Universidade do Minho
Escola de Engenharia

Relatório Trabalho Prático LI1

Tânia Rocha

15 de Fevereiro de 2020

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

Este projeto foi nos solicitado pelos docentes da unidade curricular Laboratórios de Informática III e tem como principal objetivo a realização de um programa que faça a gestão de vendas de uma Cadeia de Distribuição com vários filiais. Outros objectivos estão também associados a este, como a consolidação de conhecimentos adquiridos em unidades curriculares anteriores, dentro das quais se incluem Programação Imperativa, Algoritmos e Complexidade e Arquitetura de Computadores. O principal desafio do projecto seria a programacao em larga escala, uma vez que iam passar pelo nosso programa milhões de dados, aumentando assim a complexidade do trabalho. Para que a realização deste projecto fosse possıvel, foram-nos introduzidos novos prıncipios de programação, com especial relevo para a Modularidade e encapsulamento de dados.

Capítulo 2

Análise de Requisitos

2.1 Fase 1

Na Primeira fase deste projeto temos como objetivo começar a dar forma ao jogo que estamos a tentar construir.

Esta fase é dividida em 3 tarefas, as quais têm papeis diferentes na construção e formulação do jogo.

2.1.1 Tarefa 1

Na Tarefa 1 temos como objetivo a construção do mapa onde decorre o jogo. Este mapa deve incorporar um tabuleiro, ou seja, uma lista de listas de peças que formem um caminho por onde o jogador possa circular e uma área que seja formada por "lava"onde o jogador não o possa fazer. No caminho o jogador pode circular através de movimentos. É importante salientar que a partida do carro deve ser programada de maneira que tenha direção inicial *Este* e seja a partir duma posição onde não se encontre uma peça de lava. Todas as peças lava têm, sem exceção, altura igual a 0.

2.1.2 Tarefa 2

A Tarefa 2 tem como objetivo testar se um dado mapa é ou não válido de acordo com um conjunto de regras, as quais se encontram a seguir representadas:

-Existe apenas um percurso e todas as peças fora desse percurso são do tipo lava;

-O percurso deve corresponder a uma trajectória, tal que começando na peça de partida com a orientação inicial volta-se a chegar à peça de partida com a orientação inicial;

-A orientação inicial tem que ser compatível com a peça de partida. Como é sugerido na imagem abaixo, considera-se que a orientação é compatível com a peça se for possível entrar na peça seguindo essa orientação (note que esta questão só é relevante para as peças do tipo curva e rampa);

-As peças do percurso só podem estar ligadas a peças do percurso com alturas compatíveis;

-Todas as peças do tipo lava estão à altura 0;

-O mapa é sempre rectangular e rodeado por lava, ou seja, a primeira e última linha, assim como a primeira e última coluna são constituídas por peças necessariamente do tipo lava.

Nesta mesma tarefa é importante considerar que para cada mapa deve existir um percurso de peças de piso que permita ao carro dar voltas à pista.

2.1.3 Tarefa 3

Na tarefa 3 o objetivo presente é começar a implementar a mecânica do jogo, concretamente as movimentações do carro no mapa de jogo e as respetivas colisões com obstáculos.

Nesta tarefa, dado o estado atual dum carro, deve-se calcular o seu novo estado após um determinado período de tempo, prestando atenção às interações com o mapa que podem ocorrer durante esse período, nomeadamente:

Se o carro transita para a lava então é "destruído".

Se o carro transita para uma posição pelo menos uma unidade mais baixa que a atual, então "cai" e é "destruído";

Se o carro transita para uma posição pelo menos uma unidade mais alta que a atual, então colide contra a peça, devendo ser calculado o impacto da colisão, ou seja, qual o respectivo "ricochete".

Se o carro transita entre duas posições com diferença de alturas menor a uma unidade, então movimenta-se normalmente.

Em suma, na fase 1 podemos observar que o objetivo principal é construír mapas que sejam válidos para o funcionamento do jogo e carros que corram corretamente no jogo.

2.2 Fase 2

Tal como a primeira fase, esta segunda fase é também composta por 3 tarefas: a tarefa 4, 5, e 6.

Nesta fase 2 o principal objetivo é começar a programar o jogo de maneira a que este atualize e demostre o seu novo estado. Só assim poderá este ser jogado pois é devido a estas atualizações, que ocorrem após um comando qualquer sobre carro efetuado pelo jogador, que o jogo se vai atualizando aparecendo as antigas jogadas e as mais recentes.

2.2.1 Tarefa 4

Começando pela tarefa 4, a primeira desta fase, e pelo seu objetivo que é atualizar o estado do jogo dadas as ações efectuadas por um jogador num período de tempo, vamos observar requesitos necessários que são preciso representar:

-**Jogo** - o estado interno do jogo (que deverá ser atualizado em cada instante);

-**Ação** - algo que vai indicar, por exemplo, se o carro está a acelerar, travar, ou curvar.

Ou seja, esta tarefa tem de ser programada de maneira a que o jogo seja atualizado após um comando para que este apareça representado no jogo de maneira a torna-lo jogável.

Nesta tarefa adicionamos novas "cartas"ao jogo, como a velocidade, o atrito do piso e dos pneus, o "nitro", a aceleração, o peso ou efeito da gravidade nas rampas e a sensibilidade do guiador. Estas variantes foram designadas como propriedades e variam de percurso para percurso conforme o mapa, que representa o percurso do jogo, muda.

Em particular, o "nitro", umas das propriendades que o jogador pode colocar em funcionamento sobre si ou outro qualquer jogador, tem de ser disponibilizado numa quantidade limitada no inicio de cada jogo e este vai se gastando de cada vez que é ativado pelo jogador.

Nesta mesma tarefa temos ainda as ações do jogador, que têm efeito, num dado período de tempo, e definem se o carro do está a acelerar, travar, curvar para a direita ou esquerda ou com o "nitro" ativado.

Para a atualização do estado, nesta tarefa, esta tem que estar de acordo com o impacto das ações de um dado jogador num determinado intervalo de tempo, nomeadamente, o vetor da velocidade, que neste caso vamos chamar-lhe de vetor v, e é calculado a partir da soma de todas as forças envolvidas (dadas como vetores) num determinado intervalo de tempo. Estas forças são:

-A força de atrito

-Força de aceleração

-Força da gravidade

-Força dos pneus

Tem que haver também a atualização da direção do carro caso este esteja a rodar para a esquerda ou para a direita de acordo com a constante da sensibilidade do guiador, e temos também a atualização da quantidade de "nitro"disponível no carro, que já vimos anteriormente que é limitada. Se este estiver a ser utilizado, conforme a tempo vai decorrendo com ele ativo, a sua disponibilidade vai diminuíndo. A atualização que falta efetuar é o histórico de posições percorridas pelo carro do jogador e este adiciona uma posiçao ao histórico caso tenha mudado de posição desde a última atualização.

2.2.2 Tarefa 5

A segunda parte desta fase é a tarefa 5, a qual tem como objetivo implementar o jogo completo usando a biblioteca Gloss. Como ponto de partida deve-se começar por implementar uma versão com uma visualização gráfica simples e que use sempre um caminho fixo para gerar o mapa inicial. Embora esta tarefa deva ser construída sobre as outras tarefas anteriores, ela pode ser considerada como uma "tarefa aberta"onde podemos explorar varias possibilidades extras para melhorar o aspeto do jogo com,por exeplo, a inclusão de menús de inicio e fim de jogo, diferentes câmeras e perspetivas, gráficos apelativos e criativos, mostrar informação sobre o jogo como por exemplo o tempo de jogada e o número de volta dadas ao percurso, permitir jogar diferentes mapas e/ou carregar mapas, permitir jogar contra outros jogadores e contra bots (da Tarefa 6), suportar diferentes percursos com diferentes propriedades, suportar novas funcionalidades dos carros ou das pistas, etc.

Deve-se também, nesta tarefa, utilizar como base as funções da tarefa 4, que atualizam o estado do jogo, e as da tarefa 3,que atualiza a posição do carro de acordo com esse estado.

Devido as estas particularidades deve-se criar um novo tipo de estado do jogo para permitar a alteração do jogo, tendo-se em atenção no entanto que o

tipo "Jogo"deve permanecer inalterado.

2.2.3 Tarefa 6

Por fim, nesta fase, temos a tarefa 6, cujo objetivo é implantar bots que joguem automáticamente, tendo estes entre si diferentes graus de "inteligência".

Para isto tem que se definir uma função que dada uma ação e a sua duração, o estado do jogo e o identificador do utilizador esta deolve uma ação a realizar por um bot.

Para definir esta função tem que se definir uma estratégia assumindo os seguines pontos:

-O objetivo do jogo é dar uma volta à pista à frente dos adversários, iniciando e acabando na posição de partida;

-O jogo acaba passado 60s independentemente de algum carro ter conseguido dar a volta. Nesse caso o vencedor será o carro mais avançado no percurso;

-Os mapas são válidos de acordo com a Tarefa 2, e a física do jogo é a mesma das Tarefas 3 e 4;

-Cada jogador começa com 5s de "nitro";

-Além das mortes naturais, o carro é destruído se passar para uma posição que esteja mais de 4 peças à frente do percurso ideal (i.e., atalhos são permitidos até 4 peças);

-Quando o carro é destruído, volta ao centro da última posição visitada com velocidade zero e fica imobilizado durante 1.5s;

-O percurso será selecionado a partir de um conjunto fixo de caminhos e propriedades, que se encontram definidos no módulo auxiliar Mapas, disponível no SVN de cada grupo.

Assim, podemos concluír que na fase 2 temos como objetivo colocar o jogo em funcionamento e dar-lhe caracteristicas para que este se possa jogar de maneira interessante e de maneiras diferentes.

Capítulo 3

A Nossa Solução

Neste capitulo iremos apresentar as nossas soluções para os problemas descritos em 2.1 e 2.2.

3.1 Tarefa 1

Como explicitado anteriormente, nesta tarefa, tivemos que construír os mapas onde o jogo decorre. Para isso tivemos que construír uma função que o fizesse de maneira correta.

O tipo desta função é:

```
constroi :: Caminho -> Mapa
```

Para se chegar a esta função é necessário construír várias funções auxiliares que construam o necessário ao mapa.

A estratégia que o nosso grupo optou foi o de construír um tabuleiro totalmente em peças de lava a partir de uma dimenção dada:

```
criatabuleirolava :: Dimensao -> Tabuleiro
criatabuleirolava (x,y) = replicate y (replicate x (Peca Lava 0))
```

Desta função começamos a construír o caminho em peças não lava e para isso criamos uma função que insere uma peça do caminho no tabuleiro que é formado totalmente por lava originado da primeira função:

```
insertTabuleiro :: Peca -> Posicao -> Tabuleiro -> Tabuleiro
insertTabuleiro p (x,y) [] = []
insertTabuleiro p (x,0) (l:ls) = [(substituilinha p x l)] ++ ls
insertTabuleiro p (x,y) (l:ls) = l:(insertTabuleiro p (x,y-1) ls)
```

Como se pode observar na última função representada atrás, esta já apresenta um função auxiliar **substituilinha** que tem o papel de substituir uma peça numa lista de peças conforme o seu número de "coluna"(int), isto para que na função anterior seja possível inserir-se a peça na posição dada, visto que assim fica separada por colunas tornando assim o código mais eficiente. Esta função encontra-se a seguir representada:

```
substituilinha :: Peca -> Int -> [Peca] -> [Peca]
substituilinha _ _ [] = []
substituilinha p 0 (p1:ps) = p:ps
substituilinha p n (p1:ps) = p1:(substituilinha p (n-1) ps)
```

Tendo as duas últimas funções definidas podemos finalmente construír uma função que dada uma posição e uma peça, insere essa peça nessa posição num tabuleiro formado por lava:

```
insertTabuleiro2 :: [(Posicao,Peca)] -> Tabuleiro -> Tabuleiro
insertTabuleiro2 [] t = t
insertTabuleiro2 ((pos,p):xs) t = insertTabuleiro2 xs (insertTabuleiro p
    pos t)
```

Estando concluidas as funções para inserir as peças de um caminho no tabuleiro totalmente formado por lava temos agora de construír algumas funções para transformar o caminho em peças, visto que um caminho é uma lista de passos, ou seja movimentos, e por isso não são do mesmo tipo das funções anteriores.

Estas funções tem que ser criadas de maneira a que dada a altura onde a peça se vai encontrar, a sua posição, que neste caso é a posição inicial, e o passo esta vai ser "convertido"para uma peça que permita o seu movimento. Vão ser ainda criadas diferentes funções para diferentes orientações, visto que quando se efetua uma curva, dependendo da direção que do momento, esta terá uma orientação diferente.

A primeira função transforma o caminho em peças e as suas posições, desde a posicao inicial, com direção inicial **Este**:

```
transforma :: Altura -> Posicao -> Caminho -> [(Posicao,Peca)]
transforma a (x,y) [] = []
transforma a (x,y) (Avanca:ls) = ((x,y),Peca Recta a)       : transforma
     a (x+1,y) ls
transforma a (x,y) (Sobe:ls) = ((x,y),Peca (Rampa Este) a)  : transforma
     (a+1) (x+1,y) ls
transforma a (x,y) (Desce:ls) = ((x,y),Peca (Rampa Oeste) (a-1)) :
    transforma (a-1) (x+1,y) ls
transforma a (x,y) (CurvaEsq:ls)= ((x,y),Peca (Curva Sul) a) :
    transformaNorte a (x,y-1) ls
transforma a (x,y) (CurvaDir:ls)= ((x,y),Peca (Curva Este) a) :
    transformaSul a (x,y+1) ls
```

As restantes 3 funções que podem ser encontradas no código são para as direções iniciais *Oeste*, *Sul* e *Norte*, mas não estão aqui apresentadas porque são semelhantes à função anterior.

Desta forma, temos todas as funções auxiliares para que a função principal seja possível ser construída. Esta função funciona, junto com todas as funções criadas anteriormente, que dando um caminho esta fornecerá um mapa com um caminho onde o carro poderá mais tarde movementar-se, sendo as restantes peças de lava:

```
constroi :: Caminho -> Mapa
constroi c = (Mapa (partida c,dirInit) (insertTabuleiro2 (transforma 0 (
    partida c) c) (criatabuleirolava (dimensao c))))
```

Termina assim a resolução dos problemas e a construção da tarefa 1.

3.2 Tarefa 2

Terminada a Tarefa 1 passamos à tarefa 2 que, como afirmamos no capitulo 2, tem como objetivo a validação dos mapas construídos na tarefa 1 com base em determinadas regras.

Ora, nesta tarefa, a função utilizada para validar mapas é do tipo:

```
valida :: Mapa -> Bool
```

Neste tipo de função, podemos observar que como tem como tipo final **Bool**, as regras que são necessárias cumprir para o bom uncionamento serão também desse mesmo tipo. Se a regra der como *True* essa regra está a ser cumprida, caso contrário o mapa não será válido.

Podemos observar então que para construír a função *valida* precisamos de criar várias auxiliares que testem os mapas nas regras dadas.

Conforme o enunciado dado a primeira regra fornecida a cumprir é a seguinte:

-"Existe apenas um percurso e todas as peças fora desse percurso são do tipo lava;"

Para testarmos se num tabuleiro todas as peças que não fazem parte do percurso são peças de lava, a estratégia que optamos foi a de criar uma função que testa se o número de peças do caminho é igual ao número de peças do tabuleiro, retirando as peças de lava. Deparámo-nos, no entanto, com um problema: o tipo da função. A função *valida* tendo um tipo inicial de *Mapa* tivemos que construír esta função com um tipo inicial igual e não com um tabuleiro. Por isso tivemos que recorrer à primeira auxiliar que testa se um tabuleiro é válido após a verificação de todas as regras. A função apresenta-se a seguir representada (as funções auxiliares desta serão mais à frente explicadas):

```
validaAux :: Tabuleiro -> Bool
validaAux t = ((alturaLava t) && (lavaPorFora t))
```

Para testar se o tabuleiro é válido no que consta as peças lava onde não se encontra o percurso, voltamos à estratégia antes dita: criar uma função que testa se o número de peças do caminho é igual ao número de peças do tabuleiro, retirando as peças de lava. Para isto, temos primeiro que criar uma função que conte o número de peças de um tabuleiro

```
contaTab :: Tabuleiro -> Int
contaTab [] = 0
contaTab (l:ls) = ((length l) * (length ls)) + (length l)
```

, uma função que conte o número de peças de lava, a qual precisa de uma auxilar que separe as listas de listas que formam um tabuleiro de maneira a este se tornar mais simples

```
contaLava :: Tabuleiro -> Int
contaLava [] = 0
contaLava (l:ls) = (contaLavaLista l) + (contaLava ls)

contaLavaLista :: [Peca] -> Int
contaLavaLista []  = 0
contaLavaLista (l:ls) = if (l==Peca Lava 0)
                        then 1 + contaLavaLista ls
                        else contaLavaLista ls
```

e uma função que conte o número de peças de um caminho, ou seja o número de peças que são não lava (esta função apresenta também duas auxiliares que facilitaram a sua construção, mas que não serão aqui apresentadas):

```
contaPecasCaminho :: Mapa -> Int
contaPecasCaminho c = length (getPecasLista c)
```

Com estas funções auxiliares podemos finalmente construír uma função que testa se todas as peças que não estão no percurso são peças de lava, ficando assim a primeira regra programada com a seguinte estruturação:

```
tudoLava :: Mapa -> Bool
tudoLava (Mapa (pos,ori) tab) = (contaPecasCaminho (Mapa (pos,ori) tab))
    == ((contaTab tab) - (contaLava tab))
```

A segunda regra dada é: -*"O percurso deve corresponder a uma trajectória, tal que começando na peça de partida com a orientação inicial volta-se a chegar à peça de partida com a orientação inicial;"*

, ou seja, a peça final do percurso deve ser tal que seja compatível com a peça inicial incluíndo a sua orientação.

A terceira regra a ser comprida é:
-*"A orientação inicial tem que ser compatível com a peça de partida. Como é sugerido na imagem abaixo, considera-se que a orientação é compatível com a peça se for possível entrar na peça seguindo essa orientação (note que esta questão só é relevante para as peças do tipo curva e rampa);"*

Como a regra 2 e 3 são cpmpatíveis em termos de funções juntamos estas no mesmo tópico e funções.

Começamos então por construír funções que testam se um determinada peça é compatível com uma determinada direção representada a seguir:

```
oriCompativel :: Orientacao -> Peca -> Bool
oriCompativel ori (Peca (Curva pOri) _) = (ori == pOri) || (
    rodaOrientacaoEsq pOri == ori)
oriCompativel ori (Peca (Rampa pOri) _) = (ori == pOri) || (ori ==
    rodaOrientacao180 pOri)
oriCompativel _ (Peca Recta _) = True
oriCompativel _ (Peca Lava 0 ) = False
```

Como podemos observar, esta função tem várias auxiliares, que para não as colocar todas neste relatório será apenas feita uma breve introdução à sua utilidade. As funções atrás observadas com os nomes *rodaOrientacaoEsq* e *rodaOrientacaoDir* têm os papeis de alterar a orientação de uma peça caso esta seja uma peça curva. A outra auxiliar é denominada de *rodaOrientacao180* e "calcula"a orientação originada de rodar um determinada peça 180º. Como podemos observar, estas funções são necessárias à função *oriCompativel* para que se verifique se uma determinada peça é compatível com uma determinada direção para qualquer caso apresentado.

Há, no entanto, uma extenção à regra 3 no que se trata às alturas das peças. Por isso criamos uma função que testa se a altura de uma peça é compatível com a peça seguinte tendo em conta os casos especiais das rampas. Estas peças quando aparecem aumentam ou diminuem a altura do circuito por isso, após uma destas peças tem que haver, obrigatóriamente, alteração da altura das peças para que o circuito possa continuar corretamente. Esta função não será aqui representada devido à sua dimensão.

Não esquecendo que para além da altura também temos que testar se a orientação é compatível, construímos também a seguinte equação com a ajuda das auxiliares anteriormente construídas:

```
novaOrientacao :: Orientacao -> Peca -> Orientacao
novaOrientacao ori (Peca (Curva pOri) _) |ori == pOri = rodaOrientacaoDir
     ori
                                         |otherwise  = rodaOrientacao180 pOri
novaOrientacao ori _ = ori
```

Estão então as primeiras três regras prontas a ser testadas, passaremos então às seguintes.

A quarta regra diz o seguinte: *"As peças do percurso só podem estar ligadas a peças do percurso com alturas compatíveis;"*. O que podemos observar, já que construímos esta regra com as funções construídas anterirmente.

Passando à quinta regra *"Todas as peças do tipo lava estão à altura 0;"*.

Para esta regra tivemos apenas que construír uma função que testa se todas as peças do tipo lava de um tabueiro se encontravam à altura 0 (zero). Usamos também uma auxiliar para determinar se uma peça Lava tem ou não altura zero.

Finalmente, a sexta e última regra a ter em conta diz : *"O mapa é sempre rectangular e rodeado por lava, ou seja, a primeira e última linha, assim como"*. Ou seja, qualquer mapa criado tem que ser obrigatóriamente retangular ou quadrado e o caminho ou percurso não pode estar nas margens deste. O caminho tem que estar obrigatóriamente rodeado nas margems por peças de lava, daí as primeiras e últimas linhas e colunas têm de ser obrigatóriamente de peças de lava.

Construímos então funções que testam se na primeira e na última colunas tem peças que não sejam lava. E fazer o mesmo para as linhas. Estas funções precisaram de auxiliares que determinem as listas de peças que existem nas posições desejadas. Por isso utilizamos a função antes definida para verificar se uma peça é ou não de lava.

A função que verifica o cumprimento desta regra e já com as auxiliares atrás descritas apresenta-se a seguir:

```
lavaPorFora :: Tabuleiro -> Bool
lavaPorFora (l:ls) = let f = listaLava in (f l && f (last (l:ls)) && f (
    pColunaTab (l:ls)) && f (uColunaTab (l:ls)))
```

Terminada assim a última regra, apenas é necessário incorporá-las na função *valida*, com o auxilio da função *validaAux*, que já explicamos o seu papel ade testar se um tabuleiro é válido anteriormente.A função *validaAux* ficou assim desta maneira:

```
validaAux :: Tabuleiro -> Bool
validaAux t = ((alturaLava t) && (lavaPorFora t))
```

,pudendo assim adicioná-la à função objetivo desta tarefa:

```
valida :: Mapa -> Bool
valida (Mapa (pos,ori) tab) = (tudoLava (Mapa (pos,ori) tab) && (
    validaAux tab))
```

Temos então a Tarefa 2 terminada.

3.3 Tarefa 3

Como é possível observar no capítulo 2, o ojetivo desta tarefa é começar a implementar a mecânica do jogo, concretamente as movimentações do carro no mapa do jogo e as respetivas colisões com obstáculos.

A função que nos é dada no enunciado para construír é do tipo:

```
movimenta :: Tabuleiro -> Tempo -> Carro -> Maybe Carro
```

Observando este tipo de função e raciocionando um pouco sobre o enunciado, que nos diz que quando um carro cai ou circula na lava este é destruído, podemos concluír já que uma das partes desta função pode ser construída da seguinte maneira:

```
movimenta m t c = Nothing
```

O que nos diz neste caso que o carro é sempre destruído não importando o seu estado. Ou seja, teremos que alterar esta função de maneira a que esta retrate todos os casos em que o carro pode ser ou não destruído.

Nesta tarefa é necessário construír um carro e as suas propriedades, nomeadamente, a velocidade, a posição onde se encontra e a direção que este toma e então atualizar o estado deste após um determinado período de tempo, prestando atenção às interações com o mapa neste período de tempo.

Estas interações que temos de ter em conta são:

-*"Se o carro transita para a lava então é "destruído";"*

-*"Se o carro transita para uma posição pelo menos uma unidade mais baixa que a atual, então "cai" e é "destruído";"*

-*"Se o carro transita para uma posição pelo menos uma unidade mais alta que a atual, então colide contra a peça, devendo ser calculado o impacto da colisão, ou seja, qual o respectivo "ricochete"."* - Neste caso podem ocorrer várias colisões.

-*"Se o carro transita entre duas posições com diferença de alturas menor a uma unidade, então movimenta-se normalmente."*

As regras de colisões, devido à sua física, devem assumir que:

-*"A direção do carro não é alterada;"*

-*"O carro desloca-se a velocidade constante e nem "acelera" nem "curva", inclusive durante colisões. Mais concretamente, o "valor absoluto" da velocidade não é alterado, mas sim a sua direção de acordo com o ângulo de colisão;"*

-*"O resultado é do tipo Maybe Carro para contemplar a possibilidade de o carro ser "destruído", devendo a função devolver Nothing quando o carro é "destruído" (em qualquer momento durante a sua movimentação), ou Just e em caso contrário, sendo e o novo estado do carro."*

Para que todas estas regras sejam testadas temos que construír funções que as testem e que sejam possíveis de inserir na função *Estado*, ou seja,que tenham tipos compatíveis.

Como não terminamos esta tarefa a tempo da entrega no prazo limite da primeira fase não temos como demonstrar soluções pois não as obtivemos.

3.4 Tarefa 4

O objetivo desta tarefa é atualizar o estado do jogo dadas as ações efectuadas por um jogador num período de tempo.

Visto que esta função trabalha com vetores, como explicitado anteriormente, começamos por construír funções simples que calculem diferentes cálculos entre vários vetores. Construímos funções para multiplicar, somar, dividir e duplicar vetores.

Um importânte tópico a tomar em conta em Haskell é que esta linguangem funciona não em graus, mas sim em radianos, por isso criamos uma função que "traduz"graus para radianos.

```
degreeTorad :: Double -> Double
degreeTorad x = (x * pi) / 180
```

Construímos também outras funções mais simples para que possam ser aplicadas mais tarde, como por exeplo, uma função que calcula o grau do angulo da direção inicial e uma função que passa de pontos polares para pontos cartesianos.

Continuando com a tarefa 4 e no que se trata à atualização do seu estado, criamos uma função para atualizar a direção do carro após um determinado período de tempo e uma ação aplicada deste:

```
atualizaDirecao :: Tempo -> Jogo -> Angulo -> Acao -> Angulo
atualizaDirecao t g a (Acao _ _ e d _) | d == True = (a-t*k_roda (pista g
    ))
                                       | e == True = (a+t*k_roda (pista g))
                                       | otherwise = a
```

Outro tópico incluído nesta tarefa é o "nitro"que um carro possui, ou seja , dando uma quantidade de "nitro"inicial, este de cada vez que é utilizado vai diminuíndo. Para criar uma função que atualize então a quantidade de "nitro"no carro precisamos primeiro de contruír uma função que controle o tempo de "nitro"utilizado, à qual chamamos *tempoNitro*. Desta maneira conseguimos a função que atualiza a quantidade de nitro num carro:

```
atualizaTempoNitro :: Acao -> Int -> Tempo -> [Tempo] -> [Tempo]
atualizaTempoNitro (Acao _ _ _ _ Nothing) x t1 (h:hs) = (h:hs)
atualizaTempoNitro (Acao a t e d (Just n)) x t1 (h:hs) = if (x == 0) then
     tempoNitro h t1 : hs
                                                  else h:
                                                      atualizaTempoNitro (
                                                      Acao a t e d (Just n)
                                                      ) (x-1) t1 hs
```

Obviamente, ao aplicar o "nitro", a velocidade do carro vai aumentar, por isso construímos também uma função que atualiza a velocidade do carro com a aplicação do nitro denominada de *atualizaNitro*.

Não esquecendo os restantes vetores que terão de ser utilizados, construímos várias funções para os colocar em funcionamento. Estas funções calculam a força do atrito, da aceleração, da força exercida pelos pneus e uma função que calcula a força da gravidade no percurso dependendo das peças onde o carro se encontra. Esta funções são denominadas de *forcaAtrito*, *forcaAceleracao*, *forcaPneus*, e *forcaGrav*, respetivamente.

Para que o jogo funcione é necessário também, como é dito no inunciado, um histórico de jogadas do jogador. Para isso precisamos primeiro criar uma função que crie um histórico das posições por onde o carro circulou. Representamos assim:

```
atualizaHistorico :: Jogo -> Int -> Acao -> [[Posicao]]
atualizaHistorico (Jogo m p c n h) i Acao{} = histIdent i h (identCarro c
     i)
```

Assim, permite-nos criar uma função que aqualize o histórico dependendo do jogador, ou seja, dependendo das ações impostas ao carro. A esta função chamamos de *histIdent0*.

Esta tarefa, tal como a anterior não foi terminada, devido à data limite de entrega, por isso neste tópico de apresentação de soluções fica também incompleto.

3.5 Tarefa 5

Nesta tarefa temos o objetivo de "criar"o jogo no que se trata ao gráficos.

Criando um novo estado, diferente do da tarefa 4, devemos utilizá-lo para inserir imagens e criar menús e movimentos para jazer com que o jogo funcione.

No caso dos menús, criamos uma função *display* que nos dá a imagem base do jogo em si, cria os menús, etc. Aqui podemos editar a nossa função parar criar um menú inicial onde se possa escolher a cor do carro que se quer controlar ou a pista onde se quer jogar. Inicialmente a ideia era criar um jogo com diferentes carros e pistas, mas não obtivemos sucesso por falta de conhecimentos e tempo.

Continuando com as imagens, devemos então criar algumas funções que substituam estas imagens nos locais corretos. Por exemplo, criamos uma função que onde se encontre uma *Peca Lava 0* esta possa ser substituida por uma imagem de lava, água ou veneno, dependendo do circuito escolhido.

Para o carro e para o resto das peças do percurso, substitui-se pelas imagens corretas da mesma maneira que das *Peca Leva 0*.

É também nesta tarefa que construímos uma função para poder controlar o carro, que neste caso apenas se vai demonstrar como um poligono, de acordo com comandos que se efetuem no teclado. A função representa-se em baixo:

```
reageEvento :: Event -> Estado -> Estado
reageEvento (EventKey (SpecialKey KeyUp)Down _ _) (x,y,vx,vy)=(x,y,vx,5)
reageEvento (EventKey (SpecialKey KeyDown)Down _ _) (x,y,vx,vy)=(x,y,vx
    ,-5)
reageEvento (EventKey (SpecialKey KeyLeft)Down _ _) (x,y,vx,vy)=(x,y,-5,
    vy)
reageEvento (EventKey (SpecialKey KeyRight)Down _ _) (x,y,vx,vy)=(x,y,5,
    vy)
reageEvento (EventKey (SpecialKey KeyUp)Up _ _) (x,y,vx,vy)=(x,y,vx,0)
reageEvento (EventKey (SpecialKey KeyDown)Up _ _) (x,y,vx,vy)=(x,y,vx,0)
reageEvento (EventKey (SpecialKey KeyLeft)Up _ _) (x,y,vx,vy)=(x,y,0,vy)
reageEvento (EventKey (SpecialKey KeyRight)Up _ _) (x,y,vx,vy)=(x,y,0,vy)
reageEvento _ s = s
```

Esta função permite um movimento fluído do poligono.

Entre outras particularidades possíveis de criar nesta tarefa todas estas são depois utilizadas numa função *main* que é a que será usadapara abrir o jogo. Esta função é do tipo:

```
main :: IO ()
```

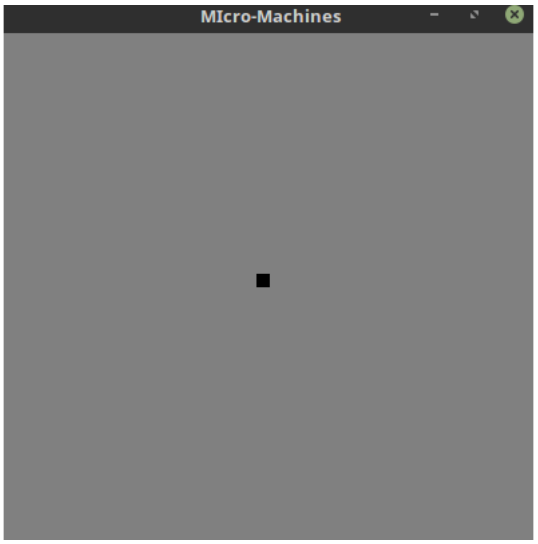

Figura 3.1: Imagem da representação gráfica do jogo conseguido

Como dito anteriormente, também não terminamos esta tarefa, daí a falta de conteúdo sobre ela no relatório. O que conseguimos obter gráficamente nesta tarefa foi o seguinte, representado na imagem em cima.

3.6 Tarefa 6

O objectivo desta tarefa é implementar um bot que jogue Micro Machines automaticamente e construír a função do tipo:

```
bot :: Tempo -> Jogo -> Int -> Acao
```

que dada a duração da ação, o estado do jogo e o identificador do jogador, devolve a ação a realizar pelo bot.

Começamos então por construír duas funções que nos vão servir de auxiliares. Uma que que converte doubles em ints e outra que dada uma lista de qualquer coisa esta dá apenas um dos constituíntes dessa lista.

Com estas funções fomos construíndo uma função ,à qual denominamos de *acaoCarro*, que tem o papel de dizer que tipo de ação um bot pode tomar dependendo do circuito em que se encontra. Incluímos nesta função o uso do "nitro", mas este pode apenas ser usado quando o bot se encontra em peças que não sejam curvas.

As restantes peças têm uma velocidade maior, graças ao "nitro". Esta função, *acaoCarro*, encontra-se representada a seguir:

```
acaoCarro :: Jogo -> Int -> Acao
acaoCarro (Jogo (Mapa _ tab) _ c n h) x5 | a == Recta = Acao True False
    False False (Just x5)
                                         | a == Curva Oeste && x1 > f3 = Acao
                                              True False False True Nothing
                                         | a == Curva Oeste && y1 > f4 = Acao
                                              True False True False Nothing
```

```
                                | a == Curva Este && x1 > f3 = Acao
                                    True False False True Nothing
                                | a == Curva Este && y1 > f4 = Acao
                                    True False True False Nothing
                                | a == Curva Norte && x1 < f3 = Acao
                                     True False True False Nothing
                                | a == Curva Norte && y1 < f4 = Acao
                                     True False False True Nothing
                                | a == Curva Sul && y1 > f4 = Acao
                                    True False False True Nothing
                                | a == Curva Sul && x1 > f3 = Acao
                                    True False True False Nothing
                                | a == Rampa Oeste = Acao True False
                                     False False Nothing
                                | a == Rampa Este = Acao True False
                                    False False Nothing
                                | a == Rampa Norte = Acao True False
                                     False False Nothing
                                | a == Rampa Sul = Acao True False
                                    False False Nothing
                                | otherwise = Acao True False False
                                    False (Just x5)
          where (Carro (x,y) u z9) = meuP c x5
                (x1,y1) = getInt (0,0) (x,y)
                (Peca a l) = getPecaTab tab (x1,y1)
                (h2:t2:y2) = meuP h x5
                (f3,f4) = t2
```

Com esta função em funcionamento, é necessário agora apenas completar a função *bot* e definir um *tick*.

A função final é esta:

```
bot tick e j = acaoCarro e j
```

Capítulo 4

Validação da Solução

Neste capítulo faremos uma descrição do método usado para validar as soluções apresentadas na secção 3.

Em todas as tarefas, à exceção da tarefa 5 e da 6, temos uma parte, logo no inicio de cada uma, que apresenta uma listagens de testes onde se podem testar as funções de cada uma das tarefas.

Das tarefa 1,2, e 4, os tipos destes exemplos encontram-se em baixo respetivamente:

```
testesT1 :: [Caminho]
testesT2 :: [Tabuleiro]
testesT3 :: [(Tabuleiro,Tempo,Carro)]
testesT4 :: [(Tempo,Jogo,Acao)]
```

Nesta tarefas, para testar se as soluções que construímos estão corretas, construímos testes para cada uma das tarefas.

No caso da tarefa 1 construímos vários caminhos, ou seja uma lista de lista de passos, não apenas um, porque assim podemos testar diferentes versões nas funções construídas.

Na tarefa 2, construímos uma lista de tabuleiros, ou seja uma lista de listas de lista de peças.

Na tarefa 3, construímos uma lista que incorpora em cada uma um tabuleiro, um determinado tempo ou período e um carro, já que, como nesta tarefa se testa o estado do carro após um determinado intervalo de tempo num tabuleiro, são necessárias todas estas variantes.

Na tarefa 4, como temos que atualizar o estado do jogo, os testes são compostos por um jogo, uma ação feita pelo jogador e um determinado intervalo de tempo em que o jogo será atualizado.

Estes testes não são apresentados aqui devido à extensão dos mesmos.

No caso da Tarefa 5, esta tarefa, que como é a construção do jogo que seja possível jogar em si, é como se fosse um teste por si mesmo. Isto significa que ao longo que fomos construíndo a tarefa 5 fomos fazendo a sua compilação em ghc no terminal e depois corre-la como um ficheiro para ver como o jogo se ia apresentando.

A tarefa 6 pode ser testada no sistema de *feedback*, através de torneios com os bots de outros grupos.

Capítulo 5

Conclusão

Terminadas assim as duas fases e os seus repetivos testes dentro dos possíveis, chegándo à conclusão do nosso projeto.

Com este projeto adquirimos novas competências de como iniciar o nosso percurso de programação, começando com a linguagem Haskell.

Este projeto, o qual consiste em criar um jogo de corridas de carros, Micro-Machines, exigiu que aprendesse-mos novas competências e que aplicasse-mos as que foram dadas nas aulas e , para além disso, fez com que nós tivevesse-mos de nos adaptarmos a novas e difíceis situações como a data limite de entrega e o facto de não podermos trabalhar em grupo com outros grupos.

Apesar de não termos concluído o projeto, fizemos uma avaliação do que este projeto significou para nós e chegamos a uma conclusão de que aprendemos novas competências e aplicações que nos poderão vir a ser úteis no futuro.