utad UNIVERSIDADE DE TRÁS-OS-MONTES E ALTO DOURO

TRABALHO PRÁTICO PARTE 2

Programação Funcional | Engenharia Informática

Trabalho Realizado por:

Francisco Ruano, al78474

Índice

1. Introdução

No contexto da unidade curricular de **Programação Funcional**, foi proposto o desafio de desenvolver uma aplicação na linguagem **Haskell** que simule a operação de uma nave alienígena em um espaço tridimensional. Esta aplicação tem como objetivo explorar conceitos fundamentais de programação funcional, como imutabilidade, recursão e a manipulação de estruturas de dados complexas, enquanto modela de forma eficiente o comportamento de naves alienígenas.

A segunda parte do projeto expande as funcionalidades inicialmente propostas, integrando a criação de múltiplas funções para gerir o estado das naves, realizar movimentações no espaço, e executar um conjunto de ações pré-definidas ou interativas.

Neste relatório, são apresentadas detalhadamente as funcionalidades implementadas, os métodos utilizados para resolver problemas específicos, e a lógica que sustenta o comportamento das naves. Destacam-se também as decisões tomadas ao longo do desenvolvimento e os desafios enfrentados, ilustrando como a linguagem Haskell foi impregnada para criar uma solução robusta e eficiente.

2. Tipos de Dados

Inicialmente, foram definidos tipos de dados específicos, utilizados ao longo do código, que representam os estados e movimentações de uma ou mais naves. Estes tipos de dados melhoram a legibilidade do código e facilitam a manipulação de informações complexas de forma clara e concisa.

- **Localização:** Define a localização de uma nave no espaço tridimensional através de uma tupla de três inteiros. Cada elemento da tupla representa uma coordenada (x,y,z), permitindo que a nave seja posicionada em um ponto específico no espaço.

- **EstadoNave:** Representa o estado completo de uma nave, combinando a localização (tipo Localizacao) e o estado de ligação da nave (valor Bool). O valor booleano indica se a nave se encontra "ligada" (*True*) ou "desligada" (*False*).

- **Movimentacao:** Este define o deslocamento de uma nave em cada eixo (x, y, z) em uma única operação de movimento e é representada por uma dupla de inteiros.

- **ID:** Este tipo de dado é um identificador único para cada nave, e facilita-nos na distinção das mesmas, caso haja movimentações ou colisões.

- **ListaNaves:** Este tipo de dado faz a combinação de uma lista de movimentos (*[Movimentacao]*) e o identificador (*ID*) de uma nave especifica.

- **Restricao:** Este indica os valores mínimos e máximos de coordenadas em que uma certa nave se pode mover.

```
type Localizacao = (Int, Int, Int)
type EstadoNave = (Localizacao, Bool)
type Movimentacao = (Int, Int, Int)
type ID = String
type ListaNaves = ([Movimentacao], ID)
type Restricao = ((Int, Int, Int), (Int, Int, Int))
```

Figura 1: Tipos de Dados

3. Funções Auxiliares

initSpace: Esta função é usada para definir os valores mínimos e máximos de coordenadas em que uma certa nave pode voar.

validaRestrições: É uma função que verifica se uma certa movimentação de uma nave se encontra dentro do espaço de valores definidos pela função **initSpace**.

verificaEmbates: É uma função que verifica se duas ou mais naves se encontram no mesmo local (colidiram).

move: É uma função que exibe o movimento das naves

validaMovimento: Esta função efetua a validação do movimento de uma determinada nave, ou seja, verifica se esta se encontra ligada ou não, se as coordenadas são negativas e as suas restrições de navegação.

atualizaAcao: Esta função faz a atualização do estado da nave (Ligado ou Desligado)

processarInstrucoes: Esta função é utilizada para processar e validar instruções que são lidas no ficheiro Alienship.txt.

```haskell
-- Restrições de movimentação e validação da mesma

initSpace :: IO Restricao
initSpace = do
    putStrLn "Digite os valores mínimos (x, y, z):"
    minVals <- readLn :: IO (Int, Int, Int)
    putStrLn "Digite os valores máximos (x, y, z):"
    maxVals <- readLn :: IO (Int, Int, Int)
    let restricao = (minVals, maxVals)
    putStrLn $ "Restrição configurada: " ++ show restricao
    return restricao

validaRestricao :: Restricao -> EstadoNave -> Bool
validaRestricao ((xMin, yMin, zMin), (xMax, yMax, zMax)) ((x, y, z), _) =
    x >= xMin && x <= xMax && y >= yMin && y <= yMax && z >= zMin && z <= zMax

-- Verifica embates

verificaEmbates :: [EstadoNave] -> [String] -> [String]
verificaEmbates estados ids =
    [id1 ++ " e " ++ id2 | ((loc1, _), id1) <- zip estados ids, ((loc2, _), id2) <- zip estados ids, loc1 == loc

-- Movimentações das naves e validação

move :: Movimentacao -> EstadoNave -> EstadoNave
move (dx, dy, dz) ((x, y, z), ligado) = ((x + dx, y + dy, z + dz), ligado)

validaMovimento :: Restricao -> Movimentacao -> EstadoNave -> Either String EstadoNave
validaMovimento restricao movimento estado@(localizacao, ligado)
    | not ligado = Left "Erro: Nave desligada."
    | otherwise =
        let novoEstado@(novaLoc@(x, y, z), _) = move movimento estado
        in if z < 0 then Left "Erro: Coordenada Z negativa."
           else if not (validaRestricao restricao novoEstado) then Left "Erro: Fora da área permitida."
           else Right novoEstado

-- Atualização de Estado

atualizaAcao :: Bool -> EstadoNave -> EstadoNave
atualizaAcao ligacao (localizacao, _) = (localizacao, ligacao)

-- Processamento de Instruções de uma ou mais naves

processarInstrucoes :: Restricao -> EstadoNave -> [String] -> IO EstadoNave
processarInstrucoes restricao estado [] = return estado
processarInstrucoes restricao estado (instr:resto) = do
    let partes = words instr
```

```haskell
-- Processamento de Instruções de uma ou mais naves

processarInstrucoes :: Restricao -> EstadoNave -> [String] -> IO EstadoNave
processarInstrucoes restricao estado [] = return estado
processarInstrucoes restricao estado (instr:resto) = do
    let partes = words instr
    case partes of
        ("init":coords:estadoInicial:_) -> do
            let novaLoc = read coords :: (Int, Int, Int)
            let ligada = read estadoInicial == 1
            let novoEstado = (novaLoc, ligada)
            if validaRestricao restricao novoEstado
                then processarInstrucoes restricao novoEstado resto
                else do
                    putStrLn "Erro: Posição inicial fora da restrição."
                    processarInstrucoes restricao estado resto
        ("move":mov:_) -> do
            let movimento = read mov :: Movimentacao
            case validaMovimento restricao movimento estado of
                Left err -> do
                    putStrLn err
                    processarInstrucoes restricao estado resto
                Right novoEstado -> processarInstrucoes restricao novoEstado resto
        ("acao":"ligar":_) -> processarInstrucoes restricao (atualizaAcao True estado) resto
        ("acao":"desligar":_) -> processarInstrucoes restricao (atualizaAcao False estado) resto
        _ -> do
            --putStrLn "Erro: Instrução inválida."
            processarInstrucoes restricao estado resto

-- Leitura e Manipulação de Ficheiros

salvarFicheiro :: FilePath -> String -> IO ()
salvarFicheiro caminho conteudo = writeFile caminho conteudo

parseFicheiro :: String -> [(ID, [String])]
parseFicheiro conteudo =
    let linhas = filter (not . null) (lines conteudo) --O filter serve para remover linhas vazias
        grupos = groupBy (\a b -> head (words a) == head (words b)) linhas --groupBy para agrupar linhas por I
    in map (\grupo -> (head (words (head grupo)), map (unwords . tail . words) grupo)) grupos

readfile :: IO ()
readfile = do
    putStrLn "Lista de naves:"
    conteudo <- readFile "Alienship.txt"  -- Lê o ficheiro
    let parsed = parseFicheiro conteudo  -- Organiza o conteúdo
    mapM_ (\(id, instrs) -> do
            putStrLn ("\nID: " ++ id)
            mapM_ putStrLn instrs) parsed  -- Exibe cada nave e suas instruções
```

Figura 4: Funções Auxiliares

já

4. Função menu

Esta função exibe um menu iterativo ao utilizador, permitindo-lhe escolher uma das várias opções apresentadas, tais como Listar naves, Executar uma nave, Execução manual de uma ou mais naves e Executar todas as naves.

Dependendo da escolha do utilizador, a função *menu* chama a função apropriada para determinada tarefa e reinicia o menu (Figura 4).

```
-- Menu Principal

menu :: IO ()
menu = do
    restricao <- initSpace
    let loop = do
            putStrLn "\nSelecione a opção: \n1- Listar naves do ficheiro \n2- Executar uma nave \n3- Execução manual de uma ou mais naves \n4- Executar todas as naves \n5- Sair"
            op <- getLine
            case op of
                "1" -> do
                    readfile
                    loop
                "2" -> do
                    executnave restricao
                    loop
                "3" -> do
                    executnaveManual restricao
                    loop
                "4" -> do
                    executarTodasNaves restricao
                    loop
                "5" -> putStrLn "Execução Terminada..."
                _ -> do
                    putStrLn "Opção inválida. Tente novamente."
                    loop
    loop
```

Figura 5: Função menu

```
Selecione a opção:
1- Listar naves do ficheiro
2- Executar ações de uma nave
3- Execução manual
4- Executar todas as naves
5- Sair
```

Figura 6: Função menu (Compilação)

5. Funções de Leitura e Manipulação de Ficheiros

salvarFicheiro: Esta função é uma função que guarda dados (*WriteFile*) do tipo *string* num ficheiro especificado pelo caminho (*FilePath*).

parserFicheiro: Esta função é usada para analisar o conteúdo do ficheiro e organizar os dados, ou seja, agrupa as linhas do ficheiro por *ID* de nave. Esta utiliza a função *lines* para dividir o conteúdo de ficheiro em linhas, remove linhas vazias através do *filter(not . null)*, usa *groupBy* para agrupar linhas por *ID* de nave, e por último cria uma lista de pares (*ID, [String]*), onde o *ID* é o identificador da nave e a lista contem as suas instruções após remover o *ID*.

readfile: Esta função inicialmente começa por ler o conteúdo do ficheiro Alienship.txt, de seguida organiza o conteúdo usando a função **parserFicheiro** e por último exibe os dados consoante o seu *ID*.

```
salvarFicheiro :: FilePath -> String -> IO ()
salvarFicheiro caminho conteudo = writeFile caminho conteudo

parseFicheiro :: String -> [(ID, [String])]
parseFicheiro conteudo =
    let linhas = filter (not . null) (lines conteudo) --O filter serve para remover linhas vaz
        grupos = groupBy (\a b -> head (words a) == head (words b)) linhas --groupBy para agru
    in map (\grupo -> (head (words (head grupo)), map (unwords . tail . words) grupo)) grupos

readfile :: IO ()
readfile = do
    putStrLn "Lista de naves:"
    conteudo <- readFile "Alienship.txt"  -- Lê o ficheiro
    let parsed = parseFicheiro conteudo  -- Organiza o conteúdo
    mapM_ (\(id, instrs) -> do
              putStrLn ("\nID: " ++ id)
              mapM_ putStrLn instrs) parsed  -- Exibe cada nave e suas instruções
```

Figura 2.Funções de Leitura e Manipulação de Ficheiros

```
Lista de naves:

ID: oi98
init (0,10,0) 1
initspace (0,0,0) (50,50,50)
move (30,0,50)
move (0,10,0)
acao desligar

ID: AS543
initspace (0,0,0) (10,10,100)
acao ligar
move (5,0,0)
move (0,10,0)

ID: EE123
move (500,0,0)
```

Figura 3: Listagem das naves (Compilação)

6. Função de execução de uma nave

A função *executnave* tem como objetivo executar as instruções associadas a uma nave específica. O utilizador inicia o processo ao fornecer o ***ID*** da nave que deseja controlar e realiza os seguintes passos:

1. **Leitura do ficheiro de texto**:
   - O ficheiro Alienship.txt é lido, e o seu conteúdo é analisado através da função *parseFicheiro*, que organiza as instruções associadas a cada nave com base no seu identificador (*ID*).
2. **Validação do ID**:
   - A função verifica se o *ID* fornecido pelo utilizador corresponde a uma nave existente no ficheiro. Se o *ID* for inválido, é apresentada uma mensagem de erro.
3. **Processamento das Instruções**:
   - Caso o *ID* seja válido, as instruções da nave são exibidas ao utilizador e processadas uma a uma. O estado inicial da nave é definido como ((0,0,0), False) (posição na origem e desligada).
   - Cada instrução é interpretada e validada. Movimentos, ações de ligar/desligar, ou outras operações são aplicadas ao estado atual da nave, verificando restrições de espaço e possíveis erros.
4. **Resultado Final**:
   - Ao término da execução das instruções, o estado final da nave é exibido no ecrã. Caso ocorram erros durante o processamento, a execução prossegue com as instruções seguintes.
5. **Mensagens de Erro**:
   - Mensagens informativas são exibidas para instruções inválidas ou para erros como restrições violadas, garantindo feedback contínuo ao utilizador.

```haskell
executnave :: Restricao -> IO ()
executnave restricao = do
    putStrLn "Introduza o ID da nave que deseja controlar:"
    idNave <- getLine
    conteudo <- catch (readFile "Alienship.txt") (\e -> do putStrLn ("Erro ao ler o ficheiro: " ++ show (e :: IOException)); return "")
    let parsed = parseFicheiro conteudo
    case lookup idNave parsed of
        Just instrs -> do
            putStrLn $ "\nInstruções encontradas para a nave " ++ idNave ++ ":"
            mapM_ putStrLn instrs
            let estadoInicial = ((0, 0, 0), False)
            estadoFinal <- processarInstrucoes restricao estadoInicial instrs
            putStrLn $ "Estado final da nave " ++ idNave ++ ": " ++ show estadoFinal
        Nothing -> putStrLn $ "ID de nave inválido: " ++ idNave
```

Figura 7: Função executnave

```
Introduza o ID da nave que deseja controlar:
oi98

Instruções encontradas para a nave oi98:
init (0,10,0) 1
initspace (0,0,0) (50,50,50)
move (30,0,50)
move (0,10,0)
acao desligar
Erro: Posição inicial fora da restrição.
Erro: Nave desligada.
Erro: Nave desligada.
Estado final da nave oi98: ((0,0,0),False)
```

Figura 8: Função executnave (Compilação)

```
Introduza o ID da nave que deseja controlar:
oi98

Instruções encontradas para a nave oi98:
init (0,10,0) 1
initspace (0,0,0) (50,50,50)
move (30,0,50)
move (0,10,0)
acao desligar
Estado final da nave oi98: ((30,20,50),False)
```

Figura 9: Função executnave (Compilação)

7. Função de execução manual de uma ou mais naves

A função *executnaveManual* permite ao utilizador controlar manualmente uma ou mais naves, introduzindo comandos de forma interativa. Ela recebe uma restrição espacial, que define os limites das coordenadas permitidas para as naves, e processa ações inseridas pelo utilizador até que este finalize a sessão.

A função começa por apresentar instruções ao utilizador, indicando o formato esperado para os comandos. Estes incluem:

- **ID init (<x>,<y>,<z>)**: Inicializar uma nave com o identificador **ID** e colocá-la na posição **(x, y, z)**.
- **ID move (<dx>,<dy>,<dz>)**: Mover uma nave para uma nova posição com base no deslocamento **(dx, dy, dz).**
- **ID acao ligar**: Ligar a nave, alterando o seu estado para **True**.
- **ID acao desligar**: Desligar a nave, alterando o seu estado para **False**.
- **fim:** Finalizar a execução manual e exibir os estados finais de todas as naves.

```haskell
executnaveManual :: Restricao -> IO ()
executnaveManual restricao = do
    putStrLn "\nIntroduza as ações para uma ou mais naves (digite init, move, acao (ligar ou desligar) e fim para terminar):"
    let loop estados = do
            putStrLn "Digite as ações que deseja ( AB123 init (1,2,3)):"
            entrada <- getLine
            if entrada == "fim"
                then do
                    putStrLn "Execução finalizada. Estados finais:"
                    mapM_ (\(id, estado) -> putStrLn $ "ID: " ++ id ++ " -> Estado: " ++ show estado) estados
                else do
                    let palavras = words entrada
                    case palavras of
                        (id:acao:resto) -> do
                            let valores = unwords resto
                            let estadoAtual = lookup id estados
                            case (acao, estadoAtual) of
                                ("init", _) -> do
                                    let loc = read valores :: Localizacao
                                    let novoEstado = (loc, False)
                                    if validaRestricao restricao novoEstado
                                        then loop ((id, novoEstado) : filter ((/= id) . fst) estados)
                                        else do
                                            putStrLn "Erro: Posição fora das restrições."
                                            loop estados
                                ("move", Just estado) -> do
                                    let mov = read valores :: Movimentacao
                                    case validaMovimento restricao mov estado of
                                        Left err -> do
                                            putStrLn err
                                            loop estados
                                        Right novoEstado -> loop ((id, novoEstado) : filter ((/= id) . fst) estados)
                                ("acao", Just estado) -> case valores of
                                    "ligar" -> loop ((id, atualizaAcao True estado) : filter ((/= id) . fst) estados)
                                    "desligar" -> loop ((id, atualizaAcao False estado) : filter ((/= id) . fst) estados)
                                    _ -> do
                                        putStrLn "Erro: Ação inválida."
                                        loop estados
                                _ -> do
                                    putStrLn "Erro: Nave não inicializada ou comando inválido."
                                    loop estados
                        _ -> do
                            putStrLn "Erro: Formato inválido."
                            loop estados
    loop []
```

Figura 10: Função executnaveManual

```
Introduza as ações para uma ou mais naves (digite init, move, acao (ligar ou desligar) e fim para terminar):
Digite as ações que deseja ( AB123 init (1,2,3)):
UI87 move (2,3,4)
Erro: Nave não inicializada ou comando inválido.
Digite as ações que deseja ( AB123 init (1,2,3)):
UI87 init (2,3,4)
Digite as ações que deseja ( AB123 init (1,2,3)):
UI87 move (2,2,4)
Erro: Nave desligada.
Digite as ações que deseja ( AB123 init (1,2,3)):
UI87 acao ligar
Digite as ações que deseja ( AB123 init (1,2,3)):
UI87 move (2,2,4)
Digite as ações que deseja ( AB123 init (1,2,3)):
fim
Execução finalizada. Estados finais:
ID: UI87 -> Estado: ((4,5,8),True)
```

Figura 11: Função executnaveManual

8. Executar todas as naves

A função *executarTodasNaves* é responsável por processar e executar todas as ações de múltiplas naves definidas em um ficheiro de texto. Essa função permite a execução automática das instruções de cada nave, realizando verificações de colisões e guarda os resultados finais em um novo ficheiro.

1. **Leitura do Ficheiro**:
    - A função lê o ficheiro Alienship.txt, onde as instruções de todas as naves estão armazenadas.
    - Caso ocorra algum erro ao abrir o ficheiro, é exibida uma mensagem informativa, e o programa continua sem processar.

2. **Parsing das Instruções:**
    - As linhas do ficheiro são agrupadas por identificador de nave (*ID*) utilizando a função *parseFicheiro*.
    - O resultado é uma lista de pares (*ID*, [Instruções]), onde cada *ID* é associado às suas ações.

3. **Processamento das Instruções:**
    - Utiliza-se uma estrutura recursiva (*loop*) para iterar sobre todas as naves e processar as suas instruções.
    - Para cada nave:
        - Se a nave não possuir um estado inicial definido, é atribuído o estado ((0, 0, 0), False).
        - As instruções são processadas uma a uma utilizando a função *processarInstrucoes*, atualizando o estado da nave.
    - Os estados atualizados são armazenados em uma lista.

4. **Verificação de Colisões:**
    - Após processar todas as naves, os estados finais são analisados utilizando a função *verificaEmbates* para identificar colisões entre naves (naves ocupando a mesma posição).
    - Se houver colisões, uma mensagem detalhada com os *IDs* das naves envolvidas é exibida.

5. **Armazenamento dos Resultados:**

   - Os estados finais de todas as naves são formatados e salvos no ficheiro Alienship_Atualizado.txt.
   - Cada linha do ficheiro contém o ID da nave, sua localização e seu estado de ligação (ligada/desligada).

6. **Mensagens ao Utilizador:**

   - O progresso da execução é exibido no ecrã, incluindo mensagens informando:
     - Os estados finais das naves.
     - A detecção ou não de embates.
     - Sucesso ou erro ao salvar o ficheiro.

```haskell
executarTodasNaves :: Restricao -> IO ()
executarTodasNaves restricao = do
    putStrLn "\nExecutando ações de todas as naves do ficheiro..."
    conteudo <- catch (readFile "Alienship.txt") (\e -> do putStrLn ("Erro ao ler o ficheiro: " ++ show (e :: IOException)); return "")
    let parsed = parseFicheiro conteudo
    let loop [] estados = do
            putStrLn "\nExecução finalizada. Estados finais:"
            mapM_ (\(id, estado) -> putStrLn $ "ID: " ++ id ++ " -> Estado: " ++ show estado) estados
            let embates = verificaEmbates (map snd estados) (map fst estados)
            if not (null embates)
                then putStrLn $ "Embates detectados entre as naves: " ++ unwords embates
                else putStrLn "Nenhum embate detectado."
            let estadosFinais = unlines [id ++ " " ++ show loc ++ " " ++ show ligado | (id, (loc, ligado)) <- estados]
            salvarFicheiro "Alienship_Atualizado.txt" estadosFinais
        loop ((id, instrs):resto) estados = do
            let estadoInicial = lookup id estados
            novoEstado <- case estadoInicial of
                Nothing -> processarInstrucoes restricao ((0, 0, 0), False) instrs
                Just estado -> processarInstrucoes restricao estado instrs
            loop resto ((id, novoEstado) : filter ((/= id) . fst) estados)
    loop parsed []
```

Figura 12: Função executarTodasNaves

```
Executando ações de todas as naves do ficheiro...
Erro: Posição inicial fora da restrição.
Erro: Nave desligada.
Erro: Nave desligada.
Erro: Fora da área permitida.
Erro: Fora da área permitida.
Erro: Nave desligada.

Execução finalizada. Estados finais:
ID: EE123 -> Estado: ((0,0,0),False)
ID: AS543 -> Estado: ((0,0,0),True)
ID: oi98 -> Estado: ((0,0,0),False)
Embates detectados entre as naves: EE123 e AS543 EE123 e oi98 AS543 e EE123 AS543 e oi98 oi98 e EE123 oi98 e AS543
```

Figura 13: Função executarTodasNaves (Compilação)

```
Executando ações de todas as naves do ficheiro...

Execução finalizada. Estados finais:
ID: CD456 -> Estado: ((1,1,1),True)
ID: AB123 -> Estado: ((1,1,1),True)
Embates detectados entre as naves: CD456 e AB123 AB123 e CD456
```

Figura 14: Função executarTodasNaves (Compilação)

9. Conclusão

Em conclusão, a realização deste trabalho prático revelou-se fundamental para aprofundar a compreensão da linguagem de programação *Haskell* e das suas particularidades no contexto da programação funcional. Através do desenvolvimento de funcionalidades para a operação de uma nave alienígena, foi possível explorar conceitos-chave, como a imutabilidade de dados, recursão e a construção de funções puras e auxiliares, além de aprender a manipular estruturas de dados complexas.

Este trabalho também demonstrou como a linguagem *Haskell*, apesar da sua abordagem rigorosa e abstrata, pode ser eficaz na resolução de problemas concretos, promovendo um pensamento lógico e estruturado.

A experiência adquirida contribuiu para consolidar habilidades analíticas, aprimorar o raciocínio algorítmico e aplicar conceitos de programação funcional a cenários práticos, reforçando o valor do *Haskell* para desenvolver soluções robustas e eficientes.