cp2021t-0.1.0.0: Trabalho para Calculo Proposicional da Universidade do Minho

Trabalho para Calculo Proposicional da Universidade do Minho

Contents

Chapter 1

BTree

```
module BTree (
    BTree(Node, Empty), inBTree, outBTree, recBTree, cataBTree,
    anaBTree, hyloBTree, baseBTree, invBTree, countBTree, inordt,
    inord, preordt, preord, postordt, qSort, qsep, part, traces,
    tunion, hanoi, present, strategy, balBTree, depthBTree, baldepth,
    tnat, monBTree, preordt', countBTree', Deriv(Dr), Zipper, plug
  ) where
```

```
data BTree a
```

Constructors

```
= Empty
| Node (a, (BTree a, BTree a))
```

```
instance Functor BTree
instance Show a => Show (BTree a)
```

```
inBTree :: Either () (b, (BTree b, BTree b)) -> BTree b
```

```
outBTree :: BTree a -> Either () (a, (BTree a, BTree a))
```

```
recBTree :: (a -> d) -> Either b1 (b2, (a, a)) -> Either b1 (b2, (d, d))
```

```
cataBTree :: (Either () (b, (d, d)) -> d) -> BTree b -> d

anaBTree :: (a -> Either () (b, (a, a))) -> a -> BTree b

hyloBTree :: (Either () (b, (c, c)) -> c) -> (a -> Either () (b, (a, a))) -> a -> c


baseBTree :: (a1 -> b1) -> (a2 -> d) -> Either b2 (a1, (a2, a2)) -> Either b2 (b1, (d, d))


invBTree :: BTree b -> BTree b

countBTree :: BTree a -> Integer

inordt :: BTree a -> [a]

inord :: Either b (a, ([a], [a])) -> [a]

preordt :: BTree a -> [a]

preord :: Either b (a, ([a], [a])) -> [a]

postordt :: BTree a -> [a]

qSort :: Ord a => [a] -> [a]

qsep :: Ord a => [a] -> Either () (a, ([a], [a]))

part :: (a -> Bool) -> [a] -> ([a], [a])

traces :: Eq a => BTree a -> [[a]]

tunion :: Eq a => (a, ([[a]], [[a]])) -> [[a]]

hanoi :: (Bool, Integer) -> [(Integer, Bool)]

present :: Either b (a, ([a], [a])) -> [a]

strategy :: Integral b => (Bool, b) -> Either () ((b, Bool), ((Bool, b), (Bool, b)))


balBTree :: BTree a -> Bool

depthBTree :: BTree a -> Integer

baldepth :: BTree a -> (Bool, Integer)

tnat :: Monoid c => (a -> c) -> Either () (a, (c, c)) -> c

monBTree :: Monoid d => (a -> d) -> BTree a -> d

preordt' :: BTree a -> [a]

countBTree' :: BTree b -> Sum Integer
```

```
data Deriv a
```

Constructors

```
=  Dr Bool a (BTree a)
```

```
type Zipper a = [Deriv a]
```

```
plug :: Zipper a -> BTree a -> BTree a
```

Chapter 2

Cp

```
module Cp (
    split, (><), (!), p1, p2, i1, i2, (-|-), cond, ap, expn, p2p,
    grd, swap, assocr, assocl, undistr, undistl, flatr, flatl, br, bl,
    coswap, coassocr, coassocl, distl, distr, BiFunctor(bmap), (.!),
    mult, ap', singl, Strong(lstr, rstr), dstr, splitm, bang, dup,
    zero, one, nil, cons, add, mul, conc, true, nothing, false,
    inMaybe, Unzipable(unzp), DistL(lamb), aap, gather, cozip, tot
  ) where
```

```
split :: (a -> b) -> (a -> c) -> a -> (b, c)

(><) :: (a -> b) -> (c -> d) -> (a, c) -> (b, d)

(!) :: a -> ()

p1 :: (a, b) -> a

p2 :: (a, b) -> b

i1 :: a -> Either a b

i2 :: b -> Either a b

(-|-) :: (a -> b) -> (c -> d) -> Either a c -> Either b d

cond :: (b -> Bool) -> (b -> c) -> (b -> c) -> b -> c
```

```
ap :: (a -> b, a) -> b

expn :: (b -> c) -> (a -> b) -> a -> c

p2p :: (a, a) -> Bool -> a

grd :: (a -> Bool) -> a -> Either a a

swap :: (a, b) -> (b, a)

assocr :: ((a, b), c) -> (a, (b, c))

assocl :: (a, (b, c)) -> ((a, b), c)

undistr :: Either (a, b) (a, c) -> (a, Either b c)

undistl :: Either (b, c) (a, c) -> (Either b a, c)

flatr :: (a, (b, c)) -> (a, b, c)

flatl :: ((a, b), c) -> (a, b, c)

br :: a -> (a, ())

bl :: a -> ((), a)

coswap :: Either a b -> Either b a

coassocr :: Either (Either a b) c -> Either a (Either b c)

coassocl :: Either b (Either a c) -> Either (Either b a) c

distl :: (Either c a, b) -> Either (c, b) (a, b)

distr :: (b, Either c a) -> Either (b, c) (b, a)


class BiFunctor f where
```

Methods

```
     bmap :: (a -> b) -> (c -> d) -> f a c -> f b d


instance BiFunctor Either
instance BiFunctor (,)


(.!) :: Monad a => (b -> a c) -> (d -> a b) -> d -> a c

mult :: Monad m => m (m b) -> m b

ap' :: Monad m => (a -> m b, m a) -> m b
```

```
singl :: a -> [a]
```

```
class (Functor f, Monad f) => Strong f where
```

Methods

```
rstr :: (f a, b) -> f (a, b)

lstr :: (b, f a) -> f (b, a)
```

```
instance Strong []
instance Strong Maybe
instance Strong IO
instance Strong LTree
```

```
dstr :: Strong m => (m a, m b) -> m (a, b)

splitm :: Strong ff => ff (a -> b) -> a -> ff b

bang :: a -> ()

dup :: c -> (c, c)

zero :: Num a => b -> a

one :: Num a => b -> a

nil :: b -> [a]

cons :: (a, [a]) -> [a]

add :: Num c => (c, c) -> c

mul :: Num c => (c, c) -> c

conc :: ([a], [a]) -> [a]

true :: b -> Bool

nothing :: b -> Maybe a

false :: b -> Bool

inMaybe :: Either () a -> Maybe a
```

```
class Functor f => Unzipable f where
```

Methods

```
    unzp :: f (a, b) -> (f a, f b)
```

```
class Functor g => DistL g where
```

Methods

```
    lamb :: Monad m => g (m a) -> m (g a)
```

```
instance DistL []
instance DistL Maybe
```

```
aap :: Monad m => m (a -> b) -> m a -> m b
```

```
gather :: [a -> b] -> a -> [b]
```

```
cozip :: Functor f => Either (f a) (f b) -> f (Either a b)
```

```
tot :: (a -> b) -> (a -> Bool) -> a -> Maybe b
```

Chapter 3

LTree

```
module LTree (
    LTree(Fork, Leaf), inLTree, outLTree, recLTree, baseLTree,
    cataLTree, anaLTree, hyloLTree, invLTree, countLTree, tips, dfac,
    dfacd, dsq', dsq, fib, fibd, mSort, merge, lsplit, dmap, dmap1, mu,
    tnat, monLTree, tips', countLTree', dlLTree, Deriv(Dr), Zipper,
    plug
  ) where
```

data LTree a

Constructors

```
= Leaf a
| Fork (LTree a, LTree a)
```

```
instance Monad LTree
instance Functor LTree
instance Applicative LTree
instance Strong LTree
instance Eq a => Eq (LTree a)
instance Show a => Show (LTree a)
```

inLTree :: Either a (LTree a, LTree a) -> LTree a

outLTree :: LTree a -> Either a (LTree a, LTree a)

```
recLTree :: (a -> d) -> Either b (a, a) -> Either b (d, d)

baseLTree :: (a1 -> b) -> (a2 -> d) -> Either a1 (a2, a2) -> Either b (d, d)


cataLTree :: (Either b (d, d) -> d) -> LTree b -> d

anaLTree :: (a1 -> Either a2 (a1, a1)) -> a1 -> LTree a2

hyloLTree :: (Either b (c, c) -> c) -> (a -> Either b (a, a)) -> a -> c


invLTree :: LTree a -> LTree a

countLTree :: LTree b -> Integer

tips :: LTree a -> [a]

dfac :: Integral p => p -> p

dfacd :: Integral b => (b, b) -> Either b ((b, b), (b, b))

dsq' :: Integral p => p -> p

dsq :: Integral p => p -> p

fib :: Integer -> Integer

fibd :: (Ord b, Num b) => b -> Either () (b, b)

mSort :: Ord a => [a] -> [a]

merge :: Ord a => ([a], [a]) -> [a]

lsplit :: [a] -> Either a ([a], [a])

dmap :: (b -> a) -> [b] -> [a]

dmap1 :: (b -> a) -> [b] -> [a]

mu :: LTree (LTree a) -> LTree a

tnat :: Monoid c => (a -> c) -> Either a (c, c) -> c

monLTree :: Monoid d => (a -> d) -> LTree a -> d

tips' :: LTree a -> [a]

countLTree' :: LTree b -> Sum Integer

dlLTree :: Strong f => LTree (f a) -> f (LTree a)
```

```
data Deriv a
```

Constructors

```
= Dr Bool (LTree a)
```

```
type Zipper a = [Deriv a]
```

```
plug :: Zipper a -> LTree a -> LTree a
```

Chapter 4

List

```
module List (
    inList, outList, cataList, recList, anaList, hyloList,
    baseList, eval, invl, look, iSort, take', fac, algMul, nats, fac',
    sq, summing, odds, sq', prefixes, suffixes, diff, nest, myfoldr,
    myfoldl, nr, ccat, mmap, lam, mcataList, dl, stream, join, sep
  ) where
```

```
inList :: Either b (a, [a]) -> [a]

outList :: [a] -> Either () (a, [a])

cataList :: (Either () (b, d) -> d) -> [b] -> d

recList :: (c -> d) -> Either b1 (b2, c) -> Either b1 (b2, d)

anaList :: (c -> Either b (a, c)) -> c -> [a]

hyloList :: (Either () (b1, c1) -> c1) -> (c2 -> Either b2 (b1, c2)) -> c2 -> c1


baseList :: (a -> b1) -> (c -> d) -> Either b2 (a, c) -> Either b2 (b1, d)


eval :: Num d => d -> [d] -> d

invl :: [a] -> [a]
```

```
look :: Eq a => a -> [(a, b)] -> Maybe b

iSort :: Ord a => [a] -> [a]

take' :: Integer -> [a] -> [a]

fac :: Integer -> Integer

algMul :: Either b (Integer, Integer) -> Integer

nats :: Integer -> Either () (Integer, Integer)

fac' :: Integer -> Integer

sq :: Integer -> Integer

summing :: Either b (Integer, Integer) -> Integer

odds :: Integer -> Either () (Integer, Integer)

sq' :: Integer -> Integer

prefixes :: Eq a => [a] -> [[a]]

suffixes :: [a] -> [[a]]

diff :: Eq a => [a] -> [a] -> [a]

nest :: Eq a => Int -> [a] -> [[a]]

myfoldr :: (a -> b -> b) -> b -> [a] -> b

myfoldl :: (a -> b -> a) -> a -> [b] -> a

nr :: Eq a => [a] -> Bool

ccat :: [a] -> [a] -> [a]

mmap :: Strong m => (a1 -> m a2) -> [a1] -> m [a2]

lam :: Strong m => [m a] -> m [a]

mcataList :: Strong ff => (Either () (b, c) -> ff c) -> [b] -> ff c

dl :: Strong m => Either () (b, m a) -> m (Either () (b, a))

stream :: (t1 -> Maybe (a, t1)) -> (t1 -> t2 -> t1) -> t1 -> [t2] -> [a]


join :: ([a], [b]) -> [Either a b]

sep :: [Either a1 a2] -> ([a1], [a2])
```

Chapter 5

Nat

```
module Nat (
    inNat, outNat, cataNat, recNat, anaNat, hyloNat, for, somar,
    multip, exp, sq, sq', sq'', fac, facfor, idiv, aux, bSort, while,
    mfor
  ) where
```

```
inNat :: Either b Integer -> Integer

outNat :: Integral b => b -> Either () b

cataNat :: Integral c => (Either () d -> d) -> c -> d

recNat :: (c -> d) -> Either b c -> Either b d

anaNat :: (c -> Either b c) -> c -> Integer

hyloNat :: (Either () c1 -> c1) -> (c2 -> Either b c2) -> c2 -> c1

for :: Integral c => (b -> b) -> b -> c -> b

somar :: (Integral c, Enum d) => d -> c -> d

multip :: (Integral c, Num d) => d -> c -> d

exp :: (Integral c, Num d) => d -> c -> d

sq :: Integral p => p -> p
```

```
sq' :: Integer -> Integer

sq'' :: (Integral c, Num b) => c -> b

fac :: Integer -> Integer

facfor :: Integer -> (Integer, Integer)

idiv :: Integer -> Integer -> Integer

aux :: (Ord c, Num c) => c -> c -> Integer

bSort :: Ord a => [a] -> [a]

while :: (a -> Bool) -> (a -> a) -> a -> a

mfor :: forall t1 m t2. (Monad m, Integral t1) => (t2 -> m t2) -> t2 -> t1 -> m t2
```

Chapter 6

Solucoes

```
module Solucoes (
    ExpAr(Un, N, X, Bin), BinOp(Product, Sum), UnOp(Negate, E),
    inExpAr, baseExpAr, cataExpAr, anaExpAr, hyloExpAr, expd, eval_exp,
    optmize_eval, sd, ad, (×), (×), ( ), ( ), ( ), BinExp, UnExp,
    OutExpAr, Injective(to), outExpAr, recExpAr, g_eval_exp, clean,
    gopt, Dup, Bin, Un, bin_aux, un_aux, sd_gen, ad_gen,
    prop_in_out_idExpAr, prop_out_in_idExpAr, prop_sum_idr,
    prop_sum_idl, prop_product_idr, prop_product_idl, prop_e_id,
    prop_negate_id, prop_double_negate,
    prop_optimize_respects_semantics, prop_const_rule, prop_var_rule,
    prop_sum_rule, prop_product_rule, prop_e_rule, prop_negate_rule,
    prop_congruent, fib', f', catdef, oracle, prop_cat, loop, inic,
    prj, cat, linear1d, NPoint, OverTime, prop_calcLine_def,
    prop_bezier_sym,  , to , from , calcLine, calc_line', myZipWithM,
    mySequenceA, myZipWith, myZip, outZip, mySequenceA', deCasteljau,
    hyloAlgForm, prop_avg, avg, avg_aux, avgLTree, e', bezier2d,
    World(World, points, time), initW, tick, actions, scaleTime,
    bezier2dAtTime, bezier2dAt, thicCirc, ps, picture, animateBezier,
    runBezier, runBezierSym, main, run, (.=?=.), (.==>.), (.<==>.),
    (.==.), (.<=.), (.&&&.)
  ) where
```

6.1 Problemas

6.1.1 Problema 1

Código Fornecido

```
data ExpAr a
```

Constructors

```
=  X
|  N a
|  Bin BinOp (ExpAr a) (ExpAr a)
|  Un UnOp (ExpAr a)
```

```
instance Eq a => Eq (ExpAr a)
instance Show a => Show (ExpAr a)
instance Arbitrary a => Arbitrary (ExpAr a)
instance Injective (ExpAr a) (OutExpAr a)
```

```
data BinOp
```

Constructors

```
=  Sum
|  Product
```

```
instance Eq BinOp
instance Show BinOp
instance Arbitrary BinOp
```

```
instance Num c => Injective BinOp ((c, c) -> c)
```

Check Symbol Interpretation

```
instance Injective (ExpAr a) (OutExpAr a)
```

```
data UnOp
```

Constructors

```
=  Negate
|  E
```

```
instance Eq UnOp
instance Show UnOp
instance Arbitrary UnOp
instance Floating c => Injective UnOp (c -> c)
```

Check Symbol Interpretation

```
instance Injective (ExpAr a) (OutExpAr a)
```

```
inExpAr :: (b   (a   (BinExp a   UnExp a))) -> ExpAr a
```

```
inExpAr   const X   N   bin   (Un  ) where bin (op, (a, b)) = Bin op a
b
```

```
baseExpAr :: (a -> b)
          -> (c -> d)
             -> (e -> f)
                -> (g -> h)
                   -> (i -> j)
                      -> (k -> l)
                         -> (m -> n)
                            -> (a   (c   ((e × (g × i))   (k × m))))
                               -> b   (d   ((f × (h × j))   (l × n)))
```

```
baseExpAr  f g h j k l z = f   g   h × j × k   l × z
```

```
cataExpAr :: ((()   (c   ((BinOp × (e × e))   (UnOp × e)))) -> e) -> ExpAr c -> e
```

```
anaExpAr :: (a -> b   (c   ((BinOp × (a × a))   (UnOp × a)))) -> a -> ExpAr c
```

```
hyloExpAr :: ((()   (c   ((BinOp × (d × d))   (UnOp × d)))) -> d)
          -> (a -> b   (c   ((BinOp × (a × a))   (UnOp × a)))) -> a -> d
```

```
expd :: Floating a => a -> a
```

```
eval_exp :: Floating a => a -> ExpAr a -> a
```

```
optmize_eval :: (Floating a, Eq a) => a -> ExpAr a -> a
```

```
sd :: Floating a => ExpAr a -> ExpAr a
```

```
ad :: Floating a => a -> ExpAr a -> a
```

Solução

```
type (×) a b = (a, b)
```

```
(×) :: (a -> b) -> (c -> d) -> (a, c) -> (b, d)
```

bimap for tuple

```
( ) :: (a -> b) -> (c -> d) -> (a   c) -> b   d
```

bimap for either

```
type ( ) = Either
```

```
( ) :: (a -> c) -> (b -> c) -> (a   b) -> c
```

```
type BinExp d = BinOp × (ExpAr d × ExpAr d)
```

```
BinOp × (ExpAr d × ExpAr d)
```

```
type UnExp d = UnOp × ExpAr d
```

```
type OutExpAr a = ()   (a   (BinExp a   UnExp a))
```

```
class Injective a b where
```

Methods

```
to :: forall b1 a1. (b1 ~ b, a1 ~ a) => a -> b
```

```
instance Floating c => Injective UnOp (c -> c)
```

Check Symbol Interpretation

```
instance Num c => Injective BinOp ((c, c) -> c)
```

Check Symbol Interpretation

```
instance Injective (ExpAr a) (OutExpAr a)
```

```
outExpAr :: ExpAr a -> OutExpAr a
```

```
recExpAr :: (a -> e)
            -> (b   (c   ((d × (a × a))   (g × a))))
               -> b   (c   ((d × (e × e))   (g × e)))
```

Read this as the "interpretation" of each `BinOp` symbol.

For example, the symbol `Sum` is interperted as the function `add`

```
g_eval_exp :: Floating c => c -> (b   (c   ((BinOp × (c × c))   (UnOp × c)))) -> c
```

```
clean :: (Eq a, Num a) => ExpAr a -> OutExpAr a
```

```
gopt :: Floating a => a -> (()   (a   ((BinOp × (a × a))   (UnOp × a)))) -> a
```

```
type Dup d = d × d
```

```
type Bin d = BinOp × (d × d)
```

```
type Un d = UnOp × d
```

```
bin_aux :: (t -> t -> t) -> (t -> t -> t) -> (BinOp, Dup (Dup t)) -> Dup t
```

```
un_aux :: (t1 -> t2) -> (t2 -> t1 -> t2) -> (t1 -> t2) -> (UnOp, Dup t1) -> Dup t2
```

```
sd_gen :: Floating a =>
          (()   (a   (Bin (Dup (ExpAr a))   Un (Dup (ExpAr a))))) -> Dup (ExpAr a)
```

```
ad_gen :: Floating a => a -> (()   (a   ((BinOp, Dup (Dup a))   (UnOp, Dup a)))) -> Dup a
```

Propriedades

```
prop_in_out_idExpAr :: Eq a => ExpAr a -> Bool
```

```
prop_out_in_idExpAr :: Eq a => OutExpAr a -> Bool
```

```
prop_sum_idr :: (Floating a, Real a) => a -> ExpAr a -> Bool
```

```
prop_sum_idl :: (Floating a, Real a) => a -> ExpAr a -> Bool
```

```
prop_product_idr :: (Floating a, Real a) => a -> ExpAr a -> Bool
```

```
prop_product_idl :: (Floating a, Real a) => a -> ExpAr a -> Bool
```

```
prop_e_id :: (Floating a, Real a) => a -> Bool
```

```
prop_negate_id :: (Floating a, Real a) => a -> Bool
```

```
prop_double_negate :: (Floating a, Real a) => a -> ExpAr a -> Bool
```

```
prop_optimize_respects_semantics :: (Floating a, Real a) => a -> ExpAr a -> Bool
```

```
prop_const_rule :: (Real a, Floating a) => a -> Bool

prop_var_rule :: Bool

prop_sum_rule :: (Real a, Floating a) => ExpAr a -> ExpAr a -> Bool

prop_product_rule :: (Real a, Floating a) => ExpAr a -> ExpAr a -> Bool

prop_e_rule :: (Real a, Floating a) => ExpAr a -> Bool

prop_negate_rule :: (Real a, Floating a) => ExpAr a -> Bool

prop_congruent :: (Floating a, Real a) => a -> ExpAr a -> Bool
```

6.1.2 Problema 2

Código Fornecido

```
fib' :: (Integral c, Num b) => c -> b

f' :: (Integral c, Num b) => b -> b -> b -> c -> b

catdef :: Integer -> Integer

oracle :: Num a => [a]
```

Propriedades

```
prop_cat :: Integer -> Property
```

Solução

```
loop :: Integral c => (c, c, c) -> (c, c, c)

inic :: (Num a, Num b, Num c) => (a, b, c)

prj :: (a, b, c) -> a

cat :: (Integral c1, Integral c2) => c1 -> c2
```

6.1.3 Problema 3

Código Fornecido

```
linear1d :: Rational -> Rational -> OverTime Rational
```

```
type NPoint = [Rational]
```

```
type OverTime a = Float -> a
```

Propriedades

```
prop_calcLine_def :: NPoint -> NPoint -> Float -> Bool
```

```
prop_bezier_sym :: [[Rational]] -> Gen Bool
```

Solução

```
type   = Rational
```

```
to  :: Real a => a -> Rational
```

```
from  :: Fractional a => Rational -> a
```

```
calcLine :: NPoint -> NPoint -> OverTime NPoint
```

Spec

```
calcLine :: NPoint -> (NPoint -> OverTime NPoint)
calcLine [] = const nil
calcLine (p : x) = curry g p (calcLine x)
 where
  g :: ( , NPoint -> OverTime NPoint) -> (NPoint -> OverTime NPoint)
  g (d, f) l = case l of
    [] -> nil
    (x : xs) -> concat . sequenceA [singl . linear1d d x, f xs]
```

```
calc_line' :: [ ] -> [ ] -> Float -> [ ]
```

```
myZipWithM :: (a1 -> b -> p -> a2) -> [a1] -> [b] -> p -> [a2]
```

```
mySequenceA :: [p -> a] -> p -> [a]
```

```
myZipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

```
myZip :: ([a], [b]) -> [(a, b)]

outZip :: ([a], [b]) -> ()  ((a, b), ([a], [b]))

mySequenceA' :: p -> [p -> a] -> [a]


deCasteljau :: [NPoint] -> OverTime NPoint
```

Spec

```
deCasteljau :: [NPoint] -> OverTime NPoint
deCasteljau [] = nil
deCasteljau [p] = const p
deCasteljau l = pt -> calcLine (p pt) (q pt) pt
 where
  p = deCasteljau (init l)
  q = deCasteljau (tail l)
```

```
hyloAlgForm :: a
```

6.1.4 Problema 4

Propriedades

```
prop_avg :: (Ord b, Fractional b) => [b] -> Property

avg :: Fractional b => [b] -> b
```

Solução

```
avg_aux :: Fractional b => [b] -> (b, b)

avgLTree :: LTree b -> c
```

6.2 Programação dinâmica por recursividade múltipla

```
e' :: (Fractional c1, Integral c2) => c1 -> c2 -> c1
```

6.3 Código Extra para Problema 3

6.3.1 2D

```
bezier2d :: [NPoint] -> OverTime (Float, Float)
```

6.3.2 Modelo

```
data World
```

Constructors

```
=  World
     { points :: [NPoint]
     , time :: Float
     }
```

```
initW :: World
```

```
tick :: Float -> World -> World
```

```
actions :: Event -> World -> World
```

```
scaleTime :: World -> Float
```

```
bezier2dAtTime :: World -> (Float, Float)
```

```
bezier2dAt :: World -> OverTime (Float, Float)
```

```
thicCirc :: Picture
```

```
ps :: [Float]
```

6.3.3 Gloss

```
picture :: World -> Picture
```

6.3.4 Animação

```
animateBezier :: Float -> [NPoint] -> Picture
```

6.3.5 Propriedades e main

```
runBezier :: IO ()
```

```
runBezierSym :: IO ()
```

Compilação e execução dentro do interpretador

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

```
run :: IO ExitCode
```

6.4 QuickCheck

6.5 Outras funções auxiliares

```
(.=?=.) :: Real a => a -> a -> Bool
```

```
(.==>.) :: Testable prop => (a -> Bool) -> (a -> prop) -> a -> Property
```

```
(.<==>.) :: (a -> Bool) -> (a -> Bool) -> a -> Property
```

```
(.==.) :: Eq b => (a -> b) -> (a -> b) -> a -> Bool
```

```
(.<=.) :: Ord b => (a -> b) -> (a -> b) -> a -> Bool
```

```
(.&&&.) :: (a -> Bool) -> (a -> Bool) -> a -> Bool
```