{EPITECH}
LEARN DIFFERENT*

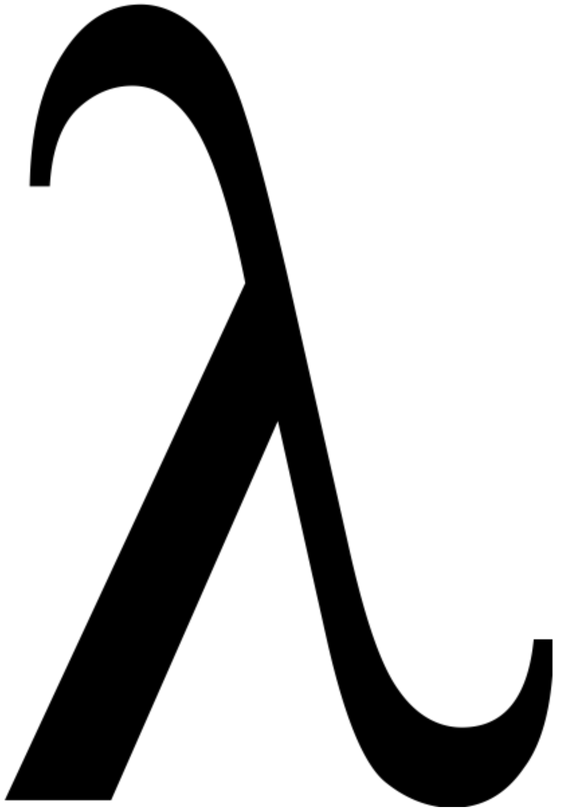
Le langage EK

Le langage du futur

Emil Pedersen, Jeremy Elalouf, Lindon Aliu, Marouan Bader, Yunse Lee — 15 février 2023

Le langage EK
Les bases

- Fonctionnel
- Impur
- Strict
- Dynamique
- Mixfixe



Le langage EK

Le Hello World

```
import std

fn main = print "Hello, World!"
```

Les fonctions

Comme en Haskell

Fonctions sans argument

```
fn key = 3
fn main = print key
```

Fonctions préfixes

```
fn createSum (a) (b) = a + b
fn main = print createSum 4 5
```

Fonctions infixes

```
fn (a) -- (b) = b ++ a
fn main = print ("i" -- "H")
```

Les fonctions

Pas comme en Haskell

Fonctions mixfixes

```
fn (a) withSum (b) (c) = a + b + c
fn main = print 3 withSum 4 5
```

Fonctions postfixes

```
fn (a) squared = a * a
fn main = print 5 squared
```

Fonctions ternaires

```
fn if (cond) then (t) else (f) = [...]
fn main = print (if 0 == 0 then 1 else 0)
```

Les types

Les types

Les atomes

```
atom true
atom false
atom void
atom empty
atom null
```

Les types

Les unions et alias de types

```
type bool = true | false
```

Les types

Les structures

```
struct Point {
    x: float,
    y: float,
}
```

Les types

Les primitives

Les strings

"Hello world"

Les floats

3.14

Les types

Les entiers

```
type digit = [0..9]
type bit = [0..1]
type bit = 0 | 1
type positive = [1..]
type negative = [..-1]
type int = [..]
type zero = 0
```

Les types

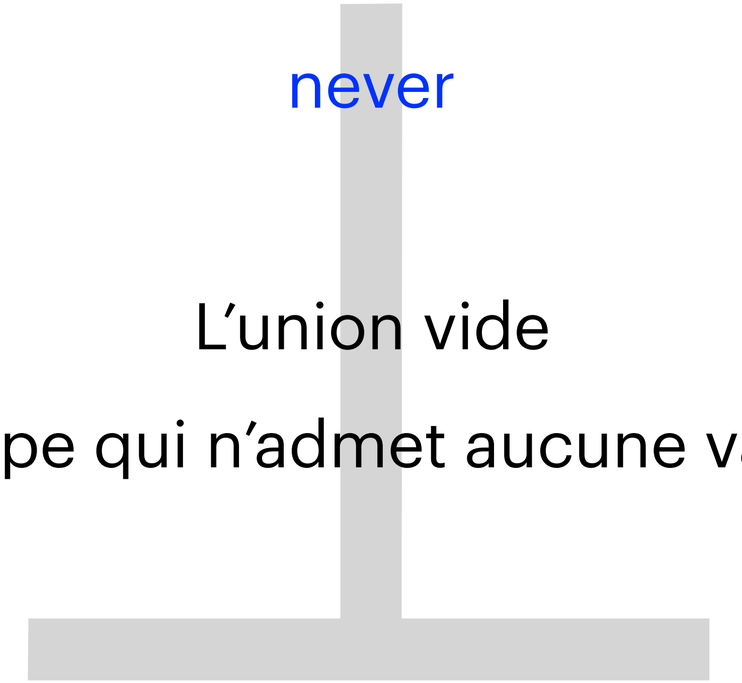
Les fonctions

```
type IntOperator = int -> int -> int
type Function = any -> any
```

Les types

Les spéciaux

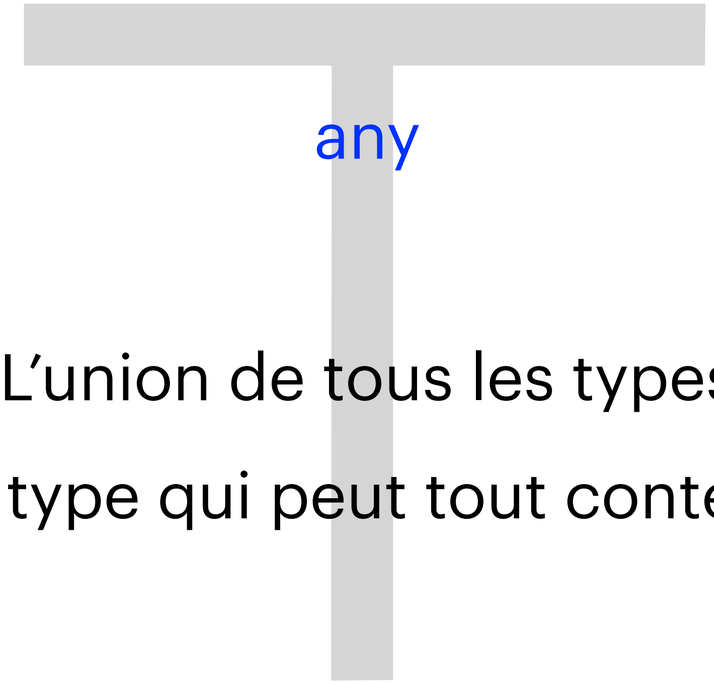
never



L’union vide

Le type qui n’admet aucune valeur

any



L’union de tous les types

Le type qui peut tout contenir

Les expressions

Les expressions

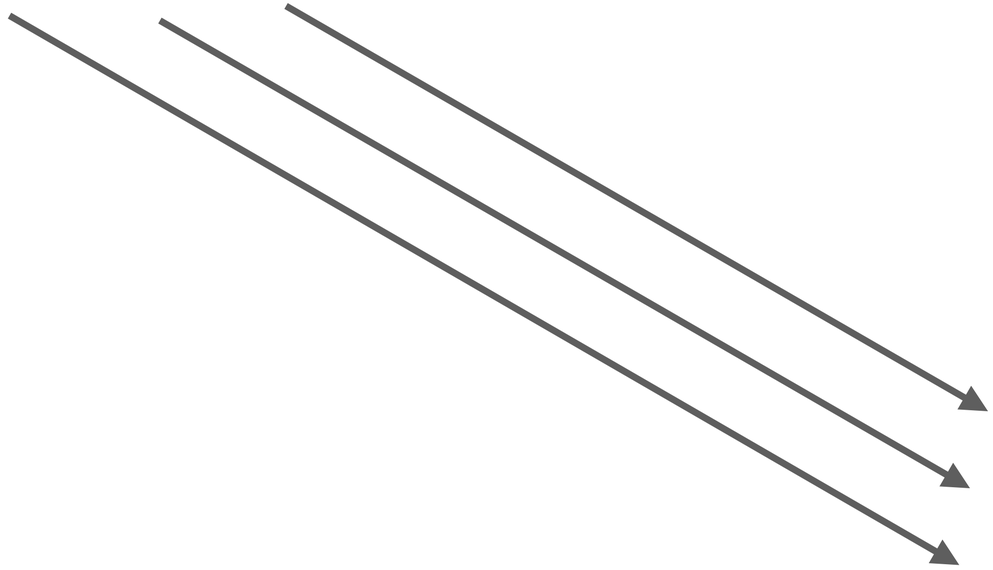
Les litéraux

Les strings	Les ints	Les floats
"Hello world"	42	3.14

Les expressions

Les ifs et appels de fonctions

```
print (if 3 > 1 then "Hello" else "World")
```



```
fn (a) > (b) = b < a
fn if (cond) then (t) else (f) = [...]
fn print (str) = [...]
```

Les expressions

Les lambdas (closures)

Lambdas

Closures

```
fn mySucc = (\a = a + 1)
fn myAdder (a) = (\b = a + b)
fn myOtherSucc = myAdder 1
```

Appel de lambda

```
fn test = mySucc $ 20
```

Les expressions

Les applications partielles

Applications partielles

Placeholders

```
fn mySucc = _ + 1
fn myAdder _ = _ + _
fn myOtherSucc = myAdder 1
```

Appel de lambda

```
fn test = mySucc $ 20
```

```
fn (f) $ _ = f
```

Les expressions

Les vérifications de types

Équivalents

```
value >= 0
value is [0..]
```

```
value is string
value is int
value is list
value is float
value is true
value is bool
```

Les expressions

Les listes

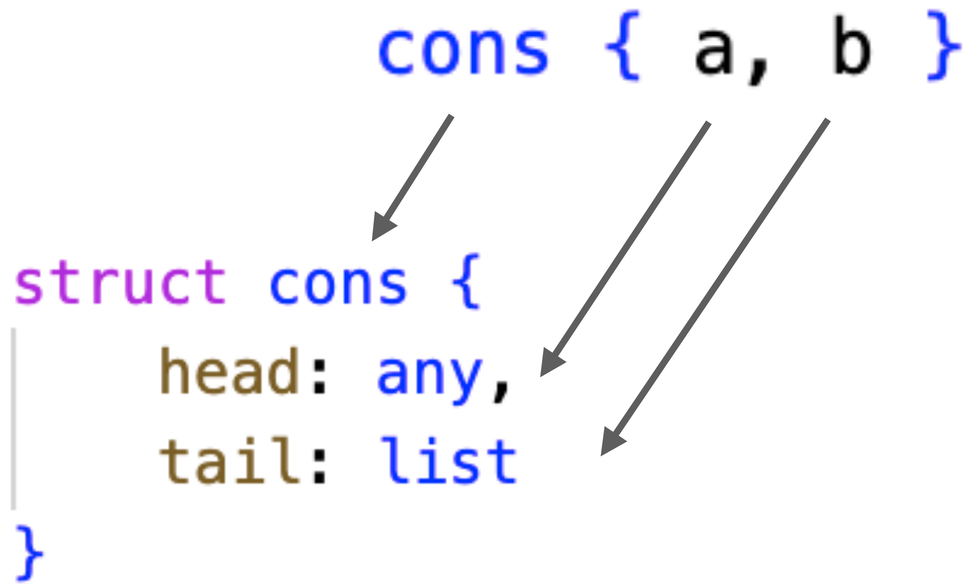
```
[1, 2, 3]
```

```
1 cons (2 cons (3 cons empty))
```

```
atom empty

struct cons {
    head: any,
    tail: list
}

type list = cons | empty

fn (a) cons (b) = cons { a, b }
```

Les expressions

Les constructeurs de structs



Les déclarations

Les déclarations

Les imports

```
import list
```

list.ek

```
atom empty

struct cons {
    head: any,
    tail: list
}

type list = cons | empty

fn (a) cons (b) = cons { a, b }

fn (l) drop (n) =
    if l is empty || n == 0 then
        l
    else
        l tail drop (n - 1)
```

Les déclarations

Les types

- Atomes
- Structs
- Type Alias

```
atom empty
```

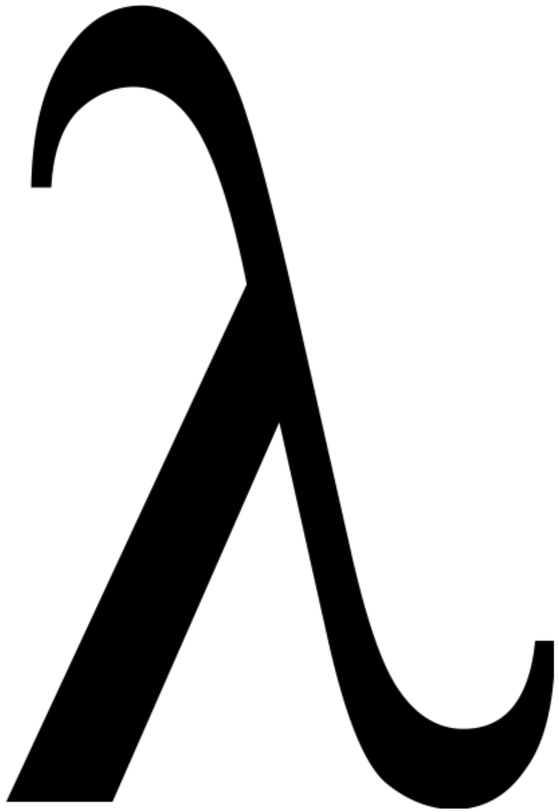
```
struct cons {
    head: any,
    tail: list
}
```

```
type list = cons | empty
```

```
fn (a) cons (b) = cons { a, b }

fn (l) drop (n) =
    if l is empty || n == 0 then
        l
    else
        l tail drop (n - 1)
```

Les déclarations

Les fonctions



```
atom empty

struct cons {
    head: any,
    tail: list
}

type list = cons | empty

fn (a) cons (b) = cons { a, b }

fn (l) drop (n) =
    if l is empty || n == 0 then
        l
    else
        l tail drop (n - 1)
```

Les déclarations

Les priorités des opérations

- Priorité personnalisable sur toutes les fonctions
- Nombre entre 0 et 10
- Par défaut à 9

```
fn (f) $ _ precedence 0 = f

fn _ + _ precedence 6 = builtin add
fn _ - _ precedence 6 = builtin sub
fn _ * _ precedence 7 = builtin mul
fn _ / _ precedence 7 = builtin div
```

```
fn _ == _ precedence 4 = builtin eq
fn (a) != (b) precedence 4 = not (a == b)
fn _ < _ precedence 4 = builtin lt
fn (a) > (b) precedence 4 = b < a
fn (a) <= (b) precedence 4 = not (a > b)
fn (a) >= (b) precedence 4 = not (a < b)
```

Les déclarations

Les fonctions lazy

```
fn if (cond) then (lazy truebr) else (lazy falsebr) precedence 1
```

- Permet de déferrer l'exécution d'un argument
- Utile si l'argument n'est utilisé que dans une branche

```
fn (a) && (lazy b) precedence 3 = if a then b else false
fn (a) || (lazy b) precedence 2 = if a then true else b
```

Pour finir

Une lib standard alternative

RPN

```
fn _ _ + = builtin add
fn _ _ - = builtin sub
fn _ _ * = builtin mul
fn _ _ / = builtin div
```

~~import std~~

import rpn

```
fn _ _ == = builtin eq
fn (a) not = a false ==
fn (a) (b) != = a b == not
fn _ _ < = builtin lt
fn (a) (b) > = b a <
fn (a) (b) <= = a b > not
fn (a) (b) >= = a b < not
```

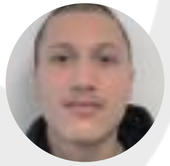
Merci de nous avoir écouté

Avez vous des questions ?

Emil Pedersen
emil.pedersen@epitech.eu

Jeremy Elalouf
jeremy.elalouf@epitech.eu

Lindon Aliu
lindon.aliu@epitech.eu



Marouan Bader
marouan.bader@epitech.eu

Yunse Lee
yunse1.lee@epitech.eu