Cálculo de Programas
Trabalho Prático
MiEI+LCC — 2018/19

Departamento de Informática
Universidade do Minho

Junho de 2019

Grupo nr.	26
a85227	João Pedro Rodrigues Azevedo
a83719	Pedro Filipe Costa Machado
a85729	Paulo Jorge da Silva Araújo

1 Preâmbulo

A disciplina de Cálculo de Programas tem como objectivo principal ensinar a programação de computadores como uma disciplina científica. Para isso parte-se de um repertório de *combinadores* que formam uma álgebra da programação (conjunto de leis universais e seus corolários) e usam-se esses combinadores para construir programas *composicionalmente*, isto é, agregando programas já existentes.

Na sequência pedagógica dos planos de estudo dos dois cursos que têm esta disciplina, restringe-se a aplicação deste método à programação funcional em Haskell. Assim, o presente trabalho prático coloca os alunos perante problemas concretos que deverão ser implementados em Haskell. Há ainda um outro objectivo: o de ensinar a documentar programas, validá-los, e a produzir textos técnico-científicos de qualidade.

2 Documentação

Para cumprir de forma integrada os objectivos enunciados acima vamos recorrer a uma técnica de programação dita "literária" [1], cujo princípio base é o seguinte:

> *Um programa e a sua documentação devem coincidir.*

Por outras palavras, o código fonte e a documentação de um programa deverão estar no mesmo ficheiro.

O ficheiro `cp1819t.pdf` que está a ler é já um exemplo de programação literária: foi gerado a partir do texto fonte `cp1819t.lhs`[1] que encontrará no material pedagógico desta disciplina descompactando o ficheiro `cp1819t.zip` e executando

```
$ lhs2TeX cp1819t.lhs > cp1819t.tex
$ pdflatex cp1819t
```

em que `lhs2tex` é um pre-processador que faz "pretty printing" de código Haskell em LaTeX e que deve desde já instalar executando

```
$ cabal install lhs2tex
```

Por outro lado, o mesmo ficheiro `cp1819t.lhs` é executável e contém o "kit" básico, escrito em Haskell, para realizar o trabalho. Basta executar

```
$ ghci cp1819t.lhs
```

[1] O suffixo 'lhs' quer dizer *literate Haskell*.

Abra o ficheiro `cp1819t.lhs` no seu editor de texto preferido e verifique que assim é: todo o texto que se encontra dentro do ambiente

```
\begin{code}
...
\end{code}
```

vai ser seleccionado pelo GHCi para ser executado.

3 Como realizar o trabalho

Este trabalho teórico-prático deve ser realizado por grupos de três alunos. Os detalhes da avaliação (datas para submissão do relatório e sua defesa oral) são os que forem publicados na página da disciplina na *internet*.

Recomenda-se uma abordagem participativa dos membros do grupo de trabalho por forma a poderem responder às questões que serão colocadas na *defesa oral* do relatório.

Em que consiste, então, o *relatório* a que se refere o parágrafo anterior? É a edição do texto que está a ser lido, preenchendo o anexo D com as respostas. O relatório deverá conter ainda a identificação dos membros do grupo de trabalho, no local respectivo da folha de rosto.

Para gerar o PDF integral do relatório deve-se ainda correr os comando seguintes, que actualizam a bibliografia (com BibTEX) e o índice remissivo (com `makeindex`),

```
$ bibtex cp1819t.aux
$ makeindex cp1819t.idx
```

e recompilar o texto como acima se indicou. Dever-se-á ainda instalar o utilitário QuickCheck, que ajuda a validar programas em Haskell e a biblioteca Gloss para geração de gráficos 2D:

```
$ cabal install QuickCheck gloss
```

Para testar uma propriedade QuickCheck *prop*, basta invocá-la com o comando:

```
> quickCheck prop
+++ OK, passed 100 tests.
```

Qualquer programador tem, na vida real, de ler e analisar (muito!) código escrito por outros. No anexo C disponibiliza-se algum código Haskell relativo aos problemas que se seguem. Esse anexo deverá ser consultado e analisado à medida que isso for necessário.

Problema 1

Um compilador é um programa que traduz uma linguagem dita de *alto nível* numa linguagem (dita de *baixo nível*) que seja executável por uma máquina. Por exemplo, o GCC compila C/C++ em código objecto que corre numa variedade de arquitecturas.

Compiladores são normalmente programas complexos. Constam essencialmente de duas partes: o *analisador sintático* que lê o texto de entrada (o programa *fonte* a compilar) e cria uma sua representação interna, estruturada em árvore; e o *gerador de código* que converte essa representação interna em código executável. Note-se que tal representação intermédia pode ser usada para outros fins, por exemplo, para gerar uma listagem de qualidade (*pretty print*) do programa fonte.

O projecto de compiladores é um assunto complexo que será assunto de outras disciplinas. Neste trabalho pretende-se apenas fazer uma introdução ao assunto, mostrando como tais programas se podem construir funcionalmente à custa de cata/ana/hilo-morfismos da linguagem em causa.

Para cumprirmos o nosso objectivo, a linguagem desta questão terá que ser, naturalmente, muito simples: escolheu-se a das expressões aritméticas com inteiros, *eg.* $1+2$, $3*(4+5)$ etc. Como representação interna adopta-se o seguinte tipo polinomial, igualmente simples:

$$\mathbf{data}\ Expr = Num\ Int \mid Bop\ Expr\ Op\ Expr$$
$$\mathbf{data}\ Op = Op\ String$$

1. Escreva as definições dos {cata, ana e hilo}-morfismos deste tipo de dados segundo o método ensinado nesta disciplina (recorde módulos como *eg.* BTree etc).

2. Como aplicação do módulo desenvolvido no ponto 1, defina como {cata, ana ou hilo}-morfismo a função seguinte:

   - $calcula :: Expr \to Int$ que calcula o valor de uma expressão;

     Propriedade QuickCheck 1 *O valor zero é um elemento neutro da adição.*

     $$
     \begin{array}{l}
     prop\_neutro1 :: Expr \to Bool \\
     prop\_neutro1 = calcula \cdot addZero \equiv calcula \ \mathbf{where} \\
     \quad addZero\ e = Bop\ (Num\ 0)\ (Op\ \texttt{"+"})\ e \\
     prop\_neutro2 :: Expr \to Bool \\
     prop\_neutro2 = calcula \cdot addZero \equiv calcula \ \mathbf{where} \\
     \quad addZero\ e = Bop\ e\ (Op\ \texttt{"+"})\ (Num\ 0)
     \end{array}
     $$

     Propriedade QuickCheck 2 *As operações de soma e multiplicação são comutativas.*

     $$
     \begin{array}{l}
     prop\_comuta = calcula \cdot mirror \equiv calcula \ \mathbf{where} \\
     \quad mirror = cataExpr\ [Num, g2] \\
     \quad g2 = \widehat{\widehat{Bop}} \cdot (swap \times id) \cdot assocl \cdot (id \times swap)
     \end{array}
     $$

3. Defina como {cata, ana ou hilo}-morfismos as funções

   - $compile :: String \to Codigo$ - trata-se do compilador propriamente dito. Deverá ser gerado código posfixo para uma máquina elementar de stack. O tipo $Codigo$ pode ser definido à escolha. Dão-se a seguir exemplos de comportamentos aceitáveis para esta função:

```
Tp4> compile "2+4"
["PUSH 2", "PUSH 4", "ADD"]
Tp4> compile "3*(2+4)"
["PUSH 3", "PUSH 2", "PUSH 4", "ADD", "MUL"]
Tp4> compile "(3*2)+4"
["PUSH 3", "PUSH 2", "MUL", "PUSH 4", "ADD"]
Tp4>
```

   - $show' :: Expr \to String$ - gera a representação textual de uma $Expr$ pode encarar-se como o *pretty printer* associado ao nosso compilador

     Propriedade QuickCheck 3 *Em anexo, é fornecido o código da função $readExp$, que é "inversa" da função $show'$, tal como a propriedade seguinte descreve:*

     $$
     \begin{array}{l}
     prop\_inv :: Expr \to Bool \\
     prop\_inv = \pi_1 \cdot head \cdot readExp \cdot show' \equiv id
     \end{array}
     $$

Valorização Em anexo é apresentado código Haskell que permite declarar $Expr$ como instância da classe $Read$. Neste contexto, $read$ pode ser vista como o analisador sintático do nosso minúsculo compilador de expressões aritméticas.

Analise o código apresentado, corra-o e escreva no seu relatório uma explicação **breve** do seu funcionamento, que deverá saber defender aquando da apresentação oral do relatório.

Exprima ainda o analisador sintático $readExp$ como um anamorfismo.

Problema 2

Pretende-se neste problema definir uma linguagem gráfica "brinquedo" a duas dimensões (2D) capaz de especificar e desenhar agregações de caixas que contêm informação textual. Vamos designar essa linguagem por $L2D$ e vamos defini-la como um tipo em Haskell:

$$\mathbf{type}\ L2D = X\ Caixa\ Tipo$$

onde X é a estrutura de dados

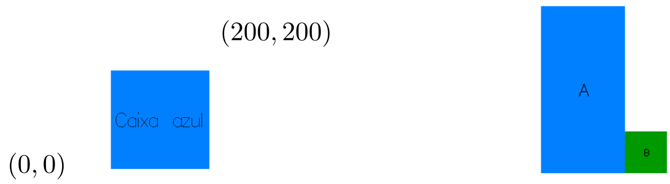

Figure 1: Caixa simples e caixa composta.

$$\textbf{data}\ X\ a\ b = Unid\ a \mid Comp\ b\ (X\ a\ b)\ (X\ a\ b)\ \textbf{deriving}\ Show$$

e onde:

$$\textbf{type}\ Caixa = ((Int, Int), (Texto, G.Color))$$
$$\textbf{type}\ Texto = String$$

Assim, cada caixa de texto é especificada pela sua largura, altura, o seu texto e a sua côr.[2] Por exemplo,

$$((200, 200), (\texttt{"Caixa azul"}, col\_blue))$$

designa a caixa da esquerda da figura 1.

O que a linguagem *L2D* faz é agregar tais caixas tipográficas umas com as outras segundo padrões especificados por vários "tipos", a saber,

$$\textbf{data}\ Tipo = V \mid Vd \mid Ve \mid H \mid Ht \mid Hb$$

com o seguinte significado:

- V - agregação vertical alinhada ao centro
- Vd - agregação vertical justificada à direita
- Ve - agregação vertical justificada à esquerda
- H - agregação horizontal alinhada ao centro
- Hb - agregação horizontal alinhada pela base
- Ht - agregação horizontal alinhada pelo topo

Como *L2D* instancia o parâmetro b de X com *Tipo*, é fácil de ver que cada "frase" da linguagem *L2D* é representada por uma árvore binária em que cada nó indica qual o tipo de agregação a aplicar às suas duas sub-árvores. Por exemplo, a frase

$$\begin{array}{l} ex2 = Comp\ Hb\ (Unid\ ((100, 200), (\texttt{"A"}, col\_blue))) \\ \qquad\qquad\qquad (Unid\ ((50, 50), (\texttt{"B"}, col\_green))) \end{array}$$

deverá corresponder à imagem da direita da figura 1. E poder-se-á ir tão longe quando a linguagem o permita. Por exemplo, pense na estrutura da frase que representa o *layout* da figura 2.

É importante notar que cada "caixa" não dispõe informação relativa ao seu posicionamento final na figura. De facto, é a posição relativa que deve ocupar face às restantes caixas que irá determinar a sua posição final. Este é um dos objectivos deste trabalho: *calcular o posicionamento absoluto de cada uma das caixas por forma a respeitar as restrições impostas pelas diversas agregações*. Para isso vamos considerar um tipo de dados que comporta a informação de todas as caixas devidamente posicionadas (i.e. com a informação adicional da origem onde a caixa deve ser colocada).

[2] Pode relacionar *Caixa* com as caixas de texto usadas nos jornais ou com *frames* da linguagem HTML usada na Internet.

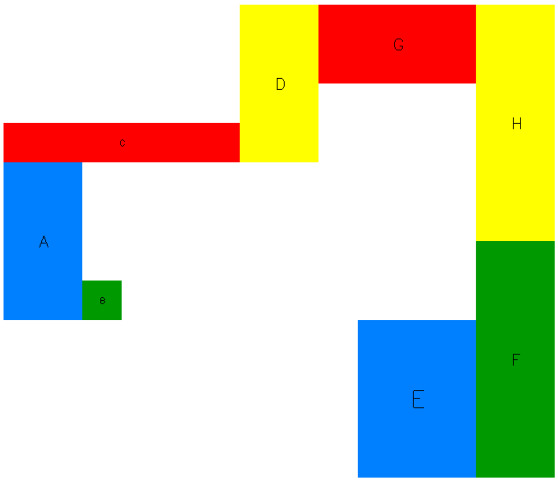

Figure 2: *Layout* feito de várias caixas coloridas.

$$\textbf{type } Fig = [(Origem, Caixa)]$$
$$\textbf{type } Origem = (Float, Float)$$

A informação mais relevante deste tipo é a referente à lista de "caixas posicionadas" (tipo $(Origem, Caixa)$). Regista-se aí a origem da caixa que, com a informação da sua altura e comprimento, permite definir todos os seus pontos (consideramos as caixas sempre paralelas aos eixos).

1. Forneça a definição da função $calc\_origems$, que calcula as coordenadas iniciais das caixas no plano:

   $$calc\_origems :: (L2D, Origem) \to X\ (Caixa, Origem)\ ()$$

2. Forneça agora a definição da função $agrup\_caixas$, que agrupa todas as caixas e respectivas origens numa só lista:

   $$agrup\_caixas :: X\ (Caixa, Origem)\ () \to Fig$$

Um segundo problema neste projecto é *descobrir como visualizar a informação gráfica calculada por* $desenho$. A nossa estratégia para superar o problema baseia-se na biblioteca Gloss, que permite a geração de gráficos 2D. Para tal disponibiliza-se a função

$$crCaixa :: Origem \to Float \to Float \to String \to G.Color \to G.Picture$$

que cria um rectângulo com base numa coordenada, um valor para a largura, um valor para a altura, um texto que irá servir de etiqueta, e a cor pretendida. Disponibiliza-se também a função

$$display :: G.Picture \to \mathsf{IO}\ ()$$

que dado um valor do tipo $G.picture$ abre uma janela com esse valor desenhado. O objectivo final deste exercício é implementar então uma função

$$mostra\_caixas :: (L2D, Origem) \to \mathsf{IO}\ ()$$

que dada uma frase da linguagem $L2D$ e coordenadas iniciais apresenta o respectivo desenho no ecrã. **Sugestão**: Use a função $G.pictures$ disponibilizada na biblioteca Gloss.

Problema 3

Nesta disciplina estudou-se como fazer programação dinâmica por cálculo, recorrendo à lei de recursividade mútua.[3]

Para o caso de funções sobre os números naturais ($\mathbb{N}_0$, com functor $\mathsf{F}\ X = 1 + X$) é fácil derivar-se da lei que foi estudada uma *regra de algibeira* que se pode ensinar a programadores que não tenham estudado Cálculo de Programas. Apresenta-se de seguida essa regra, tomando como exemplo o cálculo do ciclo-for que implementa a função de Fibonacci, recordar o sistema

$$\begin{array}{l} \mathit{fib}\ 0 = 1 \\ \mathit{fib}\ (n+1) = f\ n \\ f\ 0 = 1 \\ f\ (n+1) = \mathit{fib}\ n + f\ n \end{array}$$

Obter-se-á de imediato

$$\begin{array}{l} \mathit{fib}' = \pi_1 \cdot \mathsf{for}\ \mathit{loop}\ \mathit{init}\ \mathbf{where} \\ \quad \mathit{loop}\ (\mathit{fib}, f) = (f, \mathit{fib} + f) \\ \quad \mathit{init} = (1, 1) \end{array}$$

usando as regras seguintes:

- O corpo do ciclo *loop* terá tantos argumentos quanto o número de funções mutuamente recursivas.
- Para as variáveis escolhem-se os próprios nomes das funções, pela ordem que se achar conveniente.[4]
- Para os resultados vão-se buscar as expressões respectivas, retirando a variável n.
- Em *init* coleccionam-se os resultados dos casos de base das funções, pela mesma ordem.

Mais um exemplo, envolvendo polinómios no segundo grau a $x^2 + bx + c$ em $\mathbb{N}_0$. Seguindo o método estudado nas aulas[5], de $f\ x = ax^2 + bx + c$ derivam-se duas funções mutuamente recursivas:

$$\begin{array}{l} f\ 0 = c \\ f\ (n+1) = f\ n + k\ n \\ k\ 0 = a + b \\ k\ (n+1) = k\ n + 2\ a \end{array}$$

Seguindo a regra acima, calcula-se de imediato a seguinte implementação, em Haskell:

$$\begin{array}{l} f'\ a\ b\ c = \pi_1 \cdot \mathsf{for}\ \mathit{loop}\ \mathit{init}\ \mathbf{where} \\ \quad \mathit{loop}\ (f, k) = (f + k, k + 2 * a) \\ \quad \mathit{init} = (c, a + b) \end{array}$$

Qual é o assunto desta questão, então? Considerem fórmula que dá a série de Taylor da função coseno:

$$\mathit{cos}\ x = \sum_{i=0}^{\infty} \frac{(-1)^i}{(2i)!} x^{2i}$$

Pretende-se o ciclo-for que implementa a função $\mathit{cos}'\ x\ n$ que dá o valor dessa série tomando i até n inclusivé:

$$\mathit{cos}'\ x = \cdots \cdot \mathsf{for}\ \mathit{loop}\ \mathit{init}\ \mathbf{where}\ \cdots$$

Sugestão: Começar por estudar muito bem o processo de cálculo dado no anexo B para o problema (semelhante) da função exponencial.

Propriedade QuickCheck 4 *Testes de que $\mathit{cos}'\ x$ calcula bem o coseno de π e o coseno de $\pi / 2$:*

$$\begin{array}{l} \mathit{prop\_cos1}\ n = n \geqslant 10 \Rightarrow \mathit{abs}\ (\mathit{cos}\ \pi - \mathit{cos}'\ \pi\ n) < 0.001 \\ \mathit{prop\_cos2}\ n = n \geqslant 10 \Rightarrow \mathit{abs}\ (\mathit{cos}\ (\pi / 2) - \mathit{cos}'\ (\pi / 2)\ n) < 0.001 \end{array}$$

[3] Lei (3.94) em [2], página 98.
[4] Podem obviamente usar-se outros símbolos, mas numa primeiraleitura dá jeito usarem-se tais nomes.
[5] Secção 3.17 de [2].

Valorização Transliterar cos' para a linguagem C; compilar e testar o código. Conseguia, por intuição apenas, chegar a esta função?

Problema 4

Pretende-se nesta questão desenvolver uma biblioteca de funções para manipular *sistemas de ficheiros* genéricos. Um sistema de ficheiros será visto como uma associação de *nomes* a ficheiros ou *directorias*. Estas últimas serão vistas como sub-sistemas de ficheiros e assim recursivamente. Assumindo que a é o tipo dos identificadores dos ficheiros e directorias, e que b é o tipo do conteúdo dos ficheiros, podemos definir um tipo indutivo de dados para representar sistemas de ficheiros da seguinte forma:

$$\mathbf{data}\ FS\ a\ b = FS\ [(a, Node\ a\ b)]\ \mathbf{deriving}\ (Eq, Show)$$
$$\mathbf{data}\ Node\ a\ b = File\ b \mid Dir\ (FS\ a\ b)\ \mathbf{deriving}\ (Eq, Show)$$

Um caminho (*path*) neste sistema de ficheiros pode ser representado pelo seguinte tipo de dados:

$$\mathbf{type}\ Path\ a = [a]$$

Assumindo estes tipos de dados, o seguinte termo

$$\begin{array}{l} FS\ [(\texttt{"f1"}, File\ \texttt{"Ola"}), \\ \quad (\texttt{"d1"}, Dir\ (FS\ [(\texttt{"f2"}, File\ \texttt{"Ole"}), \\ \qquad (\texttt{"f3"}, File\ \texttt{"Ole"}) \\ \qquad])) \\ \quad] \end{array}$$

representará um sistema de ficheiros em cuja raíz temos um ficheiro chamado *f1* com conteúdo `"Ola"` e uma directoria chamada `"d1"` constituída por dois ficheiros, um chamado `"f2"` e outro chamado `"f3"`, ambos com conteúdo `"Ole"`. Neste caso, tanto o tipo dos identificadores como o tipo do conteúdo dos ficheiros é $String$. No caso geral, o conteúdo de um ficheiro é arbitrário: pode ser um binário, um texto, uma colecção de dados, etc.

A definição das usuais funções $inFS$ e $recFS$ para este tipo é a seguinte:

$$inFS = FS \cdot \mathsf{map}\ (id \times inNode)$$
$$inNode = [File, Dir]$$
$$recFS\ f = baseFS\ id\ id\ f$$

Suponha que se pretende definir como um *catamorfismo* a função que conta o número de ficheiros existentes num sistema de ficheiros. Uma possível definição para esta função seria:

$$conta :: FS\ a\ b \to Int$$
$$conta = cataFS\ (sum \cdot \mathsf{map}\ ([\underline{1}, id] \cdot \pi_2))$$

O que é para fazer:

1. Definir as funções $outFS$, $baseFS$, $cataFS$, $anaFS$ e $hyloFS$.
2. Apresentar, no relatório, o diagrama de $cataFS$.
3. Definir as seguintes funções para manipulação de sistemas de ficheiros usando, obrigatoriamente, catamorfismos, anamorfismos ou hilomorfismos:
   (a) Verificação da integridade do sistema de ficheiros (i.e. verificar que não existem identificadores repetidos dentro da mesma directoria).
   $check :: FS\ a\ b \to Bool$

   Propriedade QuickCheck 5 *A integridade de um sistema de ficheiros não depende da ordem em que os últimos são listados na sua directoria:*

   $$prop\_check :: FS\ String\ String \to Bool$$
   $$prop\_check = check \cdot (cataFS\ (inFS \cdot reverse)) \equiv check$$

(b) Recolha do conteúdo de todos os ficheiros num arquivo indexado pelo *path*.
$tar :: FS\ a\ b \rightarrow [(Path\ a, b)]$

Propriedade QuickCheck 6 *O número de ficheiros no sistema deve ser igual ao número de ficheiros listados pela função tar.*

$$prop\_tar :: FS\ String\ String \rightarrow Bool$$
$$prop\_tar = \mathsf{length} \cdot tar \equiv conta$$

(c) Transformação de um arquivo com o conteúdo dos ficheiros indexado pelo *path* num sistema de ficheiros.
$untar :: [(Path\ a, b)] \rightarrow FS\ a\ b$
Sugestão: Use a função *joinDupDirs* para juntar directorias que estejam na mesma pasta e que possuam o mesmo identificador.

Propriedade QuickCheck 7 *A composição $tar \cdot untar$ preserva o número de ficheiros no sistema.*

$$prop\_untar :: [(Path\ String, String)] \rightarrow Property$$
$$prop\_untar = validPaths \Rightarrow ((\mathsf{length} \cdot tar \cdot untar) \equiv \mathsf{length})$$
$$validPaths :: [(Path\ String, String)] \rightarrow Bool$$
$$validPaths = (\equiv 0) \cdot \mathsf{length} \cdot (filter\ (\lambda(a, \_) \rightarrow \mathsf{length}\ \ a \equiv 0))$$

(d) Localização de todos os *paths* onde existe um determinado ficheiro.
$find :: a \rightarrow FS\ a\ b \rightarrow [Path\ a]$

Propriedade QuickCheck 8 *A composição $tar \cdot untar$ preserva todos os ficheiros no sistema.*

$$prop\_find :: String \rightarrow FS\ String\ String \rightarrow Bool$$
$$prop\_find = curry\ \$$$
$$\mathsf{length} \cdot \widehat{find} \equiv \mathsf{length} \cdot \widehat{find} \cdot (id \times (untar \cdot tar))$$

(e) Criação de um novo ficheiro num determinado *path*.
$new :: Path\ a \rightarrow b \rightarrow FS\ a\ b \rightarrow FS\ a\ b$

Propriedade QuickCheck 9 *A adição de um ficheiro não existente no sistema não origina ficheiros duplicados.*

$$prop\_new :: ((Path\ String, String), FS\ String\ String) \rightarrow Property$$
$$prop\_new = ((validPath \wedge notDup) \wedge (check \cdot \pi_2)) \Rightarrow$$
$$(checkFiles \cdot \widehat{\widehat{new}})\ \textbf{where}$$
$$validPath = (\not\equiv 0) \cdot \mathsf{length} \cdot \pi_1 \cdot \pi_1$$
$$notDup = \neg \cdot \widehat{elem} \cdot (\pi_1 \times ((\mathsf{fmap}\ \pi_1) \cdot tar))$$

Questão: *Supondo-se que no código acima se substitui a propriedade checkFiles pela propriedade mais fraca check, será que a propriedade prop_new ainda é válida? Justifique a sua resposta.*

Propriedade QuickCheck 10 *A listagem de ficheiros logo após uma adição nunca poderá ser menor que a listagem de ficheiros antes dessa mesma adição.*

$$prop\_new2 :: ((Path\ String, String), FS\ String\ String) \rightarrow Property$$
$$prop\_new2 = validPath \Rightarrow ((\mathsf{length} \cdot tar \cdot \pi_2) \leqslant (\mathsf{length} \cdot tar \cdot \widehat{\widehat{new}}))\ \textbf{where}$$
$$validPath = (\not\equiv 0) \cdot \mathsf{length} \cdot \pi_1 \cdot \pi_1$$

(f) Duplicação de um ficheiro.
$cp :: Path\ a \rightarrow Path\ a \rightarrow FS\ a\ b \rightarrow FS\ a\ b$

Propriedade QuickCheck 11 *A listagem de ficheiros com um dado nome não diminui após uma duplicação.*

$$prop\_cp :: ((Path\ String, Path\ String), FS\ String\ String) \rightarrow Bool$$
$$prop\_cp = \mathsf{length} \cdot tar \cdot \pi_2 \leqslant \mathsf{length} \cdot tar \cdot \widehat{\widehat{cp}}$$

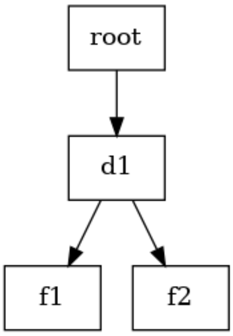

Figure 3: Exemplo de um sistema de ficheiros visualizado em Graphviz.

(g) Eliminação de um ficheiro.
$rm :: Path\ a \rightarrow FS\ a\ b \rightarrow FS\ a\ b$

Sugestão: Construir um anamorfismo $nav :: (Path\ a, FS\ a\ b) \rightarrow FS\ a\ b$ que navegue por um sistema de ficheiros tendo como base o $path$ dado como argumento.

Propriedade QuickCheck 12 *Remover duas vezes o mesmo ficheiro tem o mesmo efeito que o remover apenas uma vez.*

$$prop\_rm :: (Path\ String, FS\ String\ String) \rightarrow Bool$$
$$prop\_rm = \widehat{rm} \cdot \langle \pi_1, \widehat{rm} \rangle \equiv \widehat{rm}$$

Propriedade QuickCheck 13 *Adicionar um ficheiro e de seguida remover o mesmo não origina novos ficheiros no sistema.*

$$prop\_rm2 :: ((Path\ String, String), FS\ String\ String) \rightarrow Property$$
$$\begin{array}{l} prop\_rm2 = validPath \Rightarrow ((\mathsf{length}\ \cdot tar \cdot \widehat{rm} \cdot \langle \pi_1 \cdot \pi_1, \widehat{\widehat{new}} \rangle) \\ \quad \leqslant (\mathsf{length}\ \cdot tar \cdot \pi_2))\ \mathbf{where} \\ \quad validPath = (\not\equiv 0) \cdot \mathsf{length}\ \cdot \pi_1 \cdot \pi_1 \end{array}$$

Valorização Definir uma função para visualizar em Graphviz a estrutura de um sistema de ficheiros. A Figura 3, por exemplo, apresenta a estrutura de um sistema com precisamente dois ficheiros dentro de uma directoria chamada `"d1"`.

Para realizar este exercício será necessário apenas escrever o anamorfismo

$$cFS2Exp :: (a, FS\ a\ b) \rightarrow (Exp\ ()\ a)$$

que converte a estrutura de um sistema de ficheiros numa árvore de expressões descrita em Exp.hs. A função $dotFS$ depois tratará de passar a estrutura do sistema de ficheiros para o visualizador.

Anexos

A Como exprimir cálculos e diagramas em LaTeX/lhs2tex

Estudar o texto fonte deste trabalho para obter o efeito:[6]

$$
\begin{array}{ll}
& id = \langle f, g \rangle \\
\equiv & \quad \{ \text{ universal property } \} \\
& \left\{ \begin{array}{l} \pi_1 \cdot id = f \\ \pi_2 \cdot id = g \end{array} \right. \\
\equiv & \quad \{ \text{ identity } \} \\
& \left\{ \begin{array}{l} \pi_1 = f \\ \pi_2 = g \end{array} \right. \\
\Box &
\end{array}
$$

Os diagramas podem ser produzidos recorrendo à *package* LaTeX xymatrix, por exemplo:

$$
\begin{tikzcd}
\mathbb{N}_0 \arrow[d, "(\!| g |\!)"'] & 1 + \mathbb{N}_0 \arrow[l, "\mathsf{in}"'] \arrow[d, "id + (\!| g |\!)"] \\
B & 1 + B \arrow[l, "g"]
\end{tikzcd}
$$

B Programação dinâmica por recursividade múltipla

Neste anexo dão-se os detalhes da resolução do Exercício 3.30 dos apontamentos da disciplina[7], onde se pretende implementar um ciclo que implemente o cálculo da aproximação até $i = n$ da função exponencial $exp\ x = e^x$ via série de Taylor:

$$
exp\ x \quad = \quad \sum_{i=0}^{\infty} \frac{x^i}{i!} \tag{1}
$$

Seja $e\ x\ n = \sum_{i=0}^{n} \frac{x^i}{i!}$ a função que dá essa aproximação. É fácil de ver que $e\ x\ 0 = 1$ e que $e\ x\ (n+1) = e\ x\ n + \frac{x^{n+1}}{(n+1)!}$. Se definirmos $h\ x\ n = \frac{x^{n+1}}{(n+1)!}$ teremos $e\ x$ e $h\ x$ em recursividade mútua. Se repetirmos o processo para $h\ x\ n$ etc obteremos no total três funções nessa mesma situação:

$$
\begin{array}{l}
e\ x\ 0 = 1 \\
e\ x\ (n+1) = h\ x\ n + e\ x\ n \\
h\ x\ 0 = x \\
h\ x\ (n+1) = x\ /\ (s\ n) * h\ x\ n \\
s\ 0 = 2 \\
s\ (n+1) = 1 + s\ n
\end{array}
$$

Segundo a *regra de algibeira* descrita na página 3 deste enunciado, ter-se-á, de imediato:

$$
\begin{array}{l}
e'\ x = prj \cdot \mathsf{for}\ loop\ init\ \mathbf{where} \\
\quad init = (1, x, 2) \\
\quad loop\ (e, h, s) = (h + e, x\ /\ s * h, 1 + s) \\
\quad prj\ (e, h, s) = e
\end{array}
$$

[6] Exemplos tirados de [2].
[7] Cf. [2], página 102.

C Código fornecido

Problema 1

Tipos:

$$
\begin{array}{l}
\textbf{data } \mathit{Expr} = \mathit{Num}\ \mathit{Int} \\
\quad \mid \mathit{Bop}\ \mathit{Expr}\ \mathit{Op}\ \mathit{Expr}\ \textbf{deriving}\ (\mathit{Eq}, \mathit{Show}) \\
\textbf{data } \mathit{Op} = \mathit{Op}\ \mathit{String}\ \textbf{deriving}\ (\mathit{Eq}, \mathit{Show}) \\
\textbf{type } \mathit{Codigo} = [\mathit{String}]
\end{array}
$$

Functor de base:

$$\mathit{baseExpr}\ f\ g = \mathit{id} + (f \times (g \times g))$$

Instâncias:

$$
\begin{array}{l}
\textbf{instance } \mathit{Read}\ \mathit{Expr}\ \textbf{where} \\
\quad \mathit{readsPrec}\ \_ = \mathit{readExp}
\end{array}
$$

Read para Exp's:

$$
\begin{array}{l}
\mathit{readOp} :: \mathit{String} \to [(\mathit{Op}, \mathit{String})] \\
\mathit{readOp}\ \mathit{input} = \textbf{do} \\
\quad (x, y) \leftarrow \mathit{lex}\ \mathit{input} \\
\quad \mathit{return}\ ((\mathit{Op}\ x), y) \\
\mathit{readNum} :: \mathit{ReadS}\ \mathit{Expr} \\
\mathit{readNum} = (\mathsf{map}\ \ (\lambda(x, y) \to ((\mathit{Num}\ x), y))) \cdot \mathit{reads} \\
\mathit{readBinOp} :: \mathit{ReadS}\ \mathit{Expr} \\
\mathit{readBinOp} = (\mathsf{map}\ \ (\lambda((x, (y, z)), t) \to ((\mathit{Bop}\ x\ y\ z), t))) \cdot \\
\quad ((\mathit{readNum}\ \text{`}\mathit{ou}\text{`}\ (\mathit{pcurvos}\ \mathit{readExp})) \\
\qquad \text{`}\mathit{depois}\text{`}\ (\mathit{readOp}\ \text{`}\mathit{depois}\text{`}\ \mathit{readExp})) \\
\mathit{readExp} :: \mathit{ReadS}\ \mathit{Expr} \\
\mathit{readExp} = \mathit{readBinOp}\ \text{`}\mathit{ou}\text{`}\ (\\
\quad \mathit{readNum}\ \text{`}\mathit{ou}\text{`}\ (\\
\quad \mathit{pcurvos}\ \mathit{readExp}))
\end{array}
$$

Combinadores:

$$
\begin{array}{l}
\mathit{depois} :: (\mathit{ReadS}\ a) \to (\mathit{ReadS}\ b) \to \mathit{ReadS}\ (a, b) \\
\mathit{depois}\ \_\ \_\ [\,] = [\,] \\
\mathit{depois}\ r1\ r2\ \mathit{input} = [((x, y), i_2) \mid (x, i_1) \leftarrow r1\ \mathit{input}, \\
\quad (y, i_2) \leftarrow r2\ i_1] \\
\mathit{readSeq} :: (\mathit{ReadS}\ a) \to \mathit{ReadS}\ [a] \\
\mathit{readSeq}\ r\ \mathit{input} \\
\qquad = \textbf{case}\ (r\ \mathit{input})\ \textbf{of} \\
\qquad\quad [\,] \to [([\,], \mathit{input})] \\
\qquad\quad l \to \mathit{concat}\ (\mathsf{map}\ \ \mathit{continua}\ l) \\
\qquad\qquad \textbf{where}\ \mathit{continua}\ (a, i) = \mathsf{map}\ \ (c\ a)\ (\mathit{readSeq}\ r\ i) \\
\qquad\qquad\quad c\ x\ (\mathit{xs}, i) = ((x : \mathit{xs}), i) \\
\mathit{ou} :: (\mathit{ReadS}\ a) \to (\mathit{ReadS}\ a) \to \mathit{ReadS}\ a \\
\mathit{ou}\ r1\ r2\ \mathit{input} = (r1\ \mathit{input}) +\!\!+ (r2\ \mathit{input}) \\
\mathit{senao} :: (\mathit{ReadS}\ a) \to (\mathit{ReadS}\ a) \to \mathit{ReadS}\ a \\
\mathit{senao}\ r1\ r2\ \mathit{input} = \textbf{case}\ (r1\ \mathit{input})\ \textbf{of} \\
\quad [\,] \to r2\ \mathit{input} \\
\quad l \quad\ \ \to l \\
\mathit{readConst} :: \mathit{String} \to \mathit{ReadS}\ \mathit{String} \\
\mathit{readConst}\ c = (\mathit{filter}\ ((\equiv c) \cdot \pi_1)) \cdot \mathit{lex} \\
\mathit{pcurvos} = \mathit{parentesis}\ \texttt{'('}\ \texttt{')'}
\end{array}
$$

```
prectos = parentesis '[' ']'
chavetas = parentesis '{' '}'

parentesis :: Char → Char → (ReadS a) → ReadS a
parentesis _ _ _ [] = []
parentesis ap pa r input
  = do
    ((_, (x, _)), c) ← ((readConst [ap]) `depois` (
      r `depois` (
      readConst [pa]))) input
    return (x, c)
```

Problema 2

Tipos:

```
type Fig = [(Origem, Caixa)]
type Origem = (Float, Float)
```

"Helpers":

```
col_blue = G.azure
col_green = darkgreen

darkgreen = G.dark (G.dark G.green)
```

Exemplos:

```
ex1Caixas = G.display (G.InWindow "Problema 4" (400, 400) (40, 40)) G.white $
  crCaixa (0, 0) 200 200 "Caixa azul" col_blue
ex2Caixas = G.display (G.InWindow "Problema 4" (400, 400) (40, 40)) G.white $
  caixasAndOrigin2Pict ((Comp Hb bbox gbox), (0.0, 0.0)) where
  bbox = Unid ((100, 200), ("A", col_blue))
  gbox = Unid ((50, 50), ("B", col_green))
ex3Caixas = G.display (G.InWindow "Problema 4" (400, 400) (40, 40)) G.white mtest where
  mtest = caixasAndOrigin2Pict $ (Comp Hb (Comp Ve bot top) (Comp Ve gbox2 ybox2), (0.0, 0.0))
  bbox1 = Unid ((100, 200), ("A", col_blue))
  bbox2 = Unid ((150, 200), ("E", col_blue))
  gbox1 = Unid ((50, 50), ("B", col_green))
  gbox2 = Unid ((100, 300), ("F", col_green))
  rbox1 = Unid ((300, 50), ("C", G.red))
  rbox2 = Unid ((200, 100), ("G", G.red))
  wbox1 = Unid ((450, 200), ("", G.white))
  ybox1 = Unid ((100, 200), ("D", G.yellow))
  ybox2 = Unid ((100, 300), ("H", G.yellow))
  bot = Comp Hb wbox1 bbox2
  top = (Comp Ve (Comp Hb bbox1 gbox1) (Comp Hb rbox1 (Comp H ybox1 rbox2)))
```

A seguinte função cria uma caixa a partir dos seguintes parâmetros: origem, largura, altura, etiqueta e côr de preenchimento.

```
crCaixa :: Origem → Float → Float → String → G.Color → G.Picture
crCaixa (x, y) w h l c = G.Translate (x + (w / 2)) (y + (h / 2)) $ G.pictures [caixa, etiqueta] where
  caixa = G.color c (G.rectangleSolid w h)
  etiqueta = G.translate calc_trans_x calc_trans_y $
    G.Scale calc_scale calc_scale $ G.color G.black $ G.Text l
  calc_trans_x = (−((fromIntegral (length l)) ∗ calc_scale) / 2) ∗ base_shift_x
  calc_trans_y = (−calc_scale / 2) ∗ base_shift_y
  calc_scale = bscale ∗ (min h w)
  bscale = 1 / 700
```

$$base\_shift\_y = 100$$
$$base\_shift\_x = 64$$

Função para visualizar resultados gráficos:

$$display = G.display\ (G.InWindow\ \texttt{"Problema 4"}\ (400, 400)\ (40, 40))\ G.white$$

Problema 4

Funções para gestão de sistemas de ficheiros:

$$concatFS = inFS \cdot \widehat{(+\!\!+)} \cdot (outFS \times outFS)$$
$$mkdir\ (x, y) = FS\ [(x, Dir\ y)]$$
$$mkfile\ (x, y) = FS\ [(x, File\ y)]$$
$$joinDupDirs :: (Eq\ a) \Rightarrow (FS\ a\ b) \rightarrow (FS\ a\ b)$$
$$joinDupDirs = anaFS\ (prepOut \cdot (id \times proc) \cdot prepIn)\ \mathbf{where}$$
$$\quad prepIn = (id \times (\mathsf{map}\ \ (id \times outFS))) \cdot sls \cdot (\mathsf{map}\ \ distr) \cdot outFS$$
$$\quad prepOut = (\mathsf{map}\ \ undistr) \cdot \widehat{(+\!\!+)} \cdot ((\mathsf{map}\ \ i_1) \times (\mathsf{map}\ \ i_2)) \cdot (id \times (\mathsf{map}\ \ (id \times inFS)))$$
$$\quad proc = concat \cdot (\mathsf{map}\ \ joinDup) \cdot groupByName$$
$$\quad sls = \langle lefts, rights \rangle$$
$$joinDup :: [(a, [b])] \rightarrow [(a, [b])]$$
$$joinDup = cataList\ [nil, g]\ \mathbf{where}\ g = return \cdot \langle \pi_1 \cdot \pi_1, concat \cdot (\mathsf{map}\ \ \pi_2) \cdot \widehat{(:)} \rangle$$
$$createFSfromFile :: (Path\ a, b) \rightarrow (FS\ a\ b)$$
$$createFSfromFile\ ([a], b) = mkfile\ (a, b)$$
$$createFSfromFile\ (a : as, b) = mkdir\ (a, createFSfromFile\ (as, b))$$

Funções auxiliares:

$$checkFiles :: (Eq\ a) \Rightarrow FS\ a\ b \rightarrow Bool$$
$$checkFiles = cataFS\ (\widehat{(\wedge)} \cdot \langle f, g \rangle)\ \mathbf{where}$$
$$\quad f = nr \cdot (\mathsf{fmap}\ \pi_1) \cdot lefts \cdot (\mathsf{fmap}\ distr)$$
$$\quad g = and \cdot rights \cdot (\mathsf{fmap}\ \pi_2)$$
$$groupByName :: (Eq\ a) \Rightarrow [(a, [b])] \rightarrow [[(a, [b])]]$$
$$groupByName = (groupBy\ (curry\ p))\ \mathbf{where}$$
$$\quad p = \widehat{(\equiv)} \cdot (\pi_1 \times \pi_1)$$
$$filterPath :: (Eq\ a) \Rightarrow Path\ a \rightarrow [(Path\ a, b)] \rightarrow [(Path\ a, b)]$$
$$filterPath = filter \cdot (\lambda p \rightarrow \lambda(a, b) \rightarrow p \equiv a)$$

Dados para testes:

- Sistema de ficheiros vazio:

$$efs = FS\ [\,]$$

- Nível 0

$$f1 = FS\ [(\texttt{"f1"}, File\ \texttt{"hello world"})]$$
$$f2 = FS\ [(\texttt{"f2"}, File\ \texttt{"more content"})]$$
$$f00 = concatFS\ (f1, f2)$$
$$f01 = concatFS\ (f1, mkdir\ (\texttt{"d1"}, efs))$$
$$f02 = mkdir\ (\texttt{"d1"}, efs)$$

- Nível 1

$$f10 = mkdir\ (\texttt{"d1"}, f00)$$
$$f11 = concatFS\ (mkdir\ (\texttt{"d1"}, f00), mkdir\ (\texttt{"d2"}, f00))$$
$$f12 = concatFS\ (mkdir\ (\texttt{"d1"}, f00), mkdir\ (\texttt{"d2"}, f01))$$
$$f13 = concatFS\ (mkdir\ (\texttt{"d1"}, f00), mkdir\ (\texttt{"d2"}, efs))$$

- Nível 2

$$
\begin{aligned}
&f20 = mkdir\ (\texttt{"d1"}, f10)\\
&f21 = mkdir\ (\texttt{"d1"}, f11)\\
&f22 = mkdir\ (\texttt{"d1"}, f12)\\
&f23 = mkdir\ (\texttt{"d1"}, f13)\\
&f24 = concatFS\ (mkdir\ (\texttt{"d1"}, f10), mkdir\ (\texttt{"d2"}, f12))
\end{aligned}
$$

- Sistemas de ficheiros inválidos:

$$
\begin{aligned}
&ifs0 = concatFS\ (f1, f1)\\
&ifs1 = concatFS\ (f1, mkdir\ (\texttt{"f1"}, efs))\\
&ifs2 = mkdir\ (\texttt{"d1"}, ifs0)\\
&ifs3 = mkdir\ (\texttt{"d1"}, ifs1)\\
&ifs4 = concatFS\ (mkdir\ (\texttt{"d1"}, ifs1), mkdir\ (\texttt{"d2"}, f12))\\
&ifs5 = concatFS\ (mkdir\ (\texttt{"d1"}, f1), mkdir\ (\texttt{"d1"}, f2))\\
&ifs6 = mkdir\ (\texttt{"d1"}, ifs5)\\
&ifs7 = concatFS\ (mkdir\ (\texttt{"d1"}, f02), mkdir\ (\texttt{"d1"}, f02))
\end{aligned}
$$

Visualização em Graphviz:

$$
\begin{aligned}
&dotFS :: FS\ String\ b \to \mathsf{IO}\ ExitCode\\
&dotFS = dotpict \cdot bmap\ \underline{\texttt{""}}\ id \cdot (cFS2Exp\ \texttt{"root"})
\end{aligned}
$$

Outras funções auxiliares

Lógicas:

$$
\begin{aligned}
&\textbf{infixr}\ 0 \Rightarrow\\
&(\Rightarrow) :: (Testable\ prop) \Rightarrow (a \to Bool) \to (a \to prop) \to a \to Property\\
&p \Rightarrow f = \lambda a \to p\ a \Rightarrow f\ a\\
&\textbf{infixr}\ 0 \Leftrightarrow\\
&(\Leftrightarrow) :: (a \to Bool) \to (a \to Bool) \to a \to Property\\
&p \Leftrightarrow f = \lambda a \to (p\ a \Rightarrow property\ (f\ a))\ .\&\&.\ (f\ a \Rightarrow property\ (p\ a))\\
&\textbf{infixr}\ 4 \equiv\\
&(\equiv) :: Eq\ b \Rightarrow (a \to b) \to (a \to b) \to (a \to Bool)\\
&f \equiv g = \lambda a \to f\ a \equiv g\ a\\
&\textbf{infixr}\ 4 \leqslant\\
&(\leqslant) :: Ord\ b \Rightarrow (a \to b) \to (a \to b) \to (a \to Bool)\\
&f \leqslant g = \lambda a \to f\ a \leqslant g\ a\\
&\textbf{infixr}\ 4 \wedge\\
&(\wedge) :: (a \to Bool) \to (a \to Bool) \to (a \to Bool)\\
&f \wedge g = \lambda a \to ((f\ a) \wedge (g\ a))
\end{aligned}
$$

Compilação e execução dentro do interpretador:[8]

$$run = \textbf{do}\ \{system\ \texttt{"ghc cp1819t"}; system\ \texttt{"./cp1819t"}\}$$

D Soluções dos alunos

Os alunos devem colocar neste anexo as suas soluções aos exercícios propostos, de acordo com o "layout" que se fornece. Não podem ser alterados os nomes ou tipos das funções dadas, mas pode ser adicionado texto e/ou outras funções auxiliares que sejam necessárias.

[8]Pode ser útil em testes envolvendo Gloss. Nesse caso, o teste em causa deve fazer parte de uma função $main$.

Problema 1

Pergunta 1 - Definições base para o tipo de dados

$$
\begin{array}{l}
inExpr :: Int + (Op, (Expr, Expr)) \rightarrow Expr \\
inExpr = [Num, bopCase] \\
\quad \textbf{where } bopCase\ (a, (b, c)) = Bop\ b\ a\ c \\
outExpr :: Expr \rightarrow Int + (Op, (Expr, Expr)) \\
outExpr\ (Num\ a) = i_1\ a \\
outExpr\ (Bop\ a\ op\ b) = i_2\ (op, (a, b)) \\
recExpr\ f = baseExpr\ id\ f \\
cataExpr\ g = g \cdot (recExpr\ (cataExpr\ g)) \cdot outExpr \\
anaExpr\ g = inExpr \cdot (recExpr\ (anaExpr\ g)) \cdot g \\
hiloExpr\ h\ g = cataExpr\ h \cdot anaExpr\ g
\end{array}
$$

Pergunta 2 - Calcular o valor de uma expressão

Calcular um valor de uma expressão passa por dois casos, explicitados no tipo de dados. Ou temos um *Num* ou um *Bop* e trata-se de uma operação de redução desse tipo a um número, daí a solução passar por um catamorfismo.

Segue-se o diagrama mais apropriado para descrever este catamorfismo:

$$
\begin{tikzcd}
Expr \arrow[r, "outExpr"'] \arrow[d, "cataExpr\ g"'] & Num + Op \times Expr \times Expr \arrow[d, "id + id \times (cataExpr\ g)"] \\
Int & Num + Op \times Int \times Int \arrow[l, "g = [id, parseOp]"]
\end{tikzcd}
$$

Aqui *parseOp* auxilia a função calcula em converter uma operação e um par de Inteiros no resultado respetivo.

$$
\begin{array}{l}
calcula :: Expr \rightarrow Int \\
calcula = cataExpr\ [id, parseOp] \\
\quad \text{--} \\
parseOp :: (Op, (Int, Int)) \rightarrow Int \\
parseOp\ (Op\ op, (a, b))\ |\ op \equiv \texttt{"+"} = a + b \\
\quad |\ op \equiv \texttt{"-"} \qquad\qquad\qquad\quad = a - b \\
\quad |\ op \equiv \texttt{"*"} \qquad\qquad\qquad\quad = a * b \\
\quad |\ op \equiv \texttt{"mod"} = mod\ a\ b
\end{array}
$$

Pergunta 3 - *Pretty printer* show' e a compile

Para converter um tipo de dados *Expr* na sua representação *String* basta-o converter para uma representação literal infixa. Um catamorfismo, mais uma vez, serviu para o caso.

$$
\begin{array}{l}
show' = cataExpr\ [cnvNum, cnvPar] \\
\quad \text{--} \\
cnvNum :: (Num\ a, Ord\ a, Show\ a) \Rightarrow a \rightarrow String \\
cnvNum\ a\ |\ a < 0 = \texttt{"("} +\!\!+\ show\ a +\!\!+ \texttt{")"} \\
\quad |\ otherwise = show\ a \\
\quad \text{--} \\
cnvPar :: (Op, (String, String)) \rightarrow String \\
cnvPar\ (Op\ op, (s1, s2)) = \texttt{"("} +\!\!+\ s1 +\!\!+ op +\!\!+ s2 +\!\!+ \texttt{")"}
\end{array}
$$

A operação de **compilar** uma *String* torna-se trivial após se conseguir obter a representação da *String* num *Expr*, tendo para isso, a função *readExp* fornecida.

Após isso, basta apenas fazer um catamorfismo sobre o tipo de modo a explorar a recursividade.

$compile = cataExpr\ [numToComp, bopToCompile] \cdot \pi_1 \cdot head \cdot readExp$
  where $numToComp\ a = [$"PUSH " $+\!\!+\ show\ a]$
    $bopToCompile\ (op, (s1, s2)) = s1 +\!\!+ s2 +\!\!+ [opToCodigo\ op]$

--

$opToCodigo :: Op \rightarrow String$
$opToCodigo\ (Op\ op) \mid op \equiv$ "+" $=$ "ADD"
  $\mid op \equiv$ "-" $=$ "SUB"
  $\mid op \equiv$ "$\star$" $=$ "MUL"
  $\mid op \equiv$ "mod" $=$ "MOD"

--

Para melhor representar a operação de compile, ou seja, converter *Expr* em instruções **stack** segue-se o diagrama do catamorfismo implementado.

$$\begin{tikzcd}
Expr \arrow[r, "outExpr"'] \arrow[d, "cataExpr\ g"'] & Num + Op \times Expr \times Expr \arrow[d, "id + id \times (cataExpr\ g)"] \\
{[String]} & Num + Op \times [String] \times [String] \arrow[l, "g = [numToComp, bopToCompile]"]
\end{tikzcd}$$

Incluem-se aqui alguns testes aplicavéis às funções implementadas anteriormente:

$exp\_test :: Expr$
$exp\_test = Bop\ (Bop\ (Num\ 5)\ (Op$ "$\star$"$)\ (Num\ 6))\ (Op$ "+"$)\ (Num\ (-1))$

$string\_codigo\_1 :: String$
$string\_codigo\_1 =$ "2 + 3 $\star$ 4"

$string\_codigo\_2 :: String$
$string\_codigo\_2 =$ "2 + 1 $\star$ 3 + 4"

Problema 2

Definições base do tipo de dados L2D

O diagrama que representa um catamorfismo para este tipo é o seguinte:

$$\begin{tikzcd}
L2D \arrow[r, "outL2D"'] \arrow[d, "f = cataL2D\ g"'] & Unid + b \times L2D \times L2D \arrow[d, "id + id \times f \times f"] \\
X & Unid + b \times X \times X \arrow[l, "g"]
\end{tikzcd}$$

$inL2D :: a + (b, (X\ a\ b, X\ a\ b)) \rightarrow X\ a\ b$
$inL2D = [casoUnid, casoComp]$
  where $casoUnid\ a = Unid\ a$
    $casoComp\ (b, (l, r)) = Comp\ b\ l\ r$

--

$outL2D :: X\ a\ b \rightarrow a + (b, (X\ a\ b, X\ a\ b))$
$outL2D\ (Unid\ a) = i_1\ a$
$outL2D\ (Comp\ b\ l\ r) = i_2\ (b, (l, r))$

--

$recL2D\ f = baseL2D\ id\ f$

$$baseL2D\ f\ g = id + (f \times (g \times g))$$
$$cataL2D\ g = g \cdot (recL2D\ (cataL2D\ g)) \cdot outL2D$$
$$anaL2D\ g = inL2D \cdot (recL2D\ (anaL2D\ g)) \cdot g$$

Definições de funções dadas como undefined neste documento

$$collectLeafs\ (Unid\ a) = [a]$$
$$collectLeafs\ (Comp\ b\ l\ r) = (collectLeafs\ l) +\!\!+ (collectLeafs\ r)$$
--
$$dimen :: X\ Caixa\ Tipo \to (Float, Float)$$
$$dimen = \bot$$

Pergunta 1

Nesta pergunta era pedido uma função que dada uma origem e um conjunto de figuras em L2D fossem calculadas associações das mesmas com a origem respetiva, dados os diferentes tipo de associações.

O padrão de recursividade segue mais casos e traz uma ligeira complexidade ao tipo de dados, mas contudo, este assemelha-se muito com uma BTree.

Seguem-se as definições da pergunta 1:

$$calcOrigins :: ((X\ Caixa\ Tipo), Origem) \to X\ (Caixa, Origem)\ ()$$
$$calcOrigins\ fig = calcOriginsAux\ (\pi_2\ \$\ fig)\ fig$$
--
$$calcOriginsAux :: (Float, Float) \to ((X\ Caixa\ Tipo), Origem) \to X\ (Caixa, Origem)\ ()$$
$$calcOriginsAux\ pos\ ((Unid\ ca), orig) = Unid\ (ca, pos)$$
$$calcOriginsAux\ pos\ (Comp\ b\ (Unid\ ca)\ r, orig) = Comp\ ()\ (Unid\ (ca, pos))$$
$$\quad (calcOriginsAux\ proxPosOrig\ (r, orig))$$
$$\quad \mathbf{where}\ proxPosOrig = calc\ b\ pos\ (convCoord\ (\pi_1\ \$\ ca))$$
$$calcOriginsAux\ pos\ (Comp\ b\ l\ (Unid\ ca), orig) = Comp\ ()\ (calcOriginsAux\ proxPosOrig\ (l, orig))$$
$$\quad (Unid\ (ca, pos))$$
$$\quad \mathbf{where}\ proxPosOrig = calc\ b\ pos\ (convCoord\ (\pi_1\ \$\ ca))$$
$$calcOriginsAux\ pos\ (Comp\ b\ l\ r, orig) = Comp\ ()\ (\pi_1\ \$\ fromLeft)$$
$$\quad (calcOriginsAux\ proxPosOrig\ (r, orig))$$
$$\quad \mathbf{where}\ fromLeft = (calcOriginsAux\ pos\ (l, orig), pos)$$
$$\qquad lastOrigin = \pi_2\ fromLeft$$
$$\qquad proxPosOrig = calc\ b\ pos\ lastOrigin$$

Seguem-se algumas funções auxiliares que foram necessárias ser implementadas.

--
$$convCoord :: (Int, Int) \to (Float, Float)$$
$$convCoord\ (x, y) = (fromIntegral\ x :: Float, fromIntegral\ y :: Float)$$
--
$$calc :: Tipo \to Origem \to (Float, Float) \to Origem$$
$$calc\ (Hb)\ orig\ (box\_x, box\_y) = ((\pi_1\ orig) + box\_x, \pi_2\ orig)$$
$$calc\ (Ht)\ orig\ (box\_x, box\_y) = ((\pi_1\ orig) + box\_x, (\pi_2\ orig) + box\_y)$$
$$calc\ (H)\ orig\ (box\_x, box\_y) = ((\pi_1\ orig) + box\_x, (\pi_2\ orig) + (fromInteger\ \$\ round\ \$\ (box\_y\ /\ 2) * (10 \uparrow 5))\ /\ (10.0 \uparrow\uparrow 5))$$
$$calc\ (V)\ orig\ (box\_x, box\_y) = ((\pi_1\ orig) + (fromInteger\ \$\ round\ \$\ (box\_x\ /\ 2) * (10 \uparrow 5))\ /\ (10.0 \uparrow\uparrow 5), (\pi_2\ orig) + box\_y)$$
$$calc\ (Ve)\ orig\ (box\_x, box\_y) = ((\pi_1\ orig), (\pi_2\ orig) + box\_y)$$
$$calc\ (Vd)\ orig\ (box\_x, box\_y) = ((\pi_1\ orig) + box\_x, (\pi_2\ orig) + box\_y)$$

Pergunta 2 - Agrupar as caixas e as suas origens numa só lista

$$
\begin{array}{l}
agrup\_caixas :: X\ (Caixa, Origem)\ () \rightarrow Fig \\
agrup\_caixas\ (Unid\ (ca, or)) = [(or, ca)] \\
agrup\_caixas\ (Comp\ b\ l\ r) = (agrup\_caixas\ l) +\!\!+ (agrup\_caixas\ r)
\end{array}
$$

Display das caixas num gráfico 2D em gloss

Tendo as origems calculadas apenas é preciso agrupar as caixas com as suas origens, gerar as respetivas *G.Picture*, convertendo tudo para apenas uma *G.Picture* e apresentar na janela gloss o resultado. Para tal foi crucial ter funções como *crCaixa* e *display* para criar *G.Picture* e apresentá-las.

$$mostra\_Caixas = display \cdot caixasAndOrigin2Pict$$

$$caixasAndOrigin2Pict = G.pictures \cdot criaPict \cdot agrup\_caixas \cdot calcOrigins$$

$$
\begin{array}{l}
criaPict :: Fig \rightarrow [G.Picture] \\
criaPict\ [\,] = [\,] \\
criaPict\ (h : t) = [(crCaixa\ or\ width\ height\ text\ col)] +\!\!+ (criaPict\ t) \\
\quad \textbf{where}\ or \quad = (\pi_1\ h) \\
\qquad width \quad = (toFloat\ (\pi_1\ (\pi_1\ (\pi_2\ h)))) \\
\qquad height \quad = (toFloat\ (\pi_2\ (\pi_1\ (\pi_2\ h)))) \\
\qquad text \quad = (\pi_1\ (\pi_2\ (\pi_2\ h))) \\
\qquad col \quad = (\pi_2\ (\pi_2\ (\pi_2\ h))) \\
\qquad toFloat\ x = fromIntegral\ x :: Float
\end{array}
$$

Incluem-se aqui alguns testes aplicavéis às funções implementadas anteriormente:

$$
\begin{array}{l}
testeOrigens1 :: (X\ Caixa\ Tipo, Origem) \\
testeOrigens1 = (caixasHb1, (0, 0))
\end{array}
$$

$$
\begin{array}{l}
unidadeA :: X\ Caixa\ Tipo \\
unidadeA = Unid\ ((100, 200), (\texttt{"A"}, col\_blue))
\end{array}
$$

$$
\begin{array}{l}
unidadeB :: X\ Caixa\ Tipo \\
unidadeB = Unid\ ((50, 50), (\texttt{"B"}, col\_green))
\end{array}
$$

$$
\begin{array}{l}
caixasHb1 :: X\ Caixa\ Tipo \\
caixasHb1 = Comp\ Hb\ unidadeA\ caixasHb2
\end{array}
$$

$$
\begin{array}{l}
caixasHb2 :: X\ Caixa\ Tipo \\
caixasHb2 = Comp\ Hb\ unidadeA\ unidadeB
\end{array}
$$

$$
\begin{array}{l}
caixasBasico :: X\ Caixa\ Tipo \\
caixasBasico = Comp\ Hb\ (Unid\ ((100, 200), (\texttt{"A"}, col\_blue)))\ (Unid\ ((50, 50), (\texttt{"B"}, col\_green)))
\end{array}
$$

Problema 3

Série de Taylor do cosseno - Uma aproximação...

O objetivo do problema era implementar a série de Taylor através de uma aproximação de *n* em vez de um somatório infinito, utilizando recursividade mútua.

$$cos\ x = \sum_{i=0}^{\infty} \frac{(-1)^i}{(2i)!} x^{2i}$$

Primeiro consideramos o *cos x n* como sendo *c x n*, assim, podemos verificar, através do somatório que:

$$
\begin{array}{l}
c\ x\ 0 = 1 \\
c\ x\ (n+1) = c\ x\ n + ((-1) \uparrow (n+1) * x \uparrow (2*n+2))\ /\ ((2*n+2)!) = c\ x\ n + h\ x\ n \\
h\ x\ 0 = ((-x) \uparrow 2)\ /\ 2
\end{array}
$$

$$h\ x\ (n+1) = (h\ x\ n) * (((-x) \uparrow 2)\ /\ ((2*n+4)*(2*n+3))) = (h\ x\ n) * ((-x) \uparrow 2)\ /\ (s\ n)$$
$$s\ 0 = 12$$
$$s\ (n+1) = s\ n + 8*n + 18 = s\ n + f\ n$$
$$f\ 0 = 18$$
$$f\ (n+1) = f\ n + 8$$

Agora, em termos de *Haskell*, podemos separar o problema em 3 funções, *loop* que representa o resultado recursivo para cada função (cujo funcionamento está descrito a cima), *init* que inicializa o caso base, ou seja, o caso de paragem de cada função e por fim prj que decide a função a apresentar quando se pretende mostrar o resultado da aproximação.

$$\begin{array}{l} cos'\ x = prj \cdot \mathsf{for}\ loop\ init\ \mathbf{where} \\ \quad loop\ (c,h,s,f) = (c+h, h*(-(x \uparrow 2)\ /\ s), s+f, f+8) \\ \quad init = (1, -(x \uparrow 2)\ /\ 2, 12, 18) \\ \quad prj\ (c,h,s,f) = c \end{array}$$

Problema 4

Triologia "ana-cata-hilo" - *Sistema de ficheiros genéricos*

Seguindo a recomendação do enunciado deste problema, seguem-se as definições básicas para o tipo **FS a b**:

$$\begin{array}{l} outFS :: FS\ a\ b \to [(a, (b) + (FS\ a\ b))] \\ outFS\ (FS\ [\,]) = [\,] \\ outFS\ (FS\ ((x,y):t)) = [(x, outNode\ y)] + outFS\ (FS\ t) \\ outNode :: Node\ a\ b \to b + (FS\ a\ b) \\ outNode\ (File\ b) = i_1\ (b) \\ outNode\ (Dir\ x) = i_2\ (x) \\ baseFS\ f\ g\ h = \mathsf{map}\ \ (f \times (g+h)) \\ cataFS :: ([(a, b+c)] \to c) \to FS\ a\ b \to c \\ cataFS\ g = g \cdot (recFS\ (cataFS\ g)) \cdot outFS \\ anaFS :: (c \to [(a, b+c)]) \to c \to FS\ a\ b \\ anaFS\ g = inFS \cdot (recFS\ (anaFS\ g)) \cdot g \\ hyloFS\ g\ h = cataFS\ h \cdot anaFS\ g \end{array}$$

E também o diagrama para um catamorfismo genérico para este tipo de dados:

$$\begin{tikzcd} FS\ a\ b \arrow[r, "outFS"'] \arrow[d, "f = cataFS\ g"'] & {[(a, b + FS\ a\ b)]} \arrow[d, "\mathsf{map}\ \ (id \times (id + f))"] \\ X & {[(a, b + X)]} \arrow[l, "g"] \end{tikzcd}$$

Funções para manipulação de ficheiros

a) $check :: FS\ a\ b \to Bool$

$$\begin{array}{l} check :: (Eq\ a) \Rightarrow FS\ a\ b \to Bool \\ check = cataFS\ geneCheck \\ \quad \text{--} \\ geneCheck :: (Eq\ a) \Rightarrow [(a, b + Bool)] \to Bool \\ geneCheck\ [\,] = True \\ geneCheck\ ((x, i_1\ b) : t) = geneCheck\ t \\ geneCheck\ ((x, i_2\ b) : t)\ \mid b \equiv False = False \\ \quad \mid otherwise = geneCheck\ t \wedge \neg\ (repetidos\ (paraLista\ ((x, i_2\ b) : t))) \\ \quad \text{--} \end{array}$$

$estaNaLista :: (Eq\ a) \Rightarrow a \rightarrow [a] \rightarrow Bool$
$estaNaLista\ \_\ [\,] = False$
$estaNaLista\ x\ (h : t) \mid x \equiv h = True$
$\quad \mid otherwise = estaNaLista\ x\ t$

--

$repetidos :: (Eq\ a) \Rightarrow [a] \rightarrow Bool$
$repetidos\ [\,] = False$
$repetidos\ (h : t) \mid estaNaLista\ h\ t \equiv True = True$
$\quad \mid otherwise = repetidos\ t$

--

$paraLista :: [(a, b + c)] \rightarrow Path\ a$
$paraLista\ [\,] = [\,]$
$paraLista\ ((x, y) : t) = [x] + paraLista\ t$

b) $tar :: FS\ a\ b \rightarrow [(Path\ a, b)]$

$tar :: FS\ a\ b \rightarrow [(Path\ a, b)]$
$tar = cataFS\ geneTar$

$geneTar :: [(a, b + [(Path\ a, b)])] \rightarrow [(Path\ a, b)]$
$geneTar\ [\,] = [\,]$
$geneTar\ ((a, i_1\ b) : t) = [([a], b)] + geneTar\ t$
$geneTar\ ((a, i_2\ l) : t) = (addAllPaths\ a\ l) + (geneTar\ t)$

$addAllPaths :: a \rightarrow [(Path\ a, b)] \rightarrow [(Path\ a, b)]$
$addAllPaths\ a\ [\,] = [\,]$
$addAllPaths\ a\ ((list, b) : t) = [([a] + list, b)] + addAllPaths\ a\ t$

c) $untar :: [(Path\ a, b)] \rightarrow FS\ a\ b$

$untar :: (Eq\ a) \Rightarrow [(Path\ a, b)] \rightarrow FS\ a\ b$
$untar = joinDupDirs \cdot (anaFS\ geneUntar)$

--

$geneUntar :: [(Path\ a, b)] \rightarrow [(a, b + ([(Path\ a, b)]))]$
$geneUntar\ [\,] = [\,]$
$geneUntar\ (([\,], b) : cauda) = geneUntar\ cauda$
$geneUntar\ (([x], b) : cauda) = [(x, i_1\ b)] + geneUntar\ cauda$
$geneUntar\ (((h : t), b) : cauda) = [(h, i_2\ [(t, b)])] + geneUntar\ cauda$

Esta solução utiliza um anamorfismo que é descrito de seguida:

$$\begin{tikzcd}
FS\ a\ b & {[(a, b + FS\ a\ b)]} \arrow[l, "inFS"'] \\
{[(Path\ a, b)]} \arrow[u, "f = anaFS"] \arrow[r, "g = geneUntar"'] & {[(a, b + [(Path\ a, b)])]} \arrow[u, "\mathsf{map}\ (id \times (id + f))"']
\end{tikzcd}$$

d) $find :: a \rightarrow FS\ a\ b \rightarrow [Path\ a]$

$find :: (Eq\ a) \Rightarrow a \rightarrow FS\ a\ b \rightarrow [Path\ a]$
$find\ (a)\ (f) = findAux\ (a)\ (cataFS\ (gFind)\ (f))$

--

$findAux :: (Eq\ a) \Rightarrow a \rightarrow [(a, Path\ a)] \rightarrow [Path\ a]$
$findAux\ a\ [\,] = [\,]$
$findAux\ a\ ((a1, l) : t) \mid (a \equiv a1) = [l] + (findAux\ a\ t)$
$\quad \mid otherwise = (findAux\ a\ t)$

--

$gFind :: [(a, b + [(a, Path\ a)])] \rightarrow [(a, Path\ a)]$

$gFind\ [\,] = [\,]$
$gFind\ ((a, i_1\ b) : t) = gFind\ t$
$gFind\ ((a, i_2\ p) : t) = p +\!\!+ (gFind\ t)$

e) $new :: Path\ a \rightarrow b \rightarrow FS\ a\ b \rightarrow FS\ a\ b$

$new :: (Eq\ a) \Rightarrow Path\ a \rightarrow b \rightarrow FS\ a\ b \rightarrow FS\ a\ b$
$new\ p\ b\ fs = untar\ (tar\ (fs) +\!\!+ [(p, b)])$

--

$gNewCata :: [(a, b + [(Path\ a, b)])] \rightarrow [(Path\ a, b)]$
$gNewCata\ [\,] = [\,]$
$gNewCata\ ((a, i_1\ b) : t) = [([a], b)] +\!\!+ (gNewCata\ t)$
$gNewCata\ ((a, i_2\ l) : t) = (addAllPaths\ a\ l) +\!\!+ (gNewCata\ t)$

--

$gNewAna :: [(Path\ a, b)] \rightarrow [(a, b + [(Path\ a, b)])]$
$gNewAna\ [\,] = [\,]$
$gNewAna\ (([x], b) : t) = [(x, i_1\ b)] +\!\!+ (gNewAna\ t)$
$gNewAna\ (((h : t), b) : t1) = [(h, i_2\ [(t, b)])] +\!\!+ (gNewAna\ t1)$

f) $cp :: Path\ a \rightarrow Path\ a \rightarrow FS\ a\ b \rightarrow FS\ a\ b$

$cp :: (Eq\ a) \Rightarrow Path\ a \rightarrow Path\ a \rightarrow FS\ a\ b \rightarrow FS\ a\ b$
$cp\ list1\ list2\ (FS\ list3) = new\ list2\ (encontraFile\ list1\ (tar\ (FS\ list3)))\ (FS\ list3)$

--

$encontraFile :: (Eq\ a) \Rightarrow Path\ a \rightarrow [(Path\ a, b)] \rightarrow b$
$encontraFile\ list\ ((x, y) : t) \mid list \equiv x = y$
$\quad \mid otherwise = encontraFile\ list\ t$

g) $rm :: Path\ a \rightarrow FS\ a\ b \rightarrow FS\ a\ b$
Sugestão de implementação, definição de nav:

$nav :: (Eq\ a) \Rightarrow (Path\ a, FS\ a\ b) \rightarrow FS\ a\ b$
$nav = anaFS\ gNav$

--

$gNav :: (Eq\ a) \Rightarrow (Path\ a, FS\ a\ b) \rightarrow [(a, b + (Path\ a, FS\ a\ b))]$
$gNav\ ((h : t), FS\ [\,]) = [\,]$
$gNav\ ((h : t), FS\ ((a, File\ b) : t1)) \mid (h \equiv a) = [(a, i_1\ b)]$
$\qquad \mid otherwise = gNav\ (h : t, FS\ t1)$
$gNav\ ((h : t), FS\ ((a, Dir\ f) : t1)) \mid (h \equiv a) = [(a, i_2\ (t, f))]$
$\qquad \mid otherwise = gNav\ (h : t, FS\ t1)$

Definição de rm:

$rm :: (Eq\ a) \Rightarrow (Path\ a) \rightarrow (FS\ a\ b) \rightarrow FS\ a\ b$
$rm = \perp$

Definições extras:

$auxJoin :: ([(a, b + c)], d) \rightarrow [(a, b + (d, c))]$
$auxJoin = \perp$

$cFS2Exp :: a \rightarrow FS\ a\ b \rightarrow (Exp\ ()\ a)$
$cFS2Exp = \perp$

References

[1] D.E. Knuth. *Literate Programming*. CSLI Lecture Notes Number 27. Stanford University Center for the Study of Language and Information, Stanford, CA, USA, 1992.

[2] J.N. Oliveira. *Program Design by Calculation*, 2018. Draft of textbook in preparation. viii+297 pages. Informatics Department, University of Minho.