UNIVERSIDADE FEDERAL DO CEARÁ

GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

GUSTAVO CAMPELO DE SOUSA

PABLO KAUAN MARTIN TIMBÓ

FRANCISCO DAGOBERTO SILVA DOS SANTOS

ANTONIO CAIO OLIVEIRA NASCIMENTO

EMERSON DE SOUSA CARVALHO

TRABALHO DE PROGRAMAÇÃO FUNCIONAL

CRATEÚS

2025

TRABALHO DE PROGRAMAÇÃO FUNCIONAL

Relatório do trabalho do terceiro módulo da disciplina de Programação Funcional, tratando da implementação de algoritmos para resolução de problemas usando funções da linguagem Haskell, utilizando uma árvore de decisões binária através de tipos algébricos aprendidos em sala de aula.

Professor: Prof. Simone de Oliveira Santos

CRATEÚS

2025

SUMÁRIO

1. Divisão de Tarefas

A divisão de tarefas foi feita de forma que cada membro pudesse aplicar seus conhecimentos da melhor maneira, colaborando na criação de um sistema eficiente e funcional:

Emerson, Caio e Dagoberto: Responsáveis pelo desenvolvimento da árvore binária de decisão, na qual os dados são organizados e avaliados de acordo com as respostas do usuário. Com atribuições de refatoração extra. Foram atribuídos a questão 1.

Gustavo e Pablo: Focados na criação da interface do programa, permitindo que o usuário interaja com a árvore de decisão, como bem pedia a questão 2.

2 - Atividades dos membros

Atividades do membro - Emerson:

QUESTÃO 1

Colaborou no desenvolvimento de uma árvore binária de strings, que foi estruturada da forma como foi descrita na imagem fornecida no trabalho, tornando-a uma árvore de decisão, que a cada nível toma uma decisão para saber se um candidato foi aprovado ou não no mestrado, a partir da entrada do usuário. Além disso, fez algumas refatorações simples no código.

```
decisionTree :: BinTree String
decisionTree =
    Node "Fez Poscomp?"
        (Node "Nota do IRA >= 7?"
            (Node "Nota da Entrevista >= 7?"
                (Node "Nota da Prova de Títulos >= 5?"
                    (Leaf "Classificado")
                    (Leaf "Eliminado"))
                (Node "Nota da Prova de Títulos >= 7?"
                    (Leaf "Classificado")
                    (Leaf "Eliminado")))
```

```
            (Node "Nota da Entrevista >= 9?"
                (Node "Nota da Prova de Títulos >= 7?"
                    (Leaf "Classificado")
                    (Leaf "Eliminado"))
                (Node "Nota da Prova de Títulos >= 8?"
                    (Leaf "Classificado")
                    (Leaf "Eliminado"))))
        (Node "Nota do IRA >= 8?"
            (Node "Nota da Entrevista >= 8?"
                (Node "Nota da Prova de Títulos >= 8?"
                    (Leaf "Classificado")
                    (Leaf "Eliminado"))
                (Leaf "Eliminado"))
            (Leaf "Eliminado"))
```

Atividades dos membros - Gustavo e Pablo:

QUESTÃO 2

Os membros Gustavo e Pablo desenvolveram toda **a lógica da questão 2**: Que pede para fazer um programa interativo que receba as informações corretas vindas do usuário, e estas são compostas pelas funções main e askQuestion. Através de perguntas através do percorrimento da árvore, o resultado do candidato será dado corretamente. Abaixo segue a explicação do trecho do código:

```
askQuestion :: BinTree String -> IO String
```

Essa é uma função que receberá a árvore binária, onde cada nó contém uma String, onde essa String é o texto da pergunta a ser feita, e deve retornar a entrada do usuário, que também é uma String.

```
askQuestion tree = do
    putStrLn "\n--- Árvore atual ---"
    putStrLn $ drawTree (toDataTree tree)
    case tree of
        Leaf result -> return result
        Node question left right -> do
            putStrLn question
            answer <- getLine
```

Aqui há um print indicando o próximo passo que é mostrar a árvore. Logo após a árvore binária criada há de ser convertida para uma árvore compatível com o módulo Data.Tree (biblioteca do Haskell) para desenhar uma árvore bonitinha no terminal, sendo bem útil para a visualização e entender o fluxo da árvore.

Após isso há uma verificação em caso do nó da árvore ser uma Leaf(folha), que é o nó terminal, chegou-se ao fim do percorrimento da árvore e tendemos a ter uma resposta final, que é Classificado ou Eliminado, para retornar o resultado como resposta da função. Caso não seja uma folha, então é um nó com a pergunta e duas subárvores (esquerda e direita), ou seja, será perguntado ao usuário e decidido o caminho a ser seguido na árvore, e após isso é exibido na tela a pergunta, e na linha seguinte a resposta do usuário é lida, com o *getLine* capturando o texto digitado no terminal.

```
    if isYes answer then askQuestion left
                else if isNo answer then askQuestion right
                else do
                    putStrLn "Entrada inválida! Responda com uma
variação de SIM ou NÃO."
                    askQuestion tree
```

Essa resposta capturada no terminal possui uma verificação, caso o usuário responda positivamente (através da função *isYes*), o programa segue para o nó da esquerda da árvore. Se a resposta capturada no terminal for negativa (*isNo*), o programa segue para o nó da direita da árvore. Há ainda outra condição caso a resposta não comporte as entradas do usuário corretamente, retornará uma mensagem de erro, chamando a pergunta novamente ao usuário até que ele forneça a resposta correta.

Houve também a implementação da função main, que executará o programa interativo, possuindo as seguintes definições:

```
    main :: IO ()
    main = do
        result <- askQuestion decisionTree
        putStrLn $ "Resultado: " ++ result
        putStrLn "Digite 'sair' para terminar ou pressione Enter para
reiniciar."
        continue <- getLine
        if normalize continue /= "SAIR"
            then main
```

```
            else putStrLn "Programa encerrado."
```

Declaramos inicialmente a main, que executa ações vindas do exterior, e logo é pedido para ela fazer com que chamemos a função askQuestion explicada anteriormente, passando a árvore de decisão inteira, a fim de iniciar a interação com o usuário. Esse resultado será o resultado final de Classificado ou Eliminado, a depender das respostas dadas durante a execução; tal resultado é armazenado na variável result, e ele é exibido na tela através do *putStrLn*. Depois disso é escrito com outra *putStrLn* uma mensagem de instrução para sair do programa ou recomeçar a "entrevista".

A linha continue "continue <- getLine" é a que é responsável por ler o que o usuário digitará em seguida. Seguido disso há a condição com a transformação de que se o usuário digitar qualquer coisa diferente de "SAIR", o programa reiniciará do zero, porém se digitar "SAIR", este se encerra de maneira segura.

Atividades do membro - Dagoberto:

Questão 1 - Refatoração: Normalização de Entradas

Implementou funções que tornam o sistema mais robusto ao lidar com as entradas do usuário. Essas funções garantem que a resposta seja interpretada corretamente, independentemente de variações em letras maiúsculas, minúsculas ou espaços em branco.

Amostra de Código:

-- Funções de normalização de entrada

```
trim :: String -> String
trim = dropWhileEnd (==' ') . dropWhile (==' ')

normalize :: String -> String
normalize = map toUpper . trim

isYes :: String -> Bool
isYes ans = any (`isPrefixOf` ansNorm) ["SIM", "S", "Y", "YES"]
```

```
  where ansNorm = normalize ans

isNo :: String -> Bool
isNo ans = any (`isPrefixOf` ansNorm) ["NÃO", "NAO", "N", "NO"]
  where ansNorm = normalize ans
```

Explicação:
Estas funções permitem que o usuário insira respostas como “sim”, “Sim”, “s”, “yes” ou “YES”, pois o sistema converte a entrada para um formato padronizado antes de fazer a verificação. Essa abordagem evita erros de entrada e torna a interação mais flexível.

Atividades do Membro - Caio:

Questão 1 – Refatoração: Visualização da Árvore

Caio foi responsável por aprimorar a experiência do usuário por meio de uma visualização clara e didática da árvore de decisão. Para isso, ele realizou as seguintes mudanças no código:

1. Conversão para Data.Tree:

Caio implementou a função toDataTree, que converte a estrutura interna da árvore binária (BinTree String) para uma árvore compatível com o módulo Data.Tree. Essa conversão é crucial para aproveitar a função drawTree, que desenha a árvore com uma formatação hierárquica clara, facilitando o entendimento do fluxo de decisões.

2. Melhoria na Exibição dos Nós:

Na conversão, cada nó da árvore é encapsulado entre parênteses (por exemplo, "(" ++ s ++ ")"), o que destaca visualmente cada pergunta ou resultado. Essa formatação permite que o usuário identifique facilmente os limites de cada nó, facilitando a visualização do percurso tomado pela decisão.

3. Integração com o Fluxo Interativo:

A função drawTree é invocada a cada iteração dentro da função askQuestion, de forma que, antes de cada nova pergunta, o usuário pode ver o desenho completo da árvore. Essa abordagem não só melhora a usabilidade do programa, como também auxilia na depuração, permitindo identificar rapidamente eventuais incoerências na lógica do fluxo de decisão.

4. Separação da Lógica de Visualização:

Ao encapsular a conversão e a exibição da árvore em funções específicas (toDataTree e a chamada de drawTree dentro do askQuestion), Caio separou a lógica de visualização do processamento das decisões. Essa modularização facilita futuras manutenções e possíveis expansões do sistema, permitindo que alterações na apresentação gráfica não afetem a lógica central do programa.

Amostra de Código:

```
import qualified Data.Tree as T
import Data.Tree (drawTree)
```

-- Converte a árvore binária para uma árvore do módulo Data.Tree

```
toDataTree :: BinTree String -> T.Tree String
toDataTree (Leaf s)       = T.Node ("(" ++ s ++ ")") []
toDataTree (Node s l r)   = T.Node ("(" ++ s ++ ")") [toDataTree l,
toDataTree r]
```

-- Exemplo de visualização da árvore:

```
putStrLn $ drawTree (toDataTree decisionTree)
```

Explicação:

A conversão para o formato Data.Tree permite utilizar a função drawTree, que gera uma representação visual hierárquica da árvore no terminal. Com os nós encapsulados por parênteses, o usuário pode facilmente diferenciar cada pergunta e

resultado, compreendendo melhor o caminho percorrido durante a interação. Essa mudança não só melhora a estética e a clareza do programa, mas também auxilia na identificação e correção de possíveis erros lógicos no fluxo de decisão.

Além disso, a integração desta visualização no fluxo interativo (chamada logo no início da função askQuestion) reforça a transparência do processo, tornando a experiência do usuário mais intuitiva e facilitando o acompanhamento da decisão tomada pelo sistema.

3. Comparação entre as Versões de Código

Versão Inicial desenvolvida pelo membro Emerson, Gustavo e Pablo:

-- Aqui só definimos o tipo de dado da árvore:

```
data BinTree a = Leaf a | Node a (BinTree a) (BinTree a)
```

-- Árvore de decisão para definir o que é nó de cada qual:

```
decisionTree :: Tree String
decisionTree =
    Node "Fez Poscomp?"
        (Node "Nota do IRA >= 7?"
            (Node "Nota da Entrevista >= 7?"
                (Node "Nota da Prova de Títulos >= 5?"
                    (Leaf "Classificado")
                    (Leaf "Eliminado"))
                (Node "Nota da Prova de Títulos >= 7?"
                    (Leaf "Classificado")
                    (Leaf "Eliminado")))
            (Node "Nota da Entrevista >= 9?"
                (Node "Nota da Prova de Títulos >= 7?"
                    (Leaf "Classificado")
                    (Leaf "Eliminado"))
                (Node "Nota da Prova de Títulos >= 8?"
                    (Leaf "Classificado")
                    (Leaf "Eliminado"))))
        (Node "Nota do IRA >= 8?"
            (Node "Nota da Entrevista >= 8?"
```

```
                (Node "Nota da Prova de Títulos >= 8?"
                    (Leaf "Classificado")
                    (Leaf "Eliminado"))
                (Leaf "Eliminado"))
            (Leaf "Eliminado"))
```

-- Aqui vamos navegar pela árvore baseado na decisão:

```
askQuestion :: BinTree String -> IO String
askQuestion (Leaf result) = return result
askQuestion (Node question left right) = do
    putStrLn question
    answer <- getLine
    if answer == "SIM" then askQuestion left
    else if answer == "NÃO" then askQuestion right
    else do
        putStrLn "Entrada inválida! Responda com SIM ou NÃO."
        askQuestion (Node question left right)
```

-- Função principal para executar o programa interativo:

```
main :: IO ()
main = do
    result <- askQuestion decisionTree
    putStrLn $ "Resultado: " ++ result
    -- Aqui vai continuar pra sempre enquanto não digitar sair
    putStrLn "Digite 'sair' para terminar ou pressione Enter para
continuar."
    continue <- getLine
    if continue /= "sair"
        then main
        else putStrLn "Programa encerrado."
```

A função askQuestion apenas verificava se a resposta era exatamente "SIM" ou "NÃO", sem aceitar variações.

A visualização da árvore era inexistente, e a navegação não mostrava o caminho seguido.

Versão Refatorada:

```haskell
{-# LANGUAGE OverloadedStrings #-}
import qualified Data.Tree as T
import Data.Tree (drawTree)
import Data.Char (toUpper)
import Data.List (isPrefixOf, dropWhileEnd)
```

-- Definição da árvore binária

```haskell
data BinTree a = Leaf a | Node a (BinTree a) (BinTree a)
```

-- Árvore de decisão

```haskell
decisionTree :: BinTree String
decisionTree =
    Node "Fez Poscomp?"
        (Node "Nota do IRA >= 7?"
            (Node "Nota da Entrevista >= 7?"
                (Node "Nota da Prova de Títulos >= 5?"
                    (Leaf "Classificado")
                    (Leaf "Eliminado"))
                (Node "Nota da Prova de Títulos >= 7?"
                    (Leaf "Classificado")
                    (Leaf "Eliminado")))
            (Node "Nota da Entrevista >= 9?"
                (Node "Nota da Prova de Títulos >= 7?"
                    (Leaf "Classificado")
                    (Leaf "Eliminado"))
                (Node "Nota da Prova de Títulos >= 8?"
                    (Leaf "Classificado")
                    (Leaf "Eliminado"))))
        (Node "Nota do IRA >= 8?"
            (Node "Nota da Entrevista >= 8?"
                (Node "Nota da Prova de Títulos >= 8?"
                    (Leaf "Classificado")
                    (Leaf "Eliminado"))
                (Leaf "Eliminado"))
            (Leaf "Eliminado"))
```

-- Converte nossa árvore para uma árvore do módulo Data.Tree,

-- incluindo os parênteses para representar cada nó

```
toDataTree :: BinTree String -> T.Tree String
toDataTree (Leaf s)      = T.Node ("(" ++ s ++ ")") []
toDataTree (Node s l r)  = T.Node ("(" ++ s ++ ")") [toDataTree l,
toDataTree r]
```

-- Funções de normalização de entrada

```
trim :: String -> String
trim = dropWhileEnd (==' ') . dropWhile (==' ')

normalize :: String -> String
normalize = map toUpper . trim
```

```
isYes :: String -> Bool
isYes ans = any (`isPrefixOf` ansNorm) ["SIM", "S", "Y", "YES"]
  where ansNorm = normalize ans

isNo :: String -> Bool
isNo ans = any (`isPrefixOf` ansNorm) ["NÃO", "NAO", "N", "NO"]
  where ansNorm = normalize ans
```

-- Função que navega pela árvore interativamente e mostra o desenho atual

```
skQuestion :: BinTree String -> IO String
askQuestion tree = do
    putStrLn "\n--- Árvore atual ---"
    -- Exibe o desenho da árvore usando Data.Tree.drawTree
    putStrLn $ drawTree (toDataTree tree)
    case tree of
        Leaf result -> return result
        Node question left right -> do
            putStrLn question
            answer <- getLine
            if isYes answer then askQuestion left
            else if isNo answer then askQuestion right
            else do
                putStrLn "Entrada inválida! Responda com uma variação
de SIM ou NÃO."
                askQuestion tree
```

-- Função principal para executar o programa interativo

```
main :: IO ()
main = do
```

```
    result <- askQuestion decisionTree
    putStrLn $ "Resultado: " ++ result
    putStrLn "Digite 'sair' para terminar ou pressione Enter para
reiniciar."
    continue <- getLine
    if normalize continue /= "SAIR"
        then main
        else putStrLn "Programa encerrado.”
```

A normalização das respostas foi implementada, permitindo que variações de "sim" e "não" fossem aceitas.

A árvore de decisão foi convertida para uma estrutura compatível com Data.Tree, exibindo a árvore no terminal para o usuário acompanhar o fluxo de perguntas e respostas.

Mudanças Implementadas:

A normalização das entradas foi atribuída ao Dagoberto, garantindo maior flexibilidade no reconhecimento de respostas.

A visualização da árvore foi implementada pelo Caio, utilizando Data.Tree para desenhar a árvore de decisão no terminal.

4. Dificuldades

Integração das Funcionalidades: A integração de diferentes módulos de código foi desafiadora, especialmente nas questões de visualização e normalização de entradas.

Desenho da Árvore: A conversão de uma árvore binária para o formato compatível com Data.Tree e o uso de drawTree exigiu ajustes no código para garantir que a árvore fosse exibida corretamente.

Problemas com as palavras positivas ou negativas: O programa estava com um erro de compilação para geração do executável, porém ao tirar os acentos, tudo funcionou corretamente.

5. Aprendizados

Importância da Normalização: Aprendemos que a normalização das entradas de usuário é fundamental para tornar o sistema mais flexível e robusto.

Visualização de Dados: A implementação de uma representação gráfica das árvores de decisão ajudou a compreender a importância de exibir o fluxo de dados de forma clara e intuitiva.

Trabalho em Equipe: O desenvolvimento do trabalho evidenciou como a colaboração entre os membros e a divisão das tarefas permite uma maior eficiência e a entrega de um produto mais completo e funcional.

Este relatório descreve as atividades de cada membro no desenvolvimento do sistema, detalhando as mudanças significativas entre as versões de código e evidenciando o aprendizado adquirido durante o trabalho.