2ª Lista de Exercícios de Paradigmas de Linguagens Computacionais

Professor: Fernando Castor

Monitores:

Ciência da computação: Eric Lucena Palmeira Silva (elps), João Gabriel Santiago Maurício de Abreu (jgsma), Marcos Paulo Barros Barreto (mpbb), Tulio Paulo Lages da Silva (tpls),Victor Félix Pimenta (vfp).

CIn-UFPE – 2014.1
Disponível desde: 10/05/2014
Entrega: 22/05/2014

A lista deverá ser respondida em dupla. A falha em entregar a lista até a data estipulada implicará na **perda de 0,25 ponto** na média da disciplina para os membros da dupla. Considera-se que uma lista na qual **menos que 6** das respostas estão corretas não foi entregue. A entrega da lista com **pelo menos 11** das questões corretamente respondidas implica em um **acréscimo de 0,125 ponto** na média da disciplina para os membros da dupla. A entrega da lista com pelo menos 12 das questões corretamente respondidas, onde uma delas é a questão desafio (número 15) implica em um **acréscimo de 0,2 ponto** na média da disciplina para os membros da dupla. Se **qualquer situação de cópia de respostas for identificada**, os membros de todas as duplas envolvidas **perderão 0,5 ponto** na média da disciplina. O mesmo vale para respostas obtidas a partir da Internet. As respostas deverão ser entregues **exclusivamente em formato texto ASCII** (nada de .pdf, .doc, .docx ou .odt) e deverão ser enviadas para o monitor responsável por sua dupla, **sem** cópia para o professor. Devem ser organizadas em arquivos separados, um por questão, entregues em um único arquivo compactado, ou seja, um único arquivo .zip contendo as respostas para todas as questões. Um membro de cada dupla deve ir até a página da monitoria correspondente (CC ou EC) e registrar os nomes e logins dos membros da sua dupla sob o nome de um monitor. A escolha do monitor deve seguir uma política round-robin de modo a balancear a carga de duplas entre os monitores da maneira mais equitativa possível. A não-observância desta política pode resultar na transferência da dupla para outro monitor.

1) Uma imagem digital é a representação de uma imagem bidimensional usando números binários codificados. Uma das formas de codificar uma imagem colorida é usando o sistema RGB, onde cada pixel é representado por três valores, definindo assim a intensidade das camadas de cores vermelha (Red), verde (Green) e azul (Blue).
Um determinado banco pretende digitalizar todo o seu arquivo de documentos e chamou você para executar essa tarefa. A tarefa era simples, até você ser avisado que, por questão de gastos, para armazenar as versões digitais dos documentos, estes terão que ser convertidos para imagens em preto e branco, para assim, economizar espaço.
O seu trabalho é fazer um programa que receba uma imagem colorida de 24 bits e converta a mesma para uma versão em preto e branco.

Essa transição é feita em dois passos:
1. Converter a imagem em uma equivalente em tons de cinza (Grayscale).
2. Com a versão em grayscale, executar a conversão para preto e branco (B&W)

A conversão de RGB para Grayscale é facilmente feita através da seguinte fórmula:
GS = 0.299*R + 0.587*G + 0.114*B

Já para converter uma imagem em grayscale para preto e branco, uma das formas mais simples é através da média da imagem. A partir do valor obtido, os pixels cujo tom de cinza é igual ou menor que a média se torna preto (False) e qualquer pixel com o valor acima se torna branco (True).

A média de uma imagem é dada pela soma dos valores em grayscale de todos os pixels, dividida pelo número de pixels.

De posse dessas informações, é pedido que você faça duas funções,

rgbToGs :: [[String]] -> [[String]]

Essa função deve receber uma matriz de pixels, onde cada um é representado por uma string binária de 24 bits, e retorna a mesma matriz em tons de cinza, onde cada pixel é representado por uma string binária de 8 bits.

gsToPb :: [[String]] -> [[Bool]]

Essa função deve receber uma matriz representando uma imagem em tons de cinza, e retornar sua conversão para preto e branco, representada através de uma matriz de valores booleanos.

2) Crie em haskell um avaliador recursivo para expressões numéricas simples. Esse avaliador deve funcionar com valores inteiros, e deve conseguir executar as seguintes operações:

- Soma
- Subtração
- Multiplicação
- Divisão
- Módulo

Você tem liberdade de definir a representação das expressões. Explique em forma de comentários suas decisões, e elabore 5 expressões de teste que funcionem com a sua definição.

Abaixo segue um exemplo de execução de um desses avaliadores.

Main> eval $ Exprs [Exprs [Int 1, Operator “+”, Int 4], Operator “*”, Int 2]
10

3) Árvores Binárias são estruturas simples, porém muito interessantes, que podem ser representadas em Haskell da seguinte maneira:

data Tree t = Node t (Maybe (Tree t)) (Maybe (Tree t)) deriving (Show, Eq)

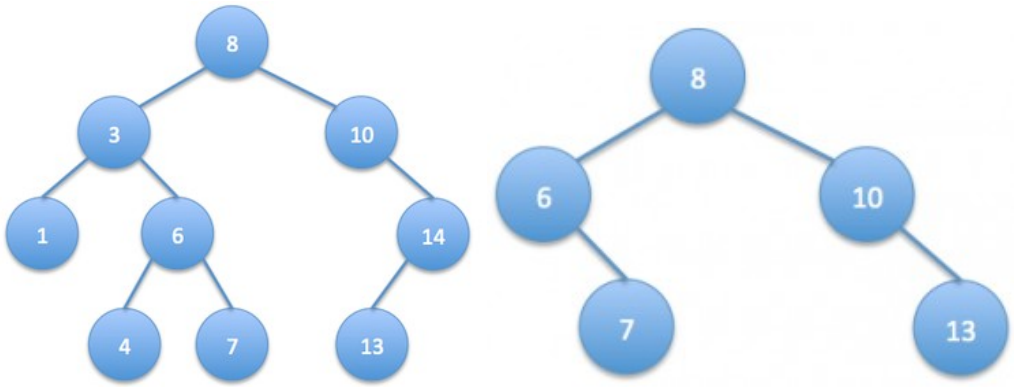
Utilizando o Tipo recursivo acima, implemente as seguintes funções sobre árvores binárias:

a) A função **trim** :: (Ord t) => t -> t -> Maybe (Tree t) -> Maybe (Tree t):

Recebe uma árvore e dois valores min e max, e poda a árvore de modo que todos os nós na árvore fiquem entre min e max (inclusive).

Exemplos:

1 - Dada a árvore na esquerda e valores min e max 5 e 13, respectivamente, obtemos a árvore na direita



b) A função **symmetric** :: (Tree t) -> Bool

Uma árvore é dita simétrica se, desenhando-se uma linha vertical que passe pela sua raiz, a subárvore esquerda for simétrica à subárvore direita. Em outras palavras, a subárvore esquerda é como a reflexão da subárvore direita em um espelho, e vice-versa. Implemente a função **symmetric**, que recebe uma árvore binária e retorna True se ela for simétrica.

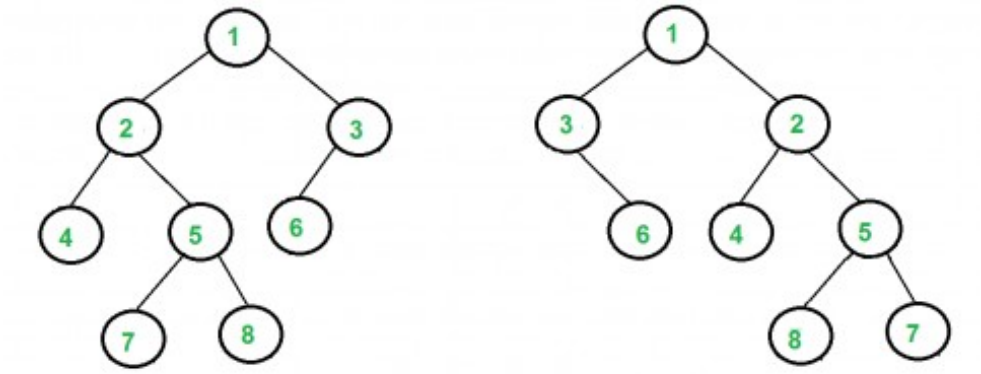
OBS: Os conteúdos dos nós não importam, a função **symmetric** indica apenas se a estrutura da árvore é simétrica.

c) A função **isomorphic** :: (Eq t) => Tree t -> Tree t -> Bool

Duas árvores são ditas isomórficas se uma puder ser obtida através da outra após uma série de “trocas” dos filhos esquerdo e direito de um mesmo nó, para um número qualquer de nós.

Exemplo:

As duas árvores abaixo são isomórficas:



4) A Lógica de Primeira Ordem (LPO) possui um papel fundamental em diversas áreas da computação, por ser um sistema formal com grande capacidade expressiva. Nela, sentenças podem ser formadas por sentenças menores ligadas por conectivos lógicos, sendo estes conjunção (∧), disjunção (∨), implicação (→) e negação (¬). Sentenças podem também conter quantificadores (∃ e ∀). Sentenças atômicas, as menores possíveis, são formadas por um predicado ou por uma igualdade de termos. Um termo, por sua vez, nada mais é do que uma representação de um elemento do domínio em questão e pode ser uma variável, uma constante ou uma função, cujos parâmetros são outros termos. Sentenças, quando associadas à uma Interpretação, possuem um valor de Verdadeiro ou Falso. Já uma Interpretação pode ser vista como uma estrutura que mapeia constantes, predicados e funções a símbolos que os representem, contidos em sentenças. Sabendo dessas informações, crie a função **value** :: Sentence -> Interpretation t -> Bool que, dada uma sentença e uma interpretação, informa se a sentença é verdadeira nessa interpretação.

OBS$_1$: Escolha a maneira que achar mais adequada para representar os componentes da LPO. Justifique essas escolhas em forma de comentário no seu código.

OBS$_2$: Para o propósito desta questão, todos os termos serão constantes ou funções, não havendo variáveis em nenhuma sentença. Consequentemente, também não haverá quantificadores.

OBS3: Para mais informações sobre Lógica de Primeira Ordem: http://en.wikipedia.org/wiki/First-order_logic

5) O Sr. Dengklek trabalha como um fabricante de correntes. Hoje, desocupado, ele gostaria de fazer uma linda corrente decorativa para presentear uma amiga. Ele fará a corrente a partir de sobras que encontrou em sua oficina, cada uma sendo um pedaço de corrente com exatamente 3 elos. Cada elo pode estar limpo ou enferrujado, e diferentes elos limpos podem ter diferentes graus de beleza.

Você recebe uma lista de strings que descreve as sobras, onde cada elemento é uma String de 3 caracteres que descreve um pedaço de corrente. Um elo enferrujado é representado por um ponto (‘.’), enquanto um elo limpo é representado por um dígito (‘0’ - ‘9’), que representa a beleza daquele elo. Por exemplo, [“.15”, “7..”, “532”, “..3”] significa que Sr. Dengklek tem 4 pedaços de corrente, e somente um deles (“532”) não possui elos enferrujados.

Todos os elos possuem o mesmo formato, o que permite ao Sr. Dengklek concatenar quaisquer dois pedaços de corrente. Entretanto, o formato dos elos não é simétrico, portanto

ele não pode reverter os pedaços de corrente. E.g. no exemplo acima, ele pode produzir a corrente “532.15”, ou a corrente “.15..37..”, mas ele não pode produzir “5323..”.

Para produzir uma corrente, o Sr. Dengklek seguirá os seguintes passos:

1. Concatenar todos os pedaços de corrente em qualquer ordem.
2. Escolher uma sequência contígua de elos que **não** contém elos enferrujados. Remover e descartar todos os outros elos.
3. A beleza da nova corrente é a beleza total (soma) de todos os elos escolhidos no segundo passo. Obviamente, o Sr. Dengklek gostaria de criar a corrente mais bela possível.

Crie a função **mostBeautifulChain** :: [String] -> Int, que recebe uma lista de pedaços de corrente e retorna a maior beleza que uma corrente pode ter de acordo com as regras anteriores. Cada pedaço de corrente terá 3 caracteres.

OBS1: O Sr. Dengklek não pode remover e descartar elos individuais antes de concatenar os pedaços de corrente.

OBS2: Se todos os elos da entrada estiverem enferrujados, o Sr. Dengklek é forçado a selecionar uma sequência vazia de elos. A beleza de uma sequência vazia é 0.

Exemplos:

```
Prelude> mostBeautifulChain [".15", "7..", "402", "..3"]
19
Prelude> mostBeautifulChain ["..1", "7..", "567", "24.", "8..", "234"]
36
Prelude> mostBeautifulChain ["...", "..."]
0
Prelude> mostBeautifulChain ["16.", "9.8", ".24", "52.", "3.1", "532", "4.4", "111"]
28
Prelude> mostBeautifulChain ["..1", "3..", "2..", ".7."]
7
```

6) Faça uma função chamada **jogoDaVelha** que receba uma String representando uma dificuldade*. (caso não seja uma escolha válida, o jogo deve ser díficil) Então iniciar um jogo da velha:

http://pt.wikipedia.org/wiki/Jogo_da_velha
entre o jogador e a “máquina”, implementada por você de forma a representar a dificuldade*.
O jogo é montado entre turnos, onde os jogadores enviam coordenadas x y, representando a posição na matriz 3x3 que deseja alterar.
(Se desejarem podem adotar outra politica para envio das marcações, mas devem mandar comentado no código exemplo de uso e explicação da politica)

*Defina uma representação para o tabuleiro e suas marcações. (explique em comentários no código)

*Quando um jogador vence o jogo, ou acontece empate, ele deve ser encerrado e uma mensagem impressa na tela.

*A cada rodada o estado do tabuleiro deve ser impresso na tela.

dificuldades**:**
“media” -> A maquina deve marcar posições aleatórias normalmente, mas toda vez que o jogador tiver duas marcações em uma mesma linha/coluna/diagonal ela deve fugir da morte.
“dificil”-> A maquina deve começar a partida e nunca perder.

7) No modelo de redes livre de escala, cada nó inserido na rede, tenta fazer n conexões com nós já presentes na rede (para essa questão n deve ser definido como 2), com probabilidade de se conectar aos nós proporcional ao número de conexões que eles já possuírem, assim os nós mais conectados, tendem a fazer ainda mais conexões. Já no modelo regular, quando um nó i entra na rede, ele se conecta com o nó i-1, i+1 e 1+2, transpondo o limite da lista de nós se for primeiro ou último.
(caso não haja nós suficientes, deve fazer apenas as conexões possíveis)

a) Implemente uma função chamada **modeloMisto** que receba um inteiro, um facionário (p) e um PathFile. Essa função deve montar uma rede, adicionando nó a nó, até a quantidade do primeiro parâmetro. Cada nó adicionado tem uma probabilidade p de ser adicionado como rede livre de escala e 1-p de ser adicionado como rede regular. Caso p>=1, será montada uma rede livre de escala. Caso p <= 0, será montada uma rede regular. Então registre todas as arestas do grafo em um arquivo com o caminho PathFile da seguinte maneira. Se o nó 1 se conecta ao nó 3. Devem constar duas linhas distintas no arquivo:
1,3
3,1

b)
Implemente uma função receba um inteiro (n) e um inteiro (m) e leia um arquivo contendo um modelo de rede gerado na letra a, então encontre e associe a cada nó uma coordenada 2d e comece a posicionar os nós. Inicialmente cada nó receberá uma posição aleatória, com coordenadas variando entre (+-m,+-m). Então cada nó deve buscar em uma vizinhança circular de raio 4 o nó que mais se pareça com ele, e tentar andar 0.5 em sua direção, caso não fique a menos do que 0.4 do nó em questão. O inteiro n recebido pela primeira função representa o número de vezes que os nós farão essa busca na vizinhança e reposicionamento.
Após esse agrupamento a função de fato deve imprimir, em um novo arquivo, os nós com suas posições finais.
ex:
1 (2.0, 3.0)
2 (4.0, 5.0)

Dicas:
(para números aleatórios pode usar randomRIO,
para imprimir no arquivo pode usar appendFile)

8) Defina uma funcao **passeioCavaleiros**, que receba um inteiro (m),como a definição de uma matriz de mxm posições, um par representando uma posição inicial, uma função **pulo** que recebe uma posição e uma lista de passos dados e devolva a lista com uma nova

posição. A função passeioCavaleiros devolve True, caso consiga a partir da aplicação da função pulo, achar um caminho que passe por cada casa do tabuleiro uma vez e retorne a posição inicial e false caso contrário.

*Considerar que funções pulo devolvem a mesma lista caso haja alguma repetição de posições.

Defina também uma função **pulo** que faça passeioCavaleiros retornar True para qualquer valor inicial.

9) Encontre os tipos de todas as funções abaixo. Depois, encontre os tipos das composições pedidas (é necessário saber o tipo de algumas das funções abaixo para se descobrir o resultado das composições).
Responda a essa questão em forma de comentário no código de resposta, assim como as justificativas/passo a passo usado para se descobrir os tipos de cada função/composição.
Lembre-se de não usar o GHCi para chegar o tipo das funções, por que na hora da prova você poderá ter que encontrá-los na mão. ;)

----------//----------

a) g a = if a == maxBound then minBound else succ a

b) p P a b = V (a+b)
   p M a b = V (a-b)
   p N a b = V (a*b)
   p D a 0 = U
   p D a b = V (a/b)

c) i xs = reverse $ tail (init (zip (f xs) (u xs)))

d) q [n] = [(C n, fromInteger n)]
   q ns = [(B e1 o e2, v) | (ns1, ns2) <- i ns,
        (e1, v1) <- q ns1,
        (e2, v2) <- q ns2,
        o <- [minBound..maxBound],
        not ((length ns2) > (length ns1)),
        v <- m (p o v1 v2)]

e) m U = []
   m (V a) = [a]

f) d l o e = [(b, round v) | (b, v) <- l, (abs $ (round v) - o) <= e]

g) a [] _ _ = error "Erro."
   a l o e = d (q l) o e

h) u [] = [[]]
   u c@(_:as) = [c] ++ u as

i) f x = reverse $ u x

----------//----------

OBS: Suponha que C e B são construtores para o tipo algébrico T1, que U e V sejam para o tipo algébrico T2 e que P, M, N e D sejam construtores para o tipo algébrico T3. Defina esses tipos e qualquer outra estrutura utilizada de forma com que o código compile!

Composições para se encontrar o tipo:

I) (.).map.funct
II) foldr.(+).next
III) (.).melhores.expressoes

10)O jogo Senha funciona da seguinte maneira: existem pinos de seis cores diferentes e o jogador deve adivinhar qual a sequência correta de quatro pinos que o desafiador escondeu. A cada rodada o desafiador faz uma combinação com quatro pinos coloridos, sem repetir as cores de cada pino, tentando adivinhar a senha. Ele recebe como resposta uma sequência quatro pinos brancos ou pretos. Pino branco significa que existe um pino na sequência escondida da cor do pino que o jogador selecionou, mas em outra posição. Pino preto significa que o pino está completamente correto. A ausência de pinos significa que não existe outro pino dessa cor na sequência escondida. O jogo termina ou quando o jogador acertar a sequência ou quando suas chances terminarem. Simule o jogo com a função "jogoSenha", que deve criar uma senha aleatória e pedir ao usuário um número máximo de turnos (que deve ser um número par). A cada turno, o usuário deve entrar com uma String contendo a sequência de cores escolhidas (cores possíveis: "az" para azul, "vm" para vermelho, "vd" para verde, "am" para amarelo, "la" para laranja e "ro" para rosa) e deve receber de volta uma String com os pinos brancos ("br"), pretos ("pr") ou sua ausência ("-"), indicando o quanto ele acertou e quantos turnos lhe resta. Caso o jogo não tenha acabado, o programa deve pedir novamente uma sequência de pinos ao usuário.

Obs: Crie o tipo Pino para guardar as cores para facilitar o andamento do jogo.

11) Dada a seguinte representação de uma árovore binária, crie as seguintes funções, usando exclusivamente recursão de cauda para percorrer a árovore:
data Tree t = Node t (Maybe (Tree t)) (Maybe (Tree t)) deriving (Show, Eq)
I)inorder :: (Ord t) => Maybe (Tree t) -> [t], que deve percorrer a árvore da esquerda pra direita, na ordem: lado esquerdo, valor do nó e lado direito;
II)postorder :: (Ord t) => Maybe (Tree t) -> [t], que deve percorrer a árvore da direita pra esquerda, vendo o lado esquerdo, depois o lado esquerdo e depois o valor do nó;
III)preorder :: (Ord t) => Maybe (Tree t) -> [t], que deve percorrer a árvore da esquerda para a direita, mas verificando o valor do nó antes do seu lado esquerdo;

IV)breadthfirstorder :: (Ord t) => Maybe (Tree t) -> [t], que deve percorrer a árvore e retornar a lista de nós da esquerda para a direita, mas agrupados do menos profundo para o mais profundo;

12) Algumas calculadoras, como a RPN, funcionam através do empilhamento de operações. Defina um tipo recursivo pilha, bem como funções de pop e push que serão utilizadas para manipular essa pilha. Então crie uma função que receba uma String contendo uma determinada operação matemática e retorne uma pilha contendo os operadores. O processo se dá pondo os operadores segundo a seguinte ordem:

"OperandoEsquerda Operador OperandoDireita" ->
Operador (topo da pilha)
OperandoDireita
OperandoEsquerda (base da pilha)

Obs: A precedência dos operadores é explícita (dada por parênteses).
Obs2: Quando não houver parênteses a precedência é dada sempre à operação mais a esquerda (sem hierarquia entre os operadores).
(http://www.ime.usp.br/~pf/mac0122-2003/aulas/rpn-calculator.html)
*Main> makeStack "(5 + 3) * 2 / (4 - 6)"
'/'
'-'
'6'
'4'
'*'
'2'
'+'
'3'
'5'

13) Implemente um simulador de TCG simplificado para haskell (um TCG consiste num duelo de cards entre 2 jogadores - http://en.wikipedia.org/wiki/Collectible_card_game).
Seu simulador deve apresentar os seguintes tipos:

- Card: pode ser do tipo "Lacaio" ou "Feitiço"
-- Lacaio: possui nome, custo de mana, ataque e vida.
-- Feitiço: possui nome, custo de mana e dano causado.
- Jogador: possui nome, uma lsita de cards, mana e vida.
- Campo: possui duas listas de lacaio* (para cada jogador).

E deve implementar a seguinte função:

turn :: (Jogador, Jogador, Campo) -> (Jogador, Jogador, Campo)

A cada turno deve-se verificar as lacaio já presentes em cada lado do campo, então as lacaio do jogador 1 atacam diretamente o jogador 2 e vice-versa. Só então os jogadores podem "evocar" seus cards, e.g. o jogador irá colocar em seu lado do campo tantas lacaio quanto puder**, ou seja, uma quantidade de lacaio cuja soma do custo de mana seja menor ou igual à mana do jogador. Os feitiços causam dano direto ao jogador adversário quando são "evocados", não precisam ser postos no campo.

*: As lacaio no campo são de um tipo diferente do tipo "Card".
**: A estratégia para decidir quais cards jogar fica a cargo da implementação (knapsack, força bruta, primeiros, últimos, etc), deve-se porém, ao menos, garantir que quando os cards tiverem um custo suficientemente baixo, seja possível "evocar" todos.

14) Jogos de partidas online usam um sistema de "matchmaking" (http://en.wikipedia.org/wiki/Matchmaking) para encontrar pessoas de capacidade equivalente para uma partida. Implemente uma função que seja capaz de, uma vez recebida uma lista de usuários emparelhe todos um duplas.

matchMaking :: [User] -> [(User, User)]

Cada usuário desse sistema deve ter um ranking (que define sua classificação de 1 a 25, sendo 1 o melhor) e um tempo de espera na fila para partidas (variando de 1 a 100 segundos). Os usuários são gerados automaticamente com valores aleatórios. A geração de números é feita pela seguinte função:

```
rng :: (Int, Int) -> IO(Int)
rng (i, j) = do
        r <- randomRIO(i,j)
        return r
```

Para encontrar as duplas é feito o seguinte algoritmo: Para cada usuário da fila procure por outro de mesmo ranking, em caso de empate o usuário que está na fila a mais tempo é escolhido. Se nenhum usuário for escolhido, então aumente o range do ranking (e.g começando com range 0 procura-se usuários de mesmo ranking, com range 1 procura-se com rank 1 maior, ou 1 menor) e procure novamente.
Crie uma função "main :: IO ()" para executar a questão.

15 - **DESAFIO**) Dada uma lista de números inteiros, encontre uma maneira correta de inserir sinais aritméticos (operadores) tal que o resultado é uma equação correta.
OBS: Divisão deve ser tratada como operando nos racionais e divisão por zero deve ser evitada.
Exemplos:

- Com a entrada [2, 3, 5, 7, 11] podemos ter como saída 2-(3-(5+7)) = 11, 2 = ((3*5)+7)/11, 2*(3-5) = 7-11, entre outras.
- Com a entrada [1, 2, 3, 4, 5] podemos ter como saída 1*(2-3) = 4-5, 1 = 2/(3+(4-5)), entre outras.

- Com a entrada [0, 1, 1, 2, 3, 5, 8, 13] podemos ter como saída 0+(1+(1+(2-(3*(5-8))))) = 13, 0+1 = 1+(2-(3+(5/(8-13)))), 0-(1*(1+2)) = 3*(5/(8-13)), entre outros.