

# Fonctions avancées

En résumé, nous avons étudié et réimplanté les fonctions suivantes du module `List` :

Fonction	Type
<code>map</code>	<code>('a -&gt; 'b) -&gt; 'a list -&gt; 'b list</code>
<code>filter</code>	<code>('a -&gt; bool) -&gt; 'a list -&gt; 'a list</code>
<code>for_all</code>	<code>('a -&gt; bool) -&gt; 'a list -&gt; bool</code>
<code>exists</code>	<code>('a -&gt; bool) -&gt; 'a list -&gt; bool</code>
<code>fold_left</code>	<code>('a -&gt; 'b -&gt; 'a) -&gt; 'a -&gt; 'b list -&gt; 'a</code>
<code>fold_right</code>	<code>('a -&gt; 'b -&gt; 'b) -&gt; 'a list -&gt; 'b -&gt; 'b</code>

# Fonctions avancées

En résumé, nous avons étudié et réimplanté les fonctions suivantes du module `List` :

Fonction	Type
<code>map</code>	<code>('a -&gt; 'b) -&gt; 'a list -&gt; 'b list</code>
<code>filter</code>	<code>('a -&gt; bool) -&gt; 'a list -&gt; 'a list</code>
<code>for_all</code>	<code>('a -&gt; bool) -&gt; 'a list -&gt; bool</code>
<code>exists</code>	<code>('a -&gt; bool) -&gt; 'a list -&gt; bool</code>
<code>fold_left</code>	<code>('a -&gt; 'b -&gt; 'a) -&gt; 'a -&gt; 'b list -&gt; 'a</code>
<code>fold_right</code>	<code>('a -&gt; 'b -&gt; 'b) -&gt; 'a list -&gt; 'b -&gt; 'b</code>

En pratique, on utilise ces fonctions sans les réimplanter.

# Fonctions avancées

En résumé, nous avons étudié et réimplanté les fonctions suivantes du module `List` :

Fonction	Type
<code>map</code>	<code>('a -&gt; 'b) -&gt; 'a list -&gt; 'b list</code>
<code>filter</code>	<code>('a -&gt; bool) -&gt; 'a list -&gt; 'a list</code>
<code>for_all</code>	<code>('a -&gt; bool) -&gt; 'a list -&gt; bool</code>
<code>exists</code>	<code>('a -&gt; bool) -&gt; 'a list -&gt; bool</code>
<code>fold_left</code>	<code>('a -&gt; 'b -&gt; 'a) -&gt; 'a -&gt; 'b list -&gt; 'a</code>
<code>fold_right</code>	<code>('a -&gt; 'b -&gt; 'b) -&gt; 'a list -&gt; 'b -&gt; 'b</code>

En pratique, on utilise ces fonctions sans les réimplanter.

Il existe aussi la fonction `sort` de type

```
('a -> 'a -> int) -> 'a list -> 'a list
```

qui permet de renvoyer une version triée d'une liste d'éléments de type `'a` au moyen d'une fonction de comparaison (1<sup>er</sup> paramètre).

# Plan

## Listes

Opérations

Non-mutabilité

Files

# Principe de non-mutabilité

Principalement en programmation fonctionnelle (mais pas uniquement), on travaille avec des données **non mutables** : une fois une donnée construite, il est **impossible de la modifier**.

# Principe de non-mutabilité

Principalement en programmation fonctionnelle (mais pas uniquement), on travaille avec des données **non mutables** : une fois une donnée construite, il est **impossible de la modifier**.

Corollaire à cela : étant donné un objet  $x$ , pour obtenir un objet  $x'$  calculé à partir de  $x$ , il faut **reconstruire**  $x'$ .

# Principe de non-mutabilité

Principalement en programmation fonctionnelle (mais pas uniquement), on travaille avec des données **non mutables** : une fois une donnée construite, il est **impossible de la modifier**.

Corollaire à cela : étant donné un objet  $x$ , pour obtenir un objet  $x'$  calculé à partir de  $x$ , il faut **reconstruire**  $x'$ .

La non-mutabilité des données présente beaucoup d'avantages :

1. écriture de programmes davantage facilitée et sécurisée;

# Principe de non-mutabilité

Principalement en programmation fonctionnelle (mais pas uniquement), on travaille avec des données **non mutables** : une fois une donnée construite, il est **impossible de la modifier**.

Corollaire à cela : étant donné un objet  $x$ , pour obtenir un objet  $x'$  calculé à partir de  $x$ , il faut **reconstruire**  $x'$ .

La non-mutabilité des données présente beaucoup d'avantages :

1. écriture de programmes davantage facilitée et sécurisée;
2. démonstration de correction de programmes facilitée;

# Principe de non-mutabilité

Principalement en programmation fonctionnelle (mais pas uniquement), on travaille avec des données **non mutables** : une fois une donnée construite, il est **impossible de la modifier**.

Corollaire à cela : étant donné un objet  $x$ , pour obtenir un objet  $x'$  calculé à partir de  $x$ , il faut **reconstruire**  $x'$ .

La non-mutabilité des données présente beaucoup d'avantages :

1. écriture de programmes davantage facilitée et sécurisée;
2. démonstration de correction de programmes facilitée;
3. gain de place mémoire.

# Principe de non-mutabilité

Principalement en programmation fonctionnelle (mais pas uniquement), on travaille avec des données **non mutables** : une fois une donnée construite, il est **impossible de la modifier**.

Corollaire à cela : étant donné un objet  $x$ , pour obtenir un objet  $x'$  calculé à partir de  $x$ , il faut **reconstruire**  $x'$ .

La non-mutabilité des données présente beaucoup d'avantages :

1. écriture de programmes davantage facilitée et sécurisée;
2. démonstration de correction de programmes facilitée;
3. gain de place mémoire.

Ces trois avantages s'appuient sur le fait que plusieurs grosses données peuvent **partager** des sous-données en commun, **sans aucune interférence**.

# Le cas des listes

Les listes sont des données non mutables. Toute « modification » d'une liste ne peut se faire que par une **construction d'une liste résultat**.

# Le cas des listes

Les listes sont des données non mutables. Toute « modification » d'une liste ne peut se faire que par une **construction d'une liste résultat**.

Considérons les phrases

```
# let lst1 = [1 ; 2 ; 3];;  
# let lst2 = 4 :: lst1;;
```

# Le cas des listes

Les listes sont des données non mutables. Toute « modification » d'une liste ne peut se faire que par une **construction d'une liste résultat**.

Considérons les phrases

```
# let lst1 = [1 ; 2 ; 3];;  
# let lst2 = 4 :: lst1;;
```

La 1<sup>re</sup> crée une liste (trois cellules) en mémoire :

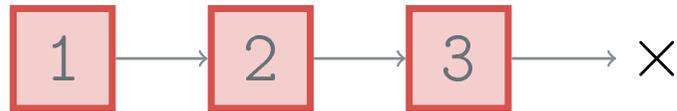
# Le cas des listes

Les listes sont des données non mutables. Toute « modification » d'une liste ne peut se faire que par une **construction d'une liste résultat**.

Considérons les phrases

```
# let lst1 = [1 ; 2 ; 3];;  
# let lst2 = 4 :: lst1;;
```

La 1<sup>re</sup> crée une liste (trois cellules) en mémoire :



# Le cas des listes

Les listes sont des données non mutables. Toute « modification » d'une liste ne peut se faire que par une **construction d'une liste résultat**.

Considérons les phrases

```
# let lst1 = [1 ; 2 ; 3];;  
# let lst2 = 4 :: lst1;;
```

La 1<sup>re</sup> crée une liste (trois cellules) en mémoire :



# Le cas des listes

Les listes sont des données non mutables. Toute « modification » d'une liste ne peut se faire que par une **construction d'une liste résultat**.

Considérons les phrases

```
# let lst1 = [1 ; 2 ; 3];;  
# let lst2 = 4 :: lst1;;
```

La 1<sup>re</sup> crée une liste (trois cellules) en mémoire :



La 2<sup>e</sup> ne crée qu'une seule cellule et **partage** les trois précédentes :

# Le cas des listes

Les listes sont des données non mutables. Toute « modification » d'une liste ne peut se faire que par une **construction d'une liste résultat**.

Considérons les phrases

```
# let lst1 = [1 ; 2 ; 3];;  
# let lst2 = 4 :: lst1;;
```

La 1<sup>re</sup> crée une liste (trois cellules) en mémoire :



La 2<sup>e</sup> ne crée qu'une seule cellule et **partage** les trois précédentes :



# Le cas des listes

Les listes sont des données non mutables. Toute « modification » d'une liste ne peut se faire que par une **construction d'une liste résultat**.

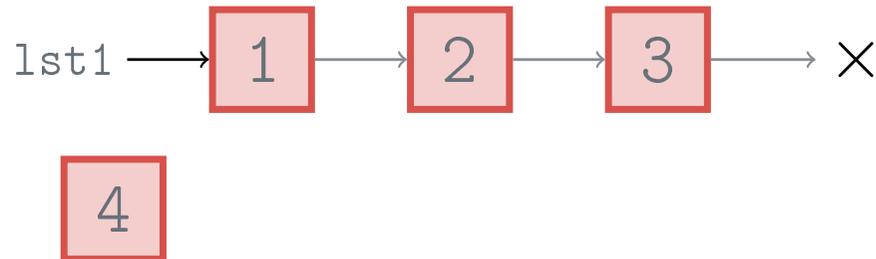
Considérons les phrases

```
# let lst1 = [1 ; 2 ; 3];;  
# let lst2 = 4 :: lst1;;
```

La 1<sup>re</sup> crée une liste (trois cellules) en mémoire :



La 2<sup>e</sup> ne crée qu'une seule cellule et **partage** les trois précédentes :



# Le cas des listes

Les listes sont des données non mutables. Toute « modification » d'une liste ne peut se faire que par une **construction d'une liste résultat**.

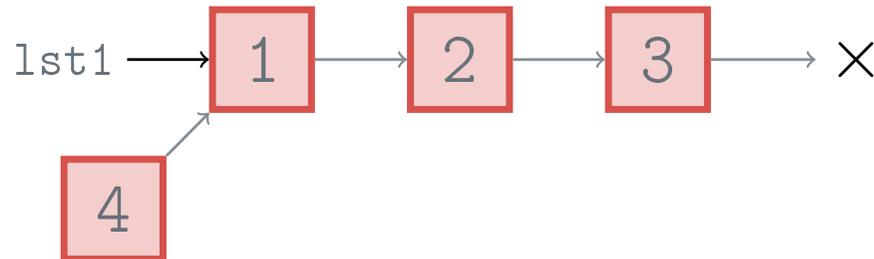
Considérons les phrases

```
# let lst1 = [1 ; 2 ; 3];;  
# let lst2 = 4 :: lst1;;
```

La 1<sup>re</sup> crée une liste (trois cellules) en mémoire :



La 2<sup>e</sup> ne crée qu'une seule cellule et **partage** les trois précédentes :



# Le cas des listes

Les listes sont des données non mutables. Toute « modification » d'une liste ne peut se faire que par une **construction d'une liste résultat**.

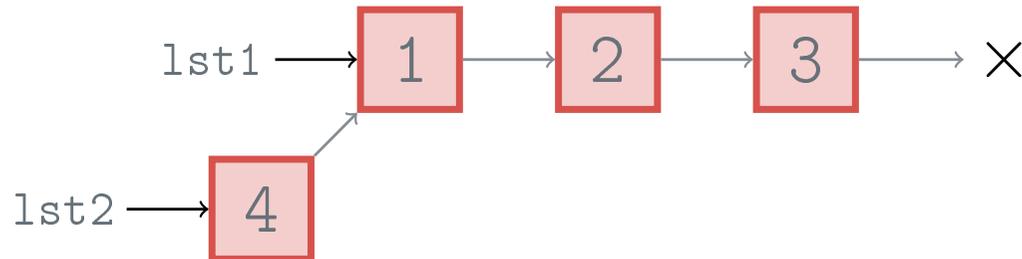
Considérons les phrases

```
# let lst1 = [1 ; 2 ; 3];;  
# let lst2 = 4 :: lst1;;
```

La 1<sup>re</sup> crée une liste (trois cellules) en mémoire :



La 2<sup>e</sup> ne crée qu'une seule cellule et **partage** les trois précédentes :



# Le cas des listes

Considérons la fonction

```
let rec concatener lst1 lst2 =  
  match lst1 with  
  | [] -> lst2  
  | e :: reste -> e :: (concatener reste lst2);;
```

# Le cas des listes

Considérons la fonction

```
let rec concatener lst1 lst2 =  
  match lst1 with  
  | [] -> lst2  
  | e :: reste -> e :: (concatener reste lst2);;
```

Elle permet de concaténer deux listes (fonction `append` du module `List`).

# Le cas des listes

Considérons la fonction

```
let rec concatener lst1 lst2 =  
  match lst1 with  
  | [] -> lst2  
  | e :: reste -> e :: (concatener reste lst2);;
```

Elle permet de concaténer deux listes (fonction `append` du module `List`).

Considérons l'effet des phrases suivantes :

# Le cas des listes

Considérons la fonction

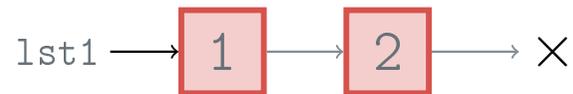
```
let rec concatener lst1 lst2 =  
  match lst1 with  
  | [] -> lst2  
  | e :: reste -> e :: (concatener reste lst2);;
```

Elle permet de concaténer deux listes (fonction `append` du module `List`).

Considérons l'effet des phrases suivantes :

```
# let lst1 = [1 ; 2];;
```

On obtient en mémoire la configuration de partage suivante :



# Le cas des listes

Considérons la fonction

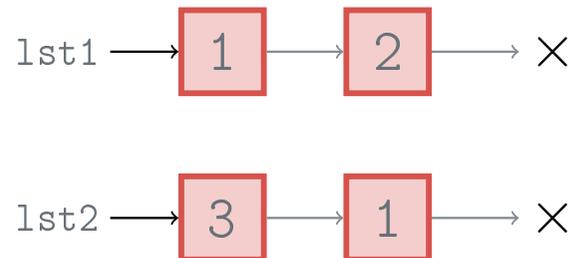
```
let rec concatener lst1 lst2 =  
  match lst1 with  
  | [] -> lst2  
  | e :: reste -> e :: (concatener reste lst2);;
```

Elle permet de concaténer deux listes (fonction `append` du module `List`).

Considérons l'effet des phrases suivantes :

```
# let lst1 = [1 ; 2];;  
# let lst2 = [3 ; 1];;
```

On obtient en mémoire la configuration de partage suivante :



# Le cas des listes

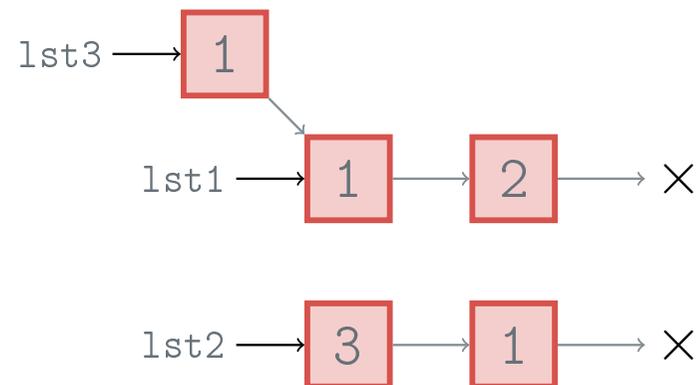
Considérons la fonction

```
let rec concatener lst1 lst2 =  
  match lst1 with  
  | [] -> lst2  
  | e :: reste -> e :: (concatener reste lst2);;
```

Elle permet de concaténer deux listes (fonction `append` du module `List`).

Considérons l'effet des phrases suivantes :      `# let lst3 = 1 :: lst1;;`  
`# let lst1 = [1 ; 2];;`  
`# let lst2 = [3 ; 1];;`

On obtient en mémoire la configuration de partage suivante :



# Le cas des listes

Considérons la fonction

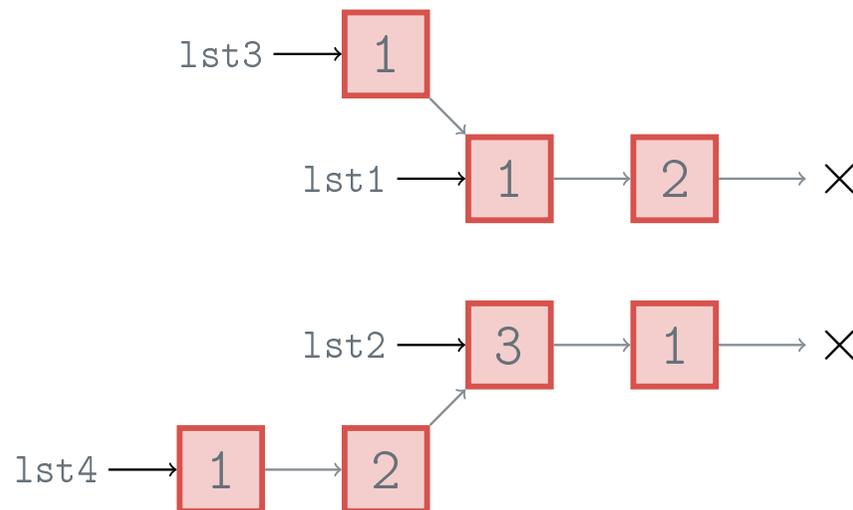
```
let rec concatener lst1 lst2 =  
  match lst1 with  
  | [] -> lst2  
  | e :: reste -> e :: (concatener reste lst2);;
```

Elle permet de concaténer deux listes (fonction `append` du module `List`).

Considérons l'effet des phrases suivantes :

```
# let lst1 = [1 ; 2];;  
# let lst2 = [3 ; 1];;  
# let lst3 = 1 :: lst1;;  
# let lst4 = (concatener lst1 lst2);;
```

On obtient en mémoire la configuration de partage suivante :



# Le cas des listes

Considérons la fonction

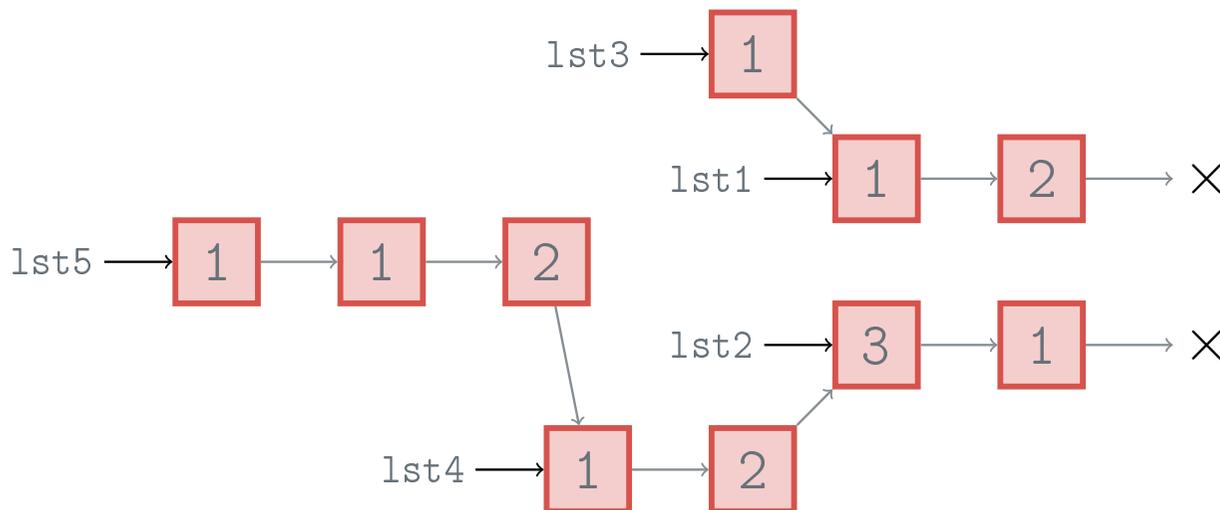
```
let rec concatener lst1 lst2 =  
  match lst1 with  
  | [] -> lst2  
  | e :: reste -> e :: (concatener reste lst2);;
```

Elle permet de concaténer deux listes (fonction `append` du module `List`).

Considérons l'effet des phrases suivantes :

```
# let lst3 = 1 :: lst1;;  
# let lst4 = (concatener lst1 lst2);;  
# let lst5 = (concatener lst3 lst4);;
```

On obtient en mémoire la configuration de partage suivante :



# Plan

## Listes

Opérations

Non-mutabilité

Files

# Files

On souhaite implanter les **files** (files First In, First Out) dont les éléments sont d'un type quelconque.

# Files

On souhaite implanter les **files** (files First In, First Out) dont les éléments sont d'un type quelconque. On doit pour cela

1. définir un type à un paramètre `file`;

# Files

On souhaite implanter les **files** (files First In, First Out) dont les éléments sont d'un type quelconque. On doit pour cela

1. définir un type à un paramètre `file`;
2. définir une constante

```
vide : 'a file
```

égale à la file vide;

# Files

On souhaite implanter les **files** (files First In, First Out) dont les éléments sont d'un type quelconque. On doit pour cela

1. définir un type à un paramètre `file`;
2. définir une constante

```
vide : 'a file
```

égale à la file vide;

3. définir une fonction

```
ancien : 'a file -> 'a
```

qui renvoie le plus ancien élément de la file;

# Files

On souhaite implanter les **files** (files First In, First Out) dont les éléments sont d'un type quelconque. On doit pour cela

1. définir un type à un paramètre `file`;
2. définir une constante

```
vide : 'a file
```

égale à la file vide;

3. définir une fonction

```
ancien : 'a file -> 'a
```

qui renvoie le plus ancien élément de la file;

4. définir une fonction

```
supprimer : 'a file -> 'a file
```

qui renvoie la file obtenue en supprimant son plus ancien élément;

# Files

On souhaite implanter les **files** (piles First In, First Out) dont les éléments sont d'un type quelconque. On doit pour cela

1. définir un type à un paramètre `file`;
2. définir une constante

```
vide : 'a file
```

égale à la file vide;

3. définir une fonction

```
ancien : 'a file -> 'a
```

qui renvoie le plus ancien élément de la file;

4. définir une fonction

```
supprimer : 'a file -> 'a file
```

qui renvoie la file obtenue en supprimant son plus ancien élément;

5. définir une fonction

```
ajouter : 'a file -> 'a -> 'a file
```

qui renvoie une nouvelle file prenant en compte de l'ajout d'un élément.

# Implantation directe

L'implantation directe consiste à représenter une file par une liste. Les éléments sont rangés du plus récent au plus ancien.

# Implantation directe

L'implantation directe consiste à représenter une file par une liste. Les éléments sont rangés du plus récent au plus ancien.

```
type 'a file = 'a list
```

# Implantation directe

L'implantation directe consiste à représenter une file par une liste. Les éléments sont rangés du plus récent au plus ancien.

```
type 'a file = 'a list
```

```
let vide = []
```

# Implantation directe

L'implantation directe consiste à représenter une file par une liste. Les éléments sont rangés du plus récent au plus ancien.

```
type 'a file = 'a list

let vide = []

let rec ancien f =
  match f with
  | [] -> (failwith "file vide")
  | [e] -> e
  | e :: reste -> (ancien reste)
```

# Implantation directe

L'implantation directe consiste à représenter une file par une liste. Les éléments sont rangés du plus récent au plus ancien.

```
type 'a file = 'a list
```

```
let vide = []
```

```
let rec ancien f =
```

```
  match f with
  | [] -> (failwith "file vide")
  | [e] -> e
  | e :: reste -> (ancien reste)
```

```
let rec supprimer f =
```

```
  match f with
  | [] -> (failwith "file vide")
  | [e] -> []
  | e :: reste -> e :: (supprimer reste)
```

# Implantation directe

L'implantation directe consiste à représenter une file par une liste. Les éléments sont rangés du plus récent au plus ancien.

```
type 'a file = 'a list
```

```
let vide = []
```

```
let rec ancien f =
```

```
  match f with
  | [] -> (failwith "file vide")
  | [e] -> e
  | e :: reste -> (ancien reste)
```

```
let rec supprimer f =
```

```
  match f with
  | [] -> (failwith "file vide")
  | [e] -> []
  | e :: reste -> e :: (supprimer reste)
```

```
let ajouter f x =
```

```
  x :: f
```

# Implantation directe

L'implantation directe consiste à représenter une file par une liste. Les éléments sont rangés du plus récent au plus ancien.

```
type 'a file = 'a list

let vide = []

let rec ancien f =
  match f with
  | [] -> (failwith "file vide")
  | [e] -> e
  | e :: reste -> (ancien reste)

let rec supprimer f =
  match f with
  | [] -> (failwith "file vide")
  | [e] -> []
  | e :: reste -> e :: (supprimer reste)

let ajouter f x =
  x :: f
```

En notant par  $n$  le nombre d'éléments de la file, on obtient les complexités

Fonction	Complexité en temps
ancien	$\Theta(n)$
supprimer	$\Theta(n)$
ajouter	$\Theta(1)$

# Implantation astucieuse

Il existe une implantation **respectant le principe de non-mutabilité** beaucoup plus performante.

Elle se base sur l'idée suivante.

# Implantation astucieuse

Il existe une implantation **respectant le principe de non-mutabilité** beaucoup plus performante.

Elle se base sur l'idée suivante. Une file est représentée par deux listes **in** et out.

# Implantation astucieuse

Il existe une implantation **respectant le principe de non-mutabilité** beaucoup plus performante.

Elle se base sur l'idée suivante. Une file est représentée par deux listes **in** et **out**.

- ▶ Les éléments prêts à sortir se situent dans **out**. Ils sont rangés du plus ancien au plus récent.

# Implantation astucieuse

Il existe une implantation **respectant le principe de non-mutabilité** beaucoup plus performante.

Elle se base sur l'idée suivante. Une file est représentée par deux listes **in** et **out**.

- ▶ Les éléments prêts à sortir se situent dans **out**. Ils sont rangés du plus ancien au plus récent.
- ▶ Les éléments ajoutés sont « mis en mémoire tampon » dans **in**. Ils sont rangés du plus récent au plus ancien.

# Implantation astucieuse

Il existe une implantation **respectant le principe de non-mutabilité** beaucoup plus performante.

Elle se base sur l'idée suivante. Une file est représentée par deux listes **in** et **out**.

- ▶ Les éléments prêts à sortir se situent dans **out**. Ils sont rangés du plus ancien au plus récent.
- ▶ Les éléments ajoutés sont « mis en mémoire tampon » dans **in**. Ils sont rangés du plus récent au plus ancien.

Les opérations sur cette structure de données se décrivent ainsi :

# Implantation astucieuse

Il existe une implantation **respectant le principe de non-mutabilité** beaucoup plus performante.

Elle se base sur l'idée suivante. Une file est représentée par deux listes **in** et **out**.

- ▶ Les éléments prêts à sortir se situent dans **out**. Ils sont rangés du plus ancien au plus récent.
- ▶ Les éléments ajoutés sont « mis en mémoire tampon » dans **in**. Ils sont rangés du plus récent au plus ancien.

Les opérations sur cette structure de données se décrivent ainsi :

- ▶ l'ajout d'un élément  $e$  à la file consiste à positionner  $e$  en tête de **in**;

# Implantation astucieuse

Il existe une implantation **respectant le principe de non-mutabilité** beaucoup plus performante.

Elle se base sur l'idée suivante. Une file est représentée par deux listes **in** et **out**.

- ▶ Les éléments prêts à sortir se situent dans **out**. Ils sont rangés du plus ancien au plus récent.
- ▶ Les éléments ajoutés sont « mis en mémoire tampon » dans **in**. Ils sont rangés du plus récent au plus ancien.

Les opérations sur cette structure de données se décrivent ainsi :

- ▶ l'ajout d'un élément  $e$  à la file consiste à positionner  $e$  en tête de **in** ;
- ▶ la suppression/renvoi du plus ancien élément consiste à supprimer/renvoyer la tête de **out** si elle est non vide.

# Implantation astucieuse

Il existe une implantation **respectant le principe de non-mutabilité** beaucoup plus performante.

Elle se base sur l'idée suivante. Une file est représentée par deux listes **in** et **out**.

- ▶ Les éléments prêts à sortir se situent dans **out**. Ils sont rangés du plus ancien au plus récent.
- ▶ Les éléments ajoutés sont « mis en mémoire tampon » dans **in**. Ils sont rangés du plus récent au plus ancien.

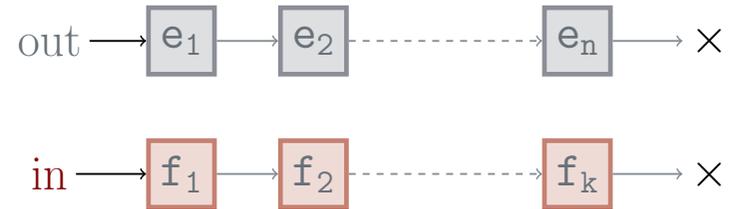
Les opérations sur cette structure de données se décrivent ainsi :

- ▶ l'ajout d'un élément  $e$  à la file consiste à positionner  $e$  en tête de **in** ;
- ▶ la suppression/renvoi du plus ancien élément consiste à supprimer/renvoyer la tête de **out** si elle est non vide.

Si **out** est vide, on remplace **out** par le **miroir** de **in**, on vide **in** et on supprime/renvoie la tête de **out**.

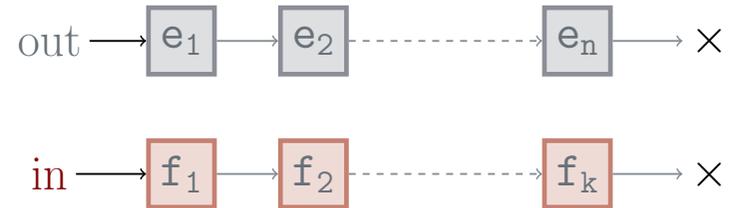
# Implantation astucieuse

Voici une file dans une situation générique :

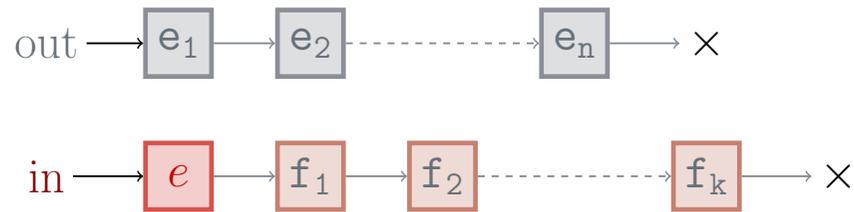


# Implantation astucieuse

Voici une file dans une situation générique :

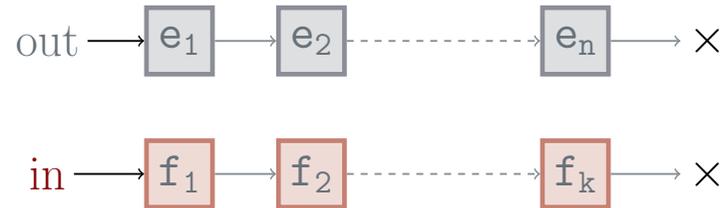


Après l'**ajout** d'un élément  $e$ , elle devient

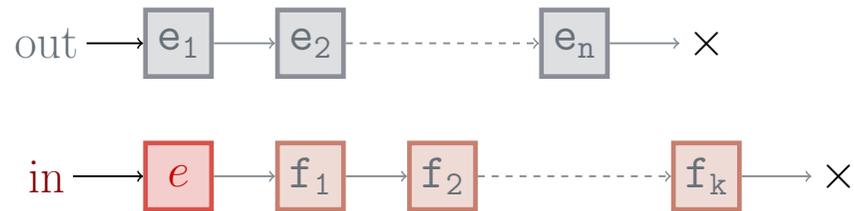


# Implantation astucieuse

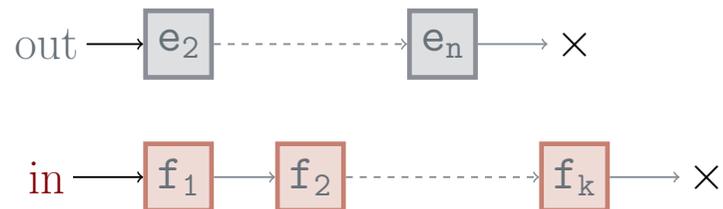
Voici une file dans une situation générique :



Après l'**ajout** d'un élément  $e$ , elle devient



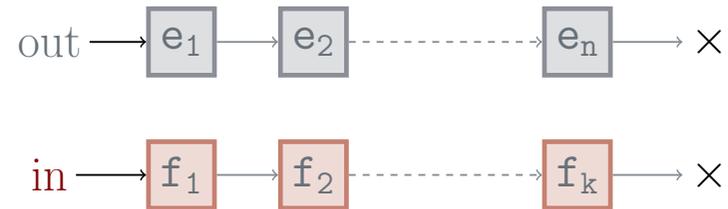
Après une **suppression** depuis la situation générique, elle devient



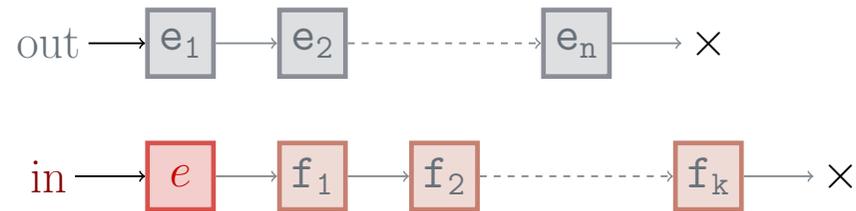
et renvoie  $e_1$  si out n'est pas vide,

# Implantation astucieuse

Voici une file dans une situation générique :



Après l'**ajout** d'un élément  $e$ , elle devient



Après une **suppression** depuis la situation générique, elle devient



et renvoie  $e_1$  si out n'est pas vide,

et renvoie  $f_k$  si out est vide.

# Implantation astucieuse – exemple

Opération	in	out
vide	[]	[]
ajout(1)		
ajout(1)		
ajout(2)		
ajout(3)		
supprimer()		
ajout(4)		
ajout(5)		
supprimer()		
supprimer()		
ajouter(6)		
supprimer()		
supprimer()		

# Implantation astucieuse – exemple

Opération	in	out
vide	[]	[]
ajout(1)	[1]	[]
ajout(1)		
ajout(2)		
ajout(3)		
supprimer()		
ajout(4)		
ajout(5)		
supprimer()		
supprimer()		
ajouter(6)		
supprimer()		
supprimer()		

# Implantation astucieuse – exemple

Opération	in	out
vide	[]	[]
ajout(1)	[1]	[]
ajout(1)	[1 ; 1]	[]
ajout(2)		
ajout(3)		
supprimer()		
ajout(4)		
ajout(5)		
supprimer()		
supprimer()		
ajouter(6)		
supprimer()		
supprimer()		

# Implantation astucieuse – exemple

Opération	in	out
vide	[]	[]
ajout(1)	[1]	[]
ajout(1)	[1 ; 1]	[]
ajout(2)	[2 ; 1 ; 1]	[]
ajout(3)		
supprimer()		
ajout(4)		
ajout(5)		
supprimer()		
supprimer()		
ajouter(6)		
supprimer()		
supprimer()		

# Implantation astucieuse – exemple

Opération	in	out
vide	[]	[]
ajout(1)	[1]	[]
ajout(1)	[1 ; 1]	[]
ajout(2)	[2 ; 1 ; 1]	[]
ajout(3)	[3 ; 2 ; 1 ; 1]	[]
supprimer()		
ajout(4)		
ajout(5)		
supprimer()		
supprimer()		
ajouter(6)		
supprimer()		
supprimer()		

# Implantation astucieuse – exemple

Opération	in	out
vide	[]	[]
ajout(1)	[1]	[]
ajout(1)	[1 ; 1]	[]
ajout(2)	[2 ; 1 ; 1]	[]
ajout(3)	[3 ; 2 ; 1 ; 1]	[]
supprimer()	[]	[1 ; 2 ; 3]
ajout(4)		
ajout(5)		
supprimer()		
supprimer()		
ajouter(6)		
supprimer()		
supprimer()		

# Implantation astucieuse – exemple

Opération	in	out
vide	[]	[]
ajout(1)	[1]	[]
ajout(1)	[1 ; 1]	[]
ajout(2)	[2 ; 1 ; 1]	[]
ajout(3)	[3 ; 2 ; 1 ; 1]	[]
supprimer()	[]	[1 ; 2 ; 3]
ajout(4)	[4]	[1 ; 2 ; 3]
ajout(5)		
supprimer()		
supprimer()		
ajouter(6)		
supprimer()		
supprimer()		

# Implantation astucieuse – exemple

Opération	in	out
vide	[]	[]
ajout(1)	[1]	[]
ajout(1)	[1 ; 1]	[]
ajout(2)	[2 ; 1 ; 1]	[]
ajout(3)	[3 ; 2 ; 1 ; 1]	[]
supprimer()	[]	[1 ; 2 ; 3]
ajout(4)	[4]	[1 ; 2 ; 3]
ajout(5)	[5 ; 4]	[1 ; 2 ; 3]
supprimer()		
supprimer()		
ajouter(6)		
supprimer()		
supprimer()		

# Implantation astucieuse – exemple

Opération	in	out
vide	[]	[]
ajout(1)	[1]	[]
ajout(1)	[1 ; 1]	[]
ajout(2)	[2 ; 1 ; 1]	[]
ajout(3)	[3 ; 2 ; 1 ; 1]	[]
supprimer()	[]	[1 ; 2 ; 3]
ajout(4)	[4]	[1 ; 2 ; 3]
ajout(5)	[5 ; 4]	[1 ; 2 ; 3]
supprimer()	[5 ; 4]	[2 ; 3]
supprimer()		
ajouter(6)		
supprimer()		
supprimer()		

# Implantation astucieuse – exemple

Opération	in	out
vide	[]	[]
ajout(1)	[1]	[]
ajout(1)	[1 ; 1]	[]
ajout(2)	[2 ; 1 ; 1]	[]
ajout(3)	[3 ; 2 ; 1 ; 1]	[]
supprimer()	[]	[1 ; 2 ; 3]
ajout(4)	[4]	[1 ; 2 ; 3]
ajout(5)	[5 ; 4]	[1 ; 2 ; 3]
supprimer()	[5 ; 4]	[2 ; 3]
supprimer()	[5 ; 4]	[3]
ajouter(6)		
supprimer()		
supprimer()		

# Implantation astucieuse – exemple

Opération	in	out
vide	[]	[]
ajout(1)	[1]	[]
ajout(1)	[1 ; 1]	[]
ajout(2)	[2 ; 1 ; 1]	[]
ajout(3)	[3 ; 2 ; 1 ; 1]	[]
supprimer()	[]	[1 ; 2 ; 3]
ajout(4)	[4]	[1 ; 2 ; 3]
ajout(5)	[5 ; 4]	[1 ; 2 ; 3]
supprimer()	[5 ; 4]	[2 ; 3]
supprimer()	[5 ; 4]	[3]
ajouter(6)	[6 ; 5 ; 4]	[3]
supprimer()		
supprimer()		

# Implantation astucieuse – exemple

Opération	in	out
vide	[]	[]
ajout(1)	[1]	[]
ajout(1)	[1 ; 1]	[]
ajout(2)	[2 ; 1 ; 1]	[]
ajout(3)	[3 ; 2 ; 1 ; 1]	[]
supprimer()	[]	[1 ; 2 ; 3]
ajout(4)	[4]	[1 ; 2 ; 3]
ajout(5)	[5 ; 4]	[1 ; 2 ; 3]
supprimer()	[5 ; 4]	[2 ; 3]
supprimer()	[5 ; 4]	[3]
ajouter(6)	[6 ; 5 ; 4]	[3]
supprimer()	[6 ; 5 ; 4]	[]
supprimer()		

# Implantation astucieuse – exemple

Opération	in	out
vide	[]	[]
ajout(1)	[1]	[]
ajout(1)	[1 ; 1]	[]
ajout(2)	[2 ; 1 ; 1]	[]
ajout(3)	[3 ; 2 ; 1 ; 1]	[]
supprimer()	[]	[1 ; 2 ; 3]
ajout(4)	[4]	[1 ; 2 ; 3]
ajout(5)	[5 ; 4]	[1 ; 2 ; 3]
supprimer()	[5 ; 4]	[2 ; 3]
supprimer()	[5 ; 4]	[3]
ajouter(6)	[6 ; 5 ; 4]	[3]
supprimer()	[6 ; 5 ; 4]	[]
supprimer()	[]	[5 ; 6]

# Implantation astucieuse

```
type 'a file =  
  {entree : 'a list ;  
   sortie : 'a list};;
```

# Implantation astucieuse

```
type 'a file =  
  {entree : 'a list ;  
    sortie : 'a list};;  
  
let vide =  
  {entree = [] ; sortie = []};;
```

# Implantation astucieuse

```
type 'a file =  
  {entree : 'a list ;  
   sortie : 'a list};;  
  
let vide =  
  {entree = [] ; sortie = []};;  
  
let ancien f =  
  match f.sortie with  
  | [] -> begin  
    match (List.rev f.entree) with  
    | [] -> (failwith  
             "file vide")  
    | e :: _ -> e  
  end  
  | e :: _ -> e;;
```

# Implantation astucieuse

```
type 'a file =
  {entree : 'a list ;
   sortie : 'a list};;

let vide =
  {entree = [] ; sortie = []};;

let ancien f =
  match f.sortie with
  | [] -> begin
      match (List.rev f.entree) with
      | [] -> (failwith "file vide")
      | e :: _ -> e
    end
  | e :: _ -> e;;

let supprimer f =
  match f.sortie with
  | [] -> begin
      match (List.rev f.entree) with
      | [] -> (failwith "file vide")
      | _ :: reste ->
          {entree = [] ;
           sortie = reste}
    end
  | _ :: reste ->
      {f with sortie = reste};;
```

# Implantation astucieuse

```
type 'a file =
  {entree : 'a list ;
   sortie : 'a list};;

let vide =
  {entree = [] ; sortie = []};;

let ancien f =
  match f.sortie with
  | [] -> begin
      match (List.rev f.entree) with
      | [] -> (failwith "file vide")
      | e :: _ -> e
    end
  | e :: _ -> e;;

let supprimer f =
  match f.sortie with
  | [] -> begin
      match (List.rev f.entree) with
      | [] -> (failwith "file vide")
      | _ :: reste ->
          {entree = [] ;
           sortie = reste}
    end
  | _ :: reste ->
      {f with sortie = reste};;

let ajouter f x =
  {f with entree = x :: f.entree};;
```

# Implantation astucieuse

```
type 'a file =
  {entree : 'a list ;
   sortie : 'a list};;

let vide =
  {entree = [] ; sortie = []};;

let ancien f =
  match f.sortie with
  | [] -> begin
      match (List.rev f.entree) with
      | [] -> (failwith "file vide")
      | e :: _ -> e
    end
  | e :: _ -> e;;

let supprimer f =
  match f.sortie with
  | [] -> begin
      match (List.rev f.entree) with
      | [] -> (failwith "file vide")
      | _ :: reste ->
          {entree = [] ;
           sortie = reste}
    end
  | _ :: reste ->
      {f with sortie = reste};;

let ajouter f x =
  {f with entree = x :: f.entree};;
```

Cette implantation est plus efficace que la précédente.

# Implantation astucieuse

```
type 'a file =
  {entree : 'a list ;
   sortie : 'a list};;

let vide =
  {entree = [] ; sortie = []};;

let ancien f =
  match f.sortie with
  | [] -> begin
      match (List.rev f.entree) with
      | [] -> (failwith "file vide")
      | e :: _ -> e
    end
  | e :: _ -> e;;

let supprimer f =
  match f.sortie with
  | [] -> begin
      match (List.rev f.entree) with
      | [] -> (failwith "file vide")
      | _ :: reste ->
          {entree = [] ;
           sortie = reste}
    end
  | _ :: reste ->
      {f with sortie = reste};;

let ajouter f x =
  {f with entree = x :: f.entree};;
```

Cette implantation est plus efficace que la précédente.

Elle est même strictement plus efficace : les trois opérations ont une complexité en temps en  $\Theta(1)$  sauf lorsque `out` est vide, ce qui demande pour `ancien` et `supprimer` une mise à jour en  $\Theta(n)$ .

# Plan

## $\lambda$ -calcul

$\lambda$ -termes

Codage

Implantation

# Généralités

Le  $\lambda$ -calcul a été introduit par Church en 1936.

# Généralités

Le  $\lambda$ -calcul a été introduit par Church en 1936.

L'objectif de Church était de fournir une **formalisation alternative** des mathématiques fondée sur la notion de fonction (et non plus sur celle d'ensemble, propre à la théorie de Zermelo-Fraenkel).

# Généralités

Le  $\lambda$ -calcul a été introduit par Church en 1936.

L'objectif de Church était de fournir une **formalisation alternative** des mathématiques fondée sur la notion de fonction (et non plus sur celle d'ensemble, propre à la théorie de Zermelo-Fraenkel).

Ceci a échoué car ce que Church parvint à découvrir, le  $\lambda$ -calcul, n'a pas un pouvoir d'expression suffisant.

# Généralités

Le  $\lambda$ -calcul a été introduit par Church en 1936.

L'objectif de Church était de fournir une **formalisation alternative** des mathématiques fondée sur la notion de fonction (et non plus sur celle d'ensemble, propre à la théorie de Zermelo-Fraenkel).

Ceci a échoué car ce que Church parvint à découvrir, le  $\lambda$ -calcul, n'a pas un pouvoir d'expression suffisant.

En revanche, le  $\lambda$ -calcul a le même pouvoir d'expression que les **machines de Turing**.

# Généralités

Le  $\lambda$ -calcul a été introduit par Church en 1936.

L'objectif de Church était de fournir une **formalisation alternative** des mathématiques fondée sur la notion de fonction (et non plus sur celle d'ensemble, propre à la théorie de Zermelo-Fraenkel).

Ceci a échoué car ce que Church parvint à découvrir, le  $\lambda$ -calcul, n'a pas un pouvoir d'expression suffisant.

En revanche, le  $\lambda$ -calcul a le même pouvoir d'expression que les **machines de Turing**.

Il offre ainsi un formalisme pour exprimer tout ce qui est calculable.

# Généralités

Le  $\lambda$ -calcul a été introduit par Church en 1936.

L'objectif de Church était de fournir une **formalisation alternative** des mathématiques fondée sur la notion de fonction (et non plus sur celle d'ensemble, propre à la théorie de Zermelo-Fraenkel).

Ceci a échoué car ce que Church parvint à découvrir, le  $\lambda$ -calcul, n'a pas un pouvoir d'expression suffisant.

En revanche, le  $\lambda$ -calcul a le même pouvoir d'expression que les **machines de Turing**.

Il offre ainsi un formalisme pour exprimer tout ce qui est calculable.

Le  $\lambda$ -calcul constitue le cœur de tous les **langages de programmation fonctionnels**.

# Plan

$\lambda$ -calcul

$\lambda$ -termes

Codage

Implantation

# $\lambda$ -termes

Un  $\lambda$ -terme est

# $\lambda$ -termes

Un  $\lambda$ -terme est

1. une **variable**  $x, y, z, t, \text{etc.}$ ,

# $\lambda$ -termes

Un  $\lambda$ -terme est

1. une **variable**  $x, y, z, t, \text{ etc.}$ , ou bien
2. une **abstraction**  $\lambda x.t$  où  $x$  est une variable et  $t$  est un  $\lambda$ -terme,

# $\lambda$ -termes

Un  $\lambda$ -terme est

1. une **variable**  $x, y, z, t, \text{ etc.}$ , ou bien
2. une **abstraction**  $\lambda x.t$  où  $x$  est une variable et  $t$  est un  $\lambda$ -terme, ou bien
3. une **application**  $st$  où  $s$  et  $t$  sont des  $\lambda$ -termes.

# $\lambda$ -termes

Un  $\lambda$ -terme est

1. une **variable**  $x, y, z, t, \text{ etc.}$ , ou bien
2. une **abstraction**  $\lambda x.t$  où  $x$  est une variable et  $t$  est un  $\lambda$ -terme, ou bien
3. une **application**  $st$  où  $s$  et  $t$  sont des  $\lambda$ -termes.

**Remarque** : il s'agit d'une définition récursive.

# $\lambda$ -termes

Un  $\lambda$ -terme est

1. une **variable**  $x, y, z, t, \text{etc.}$ , ou bien
2. une **abstraction**  $\lambda x.t$  où  $x$  est une variable et  $t$  est un  $\lambda$ -terme, ou bien
3. une **application**  $st$  où  $s$  et  $t$  sont des  $\lambda$ -termes.

**Remarque** : il s'agit d'une définition récursive.

P.ex.,

$$(\lambda x.((x y) x)) (\lambda y.\lambda x.y)$$

est un  $\lambda$ -terme.

# $\lambda$ -termes

Un  $\lambda$ -terme est

1. une **variable**  $x, y, z, t, \text{etc.}$ , ou bien
2. une **abstraction**  $\lambda x.t$  où  $x$  est une variable et  $t$  est un  $\lambda$ -terme, ou bien
3. une **application**  $st$  où  $s$  et  $t$  sont des  $\lambda$ -termes.

**Remarque** : il s'agit d'une définition récursive.

P.ex.,

$$(\lambda x.((x y) x)) (\lambda y.\lambda x.y)$$

est un  $\lambda$ -terme.

Attention à l'emploi de parenthèses pour éviter les ambiguïtés. Il existe diverses conventions pour réduire leur nombre (que nous n'emploierons pas ici).

# Arbres syntaxiques

Tout  $\lambda$ -terme  $t$  peut se représenter par son **arbre syntaxique** de la manière suivante :

# Arbres syntaxiques

Tout  $\lambda$ -terme  $t$  peut se représenter par son **arbre syntaxique** de la manière suivante :

1. si  $t$  est une variable  $x$ , l'arbre syntaxique de  $t$  est la feuille étiquetée par  $x$ ;

# Arbres syntaxiques

Tout  $\lambda$ -terme  $t$  peut se représenter par son **arbre syntaxique** de la manière suivante :

1. si  $t$  est une variable  $x$ , l'arbre syntaxique de  $t$  est la feuille étiquetée par  $x$  ;
2. si  $t$  est une abstraction  $\lambda x.t'$ , l'arbre syntaxique de  $t$  est un nœud unaire étiqueté par  $\lambda x$  qui possède comme fils l'arbre syntaxique de  $t'$  ;

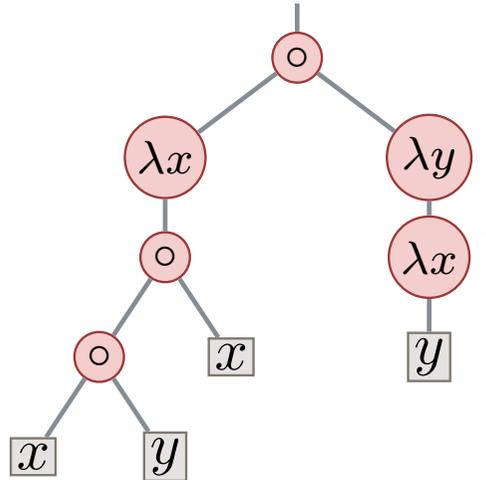
# Arbres syntaxiques

Tout  $\lambda$ -terme  $t$  peut se représenter par son **arbre syntaxique** de la manière suivante :

1. si  $t$  est une variable  $x$ , l'arbre syntaxique de  $t$  est la feuille étiquetée par  $x$  ;
2. si  $t$  est une abstraction  $\lambda x.t'$ , l'arbre syntaxique de  $t$  est un nœud unaire étiqueté par  $\lambda x$  qui possède comme fils l'arbre syntaxique de  $t'$  ;
3. si  $t$  est une application  $t' t''$ , l'arbre syntaxique de  $t$  est un nœud binaire étiqueté par  $\circ$  qui possède comme fils gauche l'arbre syntaxique de  $t'$  et comme fils droit l'arbre syntaxique de  $t''$ .

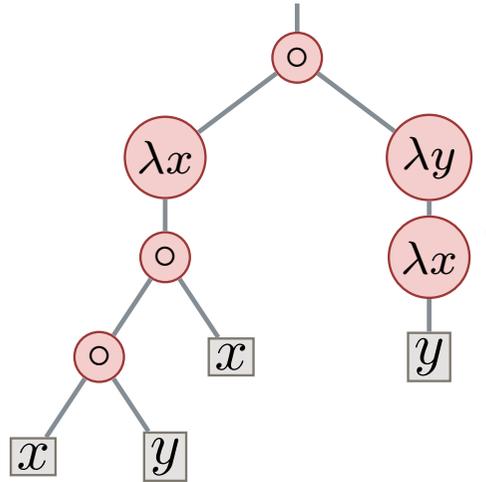
# Arbres syntaxiques

L'arbre syntaxique de  $(\lambda x.((x y) x)) (\lambda y.\lambda x.y)$  est

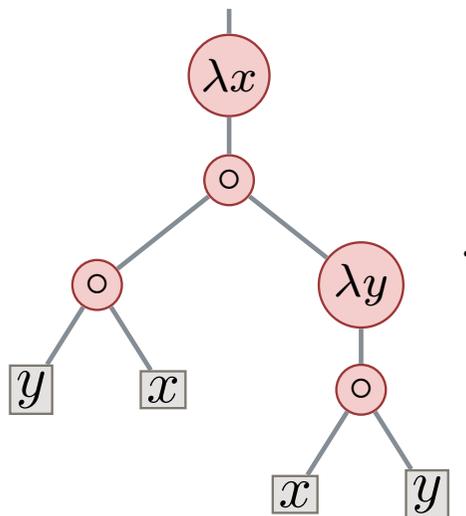


# Arbres syntaxiques

L'arbre syntaxique de  $(\lambda x.((x y) x)) (\lambda y.\lambda x.y)$  est



L'arbre syntaxique de  $\lambda x.((y x) (\lambda y.(x y)))$  est



# Variables libres/liées

Soit  $t$  un  $\lambda$ -terme et une occurrence d'une variable  $x$   $y$  apparaissant.

# Variables libres/liées

Soit  $t$  un  $\lambda$ -terme et une occurrence d'une variable  $x$   $y$  apparaissant.

Cette **occurrence** correspond à une **feuille**  $f$  de l'arbre syntaxique  $a$  de  $t$ .

# Variables libres/liées

Soit  $t$  un  $\lambda$ -terme et une occurrence d'une variable  $x$  y apparaissant.

Cette **occurrence** correspond à une **feuille**  $f$  de l'arbre syntaxique  $a$  de  $t$ .

Pour décider si cette occurrence de  $x$  dans  $t$  est libre ou liée, on considère le processus qui consiste à remonter depuis  $f$  vers la racine de  $a$  de sorte que

# Variables libres/liées

Soit  $t$  un  $\lambda$ -terme et une occurrence d'une variable  $x$  y apparaissant.

Cette **occurrence** correspond à une **feuille**  $f$  de l'arbre syntaxique  $a$  de  $t$ .

Pour décider si cette occurrence de  $x$  dans  $t$  est libre ou liée, on considère le processus qui consiste à remonter depuis  $f$  vers la racine de  $a$  de sorte que

- ▶ si le nœud visité est étiqueté par  $\lambda x$ , on s'arrête et on relie  $f$  à ce nœud;

# Variables libres/liées

Soit  $t$  un  $\lambda$ -terme et une occurrence d'une variable  $x$  y apparaissant.

Cette **occurrence** correspond à une **feuille**  $f$  de l'arbre syntaxique  $a$  de  $t$ .

Pour décider si cette occurrence de  $x$  dans  $t$  est libre ou liée, on considère le processus qui consiste à remonter depuis  $f$  vers la racine de  $a$  de sorte que

- ▶ si le nœud visité est étiqueté par  $\lambda x$ , on s'arrête et on relie  $f$  à ce nœud;
- ▶ sinon, et si ce nœud n'est pas la racine de  $a$ , on réitère ce processus depuis son parent;

# Variables libres/liées

Soit  $t$  un  $\lambda$ -terme et une occurrence d'une variable  $x$   $y$  apparaissant.

Cette **occurrence** correspond à une **feuille**  $f$  de l'arbre syntaxique  $a$  de  $t$ .

Pour décider si cette occurrence de  $x$  dans  $t$  est libre ou liée, on considère le processus qui consiste à remonter depuis  $f$  vers la racine de  $a$  de sorte que

- ▶ si le nœud visité est étiqueté par  $\lambda x$ , on s'arrête et on relie  $f$  à ce nœud;
- ▶ sinon, et si ce nœud n'est pas la racine de  $a$ , on réitère ce processus depuis son parent;
- ▶ sinon, on s'arrête.

# Variables libres/liées

Soit  $t$  un  $\lambda$ -terme et une occurrence d'une variable  $x$   $y$  apparaissant.

Cette **occurrence** correspond à une **feuille**  $f$  de l'arbre syntaxique  $a$  de  $t$ .

Pour décider si cette occurrence de  $x$  dans  $t$  est libre ou liée, on considère le processus qui consiste à remonter depuis  $f$  vers la racine de  $a$  de sorte que

- ▶ si le nœud visité est étiqueté par  $\lambda x$ , on s'arrête et on relie  $f$  à ce nœud;
- ▶ sinon, et si ce nœud n'est pas la racine de  $a$ , on réitère ce processus depuis son parent;
- ▶ sinon, on s'arrête.

Finalement, si  $f$  est liée à un nœud, on dit que l'occurrence de  $x$  est **liée** (ou encore **capturée** par le  $\lambda x$  cible); sinon, on dit qu'elle est **libre**.

# Variables libres/liées

Soit le  $\lambda$ -terme

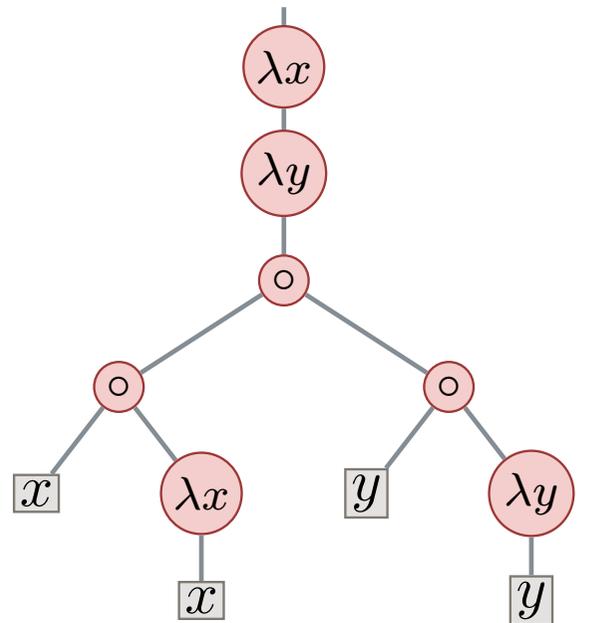
$$\lambda x. \lambda x. ((x (\lambda x. x)) (y (\lambda y. y))).$$

# Variables libres/liées

Soit le  $\lambda$ -terme

$$\lambda x. \lambda x. ((x (\lambda x. x)) (y (\lambda y. y))).$$

Son arbre syntaxique est

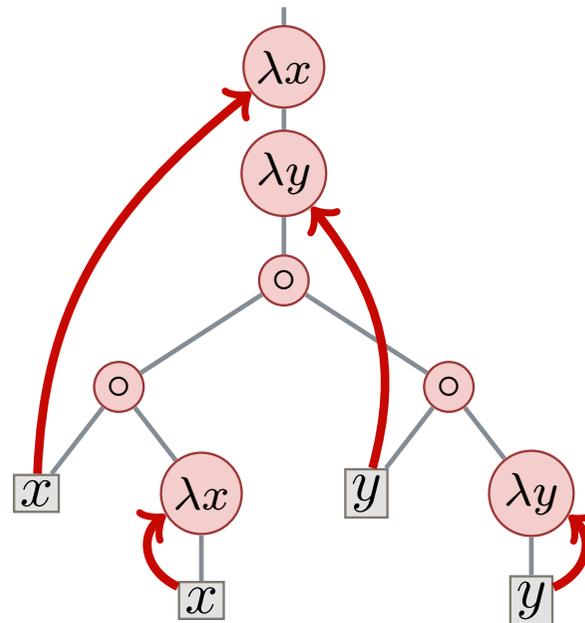


# Variables libres/liées

Soit le  $\lambda$ -terme

$$\lambda x. \lambda x. ((x (\lambda x. x)) (y (\lambda y. y))).$$

Son arbre syntaxique est



Les flèches connectent chaque occurrence de variable à son abstraction qui la capture.

# Substitutions

Soit  $t$  un  $\lambda$ -terme,  $x$  une variable et  $s$  un autre  $\lambda$ -terme.

# Substitutions

Soit  $t$  un  $\lambda$ -terme,  $x$  une variable et  $s$  un autre  $\lambda$ -terme.

La **substitution** de  $s$  à  $x$  dans  $t$  consiste à remplacer chaque **occurrence libre** de  $x$  dans  $t$  par  $s$ . On note  $t \star_x s$  le  $\lambda$ -terme ainsi obtenu.

# Substitutions

Soit  $t$  un  $\lambda$ -terme,  $x$  une variable et  $s$  un autre  $\lambda$ -terme.

La **substitution** de  $s$  à  $x$  dans  $t$  consiste à remplacer chaque **occurrence libre** de  $x$  dans  $t$  par  $s$ . On note  $t \star_x s$  le  $\lambda$ -terme ainsi obtenu.

P.ex.,

$$\blacktriangleright \lambda z.(x y) \star_x z z$$

# Substitutions

Soit  $t$  un  $\lambda$ -terme,  $x$  une variable et  $s$  un autre  $\lambda$ -terme.

La **substitution** de  $s$  à  $x$  dans  $t$  consiste à remplacer chaque **occurrence libre** de  $x$  dans  $t$  par  $s$ . On note  $t \star_x s$  le  $\lambda$ -terme ainsi obtenu.

P.ex.,

$$\blacktriangleright \lambda z.(x y) \star_x z z = \lambda z.((z z) y),$$

# Substitutions

Soit  $t$  un  $\lambda$ -terme,  $x$  une variable et  $s$  un autre  $\lambda$ -terme.

La **substitution** de  $s$  à  $x$  dans  $t$  consiste à remplacer chaque **occurrence libre** de  $x$  dans  $t$  par  $s$ . On note  $t \star_x s$  le  $\lambda$ -terme ainsi obtenu.

P.ex.,

$$\blacktriangleright \lambda z.(x y) \star_x z z = \lambda z.((z z) y),$$

$$\blacktriangleright \lambda x.(x y) \star_x z z$$

# Substitutions

Soit  $t$  un  $\lambda$ -terme,  $x$  une variable et  $s$  un autre  $\lambda$ -terme.

La **substitution** de  $s$  à  $x$  dans  $t$  consiste à remplacer chaque **occurrence libre** de  $x$  dans  $t$  par  $s$ . On note  $t \star_x s$  le  $\lambda$ -terme ainsi obtenu.

P.ex.,

$$\blacktriangleright \lambda z.(x y) \star_x z z = \lambda z.((z z) y),$$

$$\blacktriangleright \lambda x.(x y) \star_x z z = \lambda x.(x y),$$

# Substitutions

Soit  $t$  un  $\lambda$ -terme,  $x$  une variable et  $s$  un autre  $\lambda$ -terme.

La **substitution** de  $s$  à  $x$  dans  $t$  consiste à remplacer chaque **occurrence libre** de  $x$  dans  $t$  par  $s$ . On note  $t \star_x s$  le  $\lambda$ -terme ainsi obtenu.

P.ex.,

$$\blacktriangleright \lambda z.(x y) \star_x z z = \lambda z.((z z) y),$$

$$\blacktriangleright \lambda x.(x y) \star_x z z = \lambda x.(x y),$$

$$\blacktriangleright (z (\lambda z.(y z))) z \star_z \lambda x.(x x)$$

# Substitutions

Soit  $t$  un  $\lambda$ -terme,  $x$  une variable et  $s$  un autre  $\lambda$ -terme.

La **substitution** de  $s$  à  $x$  dans  $t$  consiste à remplacer chaque **occurrence libre** de  $x$  dans  $t$  par  $s$ . On note  $t \star_x s$  le  $\lambda$ -terme ainsi obtenu.

P.ex.,

$$\blacktriangleright \lambda z.(x y) \star_x z z = \lambda z.((z z) y),$$

$$\blacktriangleright \lambda x.(x y) \star_x z z = \lambda x.(x y),$$

$$\blacktriangleright (z (\lambda z.(y z))) z \star_z \lambda x.(x x) = ((\lambda x.(x x)) (\lambda z.(y z))) (\lambda x.(x x)).$$

## $\alpha$ -renommages et $\alpha$ -équivalence

Soient  $t$  un  $\lambda$ -terme,  $x$  une variable et  $y$  une variable qui n'admet aucune occurrence dans  $t$ .

## $\alpha$ -renommages et $\alpha$ -équivalence

Soient  $t$  un  $\lambda$ -terme,  $x$  une variable et  $y$  une variable qui n'admet aucune occurrence dans  $t$ .

L' $\alpha$ -renommage de  $x$  en  $y$  dans  $t$  consiste à remplacer dans  $t$  chaque  $\lambda x$  par  $\lambda y$  et chaque occurrence de  $x$  liée à un  $\lambda x$  par  $y$ .

## $\alpha$ -renommages et $\alpha$ -équivalence

Soient  $t$  un  $\lambda$ -terme,  $x$  une variable et  $y$  une variable qui n'admet aucune occurrence dans  $t$ .

L' $\alpha$ -renommage de  $x$  en  $y$  dans  $t$  consiste à remplacer dans  $t$  chaque  $\lambda x$  par  $\lambda y$  et chaque occurrence de  $x$  liée à un  $\lambda x$  par  $y$ .

P.ex.,

- ▶ l' $\alpha$ -renommage de  $x$  en  $y$  du  $\lambda$ -terme  $\lambda x.x$  est  $\lambda y.y$ ;

## $\alpha$ -renommages et $\alpha$ -équivalence

Soient  $t$  un  $\lambda$ -terme,  $x$  une variable et  $y$  une variable qui n'admet aucune occurrence dans  $t$ .

L' $\alpha$ -renommage de  $x$  en  $y$  dans  $t$  consiste à remplacer dans  $t$  chaque  $\lambda x$  par  $\lambda y$  et chaque occurrence de  $x$  liée à un  $\lambda x$  par  $y$ .

P.ex.,

- ▶ l' $\alpha$ -renommage de  $x$  en  $y$  du  $\lambda$ -terme  $\lambda x.x$  est  $\lambda y.y$ ;
- ▶ l' $\alpha$ -renommage de  $x$  en  $y$  du  $\lambda$ -terme  $(x x) (\lambda x.(x z))$  est  $(x x) (\lambda y.(y z))$ .

## $\alpha$ -renommages et $\alpha$ -équivalence

Soient  $t$  un  $\lambda$ -terme,  $x$  une variable et  $y$  une variable qui n'admet aucune occurrence dans  $t$ .

L' $\alpha$ -renommage de  $x$  en  $y$  dans  $t$  consiste à remplacer dans  $t$  chaque  $\lambda x$  par  $\lambda y$  et chaque occurrence de  $x$  liée à un  $\lambda x$  par  $y$ .

P.ex.,

- ▶ l' $\alpha$ -renommage de  $x$  en  $y$  du  $\lambda$ -terme  $\lambda x.x$  est  $\lambda y.y$ ;
- ▶ l' $\alpha$ -renommage de  $x$  en  $y$  du  $\lambda$ -terme  $(x x) (\lambda x.(x z))$  est  $(x x) (\lambda y.(y z))$ .

Deux  $\lambda$ -termes  $t$  et  $t'$  sont  $\alpha$ -équivalents s'il est possible de transformer  $t$  en  $t'$  par une suite d' $\alpha$ -renommages.

## $\alpha$ -renommages et $\alpha$ -équivalence

Soient  $t$  un  $\lambda$ -terme,  $x$  une variable et  $y$  une variable qui n'admet aucune occurrence dans  $t$ .

L' $\alpha$ -renommage de  $x$  en  $y$  dans  $t$  consiste à remplacer dans  $t$  chaque  $\lambda x$  par  $\lambda y$  et chaque occurrence de  $x$  liée à un  $\lambda x$  par  $y$ .

P.ex.,

- ▶ l' $\alpha$ -renommage de  $x$  en  $y$  du  $\lambda$ -terme  $\lambda x.x$  est  $\lambda y.y$ ;
- ▶ l' $\alpha$ -renommage de  $x$  en  $y$  du  $\lambda$ -terme  $(x x) (\lambda x.(x z))$  est  $(x x) (\lambda y.(y z))$ .

Deux  $\lambda$ -termes  $t$  et  $t'$  sont  $\alpha$ -équivalents s'il est possible de transformer  $t$  en  $t'$  par une suite d' $\alpha$ -renommages.

P.ex.,

- ▶  $\lambda x.x$  et  $\lambda y.y$  sont  $\alpha$ -équivalents;

## $\alpha$ -renommages et $\alpha$ -équivalence

Soient  $t$  un  $\lambda$ -terme,  $x$  une variable et  $y$  une variable qui n'admet aucune occurrence dans  $t$ .

L' $\alpha$ -renommage de  $x$  en  $y$  dans  $t$  consiste à remplacer dans  $t$  chaque  $\lambda x$  par  $\lambda y$  et chaque occurrence de  $x$  liée à un  $\lambda x$  par  $y$ .

P.ex.,

- ▶ l' $\alpha$ -renommage de  $x$  en  $y$  du  $\lambda$ -terme  $\lambda x.x$  est  $\lambda y.y$ ;
- ▶ l' $\alpha$ -renommage de  $x$  en  $y$  du  $\lambda$ -terme  $(x x) (\lambda x.(x z))$  est  $(x x) (\lambda y.(y z))$ .

Deux  $\lambda$ -termes  $t$  et  $t'$  sont  $\alpha$ -équivalents s'il est possible de transformer  $t$  en  $t'$  par une suite d' $\alpha$ -renommages.

P.ex.,

- ▶  $\lambda x.x$  et  $\lambda y.y$  sont  $\alpha$ -équivalents;
- ▶  $\lambda x.\lambda y.(x y)$  et  $\lambda y.\lambda x.(y x)$  sont  $\alpha$ -équivalents;

## $\alpha$ -renommages et $\alpha$ -équivalence

Soient  $t$  un  $\lambda$ -terme,  $x$  une variable et  $y$  une variable qui n'admet aucune occurrence dans  $t$ .

L' $\alpha$ -renommage de  $x$  en  $y$  dans  $t$  consiste à remplacer dans  $t$  chaque  $\lambda x$  par  $\lambda y$  et chaque occurrence de  $x$  liée à un  $\lambda x$  par  $y$ .

P.ex.,

- ▶ l' $\alpha$ -renommage de  $x$  en  $y$  du  $\lambda$ -terme  $\lambda x.x$  est  $\lambda y.y$ ;
- ▶ l' $\alpha$ -renommage de  $x$  en  $y$  du  $\lambda$ -terme  $(x x) (\lambda x.(x z))$  est  $(x x) (\lambda y.(y z))$ .

Deux  $\lambda$ -termes  $t$  et  $t'$  sont  $\alpha$ -équivalents s'il est possible de transformer  $t$  en  $t'$  par une suite d' $\alpha$ -renommages.

P.ex.,

- ▶  $\lambda x.x$  et  $\lambda y.y$  sont  $\alpha$ -équivalents;
- ▶  $\lambda x.\lambda y.(x y)$  et  $\lambda y.\lambda x.(y x)$  sont  $\alpha$ -équivalents;
- ▶  $\lambda x.(x y)$  et  $\lambda x.(x z)$  ne sont pas  $\alpha$ -équivalents.

# $\beta$ -réductions en racine

Soit  $t$  un  $\lambda$ -terme de la forme

$$t := (\lambda x.u) v.$$

# $\beta$ -réductions en racine

Soit  $t$  un  $\lambda$ -terme de la forme

$$t := (\lambda x.u) v.$$

La  $\beta$ -réduction en racine appliquée sur  $t$  consiste à transformer  $t$  en le  $\lambda$ -terme

$$u \star_x v.$$

# $\beta$ -réductions en racine

Soit  $t$  un  $\lambda$ -terme de la forme

$$t := (\lambda x.u) v.$$

La  $\beta$ -réduction en racine appliquée sur  $t$  consiste à transformer  $t$  en le  $\lambda$ -terme

$$u \star_x v.$$

**Attention** : aucune occurrence d'une variable libre de  $v$  ne doit être capturée dans  $u \star_x v$ .

# $\beta$ -réductions en racine

Soit  $t$  un  $\lambda$ -terme de la forme

$$t := (\lambda x.u) v.$$

La  $\beta$ -réduction en racine appliquée sur  $t$  consiste à transformer  $t$  en le  $\lambda$ -terme

$$u \star_x v.$$

**Attention** : aucune occurrence d'une variable libre de  $v$  ne doit être capturée dans  $u \star_x v$ .

De ce fait, dès que  $v$  admet une occurrence libre d'une variable  $y$ , on doit  $\alpha$ -renommer  $y$  en  $y'$  dans  $u$  au préalable, où  $y'$  est une nouvelle variable qui n'admet pas d'occurrence libre dans  $v$ .

# $\beta$ -réductions en racine

Soit  $t$  un  $\lambda$ -terme de la forme

$$t := (\lambda x.u) v.$$

La  $\beta$ -réduction en racine appliquée sur  $t$  consiste à transformer  $t$  en le  $\lambda$ -terme

$$u \star_x v.$$

**Attention** : aucune occurrence d'une variable libre de  $v$  ne doit être capturée dans  $u \star_x v$ .

De ce fait, dès que  $v$  admet une occurrence libre d'une variable  $y$ , on doit  $\alpha$ -renommer  $y$  en  $y'$  dans  $u$  au préalable, où  $y'$  est une nouvelle variable qui n'admet pas d'occurrence libre dans  $v$ .

On note  $t \xrightarrow{\text{r}}_{\beta} t'$  le fait qu'un  $\lambda$ -terme  $t'$  puisse être obtenu à partir du  $\lambda$ -terme  $t$  par une  $\beta$ -réduction en racine.

# $\beta$ -réductions en racine — exemples

►  $(\lambda y.y)(\lambda x.(x z))$

# $\beta$ -réductions en racine — exemples

►  $(\lambda y.y)(\lambda x.(x z)) \xrightarrow{\beta} \lambda x.(x z)$

# $\beta$ -réductions en racine — exemples

▶  $(\lambda y.y)(\lambda x.(x z)) \xrightarrow{\beta} \lambda x.(x z)$

▶  $(\lambda x.(x y)) z$

# $\beta$ -réductions en racine — exemples

▶  $(\lambda y.y)(\lambda x.(x z)) \xrightarrow{\beta} \lambda x.(x z)$

▶  $(\lambda x.(x y)) z \xrightarrow{\beta} z y$

# $\beta$ -réductions en racine — exemples

▶  $(\lambda y.y)(\lambda x.(x z)) \xrightarrow{\beta} \lambda x.(x z)$

▶  $(\lambda x.(x y)) z \xrightarrow{\beta} z y$

▶  $(\lambda x.\lambda y.((x x) (\lambda x.x))) (\lambda y.(y x))$

# $\beta$ -réductions en racine — exemples

▶  $(\lambda y.y)(\lambda x.(x z)) \xrightarrow{\beta} \lambda x.(x z)$

▶  $(\lambda x.(x y)) z \xrightarrow{\beta} z y$

▶  $(\lambda x.\lambda y.((x x) (\lambda x.x))) (\lambda y.(y x)) \xrightarrow{\beta} \lambda y.(((\lambda y.(y x)) (\lambda y(y x)))) (\lambda x.x)$

# $\beta$ -réductions en racine — exemples

▶  $(\lambda y.y)(\lambda x.(x z)) \xrightarrow{\beta} \lambda x.(x z)$

▶  $(\lambda x.(x y)) z \xrightarrow{\beta} z y$

▶  $(\lambda x.\lambda y.((x x) (\lambda x.x))) (\lambda y.(y x)) \xrightarrow{\beta} \lambda y.(((\lambda y.(y x)) (\lambda y(y x)))) (\lambda x.x)$

▶  $(\lambda x.\lambda y.x) (z y)$

# $\beta$ -réductions en racine — exemples

▶  $(\lambda y.y)(\lambda x.(x z)) \xrightarrow{\beta} \lambda x.(x z)$

▶  $(\lambda x.(x y)) z \xrightarrow{\beta} z y$

▶  $(\lambda x.\lambda y.((x x) (\lambda x.x))) (\lambda y.(y x)) \xrightarrow{\beta} \lambda y.(((\lambda y.(y x)) (\lambda y(y x)))) (\lambda x.x)$

▶  $(\lambda x.\lambda y.x) (z y) \not\xrightarrow{\beta} \lambda y.(z y)$

## $\beta$ -réductions en racine — exemples

▶  $(\lambda y.y)(\lambda x.(x z)) \xrightarrow{\beta} \lambda x.(x z)$

▶  $(\lambda x.(x y)) z \xrightarrow{\beta} z y$

▶  $(\lambda x.\lambda y.((x x) (\lambda x.x))) (\lambda y.(y x)) \xrightarrow{\beta} \lambda y.(((\lambda y.(y x)) (\lambda y(y x)))) (\lambda x.x)$

▶  $(\lambda x.\lambda y.x) (z y) \not\xrightarrow{\beta} \lambda y.(z y)$

Ici, il l'occurrence libre de  $y$  dans  $(z y)$  serait capturée par le  $\lambda y$  dans le « résultat ».

## $\beta$ -réductions en racine — exemples

▶  $(\lambda y.y)(\lambda x.(x z)) \xrightarrow{\beta} \lambda x.(x z)$

▶  $(\lambda x.(x y)) z \xrightarrow{\beta} z y$

▶  $(\lambda x.\lambda y.((x x) (\lambda x.x))) (\lambda y.(y x)) \xrightarrow{\beta} \lambda y.(((\lambda y.(y x)) (\lambda y(y x)))) (\lambda x.x)$

▶  $(\lambda x.\lambda y.x) (z y) \not\xrightarrow{\beta} \lambda y.(z y)$

Ici, il l'occurrence libre de  $y$  dans  $(z y)$  serait capturée par le  $\lambda y$  dans le « résultat ».

On  $\alpha$ -renomme donc  $y$  en  $y'$  dans  $(\lambda x.\lambda y.x)$  et on obtient

$$(\lambda x.\lambda y.x) (z y) = (\lambda x.\lambda y'.x) (z y) \xrightarrow{\beta} \lambda y'.(z y).$$

# $\beta$ -réductions

Soit  $t$  un  $\lambda$ -terme.

# $\beta$ -réductions

Soit  $t$  un  $\lambda$ -terme.

On appelle **sous-terme** de  $t$  tout  $\lambda$ -terme dont l'arbre syntaxique est un sous-arbre de l'arbre syntaxique de  $t$ .

# $\beta$ -réductions

Soit  $t$  un  $\lambda$ -terme.

On appelle **sous-terme** de  $t$  tout  $\lambda$ -terme dont l'arbre syntaxique est un sous-arbre de l'arbre syntaxique de  $t$ .

P.ex., le  $\lambda$ -terme  $x (\lambda x. \lambda y. (y y))$  possède comme sous-termes

# $\beta$ -réductions

Soit  $t$  un  $\lambda$ -terme.

On appelle **sous-terme** de  $t$  tout  $\lambda$ -terme dont l'arbre syntaxique est un sous-arbre de l'arbre syntaxique de  $t$ .

P.ex., le  $\lambda$ -terme  $x (\lambda x. \lambda y. (y y))$  possède comme sous-termes

$x$ ,

# $\beta$ -réductions

Soit  $t$  un  $\lambda$ -terme.

On appelle **sous-terme** de  $t$  tout  $\lambda$ -terme dont l'arbre syntaxique est un sous-arbre de l'arbre syntaxique de  $t$ .

P.ex., le  $\lambda$ -terme  $x (\lambda x. \lambda y. (y y))$  possède comme sous-termes

$x, y,$

# $\beta$ -réductions

Soit  $t$  un  $\lambda$ -terme.

On appelle **sous-terme** de  $t$  tout  $\lambda$ -terme dont l'arbre syntaxique est un sous-arbre de l'arbre syntaxique de  $t$ .

P.ex., le  $\lambda$ -terme  $x (\lambda x. \lambda y. (y y))$  possède comme sous-termes

$x, y, y y,$

# $\beta$ -réductions

Soit  $t$  un  $\lambda$ -terme.

On appelle **sous-terme** de  $t$  tout  $\lambda$ -terme dont l'arbre syntaxique est un sous-arbre de l'arbre syntaxique de  $t$ .

P.ex., le  $\lambda$ -terme  $x (\lambda x. \lambda y. (y y))$  possède comme sous-termes

$$x, y, y y, \lambda y. (y y),$$

# $\beta$ -réductions

Soit  $t$  un  $\lambda$ -terme.

On appelle **sous-terme** de  $t$  tout  $\lambda$ -terme dont l'arbre syntaxique est un sous-arbre de l'arbre syntaxique de  $t$ .

P.ex., le  $\lambda$ -terme  $x (\lambda x. \lambda y. (y y))$  possède comme sous-termes

$$x, y, y y, \lambda y. (y y), \lambda x. \lambda y. (y y),$$

# $\beta$ -réductions

Soit  $t$  un  $\lambda$ -terme.

On appelle **sous-terme** de  $t$  tout  $\lambda$ -terme dont l'arbre syntaxique est un sous-arbre de l'arbre syntaxique de  $t$ .

P.ex., le  $\lambda$ -terme  $x (\lambda x. \lambda y. (y y))$  possède comme sous-termes

$$x, y, y y, \lambda y. (y y), \lambda x. \lambda y. (y y), x (\lambda x. \lambda y. (y y)).$$

# $\beta$ -réductions

Soit  $t$  un  $\lambda$ -terme.

On appelle **sous-terme** de  $t$  tout  $\lambda$ -terme dont l'arbre syntaxique est un sous-arbre de l'arbre syntaxique de  $t$ .

P.ex., le  $\lambda$ -terme  $x (\lambda x. \lambda y. (y y))$  possède comme sous-termes

$$x, y, y y, \lambda y. (y y), \lambda x. \lambda y. (y y), x (\lambda x. \lambda y. (y y)).$$

La  **$\beta$ -réduction** appliquée sur  $t$  consiste à appliquer une  $\beta$ -réduction en racine sur l'un de ses sous-termes.

# $\beta$ -réductions

Soit  $t$  un  $\lambda$ -terme.

On appelle **sous-terme** de  $t$  tout  $\lambda$ -terme dont l'arbre syntaxique est un sous-arbre de l'arbre syntaxique de  $t$ .

P.ex., le  $\lambda$ -terme  $x (\lambda x. \lambda y. (y y))$  possède comme sous-termes

$$x, y, y y, \lambda y. (y y), \lambda x. \lambda y. (y y), x (\lambda x. \lambda y. (y y)).$$

La  **$\beta$ -réduction** appliquée sur  $t$  consiste à appliquer une  $\beta$ -réduction en racine sur l'un de ses sous-termes.

On note  $t \rightarrow_{\beta} t'$  le fait qu'un  $\lambda$ -terme  $t'$  puisse être obtenu à partir de  $t$  par une  $\beta$ -réduction.

# $\beta$ -réductions

Soit  $t$  un  $\lambda$ -terme.

On appelle **sous-terme** de  $t$  tout  $\lambda$ -terme dont l'arbre syntaxique est un sous-arbre de l'arbre syntaxique de  $t$ .

P.ex., le  $\lambda$ -terme  $x (\lambda x. \lambda y. (y y))$  possède comme sous-termes

$$x, y, y y, \lambda y. (y y), \lambda x. \lambda y. (y y), x (\lambda x. \lambda y. (y y)).$$

La  **$\beta$ -réduction** appliquée sur  $t$  consiste à appliquer une  $\beta$ -réduction en racine sur l'un de ses sous-termes.

On note  $t \rightarrow_{\beta} t'$  le fait qu'un  $\lambda$ -terme  $t'$  puisse être obtenu à partir de  $t$  par une  $\beta$ -réduction.

On note  $t \rightarrow_{\beta}^* t'$  le fait qu'un  $\lambda$ -terme  $t'$  puisse être obtenu à partir de  $t$  par une suite de  $\beta$ -réductions.

# $\beta$ -réductions — exemples

- ▶ Voici une suite de  $\beta$ -réductions :

$$((\lambda x.(x y)) y) ((\lambda y.y) ((\lambda x.x) x))$$

# $\beta$ -réductions — exemples

- ▶ Voici une suite de  $\beta$ -réductions :

$$((\lambda x.(x y)) y) ((\lambda y.y) ((\lambda x.x) x)) \rightarrow_{\beta} (y y) ((\lambda y.y) ((\lambda x.x) x))$$

# $\beta$ -réductions — exemples

► Voici une suite de  $\beta$ -réductions :

$$\begin{aligned} ((\lambda x.(x y)) y) ((\lambda y.y) ((\lambda x.x) x)) &\rightarrow_{\beta} (y y) ((\lambda y.y) ((\lambda x.x) x)) \\ &\rightarrow_{\beta} (y y) ((\lambda y.y) x) \end{aligned}$$

# $\beta$ -réductions — exemples

► Voici une suite de  $\beta$ -réductions :

$$\begin{aligned} ((\lambda x.(x y)) y) ((\lambda y.y) ((\lambda x.x) x)) &\rightarrow_{\beta} (y y) ((\lambda y.y) ((\lambda x.x) x)) \\ &\rightarrow_{\beta} (y y) ((\lambda y.y) x) \\ &\rightarrow_{\beta} (y y) x. \end{aligned}$$

# $\beta$ -réductions — exemples

- ▶ Voici une suite de  $\beta$ -réductions :

$$\begin{aligned} ((\lambda x.(x y)) y) ((\lambda y.y) ((\lambda x.x) x)) &\rightarrow_{\beta} (y y) ((\lambda y.y) ((\lambda x.x) x)) \\ &\rightarrow_{\beta} (y y) ((\lambda y.y) x) \\ &\rightarrow_{\beta} (y y) x. \end{aligned}$$

- ▶ En partant d'un même  $\lambda$ -terme, on peut appliquer plusieurs suites de  $\beta$ -réductions différentes :

# $\beta$ -réductions — exemples

- ▶ Voici une suite de  $\beta$ -réductions :

$$\begin{aligned} ((\lambda x.(x y)) y) ((\lambda y.y) ((\lambda x.x) x)) &\rightarrow_{\beta} (y y) ((\lambda y.y) ((\lambda x.x) x)) \\ &\rightarrow_{\beta} (y y) ((\lambda y.y) x) \\ &\rightarrow_{\beta} (y y) x. \end{aligned}$$

- ▶ En partant d'un même  $\lambda$ -terme, on peut appliquer plusieurs suites de  $\beta$ -réductions différentes :

$$(\lambda y.y) ((\lambda x.x) x)$$

# $\beta$ -réductions — exemples

- ▶ Voici une suite de  $\beta$ -réductions :

$$\begin{aligned} ((\lambda x.(x y)) y) ((\lambda y.y) ((\lambda x.x) x)) &\rightarrow_{\beta} (y y) ((\lambda y.y) ((\lambda x.x) x)) \\ &\rightarrow_{\beta} (y y) ((\lambda y.y) x) \\ &\rightarrow_{\beta} (y y) x. \end{aligned}$$

- ▶ En partant d'un même  $\lambda$ -terme, on peut appliquer plusieurs suites de  $\beta$ -réductions différentes :

$$(\lambda y.y) ((\lambda x.x) x) \rightarrow_{\beta} (\lambda y.y) x$$

# $\beta$ -réductions — exemples

- ▶ Voici une suite de  $\beta$ -réductions :

$$\begin{aligned} ((\lambda x.(x y)) y) ((\lambda y.y) ((\lambda x.x) x)) &\rightarrow_{\beta} (y y) ((\lambda y.y) ((\lambda x.x) x)) \\ &\rightarrow_{\beta} (y y) ((\lambda y.y) x) \\ &\rightarrow_{\beta} (y y) x. \end{aligned}$$

- ▶ En partant d'un même  $\lambda$ -terme, on peut appliquer plusieurs suites de  $\beta$ -réductions différentes :

$$\begin{aligned} (\lambda y.y) ((\lambda x.x) x) &\rightarrow_{\beta} (\lambda y.y) x \\ &\rightarrow_{\beta} x, \end{aligned}$$

# $\beta$ -réductions — exemples

- ▶ Voici une suite de  $\beta$ -réductions :

$$\begin{aligned} ((\lambda x.(x y)) y) ((\lambda y.y) ((\lambda x.x) x)) &\rightarrow_{\beta} (y y) ((\lambda y.y) ((\lambda x.x) x)) \\ &\rightarrow_{\beta} (y y) ((\lambda y.y) x) \\ &\rightarrow_{\beta} (y y) x. \end{aligned}$$

- ▶ En partant d'un même  $\lambda$ -terme, on peut appliquer plusieurs suites de  $\beta$ -réductions différentes :

$$\begin{aligned} (\lambda y.y) ((\lambda x.x) x) &\rightarrow_{\beta} (\lambda y.y) x \\ &\rightarrow_{\beta} x, \end{aligned}$$

$$(\lambda y.y) ((\lambda x.x) x)$$

# $\beta$ -réductions — exemples

- ▶ Voici une suite de  $\beta$ -réductions :

$$\begin{aligned} ((\lambda x.(x y)) y) ((\lambda y.y) ((\lambda x.x) x)) &\rightarrow_{\beta} (y y) ((\lambda y.y) ((\lambda x.x) x)) \\ &\rightarrow_{\beta} (y y) ((\lambda y.y) x) \\ &\rightarrow_{\beta} (y y) x. \end{aligned}$$

- ▶ En partant d'un même  $\lambda$ -terme, on peut appliquer plusieurs suites de  $\beta$ -réductions différentes :

$$\begin{aligned} (\lambda y.y) ((\lambda x.x) x) &\rightarrow_{\beta} (\lambda y.y) x \\ &\rightarrow_{\beta} x, \end{aligned}$$

$$(\lambda y.y) ((\lambda x.x) x) \rightarrow_{\beta} (\lambda x.x) x$$

# $\beta$ -réductions — exemples

- ▶ Voici une suite de  $\beta$ -réductions :

$$\begin{aligned}((\lambda x.(x y)) y) ((\lambda y.y) ((\lambda x.x) x)) &\rightarrow_{\beta} (y y) ((\lambda y.y) ((\lambda x.x) x)) \\ &\rightarrow_{\beta} (y y) ((\lambda y.y) x) \\ &\rightarrow_{\beta} (y y) x.\end{aligned}$$

- ▶ En partant d'un même  $\lambda$ -terme, on peut appliquer plusieurs suites de  $\beta$ -réductions différentes :

$$\begin{aligned}(\lambda y.y) ((\lambda x.x) x) &\rightarrow_{\beta} (\lambda y.y) x \\ &\rightarrow_{\beta} x,\end{aligned}$$

$$\begin{aligned}(\lambda y.y) ((\lambda x.x) x) &\rightarrow_{\beta} (\lambda x.x) x \\ &\rightarrow_{\beta} x.\end{aligned}$$

# $\beta$ -réductions — exemples

- ▶ Voici une suite de  $\beta$ -réductions :

$$\begin{aligned}((\lambda x.(x y)) y) ((\lambda y.y) ((\lambda x.x) x)) &\rightarrow_{\beta} (y y) ((\lambda y.y) ((\lambda x.x) x)) \\ &\rightarrow_{\beta} (y y) ((\lambda y.y) x) \\ &\rightarrow_{\beta} (y y) x.\end{aligned}$$

- ▶ En partant d'un même  $\lambda$ -terme, on peut appliquer plusieurs suites de  $\beta$ -réductions différentes :

$$\begin{aligned}(\lambda y.y) ((\lambda x.x) x) &\rightarrow_{\beta} (\lambda y.y) x \\ &\rightarrow_{\beta} x,\end{aligned}$$

$$\begin{aligned}(\lambda y.y) ((\lambda x.x) x) &\rightarrow_{\beta} (\lambda x.x) x \\ &\rightarrow_{\beta} x.\end{aligned}$$

Ainsi dans ces deux cas,  $(\lambda y.y) ((\lambda x.x) x) \xrightarrow{*}_{\beta} x$ .

# Règle de calcul

La relation de  $\beta$ -réduction  $\rightarrow_\beta$  est une règle de calcul.

# Règle de calcul

La relation de  $\beta$ -réduction  $\rightarrow_\beta$  est une **règle de calcul**. On appelle **évaluation** le processus qui consiste à appliquer des  $\beta$ -réductions sur un  $\lambda$ -terme.

# Règle de calcul

La relation de  $\beta$ -réduction  $\rightarrow_{\beta}$  est une **règle de calcul**. On appelle **évaluation** le processus qui consiste à appliquer des  $\beta$ -réductions sur un  $\lambda$ -terme.

L'évaluation permet, dans la mesure du possible, à partir d'un  $\lambda$ -terme  $t$  complexe, d'obtenir un  $\lambda$ -terme  $t'$  plus simple que l'on ne peut plus  $\beta$ -réduire.

# Règle de calcul

La relation de  $\beta$ -réduction  $\rightarrow_{\beta}$  est une **règle de calcul**. On appelle **évaluation** le processus qui consiste à appliquer des  $\beta$ -réductions sur un  $\lambda$ -terme.

L'évaluation permet, dans la mesure du possible, à partir d'un  $\lambda$ -terme  $t$  complexe, d'obtenir un  $\lambda$ -terme  $t'$  plus simple que l'on ne peut plus  $\beta$ -réduire.

On appelle un tel terme un **réduit**. Intuitivement, un réduit est une **valeur**.

# Règle de calcul

La relation de  $\beta$ -réduction  $\rightarrow_\beta$  est une **règle de calcul**. On appelle **évaluation** le processus qui consiste à appliquer des  $\beta$ -réductions sur un  $\lambda$ -terme.

L'évaluation permet, dans la mesure du possible, à partir d'un  $\lambda$ -terme  $t$  complexe, d'obtenir un  $\lambda$ -terme  $t'$  plus simple que l'on ne peut plus  $\beta$ -réduire.

On appelle un tel terme un **réduit**. Intuitivement, un réduit est une **valeur**.

Nous avons observé qu'il existe, étant donné un  $\lambda$ -terme, plusieurs processus d'évaluation différents.

# Règle de calcul

La relation de  $\beta$ -réduction  $\rightarrow_{\beta}$  est une **règle de calcul**. On appelle **évaluation** le processus qui consiste à appliquer des  $\beta$ -réductions sur un  $\lambda$ -terme.

L'évaluation permet, dans la mesure du possible, à partir d'un  $\lambda$ -terme  $t$  complexe, d'obtenir un  $\lambda$ -terme  $t'$  plus simple que l'on ne peut plus  $\beta$ -réduire.

On appelle un tel terme un **réduit**. Intuitivement, un réduit est une **valeur**.

Nous avons observé qu'il existe, étant donné un  $\lambda$ -terme, plusieurs processus d'évaluation différents.

**Question** : est-ce que les différentes évaluations d'un même  $\lambda$ -terme  $t$  aboutissent à la même valeur ?

# Règle de calcul

La relation de  $\beta$ -réduction  $\rightarrow_{\beta}$  est une **règle de calcul**. On appelle **évaluation** le processus qui consiste à appliquer des  $\beta$ -réductions sur un  $\lambda$ -terme.

L'évaluation permet, dans la mesure du possible, à partir d'un  $\lambda$ -terme  $t$  complexe, d'obtenir un  $\lambda$ -terme  $t'$  plus simple que l'on ne peut plus  $\beta$ -réduire.

On appelle un tel terme un **réduit**. Intuitivement, un réduit est une **valeur**.

Nous avons observé qu'il existe, étant donné un  $\lambda$ -terme, plusieurs processus d'évaluation différents.

**Question** : est-ce que les différentes évaluations d'un même  $\lambda$ -terme  $t$  aboutissent à la même valeur ?

**Réponse** : oui, d'après le **théorème de Church-Rosser**.

# Terminaison

**Question** : est-ce que l'évaluation de tout  $\lambda$ -terme aboutit toujours à une valeur ?

# Terminaison

**Question** : est-ce que l'évaluation de tout  $\lambda$ -terme aboutit toujours à une valeur ?

**Réponse** : considérons le  $\lambda$ -terme

$$\Delta := \lambda x.(x x).$$

# Terminaison

**Question** : est-ce que l'évaluation de tout  $\lambda$ -terme aboutit toujours à une valeur?

**Réponse** : considérons le  $\lambda$ -terme

$$\Delta := \lambda x.(x x).$$

Nous avons

$$\Delta \Delta$$

# Terminaison

**Question** : est-ce que l'évaluation de tout  $\lambda$ -terme aboutit toujours à une valeur ?

**Réponse** : considérons le  $\lambda$ -terme

$$\Delta := \lambda x.(x x).$$

Nous avons

$$\Delta \Delta = (\lambda x.(x x)) (\lambda x.(x x))$$

# Terminaison

**Question** : est-ce que l'évaluation de tout  $\lambda$ -terme aboutit toujours à une valeur?

**Réponse** : considérons le  $\lambda$ -terme

$$\Delta := \lambda x.(x x).$$

Nous avons

$$\begin{aligned} \Delta \Delta &= (\lambda x.(x x)) (\lambda x.(x x)) \\ &\rightarrow_{\beta} (\lambda x.(x x)) (\lambda x.(x x)) \end{aligned}$$

# Terminaison

**Question** : est-ce que l'évaluation de tout  $\lambda$ -terme aboutit toujours à une valeur?

**Réponse** : considérons le  $\lambda$ -terme

$$\Delta := \lambda x.(x x).$$

Nous avons

$$\begin{aligned}\Delta \Delta &= (\lambda x.(x x)) (\lambda x.(x x)) \\ &\rightarrow_{\beta} (\lambda x.(x x)) (\lambda x.(x x)) \\ &\rightarrow_{\beta} \dots\end{aligned}$$

# Terminaison

**Question** : est-ce que l'évaluation de tout  $\lambda$ -terme aboutit toujours à une valeur ?

**Réponse** : considérons le  $\lambda$ -terme

$$\Delta := \lambda x.(x x).$$

Nous avons

$$\begin{aligned}\Delta \Delta &= (\lambda x.(x x)) (\lambda x.(x x)) \\ &\rightarrow_{\beta} (\lambda x.(x x)) (\lambda x.(x x)) \\ &\rightarrow_{\beta} \dots\end{aligned}$$

Ainsi, l'évaluation de  $\Delta \Delta$  ne termine pas. La réponse est donc non.

# Plan

$\lambda$ -calcul

$\lambda$ -termes

Codage

Implantation

# Programmer en $\lambda$ -calcul

Voici quelques correspondances entre le  $\lambda$ -calcul et la programmation (fonctionnelle).

# Programmer en $\lambda$ -calcul

Voici quelques correspondances entre le  $\lambda$ -calcul et la programmation (fonctionnelle).

Tout  $\lambda$ -terme  $t$  code un **programme**.

# Programmer en $\lambda$ -calcul

Voici quelques correspondances entre le  $\lambda$ -calcul et la programmation (fonctionnelle).

Tout  $\lambda$ -terme  $t$  code un **programme**.

Étant donné que toutes les différentes évaluations d'un même  $\lambda$ -terme  $t$  aboutissent à la même valeur, l'exécution de ce programme est déterministe.

# Programmer en $\lambda$ -calcul

Voici quelques correspondances entre le  $\lambda$ -calcul et la programmation (fonctionnelle).

Tout  $\lambda$ -terme  $t$  code un **programme**.

Étant donné que toutes les différentes évaluations d'un même  $\lambda$ -terme  $t$  aboutissent à la même valeur, l'exécution de ce programme est déterministe.

L'évaluation de  $t$  correspond à l'**exécution** du programme qu'il code

# Programmer en $\lambda$ -calcul

Voici quelques correspondances entre le  $\lambda$ -calcul et la programmation (fonctionnelle).

Tout  $\lambda$ -terme  $t$  code un **programme**.

Étant donné que toutes les différentes évaluations d'un même  $\lambda$ -terme  $t$  aboutissent à la même valeur, l'exécution de ce programme est déterministe.

L'évaluation de  $t$  correspond à l'**exécution** du programme qu'il code et

- ▶ si l'évaluation de  $t$  termine, le réduit obtenu est la valeur calculée par ce programme;

# Programmer en $\lambda$ -calcul

Voici quelques correspondances entre le  $\lambda$ -calcul et la programmation (fonctionnelle).

Tout  $\lambda$ -terme  $t$  code un **programme**.

Étant donné que toutes les différentes évaluations d'un même  $\lambda$ -terme  $t$  aboutissent à la même valeur, l'exécution de ce programme est déterministe.

L'évaluation de  $t$  correspond à l'**exécution** du programme qu'il code et

- ▶ si l'évaluation de  $t$  termine, le réduit obtenu est la valeur calculée par ce programme ;
- ▶ si elle ne termine pas, l'exécution du programme diverge.

# Programmer en $\lambda$ -calcul

Voici quelques correspondances entre le  $\lambda$ -calcul et la programmation (fonctionnelle).

Tout  $\lambda$ -terme  $t$  code un **programme**.

Étant donné que toutes les différentes évaluations d'un même  $\lambda$ -terme  $t$  aboutissent à la même valeur, l'exécution de ce programme est déterministe.

L'évaluation de  $t$  correspond à l'**exécution** du programme qu'il code et

- ▶ si l'évaluation de  $t$  termine, le réduit obtenu est la valeur calculée par ce programme ;
- ▶ si elle ne termine pas, l'exécution du programme diverge.

Il reste à savoir comment donner du **sens** à nos  $\lambda$ -termes.

# Programmer en $\lambda$ -calcul

Voici quelques correspondances entre le  $\lambda$ -calcul et la programmation (fonctionnelle).

Tout  $\lambda$ -terme  $t$  code un **programme**.

Étant donné que toutes les différentes évaluations d'un même  $\lambda$ -terme  $t$  aboutissent à la même valeur, l'exécution de ce programme est déterministe.

L'évaluation de  $t$  correspond à l'**exécution** du programme qu'il code et

- ▶ si l'évaluation de  $t$  termine, le réduit obtenu est la valeur calculée par ce programme ;
- ▶ si elle ne termine pas, l'exécution du programme diverge.

Il reste à savoir comment donner du **sens** à nos  $\lambda$ -termes.

La question est de savoir comment coder des **booléens**, des **entiers** et des **constructions conditionnelles**. Ceci offre un niveau d'expressivité raisonnable.

# Booléens

On code les deux booléens `vrai` et `faux` de la manière suivante :

# Booléens

On code les deux booléens *vrai* et *faux* de la manière suivante :

$$\text{vrai} := \lambda x.\lambda y.x,$$

# Booléens

On code les deux booléens `vrai` et `faux` de la manière suivante :

`vrai` :=  $\lambda x.\lambda y.x$ ,

`faux` :=  $\lambda x.\lambda y.y$ ,

# Booléens

On code les deux booléens **vrai** et **faux** de la manière suivante :

$$\mathbf{vrai} := \lambda x. \lambda y. x,$$

$$\mathbf{faux} := \lambda x. \lambda y. y,$$

Nous avons, si  $u$  et  $v$  sont deux  $\lambda$ -termes,

$$(\mathbf{vrai} \ u) \ v$$

# Booléens

On code les deux booléens **vrai** et **faux** de la manière suivante :

$$\mathbf{vrai} := \lambda x.\lambda y.x,$$

$$\mathbf{faux} := \lambda x.\lambda y.y,$$

Nous avons, si  $u$  et  $v$  sont deux  $\lambda$ -termes,

$$(\mathbf{vrai} u) v = ((\lambda x.\lambda y.x) u) v$$

# Booléens

On code les deux booléens **vrai** et **faux** de la manière suivante :

$$\mathbf{vrai} := \lambda x. \lambda y. x,$$

$$\mathbf{faux} := \lambda x. \lambda y. y,$$

Nous avons, si  $u$  et  $v$  sont deux  $\lambda$ -termes,

$$\begin{aligned} (\mathbf{vrai} \ u) \ v &= ((\lambda x. \lambda y. x) \ u) \ v \\ &\rightarrow_{\beta} (\lambda y. u) \ v \end{aligned}$$

# Booléens

On code les deux booléens **vrai** et **faux** de la manière suivante :

$$\mathbf{vrai} := \lambda x. \lambda y. x,$$

$$\mathbf{faux} := \lambda x. \lambda y. y,$$

Nous avons, si  $u$  et  $v$  sont deux  $\lambda$ -termes,

$$\begin{aligned} (\mathbf{vrai} \ u) \ v &= ((\lambda x. \lambda y. x) \ u) \ v \\ &\rightarrow_{\beta} (\lambda y. u) \ v \\ &\rightarrow_{\beta} u. \end{aligned}$$

# Booléens

On code les deux booléens **vrai** et **faux** de la manière suivante :

**vrai** :=  $\lambda x.\lambda y.x$ ,      renvoie son 1<sup>er</sup> argument,  
**faux** :=  $\lambda x.\lambda y.y$ ,

Nous avons, si  $u$  et  $v$  sont deux  $\lambda$ -termes,

$$\begin{aligned}(\mathbf{vrai} \ u) \ v &= ((\lambda x.\lambda y.x) \ u) \ v \\ &\rightarrow_{\beta} (\lambda y.u) \ v \\ &\rightarrow_{\beta} u.\end{aligned}$$

# Booléens

On code les deux booléens **vrai** et **faux** de la manière suivante :

$$\begin{aligned}\mathbf{vrai} &:= \lambda x.\lambda y.x, && \text{renvoie son 1}^{\text{er}} \text{ argument,} \\ \mathbf{faux} &:= \lambda x.\lambda y.y,\end{aligned}$$

Nous avons, si  $u$  et  $v$  sont deux  $\lambda$ -termes,

$$\begin{aligned}(\mathbf{vrai} u) v &= ((\lambda x.\lambda y.x) u) v \\ &\rightarrow_{\beta} (\lambda y.u) v \\ &\rightarrow_{\beta} u.\end{aligned}$$

De même,

$$(\mathbf{faux} u) v$$

# Booléens

On code les deux booléens **vrai** et **faux** de la manière suivante :

$$\begin{aligned}\mathbf{vrai} &:= \lambda x.\lambda y.x, && \text{renvoie son 1<sup>er</sup> argument,} \\ \mathbf{faux} &:= \lambda x.\lambda y.y,\end{aligned}$$

Nous avons, si  $u$  et  $v$  sont deux  $\lambda$ -termes,

$$\begin{aligned}(\mathbf{vrai} u) v &= ((\lambda x.\lambda y.x) u) v \\ &\rightarrow_{\beta} (\lambda y.u) v \\ &\rightarrow_{\beta} u.\end{aligned}$$

De même,

$$(\mathbf{faux} u) v = ((\lambda x.\lambda y.y) u) v$$

# Booléens

On code les deux booléens **vrai** et **faux** de la manière suivante :

$$\begin{aligned}\mathbf{vrai} &:= \lambda x.\lambda y.x, && \text{renvoie son 1<sup>er</sup> argument,} \\ \mathbf{faux} &:= \lambda x.\lambda y.y,\end{aligned}$$

Nous avons, si  $u$  et  $v$  sont deux  $\lambda$ -termes,

$$\begin{aligned}(\mathbf{vrai} \ u) \ v &= ((\lambda x.\lambda y.x) \ u) \ v \\ &\rightarrow_{\beta} (\lambda y.u) \ v \\ &\rightarrow_{\beta} u.\end{aligned}$$

De même,

$$\begin{aligned}(\mathbf{faux} \ u) \ v &= ((\lambda x.\lambda y.y) \ u) \ v \\ &\rightarrow_{\beta} (\lambda y.y) \ v\end{aligned}$$

# Booléens

On code les deux booléens **vrai** et **faux** de la manière suivante :

$$\begin{aligned}\mathbf{vrai} &:= \lambda x.\lambda y.x, && \text{renvoie son 1<sup>er</sup> argument,} \\ \mathbf{faux} &:= \lambda x.\lambda y.y,\end{aligned}$$

Nous avons, si  $u$  et  $v$  sont deux  $\lambda$ -termes,

$$\begin{aligned}(\mathbf{vrai} u) v &= ((\lambda x.\lambda y.x) u) v \\ &\rightarrow_{\beta} (\lambda y.u) v \\ &\rightarrow_{\beta} u.\end{aligned}$$

De même,

$$\begin{aligned}(\mathbf{faux} u) v &= ((\lambda x.\lambda y.y) u) v \\ &\rightarrow_{\beta} (\lambda y.y) v \\ &\rightarrow_{\beta} v.\end{aligned}$$

# Booléens

On code les deux booléens **vrai** et **faux** de la manière suivante :

**vrai** :=  $\lambda x.\lambda y.x$ ,      renvoie son 1<sup>er</sup> argument,  
**faux** :=  $\lambda x.\lambda y.y$ ,      renvoie son 2<sup>e</sup> argument.

Nous avons, si  $u$  et  $v$  sont deux  $\lambda$ -termes,

$$\begin{aligned}(\mathbf{vrai} \ u) \ v &= ((\lambda x.\lambda y.x) \ u) \ v \\ &\rightarrow_{\beta} (\lambda y.u) \ v \\ &\rightarrow_{\beta} u.\end{aligned}$$

De même,

$$\begin{aligned}(\mathbf{faux} \ u) \ v &= ((\lambda x.\lambda y.y) \ u) \ v \\ &\rightarrow_{\beta} (\lambda y.y) \ v \\ &\rightarrow_{\beta} v.\end{aligned}$$

# Booléens

On code les deux booléens **vrai** et **faux** de la manière suivante :

**vrai** :=  $\lambda x.\lambda y.x$ ,      renvoie son 1<sup>er</sup> argument,  
**faux** :=  $\lambda x.\lambda y.y$ ,      renvoie son 2<sup>e</sup> argument.

Nous avons, si  $u$  et  $v$  sont deux  $\lambda$ -termes,

$$\begin{aligned}(\mathbf{vrai} \ u) \ v &= ((\lambda x.\lambda y.x) \ u) \ v \\ &\rightarrow_{\beta} (\lambda y.u) \ v \\ &\rightarrow_{\beta} u.\end{aligned}$$

De même,

$$\begin{aligned}(\mathbf{faux} \ u) \ v &= ((\lambda x.\lambda y.y) \ u) \ v \\ &\rightarrow_{\beta} (\lambda y.y) \ v \\ &\rightarrow_{\beta} v.\end{aligned}$$

**Attention** : ce codage, tout comme ceux qui vont suivre, ne sont pas uniques. Ils ont simplement des propriétés qui font leur intérêt.

# Opérateurs booléens

En s'appuyant sur le codage précédent des booléens, on code les principaux opérateurs booléens.

# Opérateurs booléens

En s'appuyant sur le codage précédent des booléens, on code les principaux opérateurs booléens.

On code le « et » logique *et* de la manière suivante :

$$\mathit{et} := \lambda b.\lambda c.((b\ c)\ b).$$

# Opérateurs booléens

En s'appuyant sur le codage précédent des booléens, on code les principaux opérateurs booléens.

On code le « et » logique *et* de la manière suivante :

$$\text{et} := \lambda b.\lambda c.((b\ c)\ b).$$

Par exemple,

$$(\text{et vrai})\ \text{vrai}$$

# Opérateurs booléens

En s'appuyant sur le codage précédent des booléens, on code les principaux opérateurs booléens.

On code le « **et** » logique **et** de la manière suivante :

$$\mathbf{et} := \lambda b.\lambda c.((b\ c)\ b).$$

Par exemple,

$$(\mathbf{et\ vrai})\ \mathbf{vrai} = ((\lambda b.\lambda c.((b\ c)\ b))(\lambda x.\lambda y.x))(\lambda x.\lambda y.x)$$

# Opérateurs booléens

En s'appuyant sur le codage précédent des booléens, on code les principaux opérateurs booléens.

On code le « et » logique **et** de la manière suivante :

$$\mathbf{et} := \lambda b.\lambda c.((b\ c)\ b).$$

Par exemple,

$$\begin{aligned} (\mathbf{et\ vrai})\ \mathbf{vrai} &= ((\lambda b.\lambda c.((b\ c)\ b))(\lambda x.\lambda y.x))(\lambda x.\lambda y.x) \\ &\rightarrow_{\beta} (\lambda c.(((\lambda x.\lambda y.x)\ c)\ (\lambda x.\lambda y.x)))(\lambda x.\lambda y.x) \end{aligned}$$

# Opérateurs booléens

En s'appuyant sur le codage précédent des booléens, on code les principaux opérateurs booléens.

On code le « **et** » logique **et** de la manière suivante :

$$\mathbf{et} := \lambda b.\lambda c.((b\ c)\ b).$$

Par exemple,

$$\begin{aligned}(\mathbf{et\ vrai})\ \mathbf{vrai} &= ((\lambda b.\lambda c.((b\ c)\ b))(\lambda x.\lambda y.x))(\lambda x.\lambda y.x) \\ &\rightarrow_{\beta} (\lambda c.(((\lambda x.\lambda y.x)\ c)\ (\lambda x.\lambda y.x)))(\lambda x.\lambda y.x) \\ &\rightarrow_{\beta} (\lambda c.((\lambda y.c)\ (\lambda x.\lambda y.x)))(\lambda x.\lambda y.x)\end{aligned}$$

# Opérateurs booléens

En s'appuyant sur le codage précédent des booléens, on code les principaux opérateurs booléens.

On code le « et » logique **et** de la manière suivante :

$$\mathbf{et} := \lambda b.\lambda c.((b\ c)\ b).$$

Par exemple,

$$\begin{aligned}(\mathbf{et\ vrai})\ \mathbf{vrai} &= ((\lambda b.\lambda c.((b\ c)\ b))(\lambda x.\lambda y.x))(\lambda x.\lambda y.x) \\ &\rightarrow_{\beta} (\lambda c.(((\lambda x.\lambda y.x)\ c)\ (\lambda x.\lambda y.x)))(\lambda x.\lambda y.x) \\ &\rightarrow_{\beta} (\lambda c.((\lambda y.c)\ (\lambda x.\lambda y.x)))(\lambda x.\lambda y.x) \\ &\rightarrow_{\beta} (\lambda c.c)(\lambda x.\lambda y.x)\end{aligned}$$

# Opérateurs booléens

En s'appuyant sur le codage précédent des booléens, on code les principaux opérateurs booléens.

On code le « **et** » logique **et** de la manière suivante :

$$\mathbf{et} := \lambda b.\lambda c.((b\ c)\ b).$$

Par exemple,

$$\begin{aligned}(\mathbf{et\ vrai})\ \mathbf{vrai} &= ((\lambda b.\lambda c.((b\ c)\ b))(\lambda x.\lambda y.x))(\lambda x.\lambda y.x) \\ &\rightarrow_{\beta} (\lambda c.(((\lambda x.\lambda y.x)\ c)\ (\lambda x.\lambda y.x)))(\lambda x.\lambda y.x) \\ &\rightarrow_{\beta} (\lambda c.((\lambda y.c)\ (\lambda x.\lambda y.x)))(\lambda x.\lambda y.x) \\ &\rightarrow_{\beta} (\lambda c.c)(\lambda x.\lambda y.x) \\ &\rightarrow_{\beta} \lambda x.\lambda y.x\end{aligned}$$

# Opérateurs booléens

En s'appuyant sur le codage précédent des booléens, on code les principaux opérateurs booléens.

On code le « **et** » logique **et** de la manière suivante :

$$\mathbf{et} := \lambda b.\lambda c.((b\ c)\ b).$$

Par exemple,

$$\begin{aligned}(\mathbf{et\ vrai})\ \mathbf{vrai} &= ((\lambda b.\lambda c.((b\ c)\ b))(\lambda x.\lambda y.x))(\lambda x.\lambda y.x) \\ &\rightarrow_{\beta} (\lambda c.(((\lambda x.\lambda y.x)\ c)\ (\lambda x.\lambda y.x)))(\lambda x.\lambda y.x) \\ &\rightarrow_{\beta} (\lambda c.((\lambda y.c)\ (\lambda x.\lambda y.x)))(\lambda x.\lambda y.x) \\ &\rightarrow_{\beta} (\lambda c.c)(\lambda x.\lambda y.x) \\ &\rightarrow_{\beta} \lambda x.\lambda y.x \\ &= \mathbf{vrai}.\end{aligned}$$

# Opérateurs booléens

On code le « ou » logique `ou` de la manière suivante :

$$\text{ou} := \lambda b. \lambda c. ((b b) c).$$

# Opérateurs booléens

On code le « ou » logique **ou** de la manière suivante :

$$\mathbf{ou} := \lambda b.\lambda c.((b\ b)\ c).$$

On code le « non » logique **non** de la manière suivante :

$$\mathbf{non} := \lambda b.\lambda x.\lambda y.((b\ y)\ x).$$

# Opérateurs booléens

On code le « ou » logique **ou** de la manière suivante :

$$\mathbf{ou} := \lambda b.\lambda c.((b\ b)\ c).$$

On code le « non » logique **non** de la manière suivante :

$$\mathbf{non} := \lambda b.\lambda x.\lambda y.((b\ y)\ x).$$

On code l'implication logique **imp** de la manière suivante :

$$\mathbf{imp} := \lambda a.\lambda b.((\mathbf{ou}\ (\mathbf{non}\ a))\ b).$$

# Constructions conditionnelles

En s'appuyant sur le codage précédent des booléens, on code la **construction conditionnelle** `sas` (« si alors sinon ») de la manière suivante :

$$\text{sas} := \lambda x.\lambda y.\lambda z.((x\ y)\ z).$$

# Constructions conditionnelles

En s'appuyant sur le codage précédent des booléens, on code la **construction conditionnelle** `sas` (« si alors sinon ») de la manière suivante :

$$\text{sas} := \lambda x. \lambda y. \lambda z. ((x\ y)\ z).$$

Nous avons, si  $u$  et  $v$  sont deux  $\lambda$ -termes,

$$((\text{sas}\ \text{vrai})\ u)\ v$$

# Constructions conditionnelles

En s'appuyant sur le codage précédent des booléens, on code la **construction conditionnelle** `sas` (« si alors sinon ») de la manière suivante :

$$\text{sas} := \lambda x.\lambda y.\lambda z.((x\ y)\ z).$$

Nous avons, si  $u$  et  $v$  sont deux  $\lambda$ -termes,

$$((\text{sas}\ \text{vrai})\ u)\ v = (((\lambda x.\lambda y.\lambda z.((x\ y)\ z))\ \text{vrai})\ u)\ v$$

# Constructions conditionnelles

En s'appuyant sur le codage précédent des booléens, on code la **construction conditionnelle** `sas` (« si alors sinon ») de la manière suivante :

$$\text{sas} := \lambda x. \lambda y. \lambda z. ((x\ y)\ z).$$

Nous avons, si  $u$  et  $v$  sont deux  $\lambda$ -termes,

$$\begin{aligned} ((\text{sas}\ \text{vrai})\ u)\ v &= (((\lambda x. \lambda y. \lambda z. ((x\ y)\ z))\ \text{vrai})\ u)\ v \\ &\rightarrow_{\beta} ((\lambda y. \lambda z. ((\text{vrai}\ y)\ z))\ u)\ v \end{aligned}$$

# Constructions conditionnelles

En s'appuyant sur le codage précédent des booléens, on code la **construction conditionnelle** `sas` (« si alors sinon ») de la manière suivante :

$$\text{sas} := \lambda x.\lambda y.\lambda z.((x\ y)\ z).$$

Nous avons, si  $u$  et  $v$  sont deux  $\lambda$ -termes,

$$\begin{aligned} ((\text{sas}\ \text{vrai})\ u)\ v &= (((\lambda x.\lambda y.\lambda z.((x\ y)\ z))\ \text{vrai})\ u)\ v \\ &\rightarrow_{\beta} ((\lambda y.\lambda z.((\text{vrai}\ y)\ z))\ u)\ v \\ &\rightarrow_{\beta} ((\lambda y.\lambda z.y)\ u)\ v \end{aligned}$$

# Constructions conditionnelles

En s'appuyant sur le codage précédent des booléens, on code la **construction conditionnelle** `sas` (« si alors sinon ») de la manière suivante :

$$\text{sas} := \lambda x.\lambda y.\lambda z.((x\ y)\ z).$$

Nous avons, si  $u$  et  $v$  sont deux  $\lambda$ -termes,

$$\begin{aligned} ((\text{sas}\ \text{vrai})\ u)\ v &= (((\lambda x.\lambda y.\lambda z.((x\ y)\ z))\ \text{vrai})\ u)\ v \\ &\rightarrow_{\beta} ((\lambda y.\lambda z.((\text{vrai}\ y)\ z))\ u)\ v \\ &\rightarrow_{\beta} ((\lambda y.\lambda z.y)\ u)\ v \\ &\rightarrow_{\beta} (\lambda z.u)\ v \end{aligned}$$

# Constructions conditionnelles

En s'appuyant sur le codage précédent des booléens, on code la **construction conditionnelle** `sas` (« si alors sinon ») de la manière suivante :

$$\text{sas} := \lambda x. \lambda y. \lambda z. ((x\ y)\ z).$$

Nous avons, si  $u$  et  $v$  sont deux  $\lambda$ -termes,

$$\begin{aligned} ((\text{sas}\ \text{vrai})\ u)\ v &= (((\lambda x. \lambda y. \lambda z. ((x\ y)\ z))\ \text{vrai})\ u)\ v \\ &\rightarrow_{\beta} ((\lambda y. \lambda z. ((\text{vrai}\ y)\ z))\ u)\ v \\ &\rightarrow_{\beta} ((\lambda y. \lambda z. y)\ u)\ v \\ &\rightarrow_{\beta} (\lambda z. u)\ v \\ &\rightarrow_{\beta} u. \end{aligned}$$

# Constructions conditionnelles

En s'appuyant sur le codage précédent des booléens, on code la **construction conditionnelle** `sas` (« si alors sinon ») de la manière suivante :

$$\text{sas} := \lambda x. \lambda y. \lambda z. ((x\ y)\ z).$$

Nous avons, si  $u$  et  $v$  sont deux  $\lambda$ -termes,

$$\begin{aligned} ((\text{sas}\ \text{vrai})\ u)\ v &= (((\lambda x. \lambda y. \lambda z. ((x\ y)\ z))\ \text{vrai})\ u)\ v \\ &\rightarrow_{\beta} ((\lambda y. \lambda z. ((\text{vrai}\ y)\ z))\ u)\ v \\ &\rightarrow_{\beta} ((\lambda y. \lambda z. y)\ u)\ v \\ &\rightarrow_{\beta} (\lambda z. u)\ v \\ &\rightarrow_{\beta} u. \end{aligned}$$

De même, on obtient

$$((\text{sas}\ \text{faux})\ u)\ v \rightarrow_{\beta} v.$$

# Constructions conditionnelles

En s'appuyant sur le codage précédent des booléens, on code la **construction conditionnelle** `sas` (« si alors sinon ») de la manière suivante :

$$\text{sas} := \lambda x. \lambda y. \lambda z. ((x\ y)\ z).$$

Nous avons, si  $u$  et  $v$  sont deux  $\lambda$ -termes,

$$\begin{aligned} ((\text{sas}\ \text{vrai})\ u)\ v &= (((\lambda x. \lambda y. \lambda z. ((x\ y)\ z))\ \text{vrai})\ u)\ v \\ &\rightarrow_{\beta} ((\lambda y. \lambda z. ((\text{vrai}\ y)\ z))\ u)\ v \\ &\rightarrow_{\beta} ((\lambda y. \lambda z. y)\ u)\ v \\ &\rightarrow_{\beta} (\lambda z. u)\ v \\ &\rightarrow_{\beta} u. \end{aligned}$$

De même, on obtient

$$((\text{sas}\ \text{faux})\ u)\ v \rightarrow_{\beta} v.$$

Ainsi, la valeur calculée par `sas`, appliquée à trois arguments, possède le comportement attendu.

# Entiers naturels

On code tout **entier naturel**  $n$  par  $\text{ent}(n)$  défini de la manière suivante :

$$\text{ent}(n) := \begin{cases} \lambda f. \lambda x. x & \text{si } n = 0, \\ \lambda f. \lambda x. (f ((\text{ent}(n - 1) f) x)) & \text{sinon.} \end{cases}$$

# Entiers naturels

On code tout **entier naturel**  $n$  par  $\text{ent}(n)$  défini de la manière suivante :

$$\text{ent}(n) := \begin{cases} \lambda f. \lambda x. x & \text{si } n = 0, \\ \lambda f. \lambda x. (f ((\text{ent}(n - 1) f) x)) & \text{sinon.} \end{cases}$$

Voici les 1<sup>ers</sup>  $\lambda$ -termes codant les petits entiers :

# Entiers naturels

On code tout **entier naturel**  $n$  par  $\text{ent}(n)$  défini de la manière suivante :

$$\text{ent}(n) := \begin{cases} \lambda f.\lambda x.x & \text{si } n = 0, \\ \lambda f.\lambda x.(f ((\text{ent}(n - 1) f) x)) & \text{sinon.} \end{cases}$$

Voici les 1<sup>ers</sup>  $\lambda$ -termes codant les petits entiers :

$$\text{ent}(0) = \lambda f.\lambda x.x,$$

# Entiers naturels

On code tout **entier naturel**  $n$  par  $\text{ent}(n)$  défini de la manière suivante :

$$\text{ent}(n) := \begin{cases} \lambda f. \lambda x. x & \text{si } n = 0, \\ \lambda f. \lambda x. (f ((\text{ent}(n - 1) f) x)) & \text{sinon.} \end{cases}$$

Voici les 1<sup>ers</sup>  $\lambda$ -termes codant les petits entiers :

$$\text{ent}(0) = \lambda f. \lambda x. x,$$

$$\text{ent}(1) = \lambda f. \lambda x. (f x),$$

# Entiers naturels

On code tout **entier naturel**  $n$  par  $\text{ent}(n)$  défini de la manière suivante :

$$\text{ent}(n) := \begin{cases} \lambda f.\lambda x.x & \text{si } n = 0, \\ \lambda f.\lambda x.(f ((\text{ent}(n - 1) f) x)) & \text{sinon.} \end{cases}$$

Voici les 1<sup>ers</sup>  $\lambda$ -termes codant les petits entiers :

$$\text{ent}(0) = \lambda f.\lambda x.x,$$

$$\text{ent}(1) = \lambda f.\lambda x.(f x),$$

$$\text{ent}(2) = \lambda f.\lambda x.(f (f x)),$$

# Entiers naturels

On code tout **entier naturel**  $n$  par  $\text{ent}(n)$  défini de la manière suivante :

$$\text{ent}(n) := \begin{cases} \lambda f.\lambda x.x & \text{si } n = 0, \\ \lambda f.\lambda x.(f ((\text{ent}(n - 1) f) x)) & \text{sinon.} \end{cases}$$

Voici les 1<sup>ers</sup>  $\lambda$ -termes codant les petits entiers :

$$\text{ent}(0) = \lambda f.\lambda x.x,$$

$$\text{ent}(1) = \lambda f.\lambda x.(f x),$$

$$\text{ent}(2) = \lambda f.\lambda x.(f (f x)),$$

$$\text{ent}(3) = \lambda f.\lambda x.(f (f (f x))).$$

# Entiers naturels

On code tout **entier naturel**  $n$  par  $\text{ent}(n)$  défini de la manière suivante :

$$\text{ent}(n) := \begin{cases} \lambda f.\lambda x.x & \text{si } n = 0, \\ \lambda f.\lambda x.(f ((\text{ent}(n - 1) f) x)) & \text{sinon.} \end{cases}$$

Voici les 1<sup>ers</sup>  $\lambda$ -termes codant les petits entiers :

$$\text{ent}(0) = \lambda f.\lambda x.x,$$

$$\text{ent}(1) = \lambda f.\lambda x.(f x),$$

$$\text{ent}(2) = \lambda f.\lambda x.(f (f x)),$$

$$\text{ent}(3) = \lambda f.\lambda x.(f (f (f x))).$$

Intuitivement, un entier  $n$  est une fonction à deux paramètres  $f$  et  $x$  qui applique  $f$  à  $x$  de manière itérée  $n$  fois.

# Entiers naturels

On code tout **entier naturel**  $n$  par  $\text{ent}(n)$  défini de la manière suivante :

$$\text{ent}(n) := \begin{cases} \lambda f. \lambda x. x & \text{si } n = 0, \\ \lambda f. \lambda x. (f ((\text{ent}(n - 1) f) x)) & \text{sinon.} \end{cases}$$

Voici les 1<sup>ers</sup>  $\lambda$ -termes codant les petits entiers :

$$\text{ent}(0) = \lambda f. \lambda x. x,$$

$$\text{ent}(1) = \lambda f. \lambda x. (f x),$$

$$\text{ent}(2) = \lambda f. \lambda x. (f (f x)),$$

$$\text{ent}(3) = \lambda f. \lambda x. (f (f (f x))).$$

Intuitivement, un entier  $n$  est une fonction à deux paramètres  $f$  et  $x$  qui applique  $f$  à  $x$  de manière itérée  $n$  fois.

Ce codage est connu sous le nom d'**entiers de Church**.

# Arithmétique sur les entiers de Church

La fonction `succ` calcule le `successeur` d'un entier. Elle se code par

$$\text{succ} := \lambda n. \lambda f. \lambda x. (f ((n f) x)).$$

# Arithmétique sur les entiers de Church

La fonction **succ** calcule le **successeur** d'un entier. Elle se code par

$$\mathbf{succ} := \lambda n. \lambda f. \lambda x. (f ((n f) x)).$$

Intuitivement, **succ** est une fonction paramétrée par un entier de Church  $n$  qui renvoie l'entier de Church obtenu en appliquant  $f$  à  $x$  une fois supplémentaire dans  $n$ .

# Arithmétique sur les entiers de Church

La fonction **succ** calcule le **successeur** d'un entier. Elle se code par

$$\mathbf{succ} := \lambda n. \lambda f. \lambda x. (f ((n f) x)).$$

Intuitivement, **succ** est une fonction paramétrée par un entier de Church  $n$  qui renvoie l'entier de Church obtenu en appliquant  $f$  à  $x$  une fois supplémentaire dans  $n$ .

P.ex., calculons

$$\mathbf{succ} \text{ ent}(2)$$

# Arithmétique sur les entiers de Church

La fonction **succ** calcule le **successeur** d'un entier. Elle se code par

$$\mathbf{succ} := \lambda n. \lambda f. \lambda x. (f ((n f) x)).$$

Intuitivement, **succ** est une fonction paramétrée par un entier de Church  $n$  qui renvoie l'entier de Church obtenu en appliquant  $f$  à  $x$  une fois supplémentaire dans  $n$ .

P.ex., calculons

$$\mathbf{succ} \text{ent}(2) = (\lambda n. \lambda f. \lambda x. (f ((n f) x))) (\lambda f. \lambda x. (f (f x)))$$

# Arithmétique sur les entiers de Church

La fonction **succ** calcule le **successeur** d'un entier. Elle se code par

$$\mathbf{succ} := \lambda n. \lambda f. \lambda x. (f ((n f) x)).$$

Intuitivement, **succ** est une fonction paramétrée par un entier de Church  $n$  qui renvoie l'entier de Church obtenu en appliquant  $f$  à  $x$  une fois supplémentaire dans  $n$ .

P.ex., calculons

$$\begin{aligned} \mathbf{succ} \text{ent}(2) &= (\lambda n. \lambda f. \lambda x. (f ((n f) x))) (\lambda f. \lambda x. (f (f x))) \\ &\rightarrow_{\beta} \lambda f. \lambda x. (f (((\lambda f. \lambda x. (f (f x))) f) x)) \end{aligned}$$

# Arithmétique sur les entiers de Church

La fonction **succ** calcule le **successeur** d'un entier. Elle se code par

$$\mathbf{succ} := \lambda n. \lambda f. \lambda x. (f ((n f) x)).$$

Intuitivement, **succ** est une fonction paramétrée par un entier de Church  $n$  qui renvoie l'entier de Church obtenu en appliquant  $f$  à  $x$  une fois supplémentaire dans  $n$ .

P.ex., calculons

$$\begin{aligned} \mathbf{succ} \text{ent}(2) &= (\lambda n. \lambda f. \lambda x. (f ((n f) x))) (\lambda f. \lambda x. (f (f x))) \\ &\rightarrow_{\beta} \lambda f. \lambda x. (f (((\lambda f. \lambda x. (f (f x))) f) x)) \\ &\rightarrow_{\beta} \lambda f. \lambda x. (f (\lambda x. (f (f x)) x)) \end{aligned}$$

# Arithmétique sur les entiers de Church

La fonction **succ** calcule le **successeur** d'un entier. Elle se code par

$$\mathbf{succ} := \lambda n. \lambda f. \lambda x. (f ((n f) x)).$$

Intuitivement, **succ** est une fonction paramétrée par un entier de Church  $n$  qui renvoie l'entier de Church obtenu en appliquant  $f$  à  $x$  une fois supplémentaire dans  $n$ .

P.ex., calculons

$$\begin{aligned} \mathbf{succ} \text{ent}(2) &= (\lambda n. \lambda f. \lambda x. (f ((n f) x))) (\lambda f. \lambda x. (f (f x))) \\ &\rightarrow_{\beta} \lambda f. \lambda x. (f (((\lambda f. \lambda x. (f (f x))) f) x)) \\ &\rightarrow_{\beta} \lambda f. \lambda x. (f (\lambda x. (f (f x)) x)) \\ &\rightarrow_{\beta} \lambda f. \lambda x. (f (f (f x))) \end{aligned}$$

# Arithmétique sur les entiers de Church

La fonction **succ** calcule le **successeur** d'un entier. Elle se code par

$$\mathbf{succ} := \lambda n. \lambda f. \lambda x. (f ((n f) x)).$$

Intuitivement, **succ** est une fonction paramétrée par un entier de Church  $n$  qui renvoie l'entier de Church obtenu en appliquant  $f$  à  $x$  une fois supplémentaire dans  $n$ .

P.ex., calculons

$$\begin{aligned} \mathbf{succ} \text{ent}(2) &= (\lambda n. \lambda f. \lambda x. (f ((n f) x))) (\lambda f. \lambda x. (f (f x))) \\ &\rightarrow_{\beta} \lambda f. \lambda x. (f (((\lambda f. \lambda x. (f (f x))) f) x)) \\ &\rightarrow_{\beta} \lambda f. \lambda x. (f (\lambda x. (f (f x)) x)) \\ &\rightarrow_{\beta} \lambda f. \lambda x. (f (f (f x))) \\ &= \text{ent}(3). \end{aligned}$$

# Arithmétique sur les entiers de Church

La fonction `add` calcule la **somme** de deux entiers.

# Arithmétique sur les entiers de Church

La fonction `add` calcule la **somme** de deux entiers. Elle se code par

$$\text{add} := \lambda n. \lambda m. \lambda f. \lambda x. ((n f) ((m f) x)).$$

# Arithmétique sur les entiers de Church

La fonction `add` calcule la **somme** de deux entiers. Elle se code par

$$\text{add} := \lambda n. \lambda m. \lambda f. \lambda x. ((n f) ((m f) x)).$$

La fonction `mul` calcule le **produit** de deux entiers.

# Arithmétique sur les entiers de Church

La fonction `add` calcule la **somme** de deux entiers. Elle se code par

$$\text{add} := \lambda n. \lambda m. \lambda f. \lambda x. ((n f) ((m f) x)).$$

La fonction `mul` calcule le **produit** de deux entiers. Elle se code par

$$\text{mul} := \lambda n. \lambda m. \lambda f. (n (m f)).$$

# Arithmétique sur les entiers de Church

La fonction `add` calcule la **somme** de deux entiers. Elle se code par

$$\text{add} := \lambda n. \lambda m. \lambda f. \lambda x. ((n f) ((m f) x)).$$

La fonction `mul` calcule le **produit** de deux entiers. Elle se code par

$$\text{mul} := \lambda n. \lambda m. \lambda f. (n (m f)).$$

La fonction `pui` calcule l'**exponentiation** de deux entiers.

# Arithmétique sur les entiers de Church

La fonction `add` calcule la **somme** de deux entiers. Elle se code par

$$\text{add} := \lambda n. \lambda m. \lambda f. \lambda x. ((n f) ((m f) x)).$$

La fonction `mul` calcule le **produit** de deux entiers. Elle se code par

$$\text{mul} := \lambda n. \lambda m. \lambda f. (n (m f)).$$

La fonction `pui` calcule l'**exponentiation** de deux entiers. Elle se code par

$$\text{pui} := \lambda n. \lambda m. (m n).$$

# Récurtivité

Pour écrire des fonctions récursives, il est nécessaire de mettre en place un mécanisme de **réplication** de la fonction en question.

# Récurtivité

Pour écrire des fonctions récursives, il est nécessaire de mettre en place un mécanisme de **réplication** de la fonction en question.

On utilise pour cela le **combinateur de Curry**  $\rho$  défini par

$$\rho := \lambda f.(\lambda x.((f (x x))) (\lambda x.(f (x x))))).$$

# Récurtivité

Pour écrire des fonctions récursives, il est nécessaire de mettre en place un mécanisme de **réplication** de la fonction en question.

On utilise pour cela le **combinateur de Curry**  $\rho$  défini par

$$\rho := \lambda f.(\lambda x.((f (x x))) (\lambda x.(f (x x)))).$$

Il vérifie la propriété fondamentale suivante : pour tout  $\lambda$ -terme  $u$ ,

$$\rho u \rightarrow_{\beta} u(\rho u).$$

# Récurtivité

Pour écrire des fonctions récursives, il est nécessaire de mettre en place un mécanisme de **réplication** de la fonction en question.

On utilise pour cela le **combinateur de Curry**  $\rho$  défini par

$$\rho := \lambda f.(\lambda x.((f (x x))) (\lambda x.(f (x x)))).$$

Il vérifie la propriété fondamentale suivante : pour tout  $\lambda$ -terme  $u$ ,

$$\rho u \rightarrow_{\beta} u(\rho u).$$

Ceci implique

$$\rho u \rightarrow_{\beta} u(\rho u) \rightarrow_{\beta} u(u(\rho u)) \rightarrow_{\beta} u(u(u(\rho u))) \rightarrow_{\beta} \dots ,$$

offrant un moyen d'appliquer de manière itérée le  $\lambda$ -terme (la fonction)  $u$ .

# Plan

## $\lambda$ -calcul

$\lambda$ -termes

Codage

Implantation

# Représentation de $\lambda$ -termes

On commence par définir un type pour représenter les variables :

```
type variable = {nom : string ; ref : int};;
```

# Représentation de $\lambda$ -termes

On commence par définir un type pour représenter les variables :

```
type variable = {nom : string ; ref : int};;
```

Le champ `nom` contient le nom de la variable tandis que le champ `ref` contient un entier qui permet de distinguer des variables qui ont un même nom (pour le mécanisme d' $\alpha$ -renommage en particulier).

# Représentation de $\lambda$ -termes

On commence par définir un type pour représenter les variables :

```
type variable = {nom : string ; ref : int};;
```

Le champ `nom` contient le nom de la variable tandis que le champ `ref` contient un entier qui permet de distinguer des variables qui ont un même nom (pour le mécanisme d' $\alpha$ -renommage en particulier).

Le type pour représenter les  $\lambda$ -termes est un type somme :

```
type terme =  
  | Variable of variable  
  | Abstraction of variable * terme  
  | Application of terme * terme;;
```

# Représentation de $\lambda$ -termes

On commence par définir un type pour représenter les variables :

```
type variable = {nom : string ; ref : int};;
```

Le champ `nom` contient le nom de la variable tandis que le champ `ref` contient un entier qui permet de distinguer des variables qui ont un même nom (pour le mécanisme d' $\alpha$ -renommage en particulier).

Le type pour représenter les  $\lambda$ -termes est un type somme :

```
type terme =  
  | Variable of variable  
  | Abstraction of variable * terme  
  | Application of terme * terme;;
```

La définition de ce type `terme` est conforme à la définition des  $\lambda$ -termes (récursive et en trois parties).

# Variables libres/liées

Cette fonction renvoie la liste des variables  $x$  qui admettent au moins une occurrence libre :

```
let rec variables_libres t =  
  match t with  
  |Variable v -> [v]  
  |Abstraction (v, t') ->  
    List.filter (fun x -> x <> v) (variables_libres t')  
  |Application (t', t'') ->  
    List.append (variables_libres t') (variables_libres t'')
```

# Variables libres/liées

Cette fonction renvoie la liste des variables  $x$  qui admettent au moins une occurrence libre :

```
let rec variables_libres t =  
  match t with  
  | Variable v -> [v]  
  | Abstraction (v, t') ->  
    List.filter (fun x -> x <> v) (variables_libres t')  
  | Application (t', t'') ->  
    List.append (variables_libres t') (variables_libres t'')
```

P.ex., cette fonction appliquée au  $\lambda$ -terme

$$\lambda x. \lambda x. ((x (\lambda x. x)) (y (\lambda y. y)))$$

donne

```
# let t1 = Abstraction (x, Abstraction (x,  
  Application (Application  
    (Variable x, Abstraction (x, Variable x)),  
  Application  
    (Variable y, Abstraction (y, Variable y))))))  
in  
variables_libres t1;;  
- : variable list = [nom = "y"; ref = 1]
```