2. Considere o seguinte inventário de quatro tipos de árvores:

   (a) Árvores com informação de tipo A nos nós:

   $$\mathsf{T} = \mathsf{BTree}\ A \qquad \begin{cases} \mathsf{F}\ X = 1 + A \times X^2 \\ \mathsf{F}\ f = id + id \times f^2 \end{cases} \qquad \mathsf{in} = [\underline{Empty}\ , Node]$$

   Haskell: **data** BTree a = $Empty$ | $Node$ $(a, (\mathsf{BTree}\ a, \mathsf{BTree}\ a))$

   (b) Árvores com informação de tipo A nas folhas:

   $$\mathsf{T} = \mathsf{LTree}\ A \qquad \begin{cases} \mathsf{F}\ X = A + X^2 \\ \mathsf{F}\ f = id + f^2 \end{cases} \qquad \mathsf{in} = [Leaf\ , Fork]$$

   Haskell: **data** LTree a = $Leaf$ a | $Fork$ $(\mathsf{LTree}\ a, \mathsf{LTree}\ a)$

   (c) Árvores com informação nos nós e nas folhas:

   $$\mathsf{T} = \mathsf{FTree}\ B\ A \qquad \begin{cases} \mathsf{F}\ X = B + A \times X^2 \\ \mathsf{F}\ f = id + id \times f^2 \end{cases} \qquad \mathsf{in} = [Unit\ , Comp]$$

   Haskell: **data** FTree b a = $Unit$ b | $Comp$ $(a, (\mathsf{FTree}\ b\ a, \mathsf{FTree}\ b\ a))$

   (d) Árvores de expressão:

   $$\mathsf{T} = Expr\ V\ O \qquad \begin{cases} \mathsf{F}\ X = V + O \times X^* \\ \mathsf{F}\ f = id + id \times \mathsf{map}\ f \end{cases} \qquad \mathsf{in} = [Var\ , Op]$$

   Haskell: **data** $Expr$ v o = Var v | Op $(o, [Expr\ v\ o])$

   Defina o gene g para cada um dos catamorfismos seguintes desenhando, para cada caso, o diagrama correspondente:

   - $zeros = (\!| g |\!)$ — substitui todas as folhas de uma árvore de tipo (2b) por zero.
   - $conta = (\!| g |\!)$ — conta o número de nós de uma árvore de tipo (2a).
   - $\mathsf{mirror} = (\!| g |\!)$ — espelha uma árvore de tipo (2b), i.e., roda-a de 180°.
   - $converte = (\!| g |\!)$ — converte árvores de tipo (2c) em árvores de tipo (2a) eliminando os Bs que estão na primeira.
   - $\mathsf{vars} = (\!| g |\!)$ — lista as variáveis de uma árvore expressão de tipo (2d).

Resolução / Haskell

In [1]:

```
-- loading Cp.hs
:opt no-lint
:load ../src/Cp.hs
:set -XNPlusKPatterns
```

In [2]:

```
data BTree a = Empty | Node (a, (BTree a, BTree a)) deriving (Show)

data LTree a = Leaf a | Fork (LTree a, LTree a) deriving (Show)

data FTree b a = Unit b | Comp (a, (FTree b a, FTree b a)) deriving (Show)

data Expr v o = Var v | Op (o, [Expr v o]) deriving (Show)
```

Resolução (zeros)

$k = zeros = (\!|g|\!) \equiv k.\, in = g.\, (id + (k \times k))$

{ def. in, fusão-$+$, etc. }

$[k.\, Leaf, k.\, Fork] = [g.\, i_1, g.\, i_2.\, (k \times k)]$

{ eq-$+$, **lei (27)** }

$\equiv k.\, Leaf = g.\, i_1; k.\, Fork = g.\, i2.(k \times k)$

{ igualdade extensional, **lei (71)** }

$$\equiv (k . Leaf)\ a = (g . i_1)\ a; (k . Fork)\ (a,b) = (g\ .\ i2\ .\ (k \times k))\ (a,b)$$

{ def-comp, **lei (72)**; def-$\times$, **lei (77)** }

$$\equiv k\ (Leaf\ a) = g\ (i_1\ a); k\ (Fork\ (a,b)) = g\ (i2\ (k\ a, k\ b))$$

In [3]:

```
:t Leaf
:t Fork

cata' g (Leaf a) = g . i1 $ a
cata' g (Fork (a,b)) = g . i2 $ (cata' g a, cata' g b)

-- type checking

:t cata'

-- defining g

g1 = const (Leaf 0)
g2 = Fork

-- defining zeros

zeros = cata' (either g1 g2)
:t zeros

-- testing

zeros (Fork (Fork (Leaf "a",Leaf ("b")),Leaf ("c")))

-- redefining g

g1' = singl . const 0
g2' = uncurry (++)
zeros' = cata' (either g1' g2')
:t zeros'

-- testing

zeros' (Fork (Fork (Leaf "a",Leaf ("b")),Leaf ("c")))
```

```
Leaf :: forall a. a -> LTree a
Fork :: forall a. (LTree a, LTree a) -> LTree a
cata' :: forall a b. (Either a (b, b) -> b) -> LTree a -> b
zeros :: forall a b. Num a => LTree b -> LTree a
Fork (Fork (Leaf 0,Leaf 0),Leaf 0)
zeros' :: forall a b. Num a => LTree b -> [a]
[0,0,0]
```

Resolução (conta)

$$k = conta = (\!|g|\!) \equiv k . in = g . (id + id \times (k \times k))$$

{ def. in, fusão-$+$, etc. }

$$[k . \underline{Empty}, k . Node] = [g . i_1, g . i_2 . (id \times (k \times k))]$$

{ eq-$+$, **lei (27)** }

$$\equiv k.\, \underline{Empty} = g.\, i_1 ; k.\, Node = g.\, i_2.\, (id \times (k \times k))$$

{ igualdade extensional, **lei (71)** }

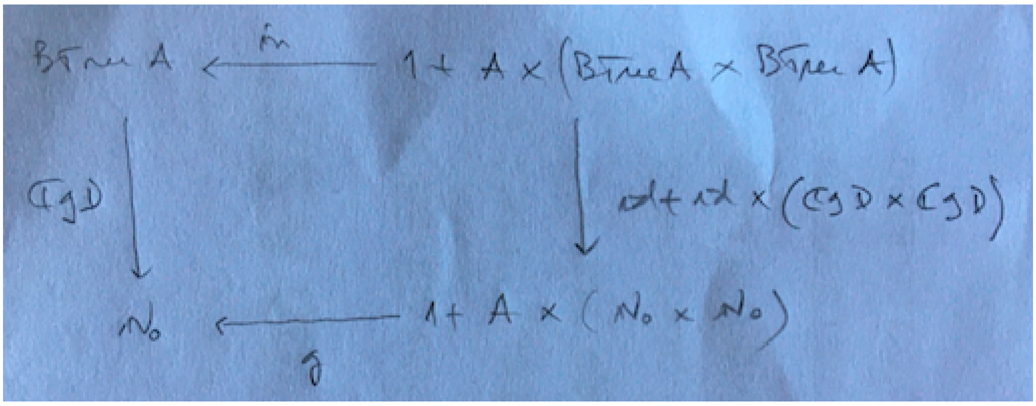
$$\equiv (k.\, \underline{Empty})\ a = (g.\, i_1)\ a; (k.\, Node)\ (a, (b, c)) = (g.\, i2.\, (id \times (k \times k)))\ (a, (b, c))$$

{ def-comp, **lei (72)**; def-$\times$, **lei (77)**, natural-id, **lei (1)** }

$$\equiv k\ (\underline{Empty}\ a) = g\ (i_1\ a); k\ (Node\ (a, (b, c))) = g(i2(a, (k\, b, k\, c)))$$

{ def. ! (bang), def-comp, **lei (72)**, def-id, **lei (73)**, def-const, **lei (74)** }

$$\equiv k\ (Empty) = g\ (i_1\ ()); k\ (Node\ (a, b, c)) = g(i2(a, (k\, b, k\, c)))$$



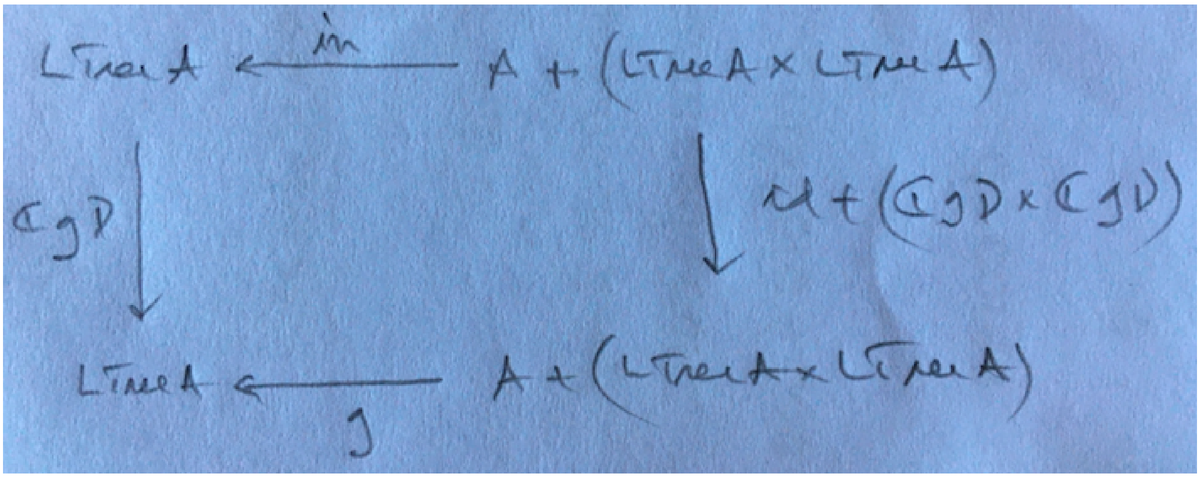
In [4]:

```
:t Empty
:t Node

cata'' g (Empty) = g . i1 $ ()
cata'' g (Node (a,(b,c))) = g . i2 $ (a, (cata'' g b, cata'' g c))

-- type checking

:t cata''

-- defining g

g1 = (const 0)
g2 = \(a,(b,c)) -> 1+b+c

-- defining conta

conta = cata'' (either g1 g2)
:t conta

-- testing

conta (Node ("a",(Node ("b",(Empty,Empty)),Node ("c",(Empty,Empty)))))
```

```
Empty :: forall a. BTree a
Node :: forall a. (a, (BTree a, BTree a)) -> BTree a
cata'' :: forall a b. (Either () (a, (b, b)) -> b) -> BTree a -> b
conta :: forall b a. Num b => BTree a -> b
```

3

Resolução (mirror)



In [5]:

```haskell
-- defining g

g1 = Leaf
g2 = Fork . swap

-- defining mirror

mirror = cata' (either g1 g2)
:t mirror

-- testing

mirror (Fork (Fork (Leaf 1,Leaf (2)),Leaf (3)))
```

mirror :: forall a. LTree a -> LTree a

```
Fork (Leaf 3,Fork (Leaf 2,Leaf 1))
```

Resolução (converte)

In [6]:

```haskell
:t Unit
:t Comp

cataFTree g (Unit b) = g . i1 $ b
cataFTree g (Comp (a,(b,c))) = g . i2 $ (a,(cataFTree g b, cataFTree g c))

-- type checking

:t cataFTree

-- defining converte

converte = cataFTree (either (const Empty) Node)
:t converte

-- testing

converte (Unit "a")
converte (Comp ("a", (Unit "b", Unit "c")))
converte (Comp ("a", (Comp ("a", (Unit "b", Unit "c")), Unit "c")))
```

Unit :: forall b a. b -> FTree b a

```
Comp :: forall a b. (a, (FTree b a, FTree b a)) -> FTree b a
cataFTree :: forall a1 a2 b. (Either a1 (a2, (b, b)) -> b) -> FTree a1
a2 -> b
converte :: forall b a. FTree b a -> BTree a
```

```
Empty
Node ("a",(Empty,Empty))
Node ("a",(Node ("a",(Empty,Empty)),Empty))
```

Resolução (vars)

In [7]:

```
:t Var
:t Op

cataExpr g (Var v) = g . i1 $ v
cataExpr g (Op (o,l)) = g . i2 $ (o,map (cataExpr g) l)

:t cataExpr

-- defining vars

vars = cataExpr (either singl (concat . p2))

-- testing

x = Op ("+",[Var "a",Var "b", Var "c"])
vars x
```

```
Var :: forall v o. v -> Expr v o
Op :: forall o v. (o, [Expr v o]) -> Expr v o
cataExpr :: forall a1 a2 b. (Either a1 (a2, [b]) -> b) -> Expr a1 a2 -
> b
```

```
["a","b","c"]
```