utad UNIVERSIDADE DE TRÁS-OS-MONTES E ALTO DOURO

Relatório

Programação Funcional

Trabalho realizado por:

Fábio Mota nº79089
Francisco Mota nº78791
Rafael Silva nº78532

Índice

Introdução

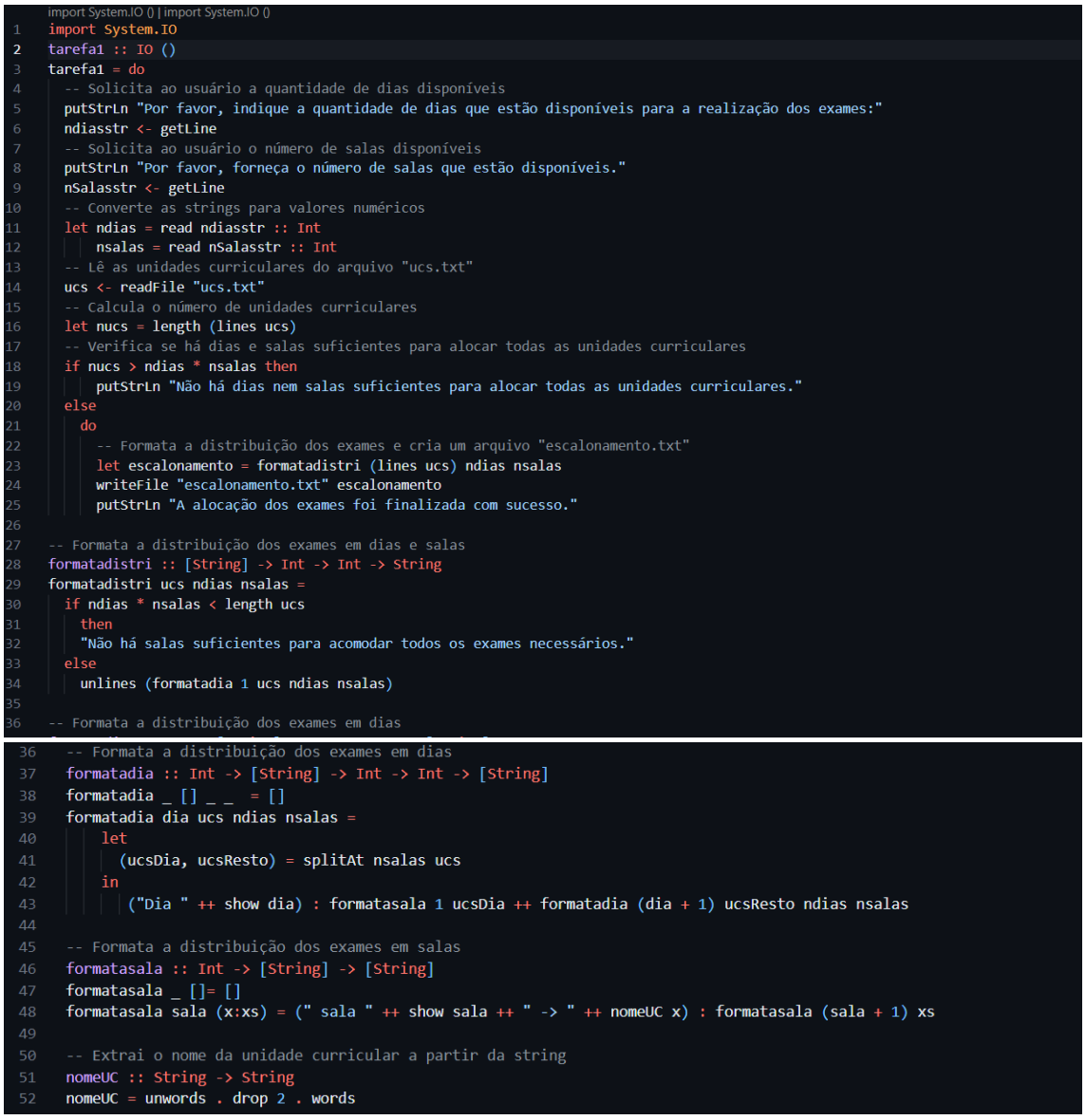
Foi pedido na unidade curricular “Programação Funcional” que fizéssemos um trabalho prático que aborda a gestão e o escalonamento de horários e salas para a realização de exames, no âmbito de uma instituição de ensino superior. Em que já tínhamos realizado a parte 1 em que consistia no desenvolvimento de soluções que permitam a gestão e visualização da informação disponível. Esta informação é relativa aos alunos, às unidades curriculares consideradas e à inscrição de alunos nas diversas unidades curriculares.

Já na parte 2 foi solicitado no desenvolvimento de soluções que permitam a proposta do escalonamento de horários e salas para a realização dos exames das diversas unidades curriculares.

Para procedermos á realização deste trabalho fizemos primeiro a análise e a compreensão dos ficheiros e relacionamos as nossas ideias e fizemos o código, separado pelas tarefas do enunciado.

Tarefa 1

Na tarefa 1 foi solicitado que criássemos um arquivo que contenha o escalonamento dos exames para as unidades curriculares em uma época de exames. O escalonamento deve levar em consideração o dia e a sala de cada exame. Caso não haja salas e dias suficientes, será exibido um aviso no terminal.



```haskell
import System.IO () | import System.IO ()
import System.IO
tarefa1 :: IO ()
tarefa1 = do
  -- Solicita ao usuário a quantidade de dias disponíveis
  putStrLn "Por favor, indique a quantidade de dias que estão disponíveis para a realização dos exames:"
  ndiasstr <- getLine
  -- Solicita ao usuário o número de salas disponíveis
  putStrLn "Por favor, forneça o número de salas que estão disponíveis."
  nSalasstr <- getLine
  -- Converte as strings para valores numéricos
  let ndias = read ndiasstr :: Int
      nsalas = read nSalasstr :: Int
  -- Lê as unidades curriculares do arquivo "ucs.txt"
  ucs <- readFile "ucs.txt"
  -- Calcula o número de unidades curriculares
  let nucs = length (lines ucs)
  -- Verifica se há dias e salas suficientes para alocar todas as unidades curriculares
  if nucs > ndias * nsalas then
      putStrLn "Não há dias nem salas suficientes para alocar todas as unidades curriculares."
  else
    do
      -- Formata a distribuição dos exames e cria um arquivo "escalonamento.txt"
      let escalonamento = formatadistri (lines ucs) ndias nsalas
      writeFile "escalonamento.txt" escalonamento
      putStrLn "A alocação dos exames foi finalizada com sucesso."

-- Formata a distribuição dos exames em dias e salas
formatadistri :: [String] -> Int -> Int -> String
formatadistri ucs ndias nsalas =
  if ndias * nsalas < length ucs
    then
    "Não há salas suficientes para acomodar todos os exames necessários."
  else
    unlines (formatadia 1 ucs ndias nsalas)

-- Formata a distribuição dos exames em dias
  -- Formata a distribuição dos exames em dias
  formatadia :: Int -> [String] -> Int -> Int -> [String]
  formatadia _ [] _ _  = []
  formatadia dia ucs ndias nsalas =
      let
        (ucsDia, ucsResto) = splitAt nsalas ucs
      in
          ("Dia " ++ show dia) : formatasala 1 ucsDia ++ formatadia (dia + 1) ucsResto ndias nsalas

  -- Formata a distribuição dos exames em salas
  formatasala :: Int -> [String] -> [String]
  formatasala _ []= []
  formatasala sala (x:xs) = (" sala " ++ show sala ++ " -> " ++ nomeUC x) : formatasala (sala + 1) xs

  -- Extrai o nome da unidade curricular a partir da string
  nomeUC :: String -> String
  nomeUC = unwords . drop 2 . words
```

Para solucionar a tarefa 1, realizou se o seguinte:

O procedimento começa por solicitar ao usuário a quantidade de dias disponíveis e o número de salas disponíveis. Essas informações são essenciais para determinar a capacidade de alocação dos exames.

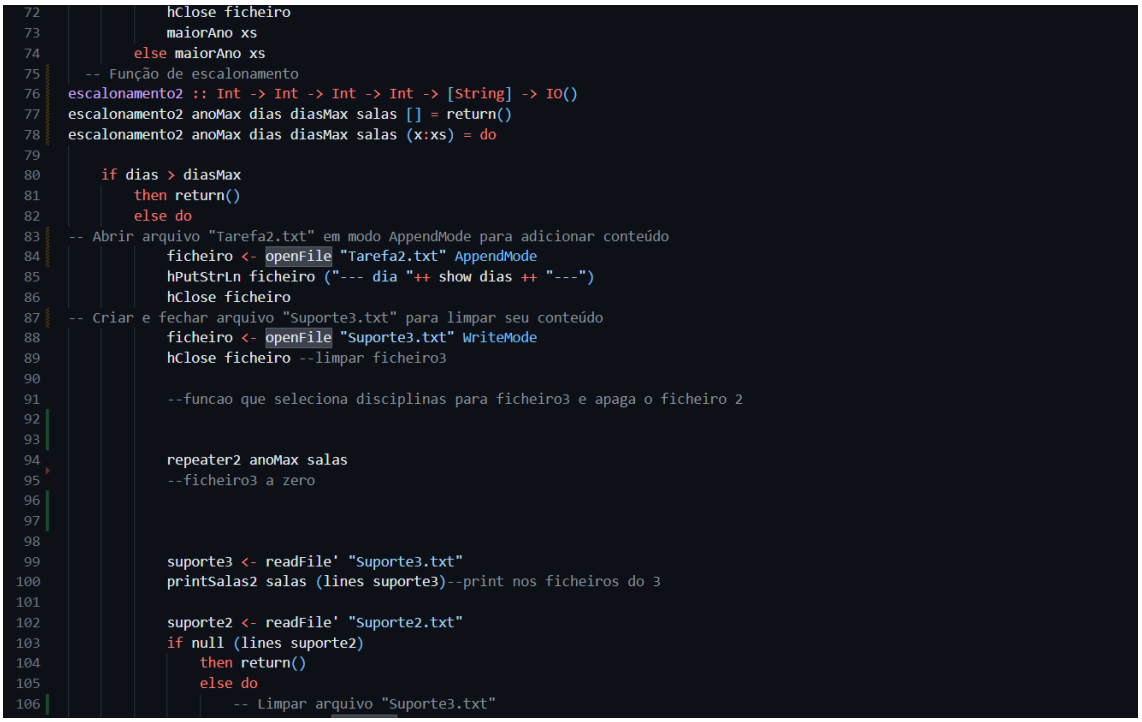
Em seguida, o código lê as informações das unidades curriculares a partir do arquivo "ucs.txt". Esse arquivo contém uma lista de unidades curriculares, cada uma com seu número, dia e nome. Com base no número de unidades curriculares lidas, o código calcula o número total de unidades curriculares. Em seguida, verifica se há dias e salas suficientes para alocar todas as unidades curriculares. Se a capacidade disponível for insuficiente, o procedimento é interrompido e exibe uma mensagem de erro. Se houver dias e salas suficientes, o código prossegue com a formatação do escalonamento dos exames. É chamada a função *formatadistri*, que organiza as unidades curriculares em dias e salas. Dentro da função *formatadistri*, é verificado se o número de salas disponíveis é suficiente para acomodar todas as unidades curriculares. Se não for, a função retorna uma mensagem de erro indicando que não há salas suficientes. Se houver salas suficientes, a função *formatadistri* chama a função *formatadia* para distribuir os exames nas salas em cada dia. A função *formatadia* divide a lista de unidades curriculares em sublistas, cada uma com o número de unidades curriculares correspondente ao número de salas disponíveis por dia. Para cada dia, a função formatadia chama a função formatasala, que formata a distribuição dos exames por sala. A função formatasala recebe o número da sala e a lista de unidades curriculares correspondentes ao dia, e retorna uma lista com a formatação adequada para cada sala.

Após a formatação do escalonamento dos exames, o resultado é escrito no arquivo "escalonamento.txt". Esse arquivo conterá o escalonamento dos exames por dia e sala. Por fim, uma mensagem de conclusão é exibida, informando que a alocação dos exames foi finalizada com sucesso.

Tarefa 2

Na tarefa 2 foi solicitado que criássemos uma solução de escalonamento de exames que garanta a restrição de não haver mais de um exame de UCs do mesmo ano no mesmo dia. O objetivo é criar um cronograma organizado e sem conflitos para a realização dos exames.

```haskell
import System.IO
import Data.List (nub)
import Prelude
      You, há 5 horas • Atualização do codigo
--tarefa2
tarefa2 :: IO()
tarefa2 = do
    conteudoDisciplina <- readFile "ucs.txt"
    -- Criar e fechar arquivo "Tarefa2.txt" para limpar seu conteúdo
    ficheiro <- openFile "Tarefa2.txt" WriteMode
    hClose ficheiro
    ---- Criar e fechar arquivo "Suporte2.txt" para limpar seu conteúdo
    ficheiro <- openFile "Suporte2.txt" WriteMode
    hClose ficheiro
 -- Criar e fechar arquivo "Suporte3.txt" para limpar seu conteúdo
    ficheiro <- openFile "Suporte3.txt" WriteMode
    hClose ficheiro
 -- Criar arquivo "Suporte.txt" com valor inicial 0
    ficheiro <- openFile "Suporte.txt" WriteMode
    hPutStrLn ficheiro "0"
    hClose ficheiro
    -- Descobrir qual é o ano mais alto e escrever no arquivo "Suporte.txt"

    copyFile (lines conteudoDisciplina)

    suporte2 <- readFile' "Suporte2.txt"
    maiorAno (lines suporte2)
    let suporte = readFile' "Suporte.txt"
    maiorAnoString <- suporte
    let maiorAnoInt = read maiorAnoString :: Int -- maior ano no ficheiro


    -- Pedir input do usuário
    putStrLn "indique numero de dias em que o exame pode ocorrer"
    dias <- getLine
    putStrLn "indique o numero de salas disponiveis por dia"
    salas <- getLine
    -- Obter o número de disciplinas no arquivo
    let n_disciplinas = length (lines conteudoDisciplina) -- numero de disciplinas é o numero de linhas no ficheiro
    --convercao-- para int
    let n_dias = read dias :: Int
    let n_salas = read salas :: Int
   -- Chamar a função de escalonamento
    escalonamento2 maiorAnoInt 1 n_dias n_salas (lines conteudoDisciplina)
-- Ler o arquivo "Suporte2.txt" para verificar se todas as disciplinas foram alocadas
    suporte2 <- readFile' "Suporte2.txt"
      You, há 5 horas • Atualização do codigo
    if null (lines suporte2)
        then return()
        else do
            putStrLn "Tempo insuficiente para acomodar todos os exames"
            return()
--funcoes tarefa 2
-- Função para encontrar o maior ano curricular no arquivo "Suporte2.txt"
maiorAno :: [String] -> IO()
maiorAno [] = return()
maiorAno (x:xs) = do

    let suporte = readFile' "Suporte.txt"

    suporteString <-  suporte
    let suporteInt = read suporteString :: Int

    let numero = head(tail(words x))
    let numeroInt = read numero :: Int

    if numeroInt > suporteInt
        then do
            ficheiro <- openFile "Suporte.txt" WriteMode
            hPrint ficheiro numeroInt
            hClose ficheiro
```



```
            hClose ficheiro
            maiorAno xs
        else maiorAno xs
  -- Função de escalonamento
escalonamento2 :: Int -> Int -> Int -> Int -> [String] -> IO()
escalonamento2 anoMax dias diasMax salas [] = return()
escalonamento2 anoMax dias diasMax salas (x:xs) = do

    if dias > diasMax
        then return()
        else do
-- Abrir arquivo "Tarefa2.txt" em modo AppendMode para adicionar conteúdo
            ficheiro <- openFile "Tarefa2.txt" AppendMode
            hPutStrLn ficheiro ("--- dia "++ show dias ++ "---")
            hClose ficheiro
-- Criar e fechar arquivo "Suporte3.txt" para limpar seu conteúdo
            ficheiro <- openFile "Suporte3.txt" WriteMode
            hClose ficheiro --limpar ficheiro3

            --funcao que seleciona disciplinas para ficheiro3 e apaga o ficheiro 2


            repeater2 anoMax salas
            --ficheiro3 a zero



            suporte3 <- readFile' "Suporte3.txt"
            printSalas2 salas (lines suporte3)--print nos ficheiros do 3

            suporte2 <- readFile' "Suporte2.txt"
            if null (lines suporte2)
                then return()
                else do
                    -- Limpar arquivo "Suporte3.txt"
```

Para solucionar esta tarefa, realizamos o seguinte:

O programa começa por importar os módulos necessários do Haskell, como System.IO e Data.List

Em seguida, há uma definição da função tarefa2, que representa a tarefa principal do programa. Essa função é do tipo IO o que quer dizer que ela realiza operações de entrada e saída. A função tarefa2 inicia a ler o conteúdo do arquivo "ucs.txt" usando a função readFile. O conteúdo lido é armazenado na variável conteudoDisciplina.

Em seguida, o programa cria e limpa o conteúdo de vários arquivos, como "Tarefa2.txt", "Suporte2.txt", "Suporte3.txt" e "Suporte.txt".

Após a limpeza dos arquivos, o programa lê o conteúdo do arquivo "Suporte2.txt" usando a função readFile. O conteúdo lido é armazenado na variável suporte2.

O programa busca o maior ano curricular encontrado no arquivo "Suporte2.txt" chamando a função maiorAno e passando as linhas lidas anteriormente. Essa função percorre cada linha do arquivo, compara o ano curricular presente na linha com o valor armazenado no arquivo "Suporte.txt" e atualiza o valor no arquivo se o ano for maior.

Em seguida, o programa solicita entrada do usuário para o número de dias em que o exame pode ocorrer e o número de salas disponíveis por dia. Os valores são lidos e armazenados nas variáveis dias e salas, respetivamente.

O programa determina o número de disciplinas no arquivo lido contando o número de linhas em conteudoDisciplina. Os valores lidos para dias, salas e o número de disciplinas são convertidos de String para Int usando a função read.

Em seguida, o programa chama a função escalonamento2 passando o maior ano curricular encontrado, o número do primeiro dia, o número total de dias, o número de salas e o conteúdo das disciplinas. Essa função é responsável pelo algoritmo de escalonamento propriamente dito.

Após a chamada da função escalonamento2, o programa lê o conteúdo do arquivo "Suporte2.txt" novamente para verificar se todas as disciplinas foram alocadas corretamente. Se houver disciplinas restantes no arquivo, o programa exibe uma mensagem indicando que o tempo foi insuficiente para acomodar todos os exames.

Tarefa 3

Na tarefa 3 consiste em criar um código que exiba no terminal as incompatibilidades entre cada par de UCs, considerando o número de alunos inscritos em cada par. Uma incompatibilidade é medida pelo número de alunos que estão inscritos simultaneamente em ambos os pares de UCs.

```haskell
import Data.List (nub, sort, subsequences)
import System.IO()
import Control.Exception (catch, IOException)
import Data.Char(isSpace)

-- Função que lê as inscrições do arquivo e retorna uma lista de tuplas (aluno, UC)
lerinscricoes :: String -> IO [(String, Int)]
lerinscricoes ficheiro = do
  conteudo <- readFile ficheiro
  let inscricoes = lines conteudo
  return $ map analiseentrada inscricoes
  where
    -- Função auxiliar para analisar cada linha do arquivo e criar uma tupla (aluno, UC)      You, há 5 horas • Atualização do codigo
    analiseentrada :: String -> (String, Int)
    analiseentrada entrada =
        case words entrada of
            [aluno, uc] -> (aluno, read uc)
            _ -> ("aluno desconhecido", 0)
        where
            -- Função auxiliar para remover espaços em branco no início e no fim de uma string
            trim :: String -> String
            trim = f . f
                where
                    f = reverse . dropWhile isSpace

-- Função que calcula as incompatibilidades entre cada par de UCs
calcularincompatibilidades :: [(String, Int)] -> [(Int, Int, Int)]
calcularincompatibilidades inscricoes =
  let ucs = sort $ nub $ map snd inscricoes -- Obtém a lista de UCs sem repetições e em ordem crescente
      pairs = [(x, y) | x <- ucs, y <- ucs, x < y] -- Cria todas as combinações possíveis de pares de UCs
  in
    map (\(x, y) -> (x, y, contarIncompatibilidades x y inscricoes)) pairs
  where
    -- Função auxiliar para contar o número de alunos incompatíveis entre duas UCs
    contarIncompatibilidades :: Int -> Int -> [(String, Int)] -> Int
    contarIncompatibilidades uc1 uc2 inscricoes =
      let alunosUC1 = filter (\(_, uc) -> uc == uc1) inscricoes -- Filtra os alunos inscritos na UC 1
          alunosUC2 = filter (\(_, uc) -> uc == uc2) inscricoes -- Filtra os alunos inscritos na UC 2
          alunosEmComum = length $ nub $ map fst $ filter (\(aluno, _) -> aluno `elem` map fst alunosUC2) alunosUC1 -- Conta os alunos em comum
      in
        alunosEmComum

-- Função que imprime as incompatibilidades na tela
printIncompatibilidades :: [(Int, Int, Int)] -> IO ()
printIncompatibilidades imcompatibilidades = do
  putStrLn "Conflitos entre combinações de unidades curriculares:"
  mapM_ (\(uc1, uc2, incompatibilidade) -> putStrLn $ "UC " ++ show uc1 ++ " e UC " ++ show uc2 ++ ": " ++ show incompatibilidade) imcompatibilidade

main :: IO ()
main = do
  inscricoes <- lerinscricoes "inscricoes.txt" -- Lê as inscrições do arquivo
  let imcompatibilidades = calcularincompatibilidades inscricoes -- Calcula as incompatibilidades entre as UCs
  printIncompatibilidades imcompatibilidades -- Imprime as incompatibilidades

```

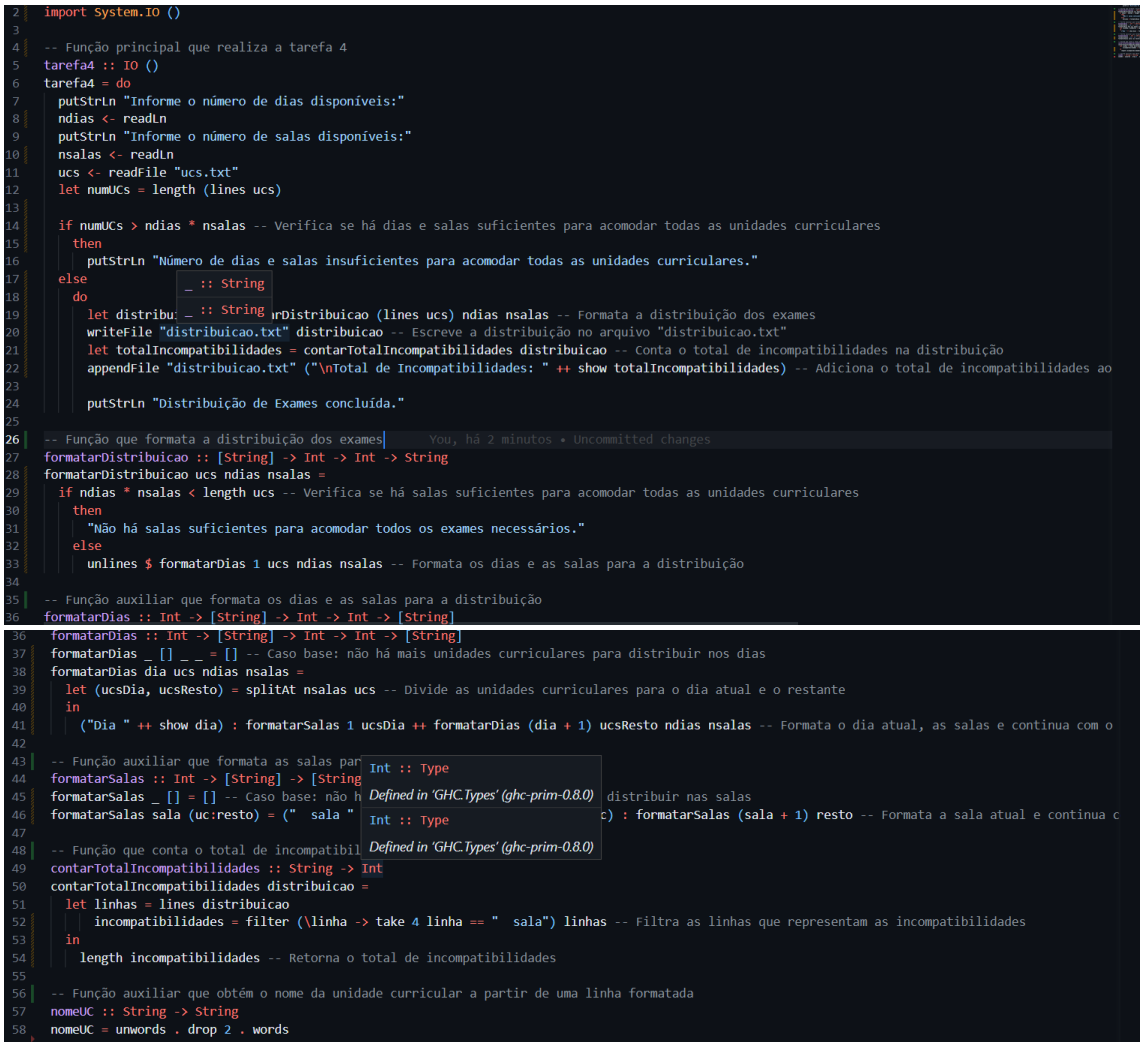
O código importa os módulos necessários, como Data.List, System.IO, Control.Exception e Data.Char, que fornecem funcionalidades adicionais utilizadas no código. Começa com a função ‘lerinscricoes’ essa função recebe como entrada o nome do arquivo de inscrições (ficheiro) e lê o conteúdo do arquivo. Em seguida, analisa cada linha do arquivo para criar uma lista de tuplas no formato (aluno, UC), onde aluno é o nome do aluno e UC é o código da unidade curricular. A função retorna a lista de tuplas. A seguir vai para a função ‘calcularincompatibilidades’ essa função recebe a lista de inscrições em UCs e calcula as incompatibilidades entre todas as combinações possíveis de pares de UCs. Primeiro, ela extrai a lista de códigos de UCs (ucs) sem repetições e em ordem crescente. Em seguida, ela cria todas as combinações possíveis de pares de UCs (pairs). Para cada par de UCs, a função chama a função auxiliar contarIncompatibilidades para contar o número de alunos que estão inscritos em ambas as UCs. O resultado é uma lista de tuplas no formato (UC1, UC2, incompatibilidade), onde UC1 e UC2 são os códigos das UCs e incompatibilidade é o número de alunos em comum. A função retorna essa lista.
A seguir a função “printIncompatibilidades” essa função recebe a lista de incompatibilidades entre as UCs e imprime na tela os pares de UCs e o número de alunos incompatíveis. Ela utiliza a função “putStrLn” para imprimir cada linha.

A função principal “main” essa é a função principal do programa. Ela chama a função lerinscricoes para ler as inscrições do arquivo "inscricoes.txt" e armazena o resultado na variável inscricoes. Em seguida, chama a função calcularincompatibilidades passando as inscrições como argumento e armazena o resultado na variável imcompatibilidades. Por fim, chama a função printIncompatibilidades para imprimir na tela as incompatibilidades encontradas.

Em resumo, o código lê as inscrições dos alunos em UCs a partir de um arquivo, calcula as incompatibilidades entre as diferentes combinações de UCs e imprime na tela os pares de UCs e o número de alunos incompatíveis.

Tarefa 4

A tarefa 4 consistiu em adicionar ao arquivo que contém a proposta de escalonamento dos exames a informação sobre o número total de incompatibilidades que essa solução apresenta, ou seja, o número de alunos que terão exames de mais de uma UC no mesmo dia. O objetivo é incluir essa informação no arquivo de escalonamento.



```haskell
import System.IO ()

-- Função principal que realiza a tarefa 4
tarefa4 :: IO ()
tarefa4 = do
  putStrLn "Informe o número de dias disponíveis:"
  ndias <- readLn
  putStrLn "Informe o número de salas disponíveis:"
  nsalas <- readLn
  ucs <- readFile "ucs.txt"
  let numUCs = length (lines ucs)

  if numUCs > ndias * nsalas -- Verifica se há dias e salas suficientes para acomodar todas as unidades curriculares
    then
      putStrLn "Número de dias e salas insuficientes para acomodar todas as unidades curriculares."
  else
    do
      let distribu _ :: String
                   _ :: String
      writeFile "distribuicao.txt" distribuicao -- Escreve a distribuição no arquivo "distribuicao.txt"
      let totalIncompatibilidades = contarTotalIncompatibilidades distribuicao -- Conta o total de incompatibilidades na distribuição
      appendFile "distribuicao.txt" ("\nTotal de Incompatibilidades: " ++ show totalIncompatibilidades) -- Adiciona o total de incompatibilidades ao

      putStrLn "Distribuição de Exames concluída."

-- Função que formata a distribuição dos exames      You, há 2 minutos • Uncommitted changes
formatarDistribuicao :: [String] -> Int -> Int -> String
formatarDistribuicao ucs ndias nsalas =
  if ndias * nsalas < length ucs -- Verifica se há salas suficientes para acomodar todas as unidades curriculares
    then
      "Não há salas suficientes para acomodar todos os exames necessários."
    else
      unlines $ formatarDias 1 ucs ndias nsalas -- Formata os dias e as salas para a distribuição

-- Função auxiliar que formata os dias e as salas para a distribuição
formatarDias :: Int -> [String] -> Int -> Int -> [String]
formatarDias :: Int -> [String] -> Int -> Int -> [String]
formatarDias _ [] _ _ = [] -- Caso base: não há mais unidades curriculares para distribuir nos dias
formatarDias dia ucs ndias nsalas =
  let (ucsDia, ucsResto) = splitAt nsalas ucs -- Divide as unidades curriculares para o dia atual e o restante
  in
    ("Dia " ++ show dia) : formatarSalas 1 ucsDia ++ formatarDias (dia + 1) ucsResto ndias nsalas -- Formata o dia atual, as salas e continua com o

-- Função auxiliar que formata as salas par
formatarSalas :: Int -> [String] -> [String
formatarSalas _ [] = [] -- Caso base: não h                                  distribuir nas salas
formatarSalas sala (uc:resto) = ("  sala "                                   c) : formatarSalas (sala + 1) resto -- Formata a sala atual e continua c

-- Função que conta o total de incompatibil
contarTotalIncompatibilidades :: String -> Int
contarTotalIncompatibilidades distribuicao =
  let linhas = lines distribuicao
      incompatibilidades = filter (\linha -> take 4 linha ==  "  sala") linhas -- Filtra as linhas que representam as incompatibilidades
  in
    length incompatibilidades -- Retorna o total de incompatibilidades

-- Função auxiliar que obtém o nome da unidade curricular a partir de uma linha formatada
nomeUC :: String -> String
nomeUC = unwords . drop 2 . words
```

Int :: Type
Defined in 'GHC.Types' (ghc-prim-0.8.0)
Int :: Type
Defined in 'GHC.Types' (ghc-prim-0.8.0)

Para solucionar esta tarefa foi realizado o seguinte:

Começa com a função tarefa4 essa é a função principal que realiza a tarefa. Ela interage com o usuário para obter o número de dias disponíveis (ndias) e o número de salas disponíveis (nsalas). Em seguida, lê o conteúdo do arquivo "ucs.txt" que contém as UCs. A variável numUCs é atribuída ao número de UCs presentes no arquivo.Se o número de UCs for maior do que o total de dias multiplicado pelo total de salas (ndias * nsalas), significa que não há dias e salas suficientes para acomodar todas as UCs. Nesse caso, é exibida uma mensagem informando a insuficiência de dias e salas. Caso contrário, a função continua com a distribuição dos exames.

Depois chamamos a função “formatarDistribuicao”, essa função recebe a lista de UCs, o número de dias disponíveis (ndias) e o número de salas disponíveis (nsalas). Ela verifica se o número total de salas é insuficiente para acomodar todas as UCs. Se for o caso, retorna uma mensagem informando a insuficiência de salas. Caso contrário, chama a função formatarDias para formatar a distribuição dos exames.

A função “formatarDias”, essa função auxiliar recebe um número de dia, a lista de UCs restantes, o número de dias disponíveis (ndias) e o número de salas disponíveis

(nsalas). Ela divide a lista de UCs em duas partes: as UCs que serão distribuídas no dia atual (ucsDia) e as UCs restantes (ucsResto). Em seguida, formata o dia atual e as salas chamando a função formatarSalas e continua formatando os próximos dias chamando recursivamente a função formatarDias com o próximo número de dia e a lista de UCs restantes.

Função formatarSalas: Essa função auxiliar recebe um número de sala e a lista de UCs a serem distribuídas nessa sala. Ela formata o número da sala e o nome da UC chamando a função nomeUC e continua formatando as próximas salas chamando recursivamente a função formatarSalas com o próximo número de sala e a lista de UCs restantes.

A função “contarTotalIncompatibilidades”, essa função recebe a distribuição dos exames formatada como uma string (distribuicao). Ela conta o total de incompatibilidades na distribuição, filtrando as linhas que representam as incompatibilidades e retornando o número de linhas filtradas.

A função nomeUC, essa função auxiliar recebe uma linha formatada contendo uma UC e retorna o nome da UC. Ela realiza a separação da linha em palavras, descarta as duas primeiras palavras que representam o número da sala e o caractere "-" e retorna o restante das palavras juntas como o nome da UC.

Em resumo, o código solicita ao usuário o número de dias e salas disponíveis, lê o arquivo "ucs.txt" contendo as UCs, verifica se há dias e salas suficientes para acomodar todas as UCs e, em seguida, formata a distribuição dos exames nos dias e salas disponíveis. A distribuição é escrita no arquivo "distribuicao.txt", juntamente com o total de incompatibilidades encontradas.

Conclusão

Em conclusão, o desenvolvimento de soluções de escalonamento de horários e salas para a realização de exames das diversas unidades curriculares utilizando a linguagem de programação Haskell mostrou-se eficaz e robusto. A abordagem funcional de Haskell proporcionou uma maneira clara e concisa de expressar algoritmos complexos, resultando em um código legível e de fácil manutenção.

Ao longo do trabalho, foram exploradas várias técnicas e conceitos de programação em Haskell, como manipulação de arquivos, leitura e escrita de dados, processamento de listas e recursão. Essas habilidades foram aplicadas para criar um sistema capaz de realizar o escalonamento de exames, levando em consideração o número de dias disponíveis e a quantidade de salas disponíveis em cada dia.

Além disso, a solução implementada também considerou a restrição do ano curricular das disciplinas, alocando-as de acordo com o maior ano curricular encontrado, garantindo assim uma distribuição equitativa dos exames.

Em suma, o trabalho sobre o escalonamento de horários e salas de exames não foi 100% sucedido, já que o resto das tarefas (tarefa 5 e 6) não foram implementadas de forma operacional.