Gabarito

Capítulo 2

Exercício 1

```
area_circ (número) >> número
area_circ (r) << x_pi * r ^ 2
```

```
dec area_circ : num -> num;
--- area_circ r <= x_pi * pow (r, 2);
```

```
:{
area_circ :: (Floating a) => a -> a
area_circ r = x_pi * r ** 2
:}
```

Exercício 2

```
qsoma (número, número) >> número
qsoma (a, b) << (a + b) ^ 2
```

```
dec qsoma : num # num -> num;
--- qsoma (a, b) <= pow (a + b, 2);
```

```
:{
qsoma :: (Num a) => a -> a -> a
qsoma a b = (a + b) ^ 2
:}
```

Exercício 3

```
x_pi_em_e (número) >> número
x_pi_em_e (x) << (x * x_pi) / x_e
```

```
dec x_pi_em_e : num -> num;
--- x_pi_em_e x <= (x * x_pi) / x_e;
```

```
:{
x_pi_em_e :: (Floating a) => a -> a
x_pi_em_e x = (x * x_pi) / x_e
:}
```

Exercício 4

```
suc (número) >> número
suc (x) << x + 1
```

```
dec suc : num -> num;
--- suc x <= x + 1;
```

```
:{
suc :: (Num a) => a -> a
suc x = x + 1
:}
```

Exercício 5

```
c2f (número) >> número
c2f (c) << (c * 9 / 5) + 32
```

```
dec c2f : num -> num;
--- c2f c <= (c * 9 / 5) + 32;
```

```
:{
c2f :: (Floating a) => a -> a
c2f c = (c * 9 / 5) + 32
:}
```

Exercício 6

```
f2c (número) >> número
f2c (f) << (f - 32) * 5 / 9
```

```
dec f2c : num -> num;
--- f2c f <= (f - 32) * 5 / 9;
```

```
:{
f2c :: (Floating a) => a -> a
f2c f = (f - 32) * 5 / 9
:}
```

Exercício 7

```
c2k (número) >> número
c2k (c) << c + 273.15
```

```
dec c2k : num -> num;
--- c2k c <= c + 273.15;
```

```
:{
c2k :: (Floating a) => a -> a
c2k c = c + 273.15
:}
```

Exercício 8

```
k2c (número) >> número
k2c (k) << k - 273.15
```

```
dec k2c : num -> num;
```

```
--- k2c k <= k - 273.15;
```

```
:{
k2c :: (Floating a) => a -> a
k2c k = k - 273.15
:}
```

Exercício 9

```
metros2centímetros (número) >> número
metros2centímetros (m) << m * 100
```

```
dec metros2centimetros : num -> num;
--- metros2centimetros m <= m * 100;
```

```
:{
metros2centimetros :: (Floating a) => a -> a
metros2centimetros m = m * 100
:}
```

Exercício 10

```
são_iguais (número, número) >> lógico
são_iguais (a, b) << a = b
```

```
dec sao_iguais : num # num -> truval;
--- sao_iguais (a, b) <= a = b;
```

```
:{
sao_iguais :: (Eq a, Floating a) => a -> a -> Bool
sao_iguais a b = a == b
:}
```

Exercício 11

```
ant (número) >> número
ant (x) << x - 1
```

```
dec ant : num -> num;
--- ant x <= x - 1;
```

```
:{
ant :: (Num a) => a -> a
ant x = x - 1
:}
```

Exercício 12

```
cubo (número) >> número
cubo (x) << x ^ 3
```

```
dec cubo : num -> num;
--- cubo x <= pow (x, 3);
```

```
:{
cubo :: (Floating a) => a -> a
cubo x = x ** 3
:}
```

Exercício 13

```
k2f (número) >> número
k2f (k) << (k - 273.15) * 9 / 5 + 32
```

```
dec k2f : num -> num;
--- k2f k <= (k - 273.15) * 9 / 5 + 32;
```

```
:{
k2f :: (Floating a) => a -> a
k2f k = (k - 273.15) * 9 / 5 + 32
:}
```

Exercício 14

```
f2k (número) >> número
f2k (f) << (f - 32) * 5 / 9 + 273.15
```

```
dec f2k : num -> num;
--- f2k f <= (f - 32) * 5 / 9 + 273.15;
```

```
:{
f2k :: (Floating a) => a -> a
f2k f = (f - 32) * 5 / 9 + 273.15
:}
```

Exercício 15

```
imc (número, número) >> número
imc (p, a) << p / (a ^ 2)
```

```
dec imc : num # num -> num;
--- imc (p, a) <= p / pow (a, 2);
```

```
:{
imc :: (Floating a) => a -> a -> a
imc p a = p / (a ** 2)
:}
```

Exercício 16

```
produto (número, número) >> número
produto (x, y) << x * y
```

```
dec produto : num # num -> num;
--- produto (x, y) <= x * y;
```

```
:{
produto :: (Num a) => a -> a -> a
```

```
produto x y = x * y
:}
```

Exercício 17

```
eq1grau (número, número) >> número
eq2grau (a, b) << -b / a
```

```
dec eq1grau : num # num -> num;
--- eq1grau (a, b) <= (0 - b) / a;
```

```
:{
eq1grau :: Double -> Double -> Double
eq1grau a b = -b / a
:}
```

Exercício 18

```
área_ret (número, número) >> número
área_ret (lado1, lado2) << lado1 * lado2
```

```
dec area_ret : num # num -> num;
--- area_ret (lado1, lado2) <= lado1 * lado2;
```

```
:{
area_ret :: Double -> Double -> Double
area_ret lado1 lado2 = lado1 * lado2
:}
```

Capítulo 3

Exercício 1

```
min (número, número) >> lógico
min (x, y) << se (x < y) então x senão y
```

```
dec min : num # num -> num;
--- min (x, y) <= if x < y then x else y;
```

```
:{
min :: (Ord a, Floating a) => a -> a -> a
min x y = if x < y then x else y
:}
```

Exercício 2

```
min3 (número, número, número) >> número
min3 (x, y, z) << min (x, min (y, z))
```

```
dec min3 : num # num # num -> num;
--- min3 (x, y, z) <= min (x, min (y, z));
```

```
:{
min3 :: (Ord a, Floating a) => a -> a -> a -> a
min3 x y z = min x (min y z)
:}
```

Exercício 3

```
somat (número) >> número
somat (0) << 0
somat (1) << 1
somat (n) << n + somat (n - 1)
```

```
dec somat : num -> num;
--- somat 0 <= 0;
--- somat 1 <= 1;
--- somat n <= n + somat (n - 1);
```

```
:{
somat :: (Eq a, Floating a) => a -> a
somat 0 = 0
somat 1 = 1
somat n = n + somat (n - 1)
:}
```

Exercício 4

```
fat (número) >> número
fat (0) << 1
fat (1) << 1
fat (n) << n * fat (n - 1)
```

```
dec fat : num -> num;
--- fat 0 <= 1;
--- fat 1 <= 1;
--- fat n <= n * fat (n - 1);
```

```
:{
fat :: (Eq a, Integral a) => a -> a
fat 0 = 1
fat 1 = 1
fat n = n * fat (n - 1)
:}
```

Exercício 5

```
somat2 (número) >> número
somat2 (n) << se (n = 0) então 0 senão se (n = 1) então 1 senão n + somat2 (n - 1)
```

```
dec somat2 : num -> num;
--- somat2 n <= if n = 0 then 0 else if n = 1 then 1 else n + somat2 (n - 1);
```

```
:{
somat2 :: (Eq a, Floating a) => a -> a
```

```
somat2 n = if n == 0 then 0 else if n == 1 then 1 else n + somat2 (n - 1)
:}
```

Exercício 6

```
fat2 (número) >> número
fat2 (n) << se (n = 0) então 1 senão se (n = 1) então 1 senão n * fat2 (n - 1)
```

```
dec fat2 : num -> num;
--- fat2 n <= if n = 0 then 1 else if n = 1 then 1 else n * fat2 (n - 1);
```

```
:{
fat2 :: (Eq a, Integral a) => a -> a
fat2 n = if n == 0 then 1 else if n == 1 then 1 else n * fat2 (n - 1)
:}
```

Exercício 7

```
fatduplo (número) >> número
fatduplo (0) << 1
fatduplo (1) << 1
fatduplo (n) << n * fatduplo (n - 2)
```

```
dec fatduplo : num -> num;
--- fatduplo 0 <= 1;
--- fatduplo 1 <= 1;
--- fatduplo n <= n * fatduplo (n - 2);
```

```
:{
fatduplo :: (Eq a, Integral a) => a -> a
fatduplo 0 = 1
fatduplo 1 = 1
fatduplo n = n * fatduplo (n - 2)
:}
```

Exercício 8

```
fattriplo (número) >> número
fattriplo (0) << 1
fattriplo (1) << 1
fattriplo (n) << n * fattriplo (n - 3)
```

```
dec fattriplo : num -> num;
--- fattriplo 0 <= 1;
--- fattriplo 1 <= 1;
--- fattriplo n <= n * fattriplo (n - 3);
```

```
:{
fattriplo :: (Eq a, Integral a) => a -> a
fattriplo 0 = 1
fattriplo 1 = 1
fattriplo n = n * fattriplo (n - 3)
:}
```

Exercício 9

```
somat3base (número, número) >> número
somat3base (0, parcial) << parcial
somat3base (n, parcial) << somat3base (n - 1, n + parcial)

somat3 (número) >> número
somat3 (n) << somat3base (n, 0)
```

```
dec somat3base : num # num -> num;
--- somat3base (0, parcial) <= parcial;
--- somat3base (n, parcial) <= somat3base (n - 1, n + parcial);

dec somat3 : num -> num;
--- somat3 n <= somat3base (n, 0);
```

```
:{
somat3base :: (Eq a, Num a) => a -> a -> a
somat3base 0 parcial = parcial
somat3base n parcial = somat3base (n - 1) (n + parcial)

somat3 :: (Eq a, Num a) => a -> a
somat3 n = somat3base n 0
:}
```

Exercício 10

```
fat3base (número, número) >> número
fat3base (0, parcial) << parcial
fat3base (n, parcial) << fat3base (n - 1, n * parcial)

fat3 (número) >> número
fat3 (n) << fat3base (n, 1)
```

```
dec fat3base : num # num -> num;
--- fat3base (0, parcial) <= parcial;
--- fat3base (n, parcial) <= fat3base (n - 1, n * parcial);

dec fat3 : num -> num;
--- fat3 n <= fat3base (n, 1);
```

```
:{
fat3base :: (Eq a, Integral a) => a -> a -> a
fat3base 0 parcial = parcial
fat3base n parcial = fat3base (n - 1) (n * parcial)

fat3 :: (Eq a, Integral a) => a -> a
fat3 n = fat3base n 1
:}
```

Exercício 11

```
intervp (número, número) >> número
```

```
intervp (m, n) >> se (m > n) então 1 senão m * intervp (m + 1, n)
```

```
dec intervp : num # num -> num;
--- intervp (m, n) <= if m > n then 1 else m * intervp (m + 1, n);
```

```
:{
intervp :: (Eq a, Integral a) => a -> a -> a
intervp m n = if m > n then 1 else m * intervp (m + 1) n
:}
```

Exercício 12

```
intervs (número, número) >> número
intervs (m, n) >> se (m > n) então 0 senão m + intervs (m + 1, n)
```

```
dec intervs : num # num -> num;
--- intervs (m, n) <= if m > n then 0 else m + intervs (m + 1, n);
```

```
:{
intervs :: (Eq a, Integral a) => a -> a -> a
intervs m n = if m > n then 0 else m + intervs (m + 1) n
:}
```

Exercício 13

```
mult (número, número) >> número
mult (x, y) >> se (y = 0) então 0 senão x + mult (x, y - 1)
```

```
dec mult : num # num -> num;
--- mult (x, y) <= if y = 0 then 0 else x + mult (x, y - 1);
```

```
:{
mult :: (Eq a, Num a) => a -> a -> a
mult x y = if y == 0 then 0 else x + mult x (y - 1)
:}
```

Exercício 14

```
potência_de_2 (número) >> número
potência_de_2 (1) >> 2
potência_de_2 (n) >> 2 * potência_de_2  (n - 1)
```

```
dec potencia_de_2 : num -> num;
--- potencia_de_2 1 <= 2;
--- potencia_de_2 n <= 2 * potencia_de_2 (n - 1);
```

```
:{
potencia_de_2 :: (Eq a, Num a) => a -> a
potencia_de_2 1 = 2
potencia_de_2 n = 2 * potencia_de_2 (n - 1)
:}
```

Exercício 15

```
hanói (número) >> número
hanói (0) >> 0
hanói (1) >> 1
hanói (n) >> 2 * hanói (n - 1) + 1
```

```
dec hanoi : num -> num;
--- hanoi 0 <= 0;
--- hanoi 1 <= 1;
--- hanoi n <= 2 * hanoi (n - 1) + 1;
```

```
:{
hanoi :: (Eq a, Integral a) => a -> a
hanoi 0 = 0
hanoi 1 = 1
hanoi n = 2 * hanoi (n - 1) + 1
:}
```

Exercício 16

```
série (número) >> número
série (0) >> 0
série (1) >> 3
série (n) >> 3 * série (n - 1) - 2
```

```
dec serie : num -> num;
--- serie 0 <= 0;
--- serie 1 <= 3;
--- serie n <= 3 * serie (n - 1) - 2;
```

```
:{
serie :: (Eq a, Integral a) => a -> a
serie 0 = 0
serie 1 = 3
serie n = 3 * serie (n - 1) - 2
:}
```

Exercício 17

```
negativo (número) >> número
negativo (n) << se (n < 0) então n senão 0 - n
```

```
dec negativo : num -> num;
--- negativo n <= if n < 0 then n else 0 - n;
```

```
:{
negativo :: (Ord a, Num a) => a -> a
negativo n = if n < 0 then n else 0 - n
:}
```

Exercício 18

```
coprimo (número, número) >> número
coprimo (x, y) << mdc (x, y) = 1
```

```
dec coprimo : num # num -> truval;
--- coprimo (x, y) <= mdc (x, y) = 1;
```

```
:{
coprimo :: (Integral a) => a -> a -> Bool
coprimo x y = mdc x y == 1
:}
```

Exercício 19

```
coprimo (número, número) >> número
coprimo (x, y) << mdc (x, y) = 1
```

```
dec mmc : num # num -> num;
--- mmc (x, y) <= x * y div (mdc (x, y));
```

```
:{
mmc :: Int -> Int -> Int
mmc x y = div (x * y) (mdc x y)
:}
```

Exercício 20

```
sinal (número, número) >> número
sinal (x, y) << se (x < y) então -1 então if x > y então 1 então 0
```

```
dec sinal : num # num -> num;
--- sinal (x, y) <= if x < y then -1 else if x > y then 1 else 0;
```

```
:{
sinal :: (Ord a, Num a) => a -> a -> a
sinal x y = if x < y then -1 else if x > y then 1 else 0
:}
```

Exercício 21

```
hms_tempo (número, número, número) >> número;
hms_tempo (h, m, s) <<
  se (h < 0) .ou. (h > 23) .ou. (m < 0) .ou. (m > 59) .ou. (s < 0) .ou. (s > 59)
  então escreva "algum dado fornecido está incorreto"
  senão h * 3600 + m * 60 + s
```

```
dec hms_tempo : num # num # num -> num;
--- hms_tempo (h, m, s) <=
      if h < 0 or h > 23 or m < 0 or m > 59 or s < 0 or s > 59
      then error "algum dado fornecido esta incorreto"
      else h * 3600 + m * 60 + s;
```

```
:{
hms_tempo :: (Int, Int, Int) -> Int
hms_tempo (h, m, s) =
  if h < 0 || h > 23 || m < 0 || m > 59 || s < 0 || s > 59
```

```
  then error "algum dado fornecido esta incorreto"
  else h * 3600 + m * 60 + s
:}
```

Exercício 22

```
tempo_hms (número) >> (número, número, número)
tempo_hms tmp <<
  se (tmp < 0) .ou. (tmp > 86399)
  então escreva "valor serial fornecido está incorreto"
  senão (tmp div 3600, tmp mod 3600 div 60, tmp mod 3600 mod 60)
```

```
dec tempo_hms : num -> num # num # num;
--- tempo_hms tmp <=
      if tmp < 0 or tmp > 86399
      then error "valor serial fornecido esta incorreto"
      else (tmp div 3600, tmp mod 3600 div 60, tmp mod 3600 mod 60);
```

```
:{
tempo_hms :: Int -> (Int, Int, Int)
tempo_hms tmp =
  if tmp < 0 || tmp > 86399
  then error "valor serial fornecido esta incorreto"
  else ((div tmp 3600), (div (mod tmp 3600) 60), (mod (mod tmp 3600) 60))
:}
```

Exercício 23

```
tempo_horário ((número, número, número), (número, número, número)) >> (número, número, número)
tempo_ horário ((h1, m1, s1), (h2, m2, s2)) <<
  se hms_tempo ((h1, m1, s1) > hms_tempo (h2, m2, s2))
  então error "início da contagem de tempo é maior que o término"
  senão tempo_hms (hms_tempo (h2, m2, s2) - hms_tempo (h1, m1, s1))
```

```
dec tempo_horario : (num # num # num) # (num # num # num) -> num # num # num;
--- tempo_horario ((h1, m1, s1), (h2, m2, s2)) <=
      if hms_tempo (h1, m1, s1) > hms_tempo (h2, m2, s2)
      then error "inicio da contagem de tempo e maior que o termino"
      else tempo_hms (hms_tempo (h2, m2, s2) - hms_tempo (h1, m1, s1));
```

```
:{
tempo_horario :: ((Int, Int, Int),  (Int, Int, Int)) -> (Int, Int, Int)
tempo_horario ((h1, m1, s1), (h2, m2, s2)) =
  if hms_tempo (h1, m1, s1) > hms_tempo (h2, m2, s2)
  then error "inicio da contagem de tempo e maior que o termino"
  else tempo_hms ((hms_tempo (h2, m2, s2)) - (hms_tempo (h1, m1, s1)))
:}
```

Exercício 24

```
binário (número) >> lista número
binário (0) << [0]
binário (1) << [1]
```

```
binário (n) << se (n mod 2 = 0)
               então binário (n div 2) # [0]
               senão binário (n div 2) # [1]
```

```
dec binario : num -> list num;
--- binario 0 <= [0];
--- binario 1 <= [1];
--- binario n <= if n mod 2 = 0
                 then binario (n div 2) <> [0]
                 else binario (n div 2) <> [1];
```

```
:{
binario :: Int -> [Int]
binario 0 = [0]
binario 1 = [1]
binario n = if mod n 2 == 0
            then binario (div n 2) ++ [0]
            else binario (div n 2) ++ [1]
:}
```

Exercício 25

```
div84 (número) >> lógico
div84 (n) << se (n mod 8 = 4) então .verdadeiro. senão .falso.
```

```
dec div84 : num -> truval;
--- div84 n <= if n mod 8 = 4 then true else false;
```

```
:{
div84 :: Int -> Bool
div84 n = if (mod n 8) == 4 then True else False
:}
```

Exercício 26

```
dec divx (número, número, número) >> lógico
--- divx (n, d, r) << se n mod d = r então .verdadeiro. senão .falso.
```

```
dec divx : num # num # num -> truval;
--- divx (n, d, r) <= if n mod d = r then true else false;
```

```
:{
divx :: Int -> Int -> Int -> Bool
divx n d r = if mod n d == r then True else False
:}
```

Capítulo 4

Exercício 1

```
simples (lista número) >> lógico
```

```
simples ([]) << .falso.
simples ([x]) << .verdadeiro.
simples (x1 :: x2 :: xs) << .falso.
```

```
dec simples : list num -> truval;
--- simples ([]) <= false;
--- simples ([x]) <= true;
--- simples (x1 :: x2 :: xs) <= false;
```

```
:{
simples :: (Num a) => [a] -> Bool
simples [] = False
simples [x] = True
simples (x1 : x2 : xs) = False
:}
```

Exercício 2

```
lista_min_max (lista número) >> lista número
lista_min_max [] << escreva "lista vazia"
lista_min_max (x :: xs) << [lista_min (x :: xs)] # [lista_max (x :: xs)]
```

```
dec lista_min_max : list num -> list num;
--- lista_min_max [] <= error "lista vazia";
--- lista_min_max (x :: xs) <= [lista_min (x :: xs)] <> [lista_max (x :: xs)];
```

```
:{
lista_min_max :: (Ord a, Num a) => [a] -> [a]
lista_min_max [] = error "lista vazia"
lista_min_max (x : xs) = [lista_min (x : xs)] ++ [lista_max (x : xs)]
:}
```

Exercício 3

```
vazia (lista número) >> lógico
vazia ([]) << .verdadeiro.
vazia (_) << .falso.
```

```
dec vazia : list num -> truval;
--- vazia [] <= true;
--- vazia _ <= false;
```

```
:{
vazia :: (Num a) => [a] -> Bool
vazia [] = True
vazia _ = False
:}
```

Exercício 4

```
intervalo (número, número) >> lista número
intervalo (m, n) << se (m > n) então [] senão m : intervalo (m + 1, n)
vazia (_) << .falso.
```

```
dec intervalo : num # num -> list num;
--- intervalo (m, n) <= if m > n then [] else m :: intervalo (m + 1, n);
```

```
:{
intervalo :: (Ord a, Num a) => a -> a -> [a]
intervalo m n = if m > n then [] else m : intervalo (m + 1) n
:}
```

Exercício 5

```
poe_ultimo (número, lista número) >> lista número
poe_ultimo (x, []) << [x]
poe_ultimo (x, y :: ys) << y :: poe_ultimo (x, ys)
```

```
dec poe_ultimo : num # list num -> list num;
--- poe_ultimo (x, []) <= [x];
--- poe_ultimo (x, y :: ys) <= y :: poe_ultimo (x, ys);
```

```
:{
poe_ultimo :: (Num a) => a -> [a] -> [a]
poe_ultimo x [] = [x]
poe_ultimo x (y : ys) = y : poe_ultimo x ys
:}
```

Exercício 6

```
soma_lista (lista número) >> número
soma_lista ([]) << 0
soma_lista (x :: xs) << x + soma_lista (xs)
```

```
dec soma_lista : list num -> num;
--- soma_lista ([]) <= 0;
--- soma_lista (x :: xs) <= x + soma_lista xs;
```

```
:{
soma_lista :: (Num a) => [a] -> a
soma_lista [] = 0
soma_lista (x : xs) = x + soma_lista xs
:}
```

Exercício 7

```
produto_lista (lista número) >> número
produto_lista ([]) << 1
produto_lista (x :: xs) << x * produto_lista (xs)
```

```
dec produto_lista : list num -> num;
--- produto_lista ([]) <= 1;
--- produto_lista (x :: xs) <= x * produto_lista xs;
```

```
:{
produto_lista :: (Num a) => [a] -> a
```

```
produto_lista [] = 1
produto_lista (x : xs) = x * produto_lista xs
:}
```

Exercício 8

```
pares (lista número) >> lista (número, número)
pares (xs) << compacta (xs, cauda (xs))
```

```
dec pares : list num -> list (num # num);
--- pares xs <= compacta (xs, cauda xs);
```

```
:{
pares :: (Num a) => [a] -> [(a, a)]
pares xs = compacta xs (cauda xs)
:}
```

Exercício 9

```
separar_em (número, lista número) >> (lista número, lista número)
separar_em (n, []) << ([], [])
separar_em (n, [x]) << ([x], [])
separar_em (n, xs) << (comeco (n, xs), final (n, xs))
```

```
dec separar_em : num # list num -> list num # list num;
--- separar_em (n, []) <= ([], []);
--- separar_em (n, [x]) <= ([x], []);
--- separar_em (n, xs) <= (comeco (n, xs), final (n, xs));
```

```
:{
separar_em :: (Ord a, Num a) => Int -> [a] -> ([a], [a])
separar_em n [] = ([], [])
separar_em n [a] = ([a], [])
separar_em n xs = (comeco n xs, final n xs)
:}
```

Exercício 10

```
troca (número, número) >> (número, número)
troca (x, y) << (y, x)
```

```
dec troca : (num # num) -> (num # num);
--- troca (x, y) <= (y, x);
```

```
:{
troca :: (Num a) => (a, a) -> (a, a)
troca (x, y) = (y, x)
:}
```

Exercício 11

```
eq2grau (número, número, número) >> lista número
eq2grau (a, b, c) <<
  se (pow (b, 2) - 4 * a * c >= 0)
```

```
  então [(-b + (b ^ 2 - 4 * a * c) ^ (1 / 2)) / (2 * a),
         (-b - (b ^ 2 - 4 * a * c) ^ (1 / 2)) / (2 * a)]
  senão error [];
```

```
dec eq2grau : num # num # num -> list num;
--- eq2grau (a, b, c) <=
      if pow (b, 2) - 4 * a * c >= 0
      then [(0 - b + pow (pow (b, 2) - 4 * a * c, (1 / 2))) / (2 * a),
            (0 - b - pow (pow (b, 2) - 4 * a * c, (1 / 2))) / (2 * a)]
      else error [];
```

```
:{
eq2grau :: Double -> Double -> Double -> [Double]
eq2grau a b c =
  if b ** 2 - 4 * a * c >= 0
  then [(-b + (b ** 2 - 4 * a * c) ** (1 / 2)) / (2 * a),
        (-b - (b ** 2 - 4 * a * c) ** (1 / 2)) / (2 * a)]
  else error []
:}
```

Exercício 12

```
pri (número, número) >> número
pri (x, _) << x
seg (número, número) >> número
seg (_, y) << y
```

```
dec pri : (num # num) -> num;
--- pri (x, _) <= x;
dec seg : (num # num) -> num;
--- seg (_, y) <= y;
```

```
:{
pri :: (Num a) => (a, a) -> a
pri (x, _) = x
seg :: (Num a) => (a, a) -> a
seg (_, y) = y
:}
```

Exercício 13

```
rotac_e (lista número) >> lista número
rotac_e [] << []
rotac_e (x :: xs) << xs # [x]
```

```
dec rotac_e : list num -> list num;
--- rotac_e [] <= [];
--- rotac_e (x :: xs) <= xs <> [x];
```

```
:{
rotac_e :: (Num a) => [a] -> [a]
rotac_e [] = []
```

```
rotac_e (x : xs) = xs ++ [x]
:}
```

Exercício 14

```
rotac_d (lista número) >> lista número
rotac_d [] << []
rotac_d (xs) << ultimo (xs) :: arranjo (xs)
```

```
dec rotac_d : list num -> list num;
--- rotac_d [] <= [];
--- rotac_d xs <= ultimo xs :: arranjo xs;
```

```
:{
rotac_d :: (Num a) => [a] -> [a]
rotac_d [] = []
rotac_d xs = ultimo xs : arranjo xs
:}
```

Exercício 15

```
troca_adj (lista número) >> lista número;
troca_adj ([]) << []
troca_adj (x1 :: x2 :: xs) << [x2, x1] # troca_adj (xs)
troca_adj (x1 :: x2) <= se (x2 = []) então [x1] senão [cabeça (x2), x1]
```

```
dec troca_adj : list num -> list num;
--- troca_adj [] <= [];
--- troca_adj (x1 :: x2 :: xs) <= [x2, x1] <> troca_adj xs;
--- troca_adj (x1 :: x2) <= if x2 = [] then [x1] else [cabeca (x2), x1];
```

```
:{
troca_adj :: (Eq a, Num a) => [a] -> [a]
troca_adj [] = []
troca_adj (x1 : x2 : xs) = [x2, x1] ++ troca_adj xs
troca_adj (x1 : x2) = if x2 == [] then [x1] else [cabeca x2, x1]
:}
```

Exercício 16

```
perfeito (número) >> lógico
perfeito (0) << .falso.
perfeito (n) << n = redução (complista (faixa (1, n - 1, 1), | x >>> múltiplo(n, x)), soma, 0)
```

```
dec perfeito : num -> truval;
--- perfeito 0 <= false;
--- perfeito n <= n = reducao (complista (faixa (1, n - 1, 1), \ x => multiplo(n, x)), soma, 0);
```

```
:{
perfeito :: Int -> Bool
perfeito 0 = False
perfeito n = n == reducao (complista (faixa 1 (n - 1) 1) (multiplo n)) soma 0
:}
```

Exercício 17

```
lista_perfeito (número) >> lista número
lista_perfeito n << se (perfeito (n))
                    então complista (faixa (1, n - 1, 1), | x >>> múltiplo(n, x))
                    senão []
```

```
dec lista_perfeito : num -> list num;
--- lista_perfeito n <= if perfeito n
                        then complista (faixa (1, n - 1, 1), \ x => multiplo(n, x))
                        else [];
```

```
:{
lista_perfeito :: Int -> [Int]
lista_perfeito n = if perfeito n
                   then complista (faixa 1 (n - 1) 1) (multiplo n)
                   else []
:}
```

Exercício 18

```
soma_impares (número) >> número
soma_impares (n) << redução (filtro ( | n >> impar (n), faixa (1, n, 1)), soma, 0)
```

```
dec soma_impares : num -> num;
--- soma_impares n <= reducao (filtro (\ n => impar n, faixa (1, n, 1)), soma, 0);
```

```
:{
soma_impares :: (Integral a) => a -> a
soma_impares n = reducao (filtro (impar) (faixa 1 n 1)) soma 0
:}
```

Exercício 19

```
duplicar (lista número) >> lista número
duplicar (x :: xs) << x :: x :: duplicar xs
```

```
dec duplicar : list num -> list num;
--- duplicar [] <= [];
--- duplicar (x :: xs) <= x :: x :: duplicar xs;
```

```
:{
duplicar :: (Num a) => [a] -> [a]
duplicar [] = []
duplicar (x : xs) = x : x : duplicar xs
:}
```

Exercício 20

```
divisores (número) >> lista número
divisores n << oposto (complista (faixa (1, n + 1, 1), | x >>> múltiplo(n, x)))
```

```
dec divisores : num -> list num;
```

```
--- divisores n <= oposto (complista (faixa (1, n + 1, 1), \ x => multiplo(n, x)));
```

```
:{
divisores :: Int -> [Int]
divisores n = oposto (complista (faixa 1 (n + 1) 1) (multiplo n))
:}
```

Exercício 21

```
amigos (número, número) >> lógico
amigos (x, y) << soma_lista (divisores (x)) - x = y .e. soma_lista (divisores (y)) - y = x
```

```
dec amigos : num # num -> truval;
--- amigos (x, y) <= soma_lista (divisores x) - x = y and soma_lista (divisores y) - y = x;
```

```
:{
amigos :: Int -> Int -> Bool
amigos x y = (soma_lista (divisores x) - x) == y && (soma_lista (divisores y) - y) == x
:}
```

Exercício 22

```
penúltimo (lista número) >> número
penúltimo (xs) << último (arranjo (xs))
```

```
dec penultimo : list num -> num;
--- penultimo xs <= ultimo (arranjo xs);
```

```
:{
penultimo :: (Num a) => [a] -> a
penultimo xs = ultimo (arranjo xs)
:}
```

Exercício 23

```
busca_ord (número, lista número) >> número
busca_ord (1, x :: xs) << x
busca_ord (i, x :: xs) << busca_ord (i - 1, xs)
busca_ord (_, _) << error "indice fora da faixa"
```

```
dec busca_ord : num # list num -> num;
--- busca_ord (1, x :: xs) <= x;
--- busca_ord (i, x :: xs) <= busca_ord (i - 1, xs);
--- busca_ord (_, _) <= error "indice fora da faixa";
```

```
:{
busca_ord :: (Num a) => Int -> [a] -> a
busca_ord 1 (x : xs) = x
busca_ord n (x : xs) = busca_ord (n - 1) xs
busca_ord _ _ = error "indice fora da faixa"
:}
```

Exercício 24

```
palindromo (lista número) >> lógico
palindromo (xs) << xs = oposto (xs)
```

```
dec palindromo : list num -> truval;
--- palindromo xs <= xs = oposto xs;
```

```
:{
palindromo :: (Eq a, Num a) => [a] -> Bool
palindromo xs = xs == (oposto xs)
:}
```

Exercício 25

```
segunda_pos (lista número) >> número
segunda_pos (xs) << cabeca (cauda (xs))
```

```
dec segunda_pos : list num -> num;
--- segunda_pos xs <= cabeca (cauda xs);
```

```
:{
segunda_pos :: (Num a) => [a] -> a
segunda_pos xs = cabeca (cauda xs)
:}
```

Exercício 26

```
fatores_primos (número) >> lista número
fatores_primos n << filtro(| n => checa_primo (n), complista (faixa (1, n, 1), | x >>> múltiplo(n, x)))
```

```
dec fatores_primos : num -> list num;
--- fatores_primos n <= filtro(\ n => checa_primo n, complista (faixa (1, n, 1), \ x => multiplo(n, x)));
```

```
:{
fatores_primos :: Int -> [Int]
fatores_primos n = filtro (checa_primo) (complista (faixa 1 n 1) (multiplo n))
:}
```

Exercício 27

```
ligacao (lista alfa, |alfa >> lista beta| >> lista beta
ligacao ([], b) << []
ligacao (x :: xs, funcao) << funcao x # ligacao (xs, funcao)

prod_cartes (list alfa, lista beta) >> lista (alfa, beta)
prod_cartes (a, b) << ligacao (a, | x >> ligacao (b, | y >> [(x, y)]))
```

```
dec ligacao : list alpha # (alpha -> list beta) -> list beta;
--- ligacao ([], b) <= [];
--- ligacao (x :: xs, funcao) <= funcao x <> ligacao (xs, funcao);

dec prod_cartes : list alpha # list beta -> list (alpha # beta);
--- prod_cartes (a, b) <= ligacao (a, \x => ligacao (b, \y => [(x, y)]));
```

```
:{
ligacao :: [a] -> (a -> [b]) -> [b]
ligacao [] b = []
ligacao (x : xs) funcao = funcao x ++ (ligacao xs funcao)

prod_cartes :: (Num a, Num b) => [a] -> [b] -> [(a, b)]
prod_cartes a b = ligacao a (\x -> ligacao b (\y -> [(x, y)]))
:}
```

Exercício 28

```
abundante (número) >> lógico
abundante n << soma_lista (complista (faixa (1, n + 1, 1), | x >>> multiplo(n, x))) - n > n
```

```
dec abundante : num -> truval;
--- abundante n <= soma_lista (complista (faixa (1, n + 1, 1), \x => multiplo(n, x))) - n > n;
```

```
:{
abundante :: Int -> Bool
abundante n = soma_lista (complista (faixa 1 (n + 1) 1) (multiplo n)) - n > n
:}
```

Exercício 29

```
insira_em (número, número, lista número) >> lista número
insira_em (x, 1, xs) << (x :: xs)
insira_em (x, n, xn :: xs) << xn :: insira_em (x, n - 1, xs)
```

```
dec insira_em : num # num # list num -> list num;
--- insira_em (x, 1, xs) <= x :: xs;
--- insira_em (x, n, xn :: xs) <= xn :: insira_em (x, n - 1, xs);
```

```
:{
insira_em :: (Num a) => a -> Int -> [a] -> [a]
insira_em x 1 xs = x : xs
insira_em x n (xn : xs) = xn : insira_em x (n - 1) xs
:}
```

Exercício 30

```
labundante (número) >> lista número
labundante (n) << filtro ( | x >> abundante (x), faixa (1, n, 1))
```

```
dec labundante : num -> list num;
--- labundante n <= filtro (\x => abundante x, faixa (1, n, 1));
```

```
:{
labundante ::  Int -> [Int]
labundante n = filtro (abundante) (faixa 1 n 1)
:}
```

Exercício 31

```
lista_n_1 (número) >> lista número
```

```
lista_n_1 n << if (n = 0) then [] else n :: lista_n_1 (n - 1)
```

```
dec lista_n_1 : num -> list num;
--- lista_n_1 n <= if n = 0 then [] else n :: lista_n_1 (n - 1);
```

```
:{
lista_n_1 :: Int -> [Int]
lista_n_1 n = if n == 0 then [] else n : lista_n_1 (n - 1)
:}
```

Exercício 32

```
lista_1_n (número) >> lista número
lista_1_n (n) << se (n = 0) então [] senão lista_1_n (n - 1) # [n]
```

```
dec lista_1_n : num -> list num;
--- lista_1_n n <= if n = 0 then [] else lista_1_n (n - 1) <> [n];
```

```
:{
lista_1_n :: Int -> [Int]
lista_1_n n = if n == 0 then [] else lista_1_n (n - 1) ++ [n]
:}
```

Exercício 33

```
calculadora (||número, número >> número||, lista número, lista número) >> lista número
calculadora (f, [], _) << []
calculadora (f, _, []) << []
calculadora (f, x :: xs, y :: ys) << f (x, y) :: calculadora (f, xs, ys)
```

```
dec calculadora : (num # num -> num) # list num # list num -> list num;
! exponenciação use: (pow)
--- calculadora (f, [], _) <= [];
--- calculadora (f, _, []) <= [];
--- calculadora (f, x :: xs, y :: ys) <= f (x, y) :: calculadora (f, xs, ys);
```

```
:{
-- exponenciacao use: (**)
calculadora :: (Num a) => (a -> a -> a) -> [a] -> [a] -> [a]
calculadora f [] _ = []
calculadora f _ [] = []
calculadora f (x : xs) (y : ys) = (f x y) : (calculadora f xs ys)
:}
```

Exercício 34

```
faixa_primo (número, número) >> lista número
faixa_primo (comeco, final) << complista (faixa (comeco, final, 1), | x >>> checa_primo (x))
```

```
dec faixa_primo : num # num -> list num;
--- faixa_primo (comeco, final) <= complista (faixa (comeco, final, 1), \ x => checa_primo (x));
```

```
:{
```

```
faixa_primo :: Int -> Int -> [Int]
faixa_primo comeco final = complista (faixa comeco final 1) (checa_primo)
:}
```

Exercício 35

```
remover (número, lista número) >> lista número
remover (n, []) << [];
remover (n, x :: xs) << se (n = x) então xs senão x :: remover (n, xs)
```

```
dec remover : num # list num -> list num;
--- remover (n, []) <= [];
--- remover (n, x :: xs) <= if n = x then xs else x :: remover (n, xs);
```

```
:{
remover :: (Eq a, Num a) => a -> [a] -> [a]
remover n [] = []
remover n (x : xs) = if x == n then xs else x : remover n xs
:}
```

Exercício 36

```
rmv_priult (lista número) >> lista número
rmv_priult ([]) << []
rmv_priult ([x]) << []
rmv_priult xs << cauda (arranjo (xs))
```

```
dec rmv_priult : list num -> list num;
--- rmv_priult [] <= [];
--- rmv_priult ([x]) <= [];
--- rmv_priult xs <= cauda (arranjo xs);
```

```
:{
rmv_priult :: (Num a) => [a] -> [a]
rmv_priult [] = []
rmv_priult [x] = []
rmv_priult xs = cauda (arranjo xs)
:}
```

Exercício 37

```
rmv_pris (número, lista número) >> lista número
rmv_pris (_, []) << []
rmv_pris (0, x) << x
rmv_pris (n, x :: xs) << rmv_pris (n - 1, xs)
```

```
dec rmv_pris : num # list num -> list num;
--- rmv_pris (_, []) <= [];
--- rmv_pris (0, x) <= x;
--- rmv_pris (n, x :: xs) <= rmv_pris (n - 1, xs);
```

```
:{
rmv_pris :: (Num a) => Int -> [a] -> [a]
```

```
rmv_pris _ [] = []
rmv_pris 0 x = x
rmv_pris n (x : xs) = rmv_pris (n - 1) xs
:}
```

Exercício 38

```
rmv_parim (lista número, número) >> lista número
rmv_parim ([], _) << []
rmv_parim (xs, 0) << filtro( | x => par(x), xs)
rmv_parim (xs, 1) << filtro( | x => impar(x), xs)
rmv_parim (xs, _) << []
```

```
dec rmv_parim : list num # num -> list num;
--- rmv_parim ([], _) <= [];
--- rmv_parim (xs, 0) <= filtro(\x => par(x), xs);
--- rmv_parim (xs, 1) <= filtro(\x => impar(x), xs);
--- rmv_parim (xs, _) <= [];
```

```
:{
rmv_parim :: (Integral a) => [a] -> Int -> [a]
rmv_parim [] _ = []
rmv_parim (xs) 0 = filtro (par) xs
rmv_parim (xs) 1 = filtro (impar) xs
rmv_parim (xs) _ = []
:}
```

Exercício 39

```
duplic_n_em_lista (número) >> lista número
duplic_n_em_lista (n) << [n, n]
```

```
dec duplic_n_em_lista : num -> list num;
--- duplic_n_em_lista n <= [n, n];
```

```
:{
duplic_n_em_lista :: (Num a) => a -> [a]
duplic_n_em_lista n = [n, n]
:}
```

Exercício 40

```
triplic_n_em_tupla (número) >> (número, número, número)
triplic_n_em_tupla (n) << (n, n, n)
```

```
dec triplic_n_em_tupla : num -> (num # num # num);
--- triplic_n_em_tupla n <= (n, n, n);
```

```
:{
triplic_n_em_tupla :: (Num a) => a -> (a, a, a)
triplic_n_em_tupla n = (n, n, n)
:}
```

Exercício 41

```
distrib_n (número, lista número) >> lista (número, número)
distrib_n (n, []) << []
distrib_n (n, x :: xs) << (n, x) :: distrib_n (n, xs)
```

```
dec distrib_n : num # list num -> list (num # num);
--- distrib_n (n, []) <= [];
--- distrib_n (n, x :: xs) <= (n, x) :: distrib_n (n, xs);
```

```
:{
distrib_n :: (Num a) => a -> [a] -> [(a, a)]
distrib_n n [] = []
distrib_n n (x : xs) = (n, x) : distrib_n n xs
:}
```

Exercício 42

```
poe_em_pos (número, número, lista número) >> lista número
poe_em_pos (n, 0, xs) << n :: xs
poe_em_pos (n, posicao, x :: xs) << x :: poe_em_pos (n, posicao - 1, xs)
```

```
dec poe_em_pos : num # num # list num -> list num;
--- poe_em_pos (n, 0, xs) <= n :: xs;
--- poe_em_pos (n, posicao, x :: xs) <= x :: poe_em_pos (n, posicao - 1, xs);
```

```
:{
poe_em_pos :: a -> Int -> [a] -> [a]
poe_em_pos n 0 xs = n : xs
poe_em_pos n posicao (x : xs) = x : poe_em_pos n (posicao - 1) xs
:}
```

Exercício 43

```
complista(faixa(1,60,1), | x >>> div84 x)
```

```
complista(faixa(1,60,1), \x => div84 x);
```

```
complista (faixa 1 60 1) (div84)
```

Exercício 44

```
mostra_priult : (lista número) >> lista número
mostra_priult (x) << [cabeca (x), último (x)]
```

```
dec mostra_priult : list num -> list num;
--- mostra_priult x <= [cabeca x, ultimo x];
```

```
:{
mostra_priult :: (Num a) => [a] -> [a]
mostra_priult x = [cabeca x, ultimo x]
:}
```

Exercício 45

```
soma_ac (lista número) >> lista número
```

```
soma_ac [] << []
soma_ac (x :: []) << x :: []
soma_ac (x1 :: x2 :: xs) << x1 :: soma_ac ((x1 + x2) :: xs)
```

```
dec soma_ac : list num -> list num;
--- soma_ac [] <= [];
--- soma_ac (x :: []) <= x :: [];
--- soma_ac (x1 :: x2 :: xs) <= x1 :: soma_ac ((x1 + x2) :: xs);
```

```
:{
soma_ac :: (Num a) => [a] -> [a]
soma_ac [] = []
soma_ac (x : []) = x : []
soma_ac (x1 : x2 : xs) = x1 : soma_ac ((x1 + x2) : xs)
:}
```

Exercício 46

```
mult (número, número) >> número;
mult (n, 0) << 0
mult (0, m) << 0
mult (n, m) << m + m * (n - 1)
```

```
dec mult : num # num -> num;
--- mult (n, 0) <= 0;
--- mult (0, m) <= 0;
--- mult (n, m) <= m + m * (n - 1);
```

```
:{
mult n 0 = 0
mult 0 m = 0
mult n m = m + m * (n - 1)
:}
```

Exercício 47

```
mapa (complista (faixa (10, 30, 1), | x >>> ímpar(x)), | x >>> potência (x, 2))
```

```
mapa (complista (faixa (10, 30, 1), \x => impar(x)), \x => pow (x, 2));
```

```
mapa (complista (faixa 10 30 1) (impar)) (^2)
```

Exercício 48

```
complista (soma_ac (faixa (1,5,1)), | x => par(x))
```

```
complista (soma_ac (faixa (1,5,1)), \x => par(x));
```

```
complista (soma_ac (faixa 1 5 1)) (par)
```

Exercício 49

```
multip_faixa (número, número) >> número
multip_faixa (x, y) << se x > y
```

```
                        então 0
                        senão se x = y
                              então y
                              senão x * multip_faixa (x + 1, y)
```

```
dec multip_faixa : num # num -> num;
--- multip_faixa (x, y) <= if x > y
                           then 0
                           else if x = y
                                then y
                                else x * multip_faixa (x + 1, y);
```

```
:{
multip_faixa :: (Ord a, Eq a, Num a) => a -> a -> a
multip_faixa x y = if x > y
                   then 0
                   else if x == y
                        then y
                        else x * multip_faixa (x + 1) y
:}
```

Exercício 50

```
fat4 (número) >> número
fat4 (0) << 1
fat4 (n) << multip_faixa (1, n)
```

```
dec fat4 : num -> num;
--- fat4 0 <= 1;
--- fat4 n <= multip_faixa (1, n);
```

```
:{
fat4 :: (Ord a, Num a) => a -> a
fat4 0 = 1
fat4 n = multip_faixa 1 n
:}
```

Exercício 51

```
somat_faixa (número, número) >> número
somat_faixa (x, y) << se x > y
                        então 0
                        senão se x = y
                              então y
                              senão x + somat_faixa (x + 1, y)
```

```
dec somat_faixa : num # num -> num;
--- somat_faixa (x, y) <= if x > y
                           then 0
                           else if x = y
                                then y
                                else x + somat_faixa (x + 1, y);
```

```
:{
somat_faixa :: (Ord a, Eq a, Num a) => a -> a -> a
somat_faixa x y = if x > y
                  then 0
                  else if x == y
                       then y
                       else x + somat_faixa (x + 1) y
:}
```

Exercício 52

```
somat4 (número) >> número
somat4 (0) << 1
somat4 (n) << somat_faixa (1, n)
```

```
dec somat4 : num -> num;
--- somat4 0 <= 1;
--- somat4 n <= somat_faixa (1, n);
```

```
:{
somat4 :: (Ord a, Num a) => a -> a
somat4 0 = 1
somat4 n = somat_faixa 1 n
:}
```

Exercício 53

```
média_arit (lista número) >> número
média_arit (x) << soma_lista x / tamanho x
```

```
dec media_arit : list num -> num;
--- media_arit x <= soma_lista x / tamanho x;
```

```
:{
media_arit :: (Fractional a) => [a] -> a
media_arit x = soma_lista x / (fromIntegral (tamanho x))
:}
```

Exercício 54

```
desvio_padrão (lista número) >> número
desvio_padrão (x) <<
  media_arit (calculadora ((^), calculadora ((-), x, replicar
  (tamanho (x), media_arit (x))), replicar (tamanho (x), 2))) ^ (1/2)
```

```
dec desvio_padrao : list num -> num;
--- desvio_padrao x <=
      pow (media_arit (calculadora ((pow), calculadora ((-), x, replicar
      (tamanho (x), media_arit (x))), replicar (tamanho (x), 2))), (1/2));
```

```
:{
desvio_padrao :: (Eq a, Floating a) => [a] -> a
```

```
desvio_padrao x =
  (media_arit (calculadora (**) (calculadora (-) (x) (replicar (fromIntegral (tamanho x))
  (media_arit x))) (replicar (fromIntegral (tamanho x)) 2))) ** (1/2)
:}
```

Exercício 55

```
var_populac (lista número) >> número
var_populac (x) <<
  media_arit (calculadora ((^), calculadora ((-), x, replicar
  (tamanho (x), media_arit (x))), replicar (tamanho (x), 2)))
```

```
dec var_populac : list num -> num;
--- var_populac x <=
      media_arit (calculadora ((pow), calculadora ((-), x, replicar
      (tamanho (x), media_arit (x))), replicar (tamanho (x), 2)));
```

```
:{
var_populac :: (Eq a, Floating a) => [a] -> a
var_populac x =
  media_arit (calculadora (**) (calculadora (-) (x) (replicar (fromIntegral (tamanho x))
  (media_arit x))) (replicar (fromIntegral (tamanho x)) 2))
:}
```

Exercício 56

```
média_geo (lista número) >> número
média_geo (x) << produto_lista (x) ^ 1 / tamanho (x)
```

```
dec media_geo : list num -> num;
--- media_geo x <= pow (produto_lista x, 1 / tamanho x);
```

```
:{
media_geo :: (Floating a) => [a] -> a
media_geo x = (produto_lista x) **  (1 /  (fromIntegral (tamanho x)))
:}
```