UNIVERSIDADE FEDERAL DE SANTA CATARINA
Disciplina: Paradigmas de Programação
Professor: Maicon Rafael Zatelli
Alunas: Gabriela Furtado da Silveira e Samantha Costa de Sousa

Relatório Trabalho II - Programação Funcional - LISP

Análise do problema

O problema resolvido foi o jogo Makaro, que consiste em um tabuleiro nxn que contém células brancas a serem preenchidas com números (sendo que alguns números já começam preenchidos), células pretas que são apenas regiões vazias ignoradas e células que apresentam setas apontando na direção vertical ou horizontal. Há também regiões de células brancas, delimitadas pelas linhas mais grossas.

Células de uma mesma região devem ser preenchidas com números de 1 a N (tamanho da região), sem repetições. Além disso, uma célula não pode conter o mesmo número que nenhuma de suas vizinhas ortogonais (estejam dentro ou fora da região). Por fim, a célula apontada por uma seta deve conter o maior número entre os vizinhos ortogonais da seta.

Solução

O tabuleiro foi modelado na forma de uma matriz, ou seja, uma lista de listas. As operações com matrizes foram implementadas utilizando uma *struct* que representa uma posição:

```
;; Struct que indica uma posição
(defstruct myposition
    row
    col
)
```

Durante a leitura do tabuleiro, são criadas duas matrizes: "*regionsMatrix*", contendo as regiões (listas de listas de tipo *String*) e "*certaintyMatrix*", contendo os valores do tabuleiro (listas de listas de tipo *Int*). Após, é criada uma terceira matriz denominada "*regions*". A estrutura "*regions*" contém o mapeamento das regiões na forma de listas de posições (*myposition*), nas quais a primeira *myposition* contém o número de elementos na região, seguida pelo número de posições não preenchidas, e as demais são as posições das células pertencentes à região

Para a solução do puzzle, há primeiramente um conjunto de funções que preenchem os números de que se tem certeza.

```
;; Chama a função fillMat1 para todas as regiões, de forma a preencher todasas regiões com
;; apenas uma posição não preenchida (só há uma opção para o número, nesse caso)
(defun fillMissingOne (i mat_n_reg)
    (let* ( (mat (car mat_n_reg))
            (regions (cadr mat_n_reg))
            (missing_num (fillMat1 i (missingNumbers (Matrix:getRow regions i) mat) mat regions))
          )
        (if (< (+ i 1) (Matrix:getRowsNumber regions))
            (fillMissingOne (+ i 1) missing_num)
            missing_num
        )
    )
)
```

```
;; Chama a função fillMat2 para todas as regiões, de forma a preencher as regiões com apenas duas posições vazias
;; nas quais é possível determinar com certeza os valores de cada uma por causa de seus vizinhos ortogonais
(defun fillMissingTwo (i mat_n_reg)
    (if (< (+ i 1) (Matrix:getRowsNumber (cadr mat_n_reg)))
        (fillMissingTwo (+ i 1) (fillMat2 i (car mat_n_reg) (cadr mat_n_reg)))
        (fillMat2 i (car mat_n_reg) (cadr mat_n_reg))
    )
)
```

A função *fillMissingOne* atua sobre as regiões, preenchendo aquelas em que resta apenas uma única célula não preenchida com o número correspondente (como as demais estarão com outros números, se tem certeza do número que deve ser colocado). Já a função *fillMissingTwo* atua de forma parecida, porém preenchendo as células de regiões onde há apenas duas posições vazias, caso seja possível determinar com certeza qual número deve ir em qual posição em virtude dos vizinhos ortogonais.

Após obter a matriz de entrada preenchida com os casos de certeza, se inicia o algoritmo de backtracking. A solução possui duas lógicas de backtracking, uma interna à região e uma referente ao puzzle inteiro.

Dentro da região (módulo *SolveRegion*): após adquirir, a partir das estruturas existentes, os números possíveis e as posições para serem preenchidas, é feita a tentativa de preencher cada número em uma dessas posições. Ao conseguir preencher um número em uma posição (passou das validações, que é uma junção de funções que implementam as regras do puzzle, no módulo *Validations*), ele passa adiante com o próximo número e as posições restantes. Ele faz isso até preencher a região inteira, guardando o caminho em uma estrutura específica para isso.

- Um caminho interno à região consiste de uma lista de inteiros referente a indexação das possibilidades de preenchimento na região. Ou seja, numa região que existem 3 casas a preencher, existem inicialmente 9 possibilidades, que são geradas pelo código (um número em uma posição). Então um caminho nesse caso poderia ser [0, 3, 6], esses números referentes ao índice da lista de possibilidades.

Porém, se chegar em um ponto no qual não é possível preencher o número da vez em nenhuma das posições restantes (não passa nas validações) então ele retorna ao preenchimento do número anterior. Para manter o controle do que falhou e não repetir o preenchimento errado, existe uma estrutura auxiliar *errorList*, que, para cada número, possui uma tupla (lista com dois itens) de:

- [Caminho acima do preenchimento atual, Posições que deram errado nesse ponto]

Essa lógica é mais visualizável se imaginarmos o preenchimento como a geração de uma árvore. As posições referente àquelas possibilidades só são uma falha naquele exato ponto da árvore, por isso é salva uma parte do caminho junto com o que causa a falha, para ser feita a verificação. Por exemplo:

- O caminho atual é [0, 4]. A próxima tentativa de preenchimento falha. Então ele salva na lista de erros, no segundo elemento da lista (referente ao segundo número a ser preenchido no caminho): [[0], [Posição referente à possibilidade 4]].

Após isso são feitas as limpezas necessárias e, ao tentar preencher um número, ele sempre exclui as posições que estão alocadas na sua lista de erros (se caminho atual == caminho salvo na lista de erros) das posições possíveis. Assim, esse erro não é repetido.

```
;; Dado um caminho, verificar se não há posições falhas a serem removidas nesse ponto da árvore.
;; (a partir da errorList). Retornar a lista de posições alterada (ou não).
(defun removeErrorPositions (path errorList possiblePositions)
    (let* ((order (length path)) ; Referente ao indice do número na lista de erros
           (currentError (nth order errorList)) ; Tupla [[int], [myposition]] (regionError)
           (lenErrorList (length (cadr currentError)))
           (isCurrentError (and (> order 0) (> lenErrorList 0) (equal (car currentError) path)))
           (isFirstNumberError (and (= order 0) (> lenErrorList 0))))
        (if (or isCurrentError isFirstNumberError)
            (removeItemsFromList possiblePositions (cadr currentError))
            possiblePositions
        )
    )
)
```

(Exclusão de posições falhas)

Todo esse procedimento pode, no fim, gerar uma região válida naquele ponto e retornar "*T*" com as devidas estruturas alteradas, ou, ao não conseguir um caminho possível, retornar "*NIL*" para falha total da região.

A captura da falha total da região ocorre no outro backtracking, externo à região (módulo *Solve*). Ele segue uma lógica muito semelhante ao backtracking interno, exceto que:

- O caminho atual que vai se formando é uma lista dos caminhos internos da região. Exemplo: [[0], [1, 2], [2, 4, 7]].
- Os erros associados à *região* no puzzle não são posições, e sim caminhos de região. Ex, o caminho [2, 4, 7] seria considerado um caminho "falha" para aquele ponto caso a próxima região obtivesse falha total. Esse valor deve ser passado em uma lista de "caminhos errados" para a região anterior à falha, que vai tentar ser preenchida novamente de uma forma que não gere algum caminho que cause falha nesse ponto da árvore.

O backtracking interno à região, então, possui mais uma verificação enquanto faz suas tentativas de preenchimento de números. A partir de uma lista de "caminhos falhas" (que não é vazia caso eu esteja tentando preencher aquela região mais uma vez no mesmo ponto da árvore), ele verifica, a cada preenchimento, se não está seguindo algum desses caminhos falhas, e tenta alterar o preenchimento; Caso sim, ele continua com o novo preenchimento que evitou um caminho falha, caso não, ele continua. No fim do preenchimento de região, há uma verificação específica para ver se o único caminho que a região conseguiu preencher é um que já falhou, então também retorna falha total para a região.

```
;; Enquanto existir números para preencher, vai para o backtracking.
;; Se não há, verifica se todas as posições foram preenchidas ou se o caminho final é um que já falhou.
(defun fillWholeRegion (mat matRegions possibleNumbers possiblePositions region possibilities path errorList wrongPaths)
    (cond
        ((and (= (length possibleNumbers) 0) (not (= (PosUtils:getSecond (car region)) 0)))
        (list nil mat region path))

        ((and (= (length possibleNumbers) 0) (= (PosUtils:getSecond (car region)) 0))
        (list (not (member path wrongPaths :test #'equalp)) mat region path))

        (t
        (backtrackingTryFillNumber mat matRegions possibleNumbers possiblePositions region possibilities path
            errorList wrongPaths))
    )
)
```

(A verificação de ter só conseguindo repetir um caminho que já falhou se refere ao "*path*" ser igual a um caminho falha existente na "*wrongPaths*", verificado com a função *member*)

Entrada e resultado

O programa recebe como entrada um tabuleiro do jogo Makaro na forma de um arquivo de texto com extensão “.txt”, cujo nome é passado na main obedecendo o seguinte formato:

- Na primeira linha há apenas um número n que corresponde ao número de linhas do puzzle (ex: para um tabuleiro 4x4, n = 4).
- As próximas n linhas devem conter as linhas do puzzle representadas de acordo com as regiões: todas as células pertencentes a uma mesma região devem ser representadas pelo mesmo número, sendo que os números correspondentes a cada região são atribuídos em ordem crescente (1, 2,…), da esquerda para a direita e de cima para baixo conforme a primeira aparição de uma célula da região. Para células pretas (vazias) deve ser colocado um“X” na posição e para as setas, devem ser usadas letras que representem o sentido para o qual apontam: L - esquerda (left), R - direita (right), U - cima (up), D - baixo (down).
- As n linhas seguintes devem conter os números de cada célula, utilizando 0 para células vazias (brancas ou pretas) e para setas.

Exemplo de entrada para um tabuleiro 8x8:

```
8
1 D 2 3 3 4 4 4
5 5 2 6 D U 4 U
R 5 5 6 7 7 8 8
9 5 U 10 7 11 L 12
9 10 10 10 11 11 11 12
9 X D 13 U 11 X 12
14 14 13 13 15 15 15 D
16 16 16 X 17 17 15 15
0 0 0 2 0 0 3 4
0 5 2 0 0 0 0 0
0 0 3 0 3 0 0 0
0 0 0 1 2 0 0 3
0 3 2 4 0 5 0 0
3 0 0 0 0 0 0 1
0 2 0 0 0 4 0 0
2 3 0 0 2 0 3 0
```

(puzzle disponível em: https://www.janko.at/Raetsel/Makaro/001.a.htm)

O puzzle resolvido é apresentado na forma da impressão no terminal da matriz nxn contendo os números da solução.

Vantagens e desvantagens entre Haskell e Lisp na implementação

Haskell é uma linguagem que é fortemente tipada, o que causa muitos erros em tempo de compilação, mas justamente por isso tem a vantagem da diminuição dos erros em tempo de execução, principalmente quando são utilizado novos tipos criados em Haskell para delimitar as estruturas usadas em argumentos e retornos de função. Como a lógica já tinha sido feita na versão em Haskell, ao usar Lisp, isso não foi um problema enorme, mas houve casos que as estruturas não eram exatamente o que esperávamos e só observamos isso em tempo de

execução. Mas caso a primeira implementação tivesse sido em Lisp, imagino que o tempo para encontrar bugs que passariam pela compilação seria bem maior.

Mudanças advindas da implementação em Lisp

Todas as funções feitas por nós na versão em Haskell tem uma certa equivalência para a versão em Lisp. Principalmente nos módulos de backtracking em Haskell, foi utilizado muito da cláusula *where* para organização e legibilidade do código, então na versão em Lisp o que foi usado para continuar nessa mesma lógica foi a cláusula *let* (principalmente *let**). Mas a diferença entre o *where* do Haskell e o *let* do Lisp é que as variáveis no *where* só são "executadas" quando são utilizadas de fato, mas no *let* é primeiro executado tudo que é necessário para a criação dessas variáveis. Isso requisitou uma avaliação do escopo dos *let*'s usados nas funções que possuem condições, pensando em quais variáveis ficariam no escopo do *let* "acima" da condição, quais ficariam dentro do *let* de resultado da condição. Essa avaliação foi necessária por dois pontos: um, evitar a execução de funções que causariam erros pois não deveriam estar sendo executadas seguindo aquelas condições, dois, evitar a execução de funções que seriam alocadas para variáveis que não seriam retornadas ou utilizadas por conta das condições.

Outro ponto é a equivalência entre as funções nativas do Haskell e as nativas do Lisp. Nem sempre elas eram equivalentes, mesmo possuindo o mesmo nome, e nem sempre as funções do Haskell existiam em Lisp, causando a necessidade de ou implementá-las, ou utilizarmos de bibliotecas externas, e preferimos a primeira opção, fazendo que a versão em Lisp tenha funções extras em relação a versão em Haskell.

Entretanto, algumas das principais mudanças que ocorreram para a implementação em Lisp foram devido à falta de uma função para definir tipos, e também à inexistência de tuplas, tal qual havíamos utilizado em Haskell. O tipo *Position* (em Haskell) foi implementado como uma *struct*, a matriz acabou sendo implementada simplesmente como uma lista de listas, sem ser um tipo, propriamente, bem como e os demais tipos criados em Haskell, que envolviam tuplas passaram a ser apenas listas.

Organização

Samantha fez a passagem de Haskell para Lisp dos módulos de solução (backtracking). Gabriela fez essa passagem nos módulos que envolviam implementação de operações com matriz, validação e leitura do arquivo de entrada com montagem das estruturas baseadas nele. Além disso, ambas se envolveram na busca de erros.

Dificuldades e soluções

Como a lógica em paradigma funcional para a solução do puzzle já existia na versão em Haskell, não houve muita dificuldade nesse ponto. Os pontos principais de dificuldade foram a sintaxe (erros de parênteses, desde os mais básicos captados em compilação até alguns mais difíceis de identificar, como por exemplo colocar uma execução fora de um *if* quando queríamos dentro do *if* como um *else*) e equivalência de funções (como por exemplo, o *last* no Haskell retorna o último elemento de uma lista, mas o *last* no Lisp retorna uma lista com o último elemento da lista passada, e essa diferença foi causa de erros).

Muitos dos erros não eram identificados facilmente nem com ferramentas como o "*ChatGPT*", que também se confundia na equivalência de funções das linguagens; Nesses pontos, além da análise de código, foram necessárias buscas na documentação da linguagem

e testes isolados de uso dessas funções para ver se funcionavam com as estruturas que queríamos utilizar, assim, adaptando o código a depender do resultado desses testes.

Além disso, muitos problemas foram causados pela utilização de uma *struct* para definir posições, já que foi necessário criar funções extra dentro do arquivo para que fosse possível criar e acessar essa estrutura através de outros módulos. Outro ponto foi a questão de comparações, pois Lisp apresenta muitos tipos diferentes de comparações *(eq, eql, equal, equalp,* =), que geram resultados diferentes dependendo do caso. Isso causou problema principalmente por causa da *struct myposition*, uma vez que as posições estavam envolvidas em inúmeras comparações ao longo do código, tanto no módulo de validações (*Validations*), quanto ao longo do backtracking. Por causa disso, toda vez que elas faziam parte, era necessário mudar para "*equalp*", tanto para comparações simples quanto para funções utilizadas, como *position* e *member*.

Todos os puzzles que são solucionados na versão em Haskell são solucionados na versão em Lisp (de tamanho 8x8, 12x12 e 15x15). A execução também continua com o tempo aparentemente instantâneo, já que a lógica ficou inalterada.