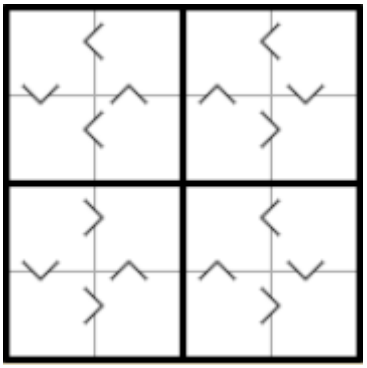
Matheus Paulon Novais - 20203078

Trabalho II: Lisp

1. Descrição

Entre as três opções disponíveis para o trabalho foi escolhida a do *Vergleichssudoku*, também conhecido como *Inequality Sudoku*(Sudoku comparativo) ou *Greater Than Sudoku*(Sudoku maior que, nome dado pelos símbolos presentes no tabuleiro). Esse jogo é uma vertente muito parecida com o sudoku comum pelos espaçamentos do tabuleiro, podendo ser 4x4(com 4 casas de 2x2), 6x6(com quatro casas de 3x2) e 9x9(com quatro casas de 3x3).

O jogo inicia com um tabuleiro de sudoku zerado, e a forma de resolvê-lo é identificar os símbolos de maior-que(“>”) juntamente com o de menor-que(“<”) presentes no tabuleiro, causando uma comparação direta entre as casas. Assim como aprendemos no começo da nossa caminhada no mundo matemático, a boca aberta do símbolo fica para o lado que é maior que o outro, devemos então completar os números que vão de 1 à *N* (*N* = tamanho de células em cada casa) em uma casa do tabuleiro a partir dessas noções.



Exemplo número 1: Vergleichssudoku

2. Desenvolvimento

Após compreender o funcionamento do jogo e também como jogar, o último passo era escolher uma estratégia para resolver o problema. Vendo semelhança com o sudoku comum, a estratégia foi desenvolver um resolvedor de Sudoku normal.

A partir desse momento surgiram duas vertentes da ideia, a primeira diminuindo a quantidade de opções possíveis para cada célula a partir de combinações específicas de símbolos, ou a cada elemento testar a possibilidade do número com os números vizinhos e os símbolos apontados para a célula atual. Junto com a opinião do professor foi escolhida a segunda opção.

Com a estratégia escolhida, agora era escolher como desenvolver ela. Iniciamos com uma forma de pegar os dados da matriz de entrada, que será explicada em sequência, e comparar sequencialmente com os dados desenvolvidos pelo algoritmo.

3. Entrada e Saída de dados

A entrada é dada por uma matriz com listas que descrevem os quatros lados de cada casa, com quatro valores para indicar se contém algum sinal, e caso tenha, qual sinal é.

Cada lista segue o exemplo de (esquerda, direita, cima, baixo), e os valores possíveis para cada um deles são:

3 -> Não contém sinal na parede indicada

2 -> Símbolo de maior que “>”

1 -> Símbolo de menor que “<”

Exemplo segundo a imagem do primeiro tópico:

(3 1 3 2)	(2 3 3 1)	(3 1 3 1)	(2 3 3 2)
(3 1 1 3)	(2 3 2 3)	(3 2 2 3)	(1 3 1 3)
(3 2 3 2)	(1 3 3 1)	(3 1 3 1)	(2 3 3 2)
(3 2 1 3)	(1 3 2 3)	(3 2 2 3)	(1 3 1 3)

O resultado é dado diretamente na saída do console após a execução do programa.

Para o LISP, a matriz de entrada teve seus números negativos retirados, para facilitar o processo.

4. Resolução do problema

Para resolver o problema, iniciamos com o resolvedor de sudoku comum, e a partir dele foi criado funções para poder processar os dados da matriz de entrada e da matriz feita pelo programa, e futuramente essas funções foram utilizadas para criar uma validação de cada elemento que é processado e adicionado a matriz final.

Começamos por desenvolver uma função para testar se cada vizinho que será procurado estará dentro dos limites da matriz, para que não ocorra erros de *out of bounds*. O segundo passo a partir desse momento foi entender como processar os dados recebidos da matriz de entrada junto aos dados da matriz sendo gerada. Para isso, junto as condições de *backtracking* do sudoku comum, como o teste de linha, coluna e caixa, foi criada primeiramente uma função que pegaria os dados da casa atual dentro da matriz de entrada a partir de uma função lambda combinada ao assessor *aref* para poder pegar o momento atual, esses dados, juntamente com uma lista auxiliar de posições são passados para função de validação.

Nesse momento é usada a lista auxiliar para saber se vamos processar a esquerda, direita, cima ou baixo a partir de o que seria um *switch-case*, dentro de cada um desses casos é então testado qual símbolo estamos trabalhando com, para depois verificarmos se o vizinho que vamos atuar em cima existe

dentro da matriz, e por último verificamos a condição de cada símbolo, assim sendo, se ele deve ser maior ou menor que o seu vizinho indicado.

5. Diferenças observadas

A maior vantagem, e também a maior diferença, no desenvolvimento foi a possibilidade de uso de loops e repetições em macros como *dotimes*, que para o grupo simplificou a compreensão para poder percorrer matrizes e vetores. A maior desvantagem, também em desenvolvimento, foi a diferença de quantidade de informações conseguidas em Haskell e na linguagem atual, por mais que LISP tenha um ranking um pouco mais alto pelo o que é pesquisado na TIOBE, acabou por ficar uma impressão de menor conteúdo ligado diretamente a linguagem, talvez até pela maior quantidade de linguagens filhas de LISP.

6. Dificuldades

Diferentemente do primeiro trabalho onde precisamos primeiro entender o problema, pensar em uma solução e por fim pensar em como aplicar ela em Haskell, dessa vez começamos diretamente do último passo, que era aplicar a nossa solução anterior em uma nova linguagem.

Não que isso seja mais fácil por si só, claro que também foram encontradas dificuldades nessa adaptação para uma nova linguagem. Um obstáculo que também foi encontrado com Haskell era entender como funcionavam os tipos dessa nova linguagem, para depois aprender a como manipulá-los e nesse trabalho que tivemos grandes manipulações sobre listas e matrizes não teria como ser simples. As regras que se encontram nas letras miúdas do contrato assinado com o Lisp, tais como compreender a noção de como e quando podemos usar cada diferente função para cada diferente tipo foi um dos atrasos encontrados no desenvolvimento.

A última e provavelmente mais constante adversidade foi ler os erros e saber quais informações disponibilizadas pelo compilador *clisp* eram importantes, visto que algo que repete a cada uso o símbolo e os nomes de seus criadores ocupando pelo menos metade do terminal não tem como não ser prolixo.