num (nth (cdr pos) (nth (car pos) board))))
                                     region-positions))
         (adjacent-numbers (get-adjacent-numbers position board))
         (is-different-from-adjacents (not (member num adjacent-numbers)))
         (is-vertical-adjacent-valid (if before-run
                                         t
                                         (every (lambda (r-pos)
                                                  (check-vertical-adjacency-validity num position board r-pos))
                                                region-positions))))
    (and is-unique-in-region is-different-from-adjacents is-vertical-adjacent-valid)))
```

Figura 7. Função *is-number-valid-for-position*.

6. DIFICULDADES ENCONTRADAS

As principais dificuldades residiram principalmente em implementar otimizações para o algoritmo de *backtracking*. Pensou-se em manter estruturas de otimização dinâmicas, em tempo de execução, tentou-se implementar, porém não obteve-se sucesso. Por isso, optou-se por manter a estrutura de possibilidades estática, o que já é um ganho. Outra dificuldade inerente é a falta de experiência com programação funcional e com a linguagem LISP. Apesar disso, gerou-se uma implementação funcional, apesar de não ser tão otimizada.