Relatório - Segundo Trabalho de Linguagens de Programação

UFJF - 2022

André Caetano Vidal - 201665010AC
Bernardo Souza Abreu Cruz - 201635019

Introdução

Este relatório tem como objetivo descrever e expandir em relação ao segundo trabalho de Linguagens de Programação no período 2021.3 da UFJF, reproduzindo o jogo Nim em Haskell. O projeto deve possuir, obrigatoriamente, as seguintes funcionalidades:

- Fidelidade às regras de Nim definidas na descrição do trabalho;
- Possibilidade de iniciar uma partida de Nim contra a CPU;
- Exibição do estado atual do tabuleiro;
- Determinação de jogada do usuário e da CPU alternadamente;
- Remoção um determinado número de palitos de uma determinada linha do tabuleiro;
- Um modo fácil, onde o jogador começa e a CPU faz jogadas aleatórias;
- Um modo difícil, onde a CPU começa e faz sempre a melhor jogada possível;
    - A melhor jogada possível deve ser sempre calculada a partir do algoritmo disponibilizado na descrição do trabalho;
- Interface intuitiva e didática;

Ferramentas utilizadas e criação do ambiente

Utilizamos a linguagem Haskell como determinado na descrição do trabalho e Stack para gerenciar as dependências após vista a necessidade do uso de aleatoriedade no algoritmo.
Para gerar o ambiente, fizemos a instalação do Stack com a versão 2.7.3, e instalamos o pacote random para fazer a geração de números aleatórios através do comando “stack install random”. Para compilação do algoritmo, utilizamos a versão 8.10.7 do ghci, contida no Stack por padrão.
Para executar o projeto, utilizamos o comando “stack ghci” para executar o compilador a partir da ferramenta Stack e por fim a instrução “:l nim.hs” para carregar o programa.

Como iniciar um jogo

Para iniciar um jogo após a execução do programa no terminal, basta digitar “start” como linha de comando. O jogo então retornará uma mensagem dizendo que o jogo foi iniciado e pedirá a seleção de dificuldade, sendo **0 como modo fácil** e **1 como modo difícil**. Digitar valores inválidos nesse momento será ignorado, e o usuário será solicitado novamente a inserção de um valor relacionado a dificuldade.
Após a seleção, o jogo inicia o tabuleiro com o padrão solicitado na demanda, com um palito na primeira linha, três na segunda, cinco na terceira e sete na quarta.

Representação do jogo

O jogo gira em torno de um tabuleiro, representado por meio de uma lista de inteiros. A lista é criada com os valores [1, 3, 5, 7], representando cada uma das fileiras de palitos existentes no jogo com a quantidade inicial de palitos em cada uma delas. Esse tabuleiro é mostrado em tela ordenadamente, da seguinte forma:

1
3
5
7

A cada rodada, alterações também são mostradas em tela após a rodada de cada jogador e antes do início da próxima, permitindo que usuários saibam constantemente o estado do tabuleiro e pensem qual deve ser a próxima jogada de maneira bem visual. Fileiras que não contém mais palitos são representadas com o número 0.

Rotina de turnos

Cada rodada é controlada inteiramente por um único método chamado **play**, utilizando como parâmetros o atual estado do tabuleiro, o jogador da rodada atual e a dificuldade selecionada. O método é recursivo, de forma que ao chegar ao final de cada iteração, as instruções são executadas novamente e alterando o jogador atual.
O método também controla a execução da rodada de cada jogador, requisitando ao usuário os inputs necessários para sua ação e realizando ações da CPU de acordo com a dificuldade selecionada.
A rodada é iniciada checando o atual estado do tabuleiro e verificando se todas as fileiras estão com número de palitos zerados, significando o fim do jogo. Caso isso seja verdade, o jogador atual é considerado como derrotado na partida, já que demonstra que o último jogador retirou o último palito do tabuleiro.

Rodada do usuário

O movimento do jogador é feito através do método **playerMove**, executado através de inputs do usuário com o seguinte comportamento:

- Será solicitado para o mesmo o número da fileira que ele deseja atuar entre todas as fileiras do tabuleiro.

    - O jogo então valida se o número da fileira é válido, estando entre as fileiras 1 e 4.
- Imediatamente depois, o jogo solicita o número de palitos que o usuário deseja remover.
    - O jogo então valida se o número de palitos solicitados é válido, ou seja, é menor ou igual o número de palitos atuais na fileira.
- Caso alguma das validações não tenha sido confirmada, o usuário é solicitado a inserção dessas informações novamente, atendendo a possibilidade do jogador repensar sua jogada caso a mesma não esteja inicialmente disponível.

Rodada da CPU

Diferentemente do usuário, a rodada da CPU é definida a partir de métodos relacionados a atual dificuldade da partida, seguindo aleatoriedade no caso do modo fácil e a melhor jogada possível no modo difícil.
Como falado, o **modo fácil** tem os movimentos definidos de maneira completamente randômica, desde a fileira até o número de palitos retirados, a partir das funções citadas abaixo:

- **randomMove**: responsável por calcular isso, utiliza outros métodos auxiliares para garantir que a jogada sempre é válida, redefinindo a margem de intervalo de aleatoriedade de acordo com o número de fileiras e palitos disponíveis.

- **getRandomRow**: método utilizado para escolher uma fileira aleatória entre as fileiras que possuem pelo menos um palito disponível para ser removido.

- **getNonZeroRows e getNonZeroRows**Helper: responsáveis por retornar uma lista das fileiras não-zeradas, em que utilizamos tuplas para garantir a manutenção do índice no tabuleiro original e o número de palitos nela disponível.

Já o **modo difícil,** é calculado a partir de um algoritmo que garante sempre a melhor jogada para cada situação, assegurando o jogador que realizar a primeira jogada a vitória na partida. A estratégia vencedora consiste em sempre passar a rodada para o próximo jogador de forma que a soma decimal das fileiras no tabuleiro em representação binária possua apenas algarismos pares. Para isso, utilizamos os métodos abaixo:

- **bestPlay e bestPlayHelper:** observando o estado atual do tabuleiro, esses métodos avaliam todas as possibilidade de jogada em cada fileira a partir da primeira, iniciando sempre com uma possível remoção do número total de palitos existentes naquela linha. Em primeiro momento, se o valor removido não atende a estratégia vencedora, o valor de palitos removidos é decrementado por uma unidade até chegar a zero. Ao chegar a zero, o método inicia o processo novamente para a próxima fileira. Caso não exista uma jogada que se encaixe perfeitamente na estratégia, é realizada uma jogada aleatória.
  Ao final do método, é retornado

- **sumDigitsBin e sumDigits:** métodos utilizados para calcular a soma dos dígitos do número de palitos representados em binário, resultando numa lista de algarismos na base decimal.

- **toBin e toBin3**: utilizados para representar o número de palitos de cada fileira em base 2, retornando uma lista de inteiros com 3 elementos, cada um representando um dígito binário.

- **checkDigitsEven:** retorna o valor booleano caso todos os elementos de uma lista de inteiros sejam pares, usado para validação da soma dos números binários representando cada fileira.

Métodos auxiliares

Ao longo da execução do projeto, notamos a necessidade da criação de alguns métodos utilitários para a construção de todas as operações previstas no jogo.

- **getDigit:** para pegar um dígito inserido pelo usuário.

- **printRows:** para imprimir a quantidade de palitos por fileira, representando uma fileira por linha.

- **removeSticks:** método que recebe o estado atual do tabuleiro, a fileira escolhida e a quantidade de palitos removidos e realiza a remoção do número de palitos da linha.

- **nextPlayer:** método que determina o próximo jogador a partir do jogador atual.

Conclusão

Utilizando os métodos explicitados acima, conseguimos colocar em prática os conhecimentos adquiridos em cima da linguagem Haskell durante a disciplina, criando um modelo funcional do jogo Nim, com a possibilidade de jogar contra a CPU em diferentes dificuldades.