Functional programming
Lecture 02 – Functions 101

(version: 2026-02-26–11:53:37)

Stéphane Vialette
stephane.vialette@univ-eiffel.fr

Laboratoire d'Informatique Gaspard-Monge, UMR CNRS 8049,
Université Gustave Eiffel

Conditional

Conditional expressions

For processing conditions, the `if-then-else` syntax was defined in Haskell98.

```
if <condition> then <true-value> else <false-value>
```

`if` is an expression (which is converted to a value) and not a statement (which is executed) as in many imperative languages. As a consequence, the `else` is mandatory in Haskell. Since `if` is an expression, it must evaluate to a result whether the condition is true or false, and the `else` ensures this.

Conditional expressions

```
abs :: Int -> Int
abs n = if n >= 0 then n else -n

signum :: Int -> Int
signum n = if n < 0 then -1
           else if n == 0 then 0 else 1

describeLetter :: Char -> String
describeLetter c = if c >= 'a' && c <= 'z'
                   then "Lower case"
                   else if c >= 'A' && c <= 'Z'
                        then "Upper case"
                        else "Not an ASCII letter"
```

Conditional expressions

```haskell
addOneIfEven1 :: Integral a => a -> a
addOneIfEven1 n = if even n then n+1 else n

addOneIfEven2 :: Integral a => a -> a
addOneIfEven2 n = n + if even n then 1 else 0

addOneIfEven3 :: Integral a => a -> a
addOneIfEven3 n = (if even n then (+ 1) else (+ 0)) n

addOneIfEven4 :: Integral a => a -> a
addOneIfEven4 n = (if even n then (+ 1) else id) n
```

Conditional expressions

Remember that

```
isNullLength :: Foldable t => t a -> Bool
isNullLength xs = if length xs == 0 then True else False
```

is nothing but

```
isNullLength :: Foldable t => t a -> Bool
isNullLength xs = length xs == 0
```

or (as we we shall see soon . . . but not really better here!)

```
isNullLength :: Foldable t => t a -> Bool
isNullLength = (== 0) . length
```

Exercices

The function `Data.Char.isSpace` returns `True` for any Unicode space character, and the control characters `\t`, `\n`, `\r`, `\f` and `\v`.

```
import Data.Char

hasSpace (x : xs) = if isSpace a then True
                    else hasSpace xs
```

Write `hasSpace` without `if ... then ... else` (hint: use `(||) :: Bool -> Bool -> Bool`).

Guarded expressions

As an alternative to using conditional expressions, functions can also be defined using guarded expressions, in which a sequence of logical expressions called guards is used to choose between a sequence of results of the same type.

- If the first guard is `True`, then the first result is chosen.
- Otherwise, if the second guard is `True`, then the second result is chosen.
- And so on.

Guarded expressions

```
abs1 :: Int -> Int
abs1 n = if n >= 0 then n else -n

abs2 :: Int -> Int
abs2 n
  | n >= 0    = n
  | otherwise = -n
```

Guarded expressions

```haskell
signum1 :: Int -> Int
signum1 n = if n < 0 then -1 else
              if n == 0 then 0 else 1

signum2 :: Int -> Int
signum2 n
  | n < 0     = -1
  | n == 0    = 0
  | otherwise = 1
```

Guarded expressions

```
describeLetter1 :: Char -> String
describeLetter1 c = if c >= 'a' && c <= 'z'
                    then "Lower case"
                    else if c >= 'A' && c <= 'Z'
                         then "Upper case"
                         else "Not an ASCII letter"

describeLetter2 :: Char -> String
describeLetter2 c
  | c >= 'a' && c <= 'z' = "Lower case"
  | c >= 'A' && c <= 'Z' = "Upper case"
  | otherwise            = "Not an ASCII letter"
```

Guarded expressions

```
fact :: (Eq a, Num a) => a -> a
fact n
  | n == 0    = 1
  | otherwise = n * fact (n-1)

mult :: (Eq a, Num a, Num b) => b -> a -> b
mult n m
  | m == 0    = 0
  | otherwise = n + mult n (m - 1)
```

Guarded expressions

```haskell
-- Bad implementation:
fact :: Integer -> Integer
fact n
  | n == 0 = 1
  | n /= 0 = n * fact (n-1)

-- Slightly improved implementation:
fact :: Integer -> Integer
fact n
  | n == 0    = 1
  | otherwise = n * fact (n-1)
```

Exercices

Using guards, define a function

```
max4 :: Int -> Int -> Int -> Int > Int
```

that returns the maximum of four integers.

Pattern matching

Conditional

Pattern matching

Lambdas

`let` and `where`

Some functions

Pattern matching

Many functions have a simple and intuitive definition using pattern matching, in which a sequence of syntactic expressions called patterns is used to choose between a sequence of results of the same type.

The wildcard pattern `_` matches any value.

- If the first pattern is matched, then the first result is chosen.
- Otherwise, if the second pattern is matched, then the second result is chosen.
- And so on. . .

Pattern matching

```haskell
-- conditional expression
not :: Bool -> Bool
not b = if b then False else True

-- guarded function
not :: Bool -> Bool
not b
  | b         = False
  | otherwise = True

-- pattern matching
not :: Bool -> Bool
not False = True
not True  = False
```

Pattern matching

```
(&&) :: Bool -> Bool -> Bool
True  && True  = True
True  && False = False
False && True  = False
False && False = False

(&&) :: Bool -> Bool -> Bool
True && True  = True
_    && _     = False

(&&) :: Bool -> Bool -> Bool
True  && b = b
False && _ = False
```

Pattern matching

```haskell
guess :: Int -> String
guess 0 = "I am zero"
guess 1 = "I am one"
guess 2 = "I am two"
guess _ = "I am at least three"

-- be careful with the wildcard pattern !
guess :: Int -> String
guess _ = "I am at least three"
guess 0 = "I am zero"
guess 1 = "I am one"
guess 2 = "I am two"
```

Short circuiting

```haskell
(&&)            :: Bool -> Bool -> Bool
True  && x      =  x
False && _      =  False
```

```
λ> 1 `div` 0
*** Exception: divide by zero
λ> f x = x > 0 && x `div` 0 == 0
λ> f 1
*** Exception: divide by zero
λ> f (-1)
False
```

Short circuiting

```
(||)            :: Bool -> Bool -> Bool
True  && _      =  True
_     && x      =  x
```

```
λ> 1 `div` 0
*** Exception: divide by zero
λ> g x = x > 0 || x `div` 0 == 0
λ> g 1
True
λ> g (-1)
*** Exception: divide by zero
```

Short circuiting

Consider the valid function definition:

```
f ((x : _) : xs) = x + f xs
```

Give the type of `f`. Is it a safe function? Why? Improve.

Pattern matching – Tuple patterns

A tuple of patterns is itself a pattern, which matches any tuple of the same arity whose components all match the corresponding patterns in order.

Functions `fst` and `snd` are defined in the module `Data.Tuple`:

```
λ> :type fst
fst :: (a, b) -> a
λ> fst (1,2)
1
λ> :type snd
snd :: (a, b) -> b
λ> snd (1,2)
2
```

Pattern matching – Tuple patterns

A tuple of patterns is itself a pattern, which matches any tuple of the same arity whose components all match the corresponding patterns in order.

Functions `fst` and `snd` are defined in the module `Data.Tuple`:

```
fst :: (a, b) -> a
fst (x, _) = x
snd :: (a, b) -> b
snd (_, x) = x
```

Pattern matching – Tuple patterns

A tuple of patterns is itself a pattern, which matches any tuple of the same arity whose components all match the corresponding patterns in order.

```
first3 :: (a, b, c) -> a
first3 (x, _, _) = x
second3 :: (a, b, c) -> b
second3 (_, x, _) = x
third3 :: (a, b, c) -> c
third3 (_, _, x) = x
```

Pattern matching – Tuple patterns

A tuple of patterns is itself a pattern, which matches any tuple of the same arity whose components all match the corresponding patterns in order.

```
first4 :: (a, b, c, d) -> a
first4 (x, _, _, _) = x
second4 :: (a, b, c, d) -> b
second4 (_, x, _, _) = x
third4 :: (a, b, c, d) -> c
third4 (_, _, x, _) = x
fourth4 :: (a, b, c, d) -> d
fourth4 (_, _, _, x) = x
```

Short circuiting

Consider the valid function definition:

```
f []              = 0
f ((x, y) : xys) = x + y + f xys
```

Give the type of `f`. Is it a safe function? Why?

Pattern matching – List patterns

A list of patterns is itself a pattern, which matches any list of the same length whose components all match the corresponding patterns in order.

```
-- three characters beginning with the letter 'a'
test :: [Char] -> Bool
test ['a', _, _] = True
test _           = False

-- four characters ending with the letter 'z'
test :: [Char] -> Bool
test [_, _, _, 'z'] = True
test _              = False
```

Pattern matching – List patterns

A list of patterns is itself a pattern, which matches any list of the same length whose components all match the corresponding patterns in order.

These are two different functions

```
-- three characters beginning with the letter 'a'
test :: [Char] -> Bool
test ['a', _, _] = True
test _           = False

-- three characters beginning with the letter 'a'
test :: (Char, Char, Char) -> Bool
test ('a', _, _) = True
test _           = False
```

Exercices

Define a function `intersperse'` that takes an element and a list and *intersperses* that element between the elements of the list. The type definition should be

```haskell
intersperse' :: a -> [a] -> [a]
```

```
λ> intersperse' ',' ""
""
λ> intersperse' ',' "a"
"a"
λ> intersperse' ',' "abcd"
"a,b,c,d"
λ> intersperse' "--" ["a", "cd", "edf"]
["a","--","cd","--","edf"]
```

Exercices

Define the function

```
intercalate' :: [a] -> [[a]] -> [a]
```

The call `intercalate' xs xss` inserts the list `xs` in between the lists in `xss` and concatenates the result.

```
λ> intercalate' "--" []
""
λ> intercalate' "--" ["ab"]
"ab"
λ> intercalate' "--" ["ab", "cde", "fg"]
"ab--cde--fg"
λ> intercalate' ["--"] [["ab"], ["cde"], ["fg"]]
["ab","--","cde","--","fg"]
λ> intersperse' "--" ["ab", "cde", "fg"] -- cf previous exercice
["ab","--","cde","--","fg"]
```

Lambdas

Pattern matching – Lambda expression

- An anonymous function is a function without a name.
- It is a Lambda abstraction and might look like this:
  `\x -> x + 1`.

That backslash is Haskell's way of expressing a λ and is supposed to look like a Lambda ... if one has enough imagination!

```
λ> :type (\x -> x+1)
(\x -> x+1) :: Num a => a -> a
λ> (\x -> x+1) 2
3
```

Pattern matching – Lambda expression

The definition

```
add :: Int -> Int -> Int -> Int
add x y z = x + y + z
```

can be understood as meaning

```
add :: Int -> Int -> Int -> Int
add = \x -> (\y -> (\z -> x + y + z))
```

which makes precise that `add` is a function that takes an integer `x` and returns a function which in turn takes another integer `y` and returns a function which in turn takes another integer `z` and returns the result `x+y+z`.

Pattern matching – Lambda expression

λ-expressions are useful when defining functions that returns function as results by their very nature, rather than a consequence of currying.

```
const :: a -> b -> a
const x _ = x

-- emphasis const :: a -> (b -> a)
const :: a -> b -> a
const x = \_ -> x
```

Pattern matching – Lambda expression

A closure (the opposite of a combinator) is a function that makes use of free variables in its definition. It closes around some portion of its environment.

```haskell
f :: Num a => a -> a -> a
f x = \y -> x + y
```

`f` returns a closure, because the variable `x`, which is bounded outside of the lambda abstraction is used inside its definition.

```
λ> g = f 1
λ> g 2
3
λ> g 3
4
λ> g 4
5
```

Exercices

Consider the function

```haskell
f :: (a -> a -> b) -> (b -> b -> c) -> a -> a -> c
f = \g h x y -> (x `g` x) `h` (y `g` y)
```

Explain the following session.

```
λ> f (+) (*) 2 5
40
λ> f (*) (+) 2 5
29
λ> let o1 = (+); o2 = (*) in f o1 o2 2 5
40
λ> let o = (+) in f o o 2 5
14
λ> f (\ x y -> x + 2*y) (\x y -> x - y) 2 5
-9
```

Exercices

Explain the following functions:

```
g :: Int -> Int
g = \x -> x * x

h :: Int -> Int
h = \x -> g (g x)

i :: Int -> Int
i x = h (h x)
```

Pattern matching – Operator sections

- Functions such as + that are written between their two arguments are called section
- Any operator can be converted into a curried function by enclosing the name of the operator in parentheses, such as `(+) 1 2`.
- More generally, if o is an operator, then expression of the form `(o)`, `(x o)` and `(o y)` are called sections whose meaning as functions can be formalised using λ-expressions as follows:

```
(o)   = \x -> (\y -> x o y))
(x o) = \y -> x o y
(o y) = \x -> x o y
```

Pattern matching – Operator sections

- `(+)` is the addition function `\x -> (\y -> x+y)`.
- `(1 +)` is the successor function `\y -> 1+y`.
- `(1 /)` is the reciprocation function `\y -> 1/y`.
- `(* 2)` is the doubling function `\x -> x*2`.
- `(/ 2)` is the halving function `\x -> x/2`.

Sections

Explain the following functions:

```
f :: [Char] -> [Char]
f = ("A" ++)

g :: [Char] -> [Char]
g = (++ "Z")

h :: [Char] -> [Char]
h = \x -> f (g x)

i :: [Char] -> [Char]
i = \x -> g (f x)
```

Do we have `h xs == i xs` for every string `xs`?

Sections

Explain the difference between the following two sessions:

```
λ> f = (+ 1)
λ> :type f
f :: Num a => a -> a
λ> f 2
3
```

```
λ> g = (- 3)
λ> :type g
g :: Num a => a
λ> g 1
<interactive>:32:1: error:
λ> g
-3
```

let **and** where

Pattern matching – Bindings

- A `where` clause is used to divide the more complex logic or calculation into smaller parts, which makes the logic or calculation easy to understand and handle
- A `where` clause is bound to a surrounding syntactic construct, like the pattern matching line of a function definition.
- A `where` clause is a syntactic construct

Pattern matching – Bindings

```haskell
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
  | weight / height ^ 2 <= 18.5 = "Underweight"
  | weight / height ^ 2 <  25.0 = "Healthy weight"
  | weight / height ^ 2 <  30.0 = "Overweight"
  | otherwise                   = "Obese"
```

```haskell
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
  | bmi <= 18.5 = "Underweight"
  | bmi < 25.0  = "Healthy weight"
  | bmi < 30.0  = "Overweight"
  | otherwise   = "Obese"
  where
    bmi = weight / height ^ 2
```

Pattern matching – Bindings

```
bmiTell :: (RealFloat a) => a -> a -> String
bmiTell weight height
  | bmi <= underweight = "Underweight"
  | bmi <  healthy     = "Healthy weight"
  | bmi <  overweight  = "Overweight"
  | otherwise          = "Obese"
  where
    bmi         = weight / height ^ 2
    underweight = 18.5
    healthy     = 25
    overweight  = 30
```

Pattern matching – Bindings

- A let binding binds variables anywhere and is an expression itself, but its scope is tied to where the let expression appears.
- if a let binding is defined within a guard, its scope is local and it will not be available for another guard.
- A let binding can take global scope overall pattern-matching clauses of a function definition if it is defined at that level.

Pattern matching – Bindings

```haskell
cylinder :: (RealFloat a) => a -> a -> a
cylinder r h =
  let sideArea = 2 * pi * r * h
      topArea  = pi * r ^2
  in  sideArea + 2*topArea
```

Pattern matching – Bindings

```
λ >  let zoot x y z = x*y + z
λ >  :type zoot
zoot :: Num a => a -> a -> a -> a
λ >  zoot 3 9 2
29
λ >  let boot x y z = x*y + z in boot 3 9 2
29
λ >  :type boot
<interactive>: error:
  o Variable not in scope: boot
```

Pattern matching – Bindings

```
λ> let a = 1; b = 2 in a + b
3
λ> let a = 1; b = a + 2 in a + b
4
λ> let a = 1; a = 2 in a
<interactive>:: error:
    Conflicting definitions for 'a'
λ> let a = 1; b = 2+a; c = 3+a+b in (a, b, c)
(1,3,7)
```

Pattern matching – Bindings

```
λ > let a = 1 in let a = 2; b = 3+a in b
5
λ > let a = 1 in let a = a+2 in let b = 3+a in b
^CInterrupted.
λ > let f x y = x+y+1 in f 3 5
9
λ > let f x y = x+y; g x = f x (x+1) in g 5
11
```

Pattern matching – Bindings

```haskell
dist :: Floating a => (a, a) -> (a, a) -> a
dist (x1,y1) (x2,y2) =
      let xdist = x2 - x1
          ydist = y2 - y1
          sqr z = z*z
      in sqrt ((sqr xdist) + (sqr ydist))

dist :: Floating a => (a, a) -> (a, a) -> a
dist (x1,y1) (x2,y2) = sqrt ((sqr xdist) + (sqr ydist))
  where
    xdist = x2 - x1
    ydist = y2 - y1
    sqr z = z*z
```

Pattern matching – Bindings

We can pattern match with `let` bindings. E.g., we can dismantle a tuple into components and bind the components to names.

```
λ> f x y z = let (sx,sy,sz) = (x*x,y*y,z*z) in (sx,sy,sz)
λ> f 1 2 3
(1,4,9)
λ> g x y = let (sx,_) = (x*x,y*y) in sx
λ> g 2 3
4
λ> h x = let ((sx,cx),qx) = ((x*x,x*x*x),x*x*x*x) in (sx,cx,qx)
λ> h 2
(4,8,16)
```

Pattern matching – Bindings

`let` bindings are expressions.

```
λ> 1 + let x = 2 in x*x
5
λ> (let x = 2 in x*x) + 1
5
λ> (let (x,y,z) = (1,2,3) in x+y+z) * 100
600
λ> (let x = 2 in (+ x)) 3
5
λ> let x=3 in x*x + let x=4 in x*x
25
```

Some functions

Conditional

Pattern matching

Lambdas

`let` and `where`

Some functions

Double factorial

The double factorial (or semifactorial of a number n, denoted by $n!!$, is the product of all the integers from 1 up to n that have the same parity (odd or even) as n

Double factorial

The double factorial (or semifactorial of a number *n*, denoted by *n*!!, is the product of all the integers from 1 up to *n* that have the same parity (odd or even) as *n*

```
dblFact1 :: Int -> Int
dblFact1 n = go n
  where
    go 0 = 1
    go m
      | p m       = m * go (m-1)
      | otherwise = go (m-1)
      where
        p m = (even n && even m) || (odd n && odd m)
```

Double factorial

The double factorial (or semifactorial of a number n, denoted by $n!!$, is the product of all the integers from 1 up to n that have the same parity (odd or even) as n

```
dblFact2 :: Int -> Int
dblFact2 n = go n
  where
    go 0 = 1
    go m
      | p m       = m * go (m-1)
      | otherwise = go (m-1)
      where
        nParity2 = n `mod` 2
        p m = m `mod` 2 == nParity2
```

Double factorial

The double factorial (or semifactorial of a number *n*, denoted by *n*!!, is the product of all the integers from 1 up to *n* that have the same parity (odd or even) as *n*

```
dblFact3 :: Int -> Int
dblFact3 0 = 1
dblFact3 1 = 1
dblFact3 n = n * dblFact3 (n-2)
```

Double factorial

The double factorial (or semifactorial of a number n, denoted by $n!!$, is the product of all the integers from 1 up to n that have the same parity (odd or even) as n

```
dblFact4 :: Int -> Int
dblFact4 n = product [n,n-2..1]
```

Collatz conjecture

The Collatz conjecture is one of the most famous unsolved problems in mathematics. It concerns sequences of integers in which each term is obtained from the previous term as follows:

$$u_n = \begin{cases} u_{n-1}/2 & \text{if } u_{n-1} \text{ is even} \\ 3u_{n-1}+1 & \text{if } u_{n-1} \text{ is odd} \end{cases}$$

For instance, starting with $n = 19$, one gets the sequence $19, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1.$

Collatz conjecture

The Collatz conjecture is one of the most famous unsolved problems in mathematics. It concerns sequences of integers in which each term is obtained from the previous term as follows:

$$u_n = \begin{cases} u_{n-1}/2 & \text{if } u_{n-1} \text{ is even} \\ 3u_{n-1}+1 & \text{if } u_{n-1} \text{ is odd} \end{cases}$$

For instance, starting with $n = 19$, one gets the sequence $19, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1.$

```
collatz1 1 = "win"
collatz1 n = collatz1 (if even n
                       then n `div` 2
                       else 3*n + 1)
```

Collatz conjecture

The Collatz conjecture is one of the most famous unsolved problems in mathematics. It concerns sequences of integers in which each term is obtained from the previous term as follows:

$$u_n = \begin{cases} u_{n-1}/2 & \text{if } u_{n-1} \text{ is even} \\ 3u_{n-1} + 1 & \text{if } u_{n-1} \text{ is odd} \end{cases}$$

For instance, starting with $n = 19$, one gets the sequence $19, 58, 29, 88, 44, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1$.

```haskell
collatz2 :: Integral a => a -> String
collatz2 1 = "win"
collatz2 n
  | even n    = collatz2 (n `div` 2)
  | otherwise = collatz2 (3*n + 1)
```

Ackermann–Péter function

$$
\begin{aligned}
A(0, n) &= n + 1 \\
A(m + 1, 0) &= A(m, 1) \\
A(m + 1, n + 1) &= A(m, A(m + 1, n))
\end{aligned}
$$

```haskell
aP :: (Num a, Eq a, Num b, Eq b) => a -> b -> b
aP 0 n = n+1
aP m 0 = aP (m-1) 1
aP m n = aP (m-1) (aP m (n-1))
```

Prime numbers

A prime number (or a prime) is a natural number greater than 1 that is not a product of two smaller natural numbers.

```haskell
-- very naive
isPrime :: Integral a => a -> Bool
isPrime 0 = False
isPrime 1 = False
isPrime n = go 2
  where
    go k
      | k >= n    = True
      | otherwise = n `mod` k /= 0 && go (k+1)
```

Ping-pong programming

```haskell
-- odd number predicate
isOdd :: (Eq a, Num a) => a -> Bool
isOdd 0 = False
isOdd 1 = True
isOdd n = isEven (n-1)

-- even number predicate
isEven :: (Eq a, Num a) => a -> Bool
isEven 0 = True
isEven 1 = False
isEven n = isOdd (n-1)
```

Factorial

```haskell
fact1 :: (Eq a, Num a) => a -> a
fact1 n = if n == 0 then 1 else n * fact1 (n-1)

fact2 :: (Eq a, Num a) => a -> a
fact2 n
  | n == 0 = 1
  | otherwise = n * fact2 (n-1)
```

Factorial

```
fact3 :: (Ord a, Num a) => a -> a
fact3 = go 1
  where
    go m n
      | m > n     = 1
      | otherwise = m * go (m+1) n

fact4 :: (Eq t, Num t) => t -> t
fact4 n = go 1 n
  where
    go acc 0 = acc
    go acc m = go (acc*m) (m-1)
```

Factorial

```
fact5 :: (Enum a, Num a) => a -> a
fact5 n = product [1..n]
```

Pascal triangle

						1						
					1		1					
				1		2		1				
			1		3		3		1			
		1		4		6		4		1		
	1		5		10		10		5		1	
1		6		15		20		15		6		1

Pascal triangle

$$
\begin{array}{lllllll}
\binom{0}{0} = 1 & & & & & & \\
\binom{1}{0} = 1 & \binom{1}{1} = 1 & & & & & \\
\binom{2}{0} = 1 & \binom{2}{1} = 2 & \binom{2}{2} = 1 & & & & \\
\binom{3}{0} = 1 & \binom{3}{1} = 3 & \binom{3}{2} = 3 & \binom{3}{3} = 1 & & & \\
\binom{4}{0} = 1 & \binom{4}{1} = 4 & \binom{4}{2} = 6 & \binom{4}{3} = 4 & \binom{4}{4} = 1 & & \\
\binom{5}{0} = 1 & \binom{5}{1} = 5 & \binom{5}{2} = 10 & \binom{5}{3} = 10 & \binom{5}{4} = 5 & \binom{5}{5} = 1 & \\
\binom{6}{0} = 1 & \binom{6}{1} = 5 & \binom{6}{2} = 15 & \binom{6}{3} = 20 & \binom{4}{4} = 20 & \binom{6}{5} = 6 & \binom{6}{6} = 1
\end{array}
$$

Pascal triangle

$$\binom{0}{0} = 1$$

$$\binom{1}{0} = 1 \quad \binom{1}{1} = 1$$

$$\binom{2}{0} = 1 \quad \binom{2}{1} = 2 \quad \binom{2}{2} = 1$$

$$\binom{3}{0} = 1 \quad \binom{3}{1} = 3 \quad \binom{3}{2} = 3 \quad \binom{3}{3} = 1$$

$$\binom{4}{0} = 1 \quad \binom{4}{1} = 4 \quad \binom{4}{2} = 6 \quad \binom{4}{3} = 4 \quad \binom{4}{4} = 1$$

$$\binom{5}{0} = 1 \quad \binom{5}{1} = 5 \quad \binom{5}{2} = 10 \quad \binom{5}{3} = 10 \quad \binom{5}{4} = 5 \quad \binom{5}{5} = 1$$

$$\binom{6}{0} = 1 \quad \binom{6}{1} = 5 \quad \binom{6}{2} = 15 \quad \binom{6}{3} = 20 \quad \binom{4}{4} = 20 \quad \binom{6}{5} = 6 \quad \binom{6}{6} = 1$$

Pascal's relation

$$\binom{n+1}{k+1} = \binom{n}{k} + \binom{n}{k+1}$$

Pascal triangle

$$\binom{0}{0} = 1$$

$$\binom{1}{0} = 1 \quad \binom{1}{1} = 1$$

$$\binom{2}{0} = 1 \quad \binom{2}{1} = 2 \quad \binom{2}{2} = 1$$

$$\binom{3}{0} = 1 \quad \binom{3}{1} = 3 \quad \binom{3}{2} = 3 \quad \binom{3}{3} = 1$$

$$\binom{4}{0} = 1 \quad \binom{4}{1} = 4 \quad \binom{4}{2} = 6 \quad \binom{4}{3} = 4 \quad \binom{4}{4} = 1$$

$$\binom{5}{0} = 1 \quad \binom{5}{1} = 5 \quad \binom{5}{2} = 10 \quad \binom{5}{3} = 10 \quad \binom{5}{4} = 5 \quad \binom{5}{5} = 1$$

$$\binom{6}{0} = 1 \quad \binom{6}{1} = 5 \quad \binom{6}{2} = 15 \quad \binom{6}{3} = 20 \quad \binom{4}{4} = 20 \quad \binom{6}{5} = 6 \quad \binom{6}{6} = 1$$

```haskell
pT :: (Num a, Ord a, Num b) => a -> a -> b
pT n k
  | n == 1 && k == 1 = 1
  | k < 1  || k > n  = 0
  | otherwise        = pT (n-1) (k-1) + pT (n-1) k
```

Exercices

Consider the following functions :

```haskell
fix :: (a -> a) -> a
fix f = f (fix f)

fact :: Integer -> Integer
fact = fix (\ r n -> if n == 0 then 1 else n * r (n-1))
```

Explain :

```
λ> fact 5
120
```

The "Lazy Caterer's" Recursive Puzzle

Imagine you are a chef cutting a giant circular pancake. You want to find the maximum number of pieces you can create using only straight vertical cuts.

- With 0 cuts, you have 1 piece.
- With 1 cut, you have 2 pieces.
- With 2 cuts, you can get a maximum of 4 pieces.
- With 3 cuts, if you arrange them so they don't all cross at the same point, you get 7 pieces.

Write a function `cuts :: Int -> Int` that calculates the maximum number of pieces for *n* cuts.