

WORKSHOP HASKELL

THIS IS WHERE THE FUN BEGINS



INTRODUCTION

LE PARADIGME FONCTIONNEL

PARADIGME ?

INTRODUCTION

LE PARADIGME FONCTIONNEL

PARADIGME ?

Modèle de conception



Modèle de réflexion

INTRODUCTION

LE PARADIGME FONCTIONNEL

PARADIGME ?

Modèle de conception

.....

Modèle de réflexion

.....

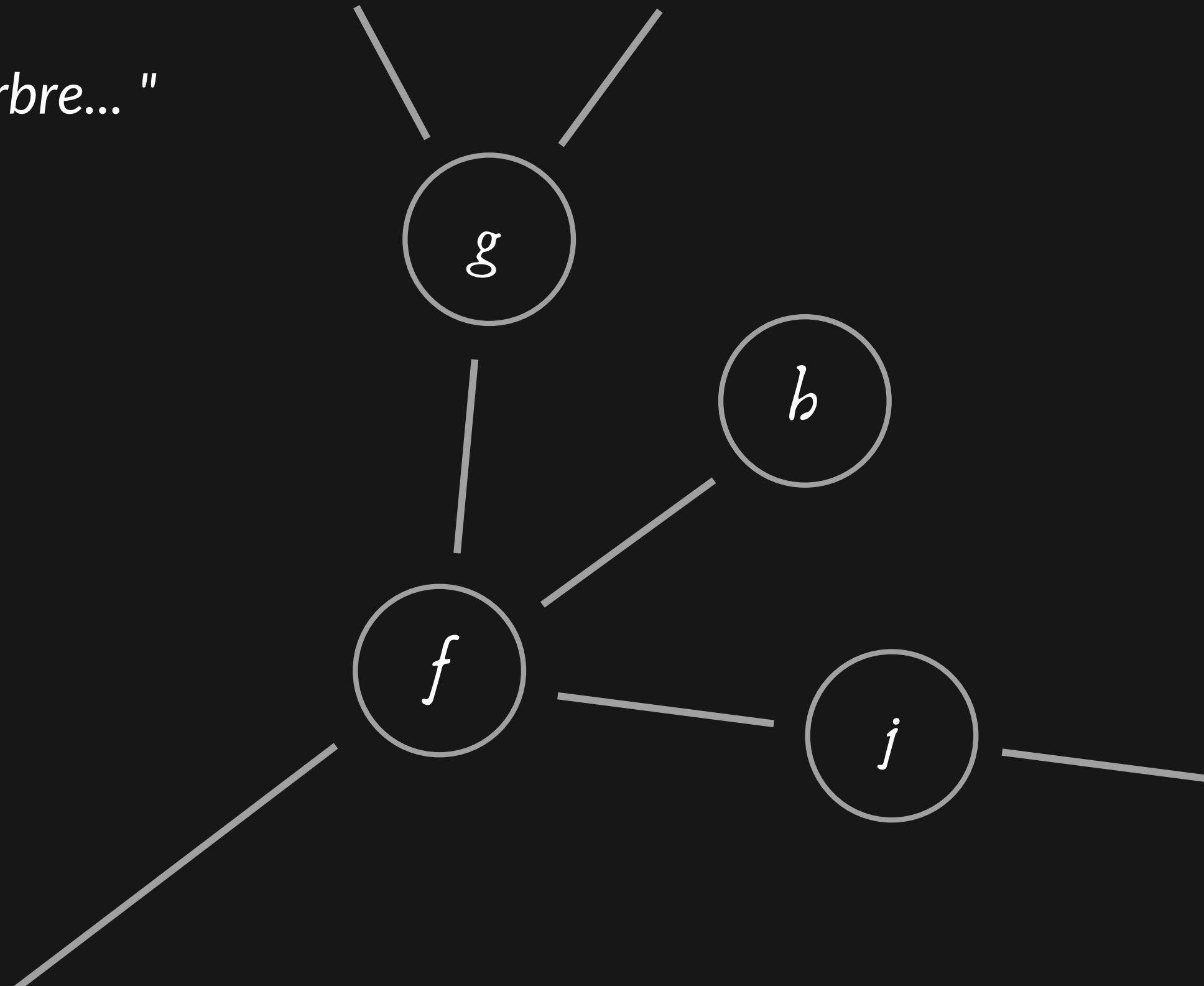
Fonctionnement *interne* du langage

INTRODUCTION

LE PARADIGME FONCTIONNEL

" L'Haskell c'est un arbre... "

- a smart person

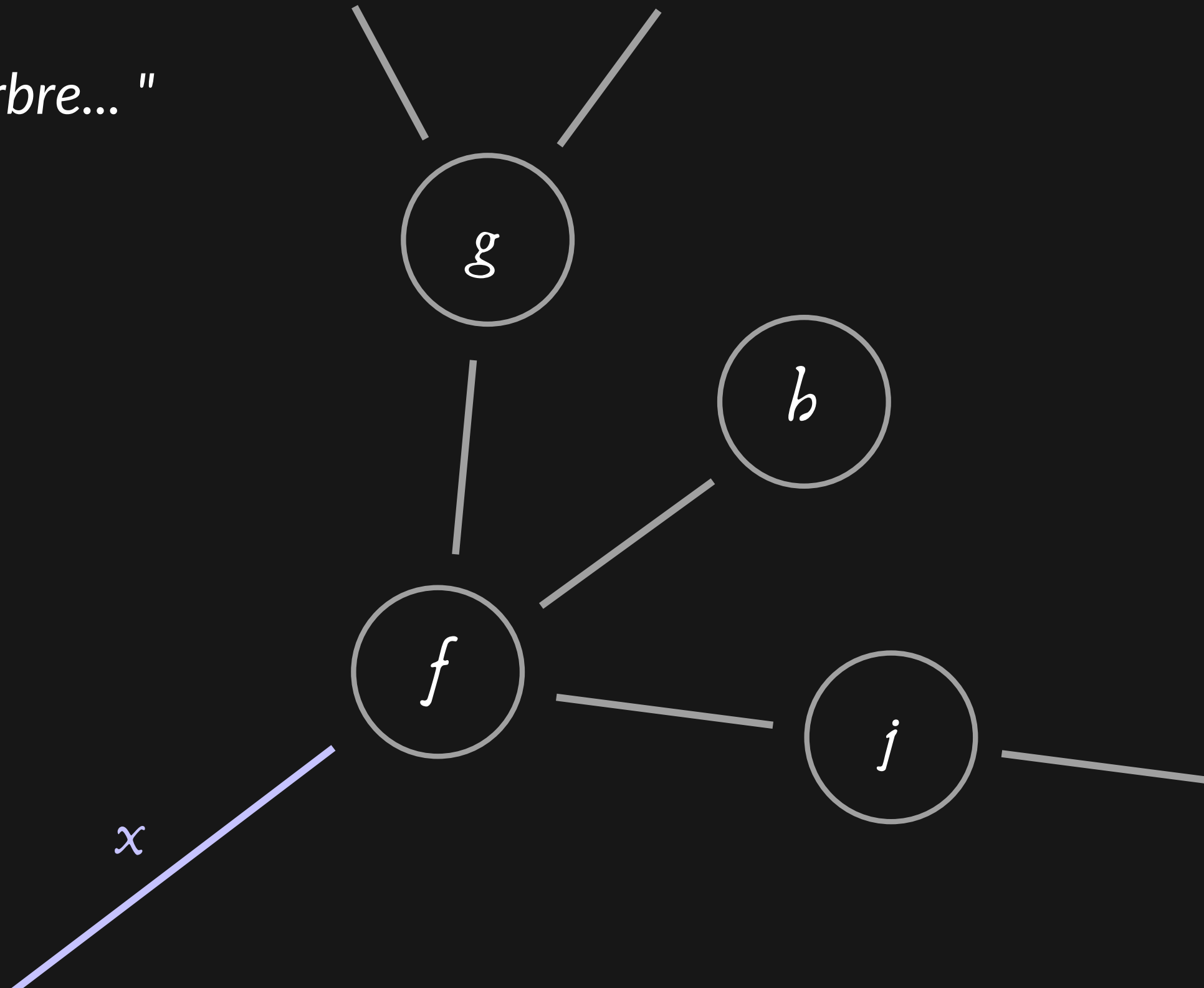


INTRODUCTION

LE PARADIGME FONCTIONNEL

" L'Haskell c'est un arbre... "

- a smart person

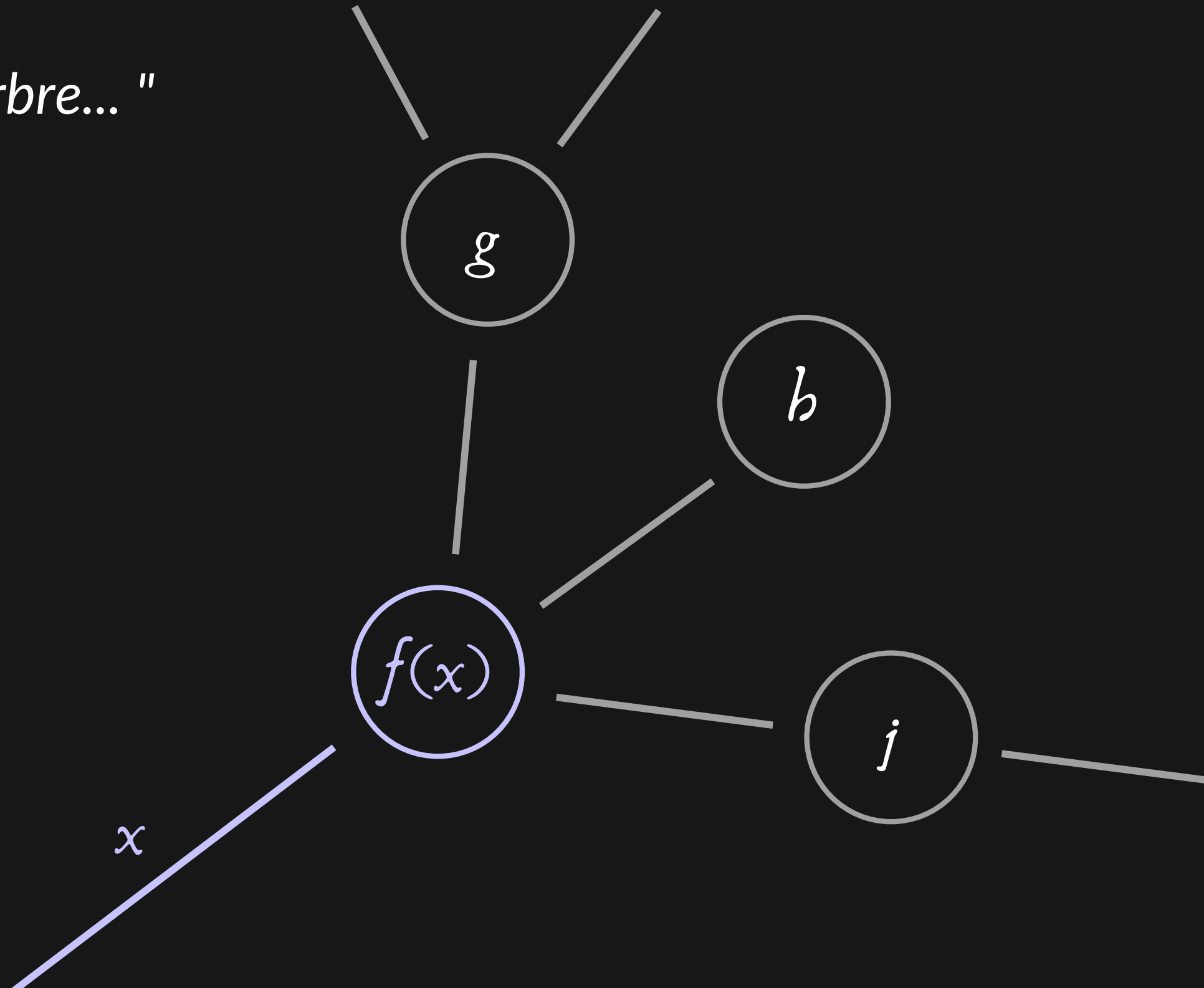


INTRODUCTION

LE PARADIGME FONCTIONNEL

" L'Haskell c'est un arbre... "

- a smart person

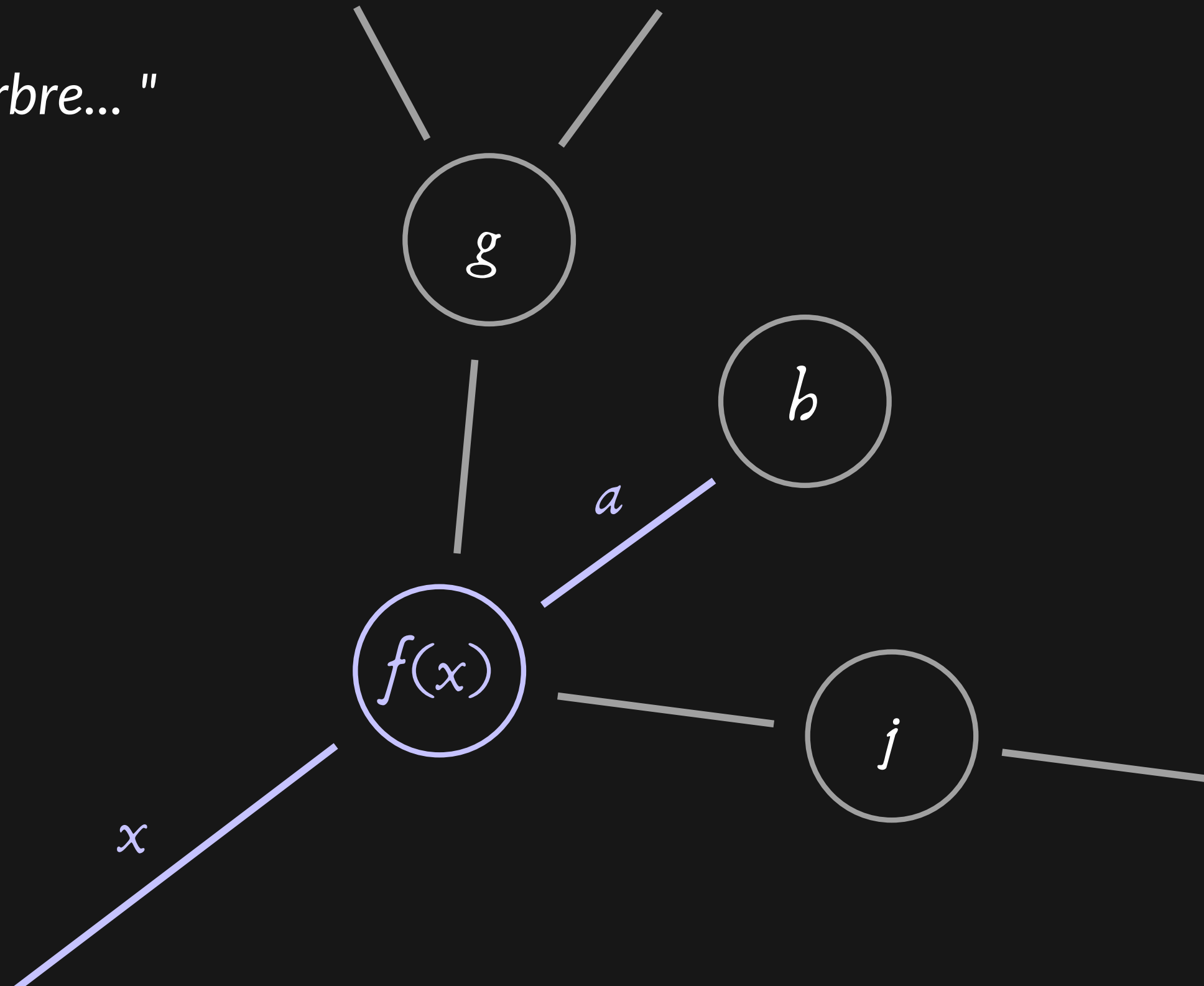


INTRODUCTION

LE PARADIGME FONCTIONNEL

" L'Haskell c'est un arbre... "

- a smart person

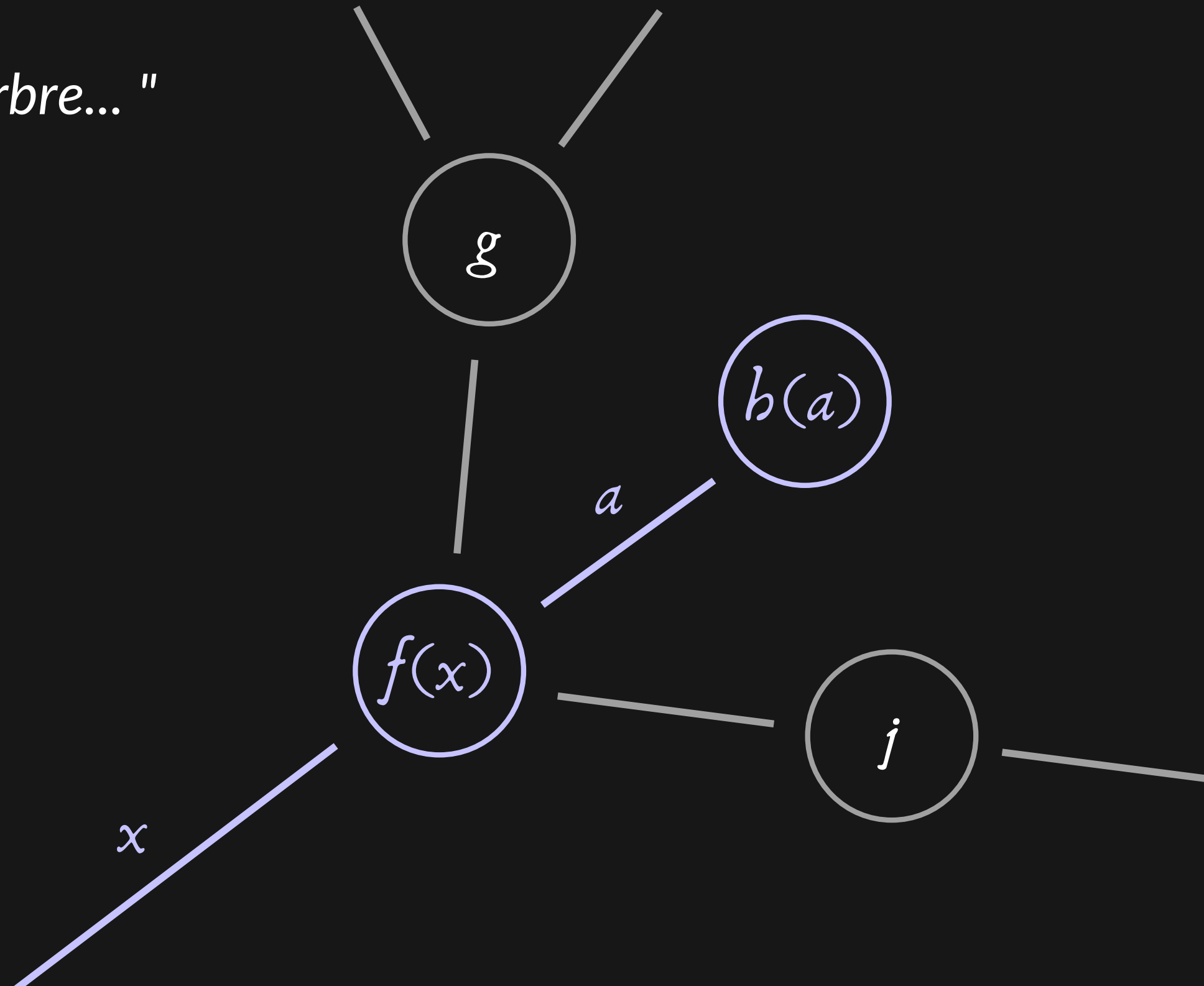


INTRODUCTION

LE PARADIGME FONCTIONNEL

" *L'Haskell c'est un arbre...* "

- a smart person

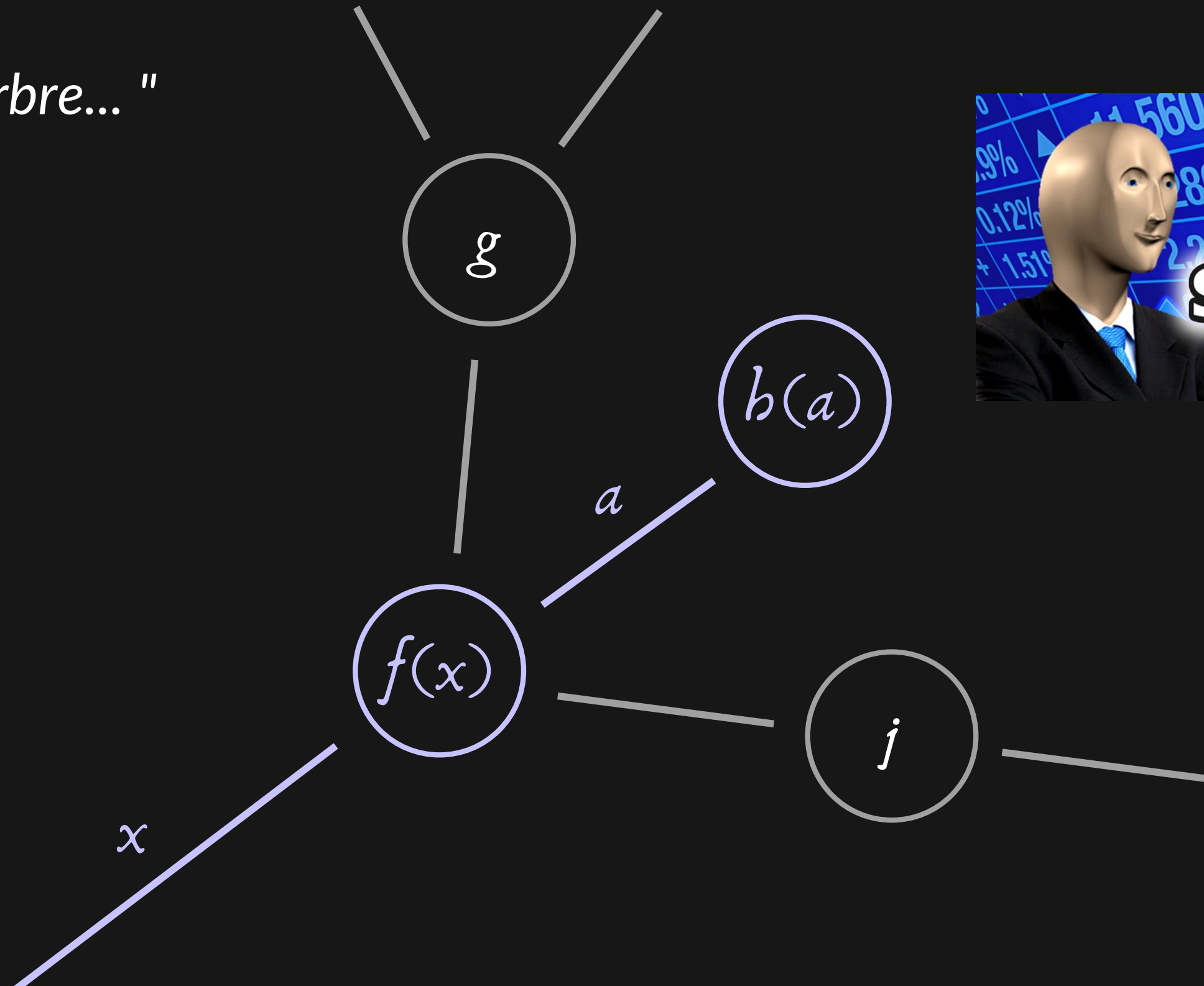


INTRODUCTION

LE PARADIGME FONCTIONNEL

" *L'Haskell c'est un arbre...* "

- a smart person



C VS HASKELL

FIBONACCI



C VS HASKELL

FIBONACCI

Itératif :

```
int fib(int n)
{
    int first = 0;
    int second = 1;
    int tmp = 0;

    while (n--)
    {
        tmp = first + second;
        first = second;
        second = tmp;
    }
    return first;
}
```



C VS HASKELL

FIBONACCI

Itératif :

```
int fib(int n)
{
    int first = 0;
    int second = 1;
    int tmp = 0;

    while (n--)
    {
        tmp = first + second;
        first = second;
        second = tmp;
    }
    return first;
}
```

Récurusif :

```
int fib(int n)
{
    if (n < 2)
    {
        return n;
    }
    else
    {
        return fib(n - 1) + fib(n - 2);
    }
}
```



C VS HASKELL

```
fib :: Int → Int  
fib 0 = 0  
fib 1 = 1  
fib n = fib (n - 1) + fib (n - 2)
```



$\text{fib} :: \text{Int} \rightarrow \text{Int}$

$\text{fib } 0 = 0$

$\text{fib } 1 = 1$

$\text{fib } n = \text{fib } (n - 1) + \text{fib } (n - 2)$



fib :: Int → Int

fib 0 = 0

fib 1 = 1

fib n = fib (n - 1) + fib (n - 2)



fib :: Int → Int

fib 0 = 0

fib 1 = 1

fib n = fib (n - 1) + fib (n - 2)



Argument

Retour de fonction

fib :: Int → Int

fib 0 = 0

fib 1 = 1

fib n = fib (n - 1) + fib (n - 2)



Argument

Retour de fonction

fib :: Int → Int

fib 0 = 0
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)



Argument

Retour de fonction

fib :: Int → Int

```
fib 0 = 0
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)
```

♥ Pattern
Matching



LE FILTRAGE PAR MOTIF

ou pattern matching



LE FILTRAGE PAR MOTIF

ou pattern matching

```
tellMe :: Int → String  
tellMe 0 = "It's a zero !"  
tellMe 1 = "It's a one !"  
tellMe 2 = "It's a two !"
```

LE FILTRAGE PAR MOTIF

ou pattern matching

```
tellMe :: Int → String  
tellMe 0 = "It's a zero !"  
tellMe 1 = "It's a one !"  
tellMe 2 = "It's a two !"
```

Pattern non exhaustif !

LE FILTRAGE PAR MOTIF

ou pattern matching

```
tellMe :: Int → String  
tellMe 0 = "It's a zero !"  
tellMe 1 = "It's a one !"  
tellMe 2 = "It's a two !"  
tellMe _ = "Other number."
```



LET'S GO FURTHER

types and newtypes

```
type X = Int  
type Y = Int
```

```
newtype Point = Point (X, Y)
```

LET'S GO FURTHER

types and newtypes

```
type X = Int  
type Y = Int
```

type = alias

```
newtype Point = Point (X, Y)
```

LET'S GO FURTHER

types and newtypes

```
type X = Int  
type Y = Int
```

type = alias

```
newtype Point = Point (X, Y)
```

newtype ?

LET'S GO FURTHER

Algebraic data types

```
data Shape = Square  
           | Triangle  
           | Octagon  
           | Circle  
           | Star Int
```

LET'S GO FURTHER

Algebraic data types

```
data Shape = Square  
           | Triangle  
           | Octagon  
           | Circle  
           | Star Int  
           enhanced
```

← Plusieurs
constructeurs !

LET'S GO FURTHER

Algebraic data types

```
data Shape = Square  
           | Triangle  
           | Octagon  
           | Circle  
           | Star Int
```

← Plusieurs
constructeurs !

Ce type de structure de données
s'apparente à des tagged unions.

PATTERN MATCHING BUT WITH MORE PATTERNS

La déconstruction de type !



`box :: Shape → IO ()`

PATTERN MATCHING BUT WITH MORE PATTERNS

La déconstruction de type !

```
data Shape = Square  
           | Triangle  
           | Octagon  
           | Circle  
           | Star Int
```

```
box :: Shape → IO ()  
box Square    = putStrLn "Square"  
box Triangle  = putStrLn "Triangle"  
box Octagon   = putStrLn "Octagon"  
box Circle    = putStrLn "Circle"  
box (Star a)  = putStrLn ("Star. Number ?" ++ tellMe a)
```


MORE PATTERNS ? AGAIN ??

Déconstruction des listes et des n-uplés

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

`toto (x, y) = ... -- x et y sont les deux membres du tuple`

MORE PATTERNS ? AGAIN ??

Déconstruction des listes et des n-uplés

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

`toto (x, y) = ...` -- `x` et `y` sont les deux membres du tuple

`toto :: [a] -> b`

`toto []` = ... -- représente une liste vide en entrée

`toto [e, e1] = ...` -- `e` et `e1` sont les seuls éléments de la liste

`toto (l:xs)` = ... -- sépare la tête de la liste du reste

ET LES CHAINES DE CARACTÈRES ?

String = [Char]

toto :: String -> a

toto [] = ... -- représente une chaine vide en entrée

toto "toto" = ... -- représente une chaine "toto" en entrée

toto ('t':lx) = ... -- représente une chaine commençant par 't'

toto ('a':'o':lx) = ... -- représente une chaine commençant par 'a' suivi d'un 'o'

AH OUI, ENCORE DES PATTERNS.

Le fun ne s'arrête jamais.

toto :: (Int, Int) -> c

toto (x, y) = ... -- x et y sont les deux membres du tuple

AH OUI, ENCORE DES PATTERNS.

Le fun ne s'arrête jamais.

```
toto :: (Int, Int) -> c
toto (x, y) | x > 10    = ... -- sera matché si x > 10
            | otherwise = ... -- tous les autres cas
```

Plus généralement :

```
f :: a -> b
f a | Condition = ...
    | etc       = ...
    | Sinon     = ...
```



DES QUESTIONS ?





DES QUESTIONS ?

Et maintenant, on va faire un **fizzbuzz**.

