Déconstruction

L'opérateur de construction est également un opérateur de **déconstruction** lorsqu'on s'en sert avec un filtrage de motifs.

P.ex.,

```
# let tete lst =
      match lst with
          |[] -> (failwith "liste vide")
          |e :: _ -> e;;
val tete : 'a list -> 'a = <fun>
```

déconstruit la liste en argument pour accéder à son 1er élément.

```
# let rec un_sur_deux lst =
      match lst with
          |[] -> []
          |[e] -> [e]
          |e1 :: e2 :: reste -> e1 :: (un_sur_deux reste);;
val un_sur_deux : 'a list -> 'a list = <fun>
```

renvoie la liste des éléments pris un sur deux à partir de la liste en argument.

Note : ce sont des fonctions polymorphes.

Fonctions élémentaires

La librairie de CAML propose dans le module `List` diverses fonctions de manipulation de listes.

Exercice : donner (en examinant leurs implantations) les complexités en temps et en espace de ces fonctions relativement au nombre d'éléments dans les listes impliquées.

Fonctions élémentaires

La librairie de CAML propose dans le module `List` diverses fonctions de manipulation de listes. Parmi elles :

Fonction	Type	Valeur renvoyée
`hd`	`'a list -> 'a`	Tête de la liste

Exercice : donner (en examinant leurs implantations) les complexités en temps et en espace de ces fonctions relativement au nombre d'éléments dans les listes impliquées.

Fonctions élémentaires

La librairie de CAML propose dans le module `List` diverses fonctions de manipulation de listes. Parmi elles :

Fonction	Type	Valeur renvoyée
`hd`	`'a list -> 'a`	Tête de la liste
`tl`	`'a list -> 'a list`	Liste privée de sa tête

Exercice : donner (en examinant leurs implantations) les complexités en temps et en espace de ces fonctions relativement au nombre d'éléments dans les listes impliquées.

Fonctions élémentaires

La librairie de CAML propose dans le module `List` diverses fonctions de manipulation de listes. Parmi elles :

Fonction	Type	Valeur renvoyée
`hd`	`'a list -> 'a`	Tête de la liste
`tl`	`'a list -> 'a list`	Liste privée de sa tête
`nth`	`'a list -> int -> 'a`	L'élément de la liste à l'indice donné

Exercice : donner (en examinant leurs implantations) les complexités en temps et en espace de ces fonctions relativement au nombre d'éléments dans les listes impliquées.

Fonctions élémentaires

La librairie de CAML propose dans le module `List` diverses fonctions de manipulation de listes. Parmi elles :

Fonction	Type	Valeur renvoyée
`hd`	`'a list -> 'a`	Tête de la liste
`tl`	`'a list -> 'a list`	Liste privée de sa tête
`nth`	`'a list -> int -> 'a`	L'élément de la liste à l'indice donné
`length`	`'a list -> int`	longueur de la liste

Exercice : donner (en examinant leurs implantations) les complexités en temps et en espace de ces fonctions relativement au nombre d'éléments dans les listes impliquées.

Fonctions élémentaires

La librairie de CAML propose dans le module `List` diverses fonctions de manipulation de listes. Parmi elles :

Fonction	Type	Valeur renvoyée
`hd`	`'a list -> 'a`	Tête de la liste
`tl`	`'a list -> 'a list`	Liste privée de sa tête
`nth`	`'a list -> int -> 'a`	L'élément de la liste à l'indice donné
`length`	`'a list -> int`	longueur de la liste
`mem`	`'a -> 'a list -> bool`	Présence de l'élément dans la liste

Exercice : donner (en examinant leurs implantations) les complexités en temps et en espace de ces fonctions relativement au nombre d'éléments dans les listes impliquées.

Fonctions élémentaires

La librairie de CAML propose dans le module `List` diverses fonctions de manipulation de listes. Parmi elles :

Fonction	Type	Valeur renvoyée
`hd`	`'a list -> 'a`	Tête de la liste
`tl`	`'a list -> 'a list`	Liste privée de sa tête
`nth`	`'a list -> int -> 'a`	L'élément de la liste à l'indice donné
`length`	`'a list -> int`	longueur de la liste
`mem`	`'a -> 'a list -> bool`	Présence de l'élément dans la liste
`rev`	`'a list -> 'a list`	Liste miroir

Exercice : donner (en examinant leurs implantations) les complexités en temps et en espace de ces fonctions relativement au nombre d'éléments dans les listes impliquées.

Fonctions élémentaires

La librairie de CAML propose dans le module `List` diverses fonctions de manipulation de listes. Parmi elles :

Fonction	Type	Valeur renvoyée
`hd`	`'a list -> 'a`	Tête de la liste
`tl`	`'a list -> 'a list`	Liste privée de sa tête
`nth`	`'a list -> int -> 'a`	L'élément de la liste à l'indice donné
`length`	`'a list -> int`	longueur de la liste
`mem`	`'a -> 'a list -> bool`	Présence de l'élément dans la liste
`rev`	`'a list -> 'a list`	Liste miroir
`append`	`'a list -> 'a list -> 'a list`	Concaténation des deux listes

Exercice : donner (en examinant leurs implantations) les complexités en temps et en espace de ces fonctions relativement au nombre d'éléments dans les listes impliquées.

Opérations habituelles sur les listes

La plupart des opérations réalisées en pratique sur les listes rentrent dans l'une des catégories suivantes :

Opérations habituelles sur les listes

La plupart des opérations réalisées en pratique sur les listes rentrent dans l'une des catégories suivantes :

1. **transformer** les éléments d'une liste ;

Opérations habituelles sur les listes

La plupart des opérations réalisées en pratique sur les listes rentrent dans l'une des catégories suivantes :

1. **transformer** les éléments d'une liste ;
2. **sélectionner** les éléments d'une liste qui vérifient une propriété ;

Opérations habituelles sur les listes

La plupart des opérations réalisées en pratique sur les listes rentrent dans l'une des catégories suivantes :

1. **transformer** les éléments d'une liste ;
2. **sélectionner** les éléments d'une liste qui vérifient une propriété ;
3. **tester** si tous les (resp. au moins un) élément(s) d'une liste vérifie(nt) une propriété ;

Opérations habituelles sur les listes

La plupart des opérations réalisées en pratique sur les listes rentrent dans l'une des catégories suivantes :

1. **transformer** les éléments d'une liste ;
2. **sélectionner** les éléments d'une liste qui vérifient une propriété ;
3. **tester** si tous les (resp. au moins un) élément(s) d'une liste vérifie(nt) une propriété ;
4. **combiner** les éléments d'une liste pour calculer une valeur ;

Opérations habituelles sur les listes

La plupart des opérations réalisées en pratique sur les listes rentrent dans l'une des catégories suivantes :

1. **transformer** les éléments d'une liste ;
2. **sélectionner** les éléments d'une liste qui vérifient une propriété ;
3. **tester** si tous les (resp. au moins un) élément(s) d'une liste vérifie(nt) une propriété ;
4. **combiner** les éléments d'une liste pour calculer une valeur ;
5. **permuter** les éléments d'une liste.

Opérations habituelles sur les listes

Exemples :

1. **[transformation]** multiplier les éléments d'une liste d'entiers par 3, convertir une liste de caractères en la liste des codes ASCII correspondants ;

Opérations habituelles sur les listes

Exemples :

1. **[transformation]** multiplier les éléments d'une liste d'entiers par 3, convertir une liste de caractères en la liste des codes ASCII correspondants ;

2. **[sélection]** obtenir la sous-liste des nombres pairs d'une liste d'entiers ;

Opérations habituelles sur les listes

Exemples :

1. **[transformation]** multiplier les éléments d'une liste d'entiers par 3, convertir une liste de caractères en la liste des codes ASCII correspondants ;

2. **[sélection]** obtenir la sous-liste des nombres pairs d'une liste d'entiers ;

3. **[test]** tester si toutes les chaînes de caractères contenues dans une liste commencent par `'a'`, tester la présence de l'élément 7 dans une liste d'entiers ;

Opérations habituelles sur les listes

Exemples :

1. **[transformation]** multiplier les éléments d'une liste d'entiers par `3`, convertir une liste de caractères en la liste des codes ASCII correspondants ;

2. **[sélection]** obtenir la sous-liste des nombres pairs d'une liste d'entiers ;

3. **[test]** tester si toutes les chaînes de caractères contenues dans une liste commencent par `'a'`, tester la présence de l'élément `7` dans une liste d'entiers ;

4. **[combinaison]** calculer la somme des éléments d'une liste d'entiers, extraire la plus grande valeur d'une liste d'entiers ;

Opérations habituelles sur les listes

Exemples :

1. **[transformation]** multiplier les éléments d'une liste d'entiers par `3`, convertir une liste de caractères en la liste des codes ASCII correspondants ;

2. **[sélection]** obtenir la sous-liste des nombres pairs d'une liste d'entiers ;

3. **[test]** tester si toutes les chaînes de caractères contenues dans une liste commencent par `'a'`, tester la présence de l'élément `7` dans une liste d'entiers ;

4. **[combinaison]** calculer la somme des éléments d'une liste d'entiers, extraire la plus grande valeur d'une liste d'entiers ;

5. **[permutation]** tri d'une liste, image miroir d'une liste.

Opérations habituelles sur les listes

Exemples :

1. **[transformation]** multiplier les éléments d'une liste d'entiers par 3, convertir une liste de caractères en la liste des codes ASCII correspondants ;

2. **[sélection]** obtenir la sous-liste des nombres pairs d'une liste d'entiers ;

3. **[test]** tester si toutes les chaînes de caractères contenues dans une liste commencent par `'a'`, tester la présence de l'élément 7 dans une liste d'entiers ;

4. **[combinaison]** calculer la somme des éléments d'une liste d'entiers, extraire la plus grande valeur d'une liste d'entiers ;

5. **[permutation]** tri d'une liste, image miroir d'une liste.

Tout ceci se fait à l'aide de fonctions d'ordre supérieur.

Transformation de listes

Une bonne manière de spécifier la manière de transformer les éléments d'une liste d'éléments de type `'a` consiste à donner une fonction `tr` de type `'a -> 'b` où `'b` est un type cible.

Transformation de listes

Une bonne manière de spécifier la manière de transformer les éléments d'une liste d'éléments de type `'a` consiste à donner une fonction `tr` de type `'a -> 'b` où `'b` est un type cible.

Ainsi, l'opération de transformation `transformer` est de type

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

Transformation de listes

Une bonne manière de spécifier la manière de transformer les éléments d'une liste d'éléments de type `'a` consiste à donner une fonction `tr` de type `'a -> 'b` où `'b` est un type cible.

Ainsi, l'opération de transformation `transformer` est de type

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

et sa définition est

```
let rec transformer tr lst =
    match lst with
        |[] -> []
        |e :: reste -> (tr e) :: (transformer tr reste);;
```

Transformation de listes

Une bonne manière de spécifier la manière de transformer les éléments d'une liste d'éléments de type `'a` consiste à donner une fonction `tr` de type `'a -> 'b` où `'b` est un type cible.

Ainsi, l'opération de transformation `transformer` est de type

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

et sa définition est

```
let rec transformer tr lst =
    match lst with
        |[] -> []
        |e :: reste -> (tr e) :: (transformer tr reste);;
```

Exemples d'utilisation :

```
# (transformer (fun x -> x * 3) [1 ; 2 ; 3 ; 4 ; 5 ; 6]);;
- : int list = [3; 6; 9; 12; 15; 18]
```

Transformation de listes

Une bonne manière de spécifier la manière de transformer les éléments d'une liste d'éléments de type `'a` consiste à donner une fonction `tr` de type `'a -> 'b` où `'b` est un type cible.

Ainsi, l'opération de transformation `transformer` est de type

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

et sa définition est

```
let rec transformer tr lst =
    match lst with
        |[] -> []
        |e :: reste -> (tr e) :: (transformer tr reste);;
```

Exemples d'utilisation :

```
# (transformer (fun x -> x * 3) [1 ; 2 ; 3 ; 4 ; 5 ; 6]);;
- : int list = [3; 6; 9; 12; 15; 18]

# (transformer int_of_char ['a' ; 'b' ; 'c' ; '1' ; '2' ; '3']);;
- : int list = [97; 98; 99; 49; 50; 51]
```

Transformation de listes

Une bonne manière de spécifier la manière de transformer les éléments d'une liste d'éléments de type `'a` consiste à donner une fonction `tr` de type `'a -> 'b` où `'b` est un type cible.

Ainsi, l'opération de transformation `transformer` est de type

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

et sa définition est

```
let rec transformer tr lst =
    match lst with
        |[] -> []
        |e :: reste -> (tr e) :: (transformer tr reste);;
```

Exemples d'utilisation :

```
# (transformer (fun x -> x * 3) [1 ; 2 ; 3 ; 4 ; 5 ; 6]);;
- : int list = [3; 6; 9; 12; 15; 18]

# (transformer int_of_char ['a' ; 'b' ; 'c' ; '1' ; '2' ; '3']);;
- : int list = [97; 98; 99; 49; 50; 51]
```

Nous avons réimplanté ici la fonction `map` du module `List`.

Sélection des éléments d'une liste

Une bonne manière de spécifier les éléments à garder d'une liste consiste à donner une fonction `sel` de type `'a -> bool`. Les éléments de type `'a` qui ont `true` pour image étant exactement ceux à conserver.

Sélection des éléments d'une liste

Une bonne manière de spécifier les éléments à garder d'une liste consiste à donner une fonction `sel` de type `'a -> bool`. Les éléments de type `'a` qui ont `true` pour image étant exactement ceux à conserver.

Ainsi, l'opération de sélection `selectionner` est de type

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

Sélection des éléments d'une liste

Une bonne manière de spécifier les éléments à garder d'une liste consiste à donner une fonction `sel` de type `'a -> bool`. Les éléments de type `'a` qui ont `true` pour image étant exactement ceux à conserver.

Ainsi, l'opération de sélection `selectionner` est de type

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

et sa définition est

```
let rec selectionner sel lst =
    match lst with
        |[] -> []
        |e :: reste ->
            let suite = (selectionner sel reste) in
            if (sel e) then e :: suite
            else suite;;
```

Sélection des éléments d'une liste

Une bonne manière de spécifier les éléments à garder d'une liste consiste à donner une fonction `sel` de type `'a -> bool`. Les éléments de type `'a` qui ont `true` pour image étant exactement ceux à conserver.

Ainsi, l'opération de sélection `selectionner` est de type

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

et sa définition est

```
let rec selectionner sel lst =
    match lst with
        |[] -> []
        |e :: reste ->
            let suite = (selectionner sel reste) in
            if (sel e) then e :: suite
            else suite;;
```

Exemple d'utilisation :

```
# (selectionner (fun x -> x mod 2 = 0) [13 ; 8 ; 9 ; 8 ; 6 ; 15 ; 2]);;
- : int list = [8; 8; 6; 2]
```

Sélection des éléments d'une liste

Une bonne manière de spécifier les éléments à garder d'une liste consiste à donner une fonction `sel` de type `'a -> bool`. Les éléments de type `'a` qui ont `true` pour image étant exactement ceux à conserver.

Ainsi, l'opération de sélection `selectionner` est de type

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

et sa définition est

```
let rec selectionner sel lst =
    match lst with
        |[] -> []
        |e :: reste ->
            let suite = (selectionner sel reste) in
            if (sel e) then e :: suite
            else suite;;
```

Exemple d'utilisation :

```
# (selectionner (fun x -> x mod 2 = 0) [13 ; 8 ; 9 ; 8 ; 6 ; 15 ; 2]);;
- : int list = [8; 8; 6; 2]
```

Nous avons réimplanté ici la fonction `filter` du module `List`.

Test universel des éléments d'une liste

Une bonne manière de spécifier les éléments qui vérifient une propriété donnée consiste à fournir une fonction `prop` de type `'a -> bool`. Les éléments de type `'a` qui ont `true` pour image étant exactement ceux vérifiant la propriété.

Test universel des éléments d'une liste

Une bonne manière de spécifier les éléments qui vérifient une propriété donnée consiste à fournir une fonction `prop` de type `'a -> bool`. Les éléments de type `'a` qui ont `true` pour image étant exactement ceux vérifiant la propriété.

Ainsi, l'opération `tester_univ` qui teste si tous les éléments de la liste vérifient la propriété est de type

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

Test universel des éléments d'une liste

Une bonne manière de spécifier les éléments qui vérifient une propriété donnée consiste à fournir une fonction `prop` de type `'a -> bool`. Les éléments de type `'a` qui ont `true` pour image étant exactement ceux vérifiant la propriété.

Ainsi, l'opération `tester_univ` qui teste si tous les éléments de la liste vérifient la propriété est de type

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

et sa définition est

```
let rec tester_univ prop lst =
    match lst with
        |[] -> true
        |x :: _ when (not (prop x)) -> false
        |_ :: reste -> (tester_univ prop reste);;
```

Test universel des éléments d'une liste

Une bonne manière de spécifier les éléments qui vérifient une propriété donnée consiste à fournir une fonction `prop` de type `'a -> bool`. Les éléments de type `'a` qui ont `true` pour image étant exactement ceux vérifiant la propriété.

Ainsi, l'opération `tester_univ` qui teste si tous les éléments de la liste vérifient la propriété est de type

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

et sa définition est

```
let rec tester_univ prop lst =
    match lst with
        |[] -> true
        |x :: _ when (not (prop x)) -> false
        |_ :: reste -> (tester_univ prop reste);;
```

Exemple d'utilisation :

```
# (tester_univ (fun u->u.[0]='a') ["a" ; "aba" ; "abacaba" ; "abacabadabacaba"]);;
- : bool = true
```

Test universel des éléments d'une liste

Une bonne manière de spécifier les éléments qui vérifient une propriété donnée consiste à fournir une fonction `prop` de type `'a -> bool`. Les éléments de type `'a` qui ont `true` pour image étant exactement ceux vérifiant la propriété.

Ainsi, l'opération `tester_univ` qui teste si tous les éléments de la liste vérifient la propriété est de type

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

et sa définition est

```
let rec tester_univ prop lst =
    match lst with
        |[] -> true
        |x :: _ when (not (prop x)) -> false
        |_ :: reste -> (tester_univ prop reste);;
```

Exemple d'utilisation :

```
# (tester_univ (fun u->u.[0]='a') ["a" ; "aba" ; "abacaba" ; "abacabadabacaba"]);;
- : bool = true
```

Nous avons réimplanté ici la fonction `for_all` du module `List`.

Test existentiel des éléments d'une liste

Une bonne manière de spécifier les éléments qui vérifient une propriété donnée consiste à fournir une fonction `prop` de type `'a -> bool`. Les éléments de type `'a` qui ont `true` pour image étant exactement ceux vérifiant la propriété.

Test existentiel des éléments d'une liste

Une bonne manière de spécifier les éléments qui vérifient une propriété donnée consiste à fournir une fonction `prop` de type `'a -> bool`. Les éléments de type `'a` qui ont `true` pour image étant exactement ceux vérifiant la propriété.

Ainsi, l'opération `tester_exist` qui test s'il existe un élément de la liste qui vérifie la propriété est de type

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

Test existentiel des éléments d'une liste

Une bonne manière de spécifier les éléments qui vérifient une propriété donnée consiste à fournir une fonction `prop` de type `'a -> bool`. Les éléments de type `'a` qui ont `true` pour image étant exactement ceux vérifiant la propriété.

Ainsi, l'opération `tester_exist` qui test s'il existe un élément de la liste qui vérifie la propriété est de type

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

et sa définition est

```
let rec tester_exist prop lst =
    match lst with
        |[] -> false
        |x :: _ when (prop x) -> true
        |_ :: reste -> (tester_exist prop reste);;
```

Test existentiel des éléments d'une liste

Une bonne manière de spécifier les éléments qui vérifient une propriété donnée consiste à fournir une fonction `prop` de type `'a -> bool`. Les éléments de type `'a` qui ont `true` pour image étant exactement ceux vérifiant la propriété.

Ainsi, l'opération `tester_exist` qui test s'il existe un élément de la liste qui vérifie la propriété est de type

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

et sa définition est

```
let rec tester_exist prop lst =
    match lst with
        |[] -> false
        |x :: _ when (prop x) -> true
        |_ :: reste -> (tester_exist prop reste);;
```

Exemple d'utilisation :

```
# (tester_exist (fun x -> x = 7) [0 ; 1 ; 1 ; 2 ; 3 ; 5 ; 8]);;
- : bool = false
```

Test existentiel des éléments d'une liste

Une bonne manière de spécifier les éléments qui vérifient une propriété donnée consiste à fournir une fonction `prop` de type `'a -> bool`. Les éléments de type `'a` qui ont `true` pour image étant exactement ceux vérifiant la propriété.

Ainsi, l'opération `tester_exist` qui test s'il existe un élément de la liste qui vérifie la propriété est de type

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

et sa définition est

```
let rec tester_exist prop lst =
    match lst with
        |[] -> false
        |x :: _ when (prop x) -> true
        |_ :: reste -> (tester_exist prop reste);;
```

Exemple d'utilisation :

```
# (tester_exist (fun x -> x = 7) [0 ; 1 ; 1 ; 2 ; 3 ; 5 ; 8]);;
- : bool = false
```

Nous avons réimplanté ici la fonction `exists` du module `List`.

Association des éléments d'une liste

Le problème est le suivant.

Association des éléments d'une liste

Le problème est le suivant. On dispose

1. d'un élément `gr` ;

Association des éléments d'une liste

Le problème est le suivant. On dispose

1. d'un élément `gr` ;
2. d'une liste `lst` de la forme `[e1 ; e2 ; ... ; en]` ;

Association des éléments d'une liste

Le problème est le suivant. On dispose

1. d'un élément `gr` ;
2. d'une liste `lst` de la forme `[e1 ; e2 ; ... ; en]` ;
3. d'une opération binaire associative $\times$;

Association des éléments d'une liste

Le problème est le suivant. On dispose

1. d'un élément `gr` ;
2. d'une liste `lst` de la forme `[e1 ; e2 ; ... ; en]` ;
3. d'une opération binaire associative $\times$;

et on souhaite calculer la valeur

$$\texttt{gr} \times \texttt{e1} \times \texttt{e2} \times \cdots \times \texttt{en}.$$

Association des éléments d'une liste

Le problème est le suivant. On dispose

1. d'un élément `gr` ;
2. d'une liste `lst` de la forme `[e1 ; e2 ; ... ; en]` ;
3. d'une opération binaire associative $\times$;

et on souhaite calculer la valeur

$$\texttt{gr} \times \texttt{e1} \times \texttt{e2} \times \cdots \times \texttt{en}.$$

Ce problème admet de nombreuses instances :

- lorsque `gr = 0`, `lst` est une liste d'entiers et $\times$ est la fonction d'**addition** des entiers, ceci calcule la somme des éléments de `lst` ;

Association des éléments d'une liste

Le problème est le suivant. On dispose

1. d'un élément `gr` ;
2. d'une liste `lst` de la forme `[e1 ; e2 ; ... ; en]` ;
3. d'une opération binaire associative $\times$;

et on souhaite calculer la valeur

$$\texttt{gr} \times \texttt{e1} \times \texttt{e2} \times \cdots \times \texttt{en}.$$

Ce problème admet de nombreuses instances :

- lorsque `gr = 0`, `lst` est une liste d'entiers et $\times$ est la fonction d'**addition** des entiers, ceci calcule la somme des éléments de `lst` ;
- lorsque `gr = ""`, `lst` est une liste de chaînes de caractères et $\times$ est l'opération de **concaténation**, ceci calcule de gauche à droite la concaténation des chaînes de caractères de `lst` ;

Association des éléments d'une liste

Le problème est le suivant. On dispose

1. d'un élément `gr` ;
2. d'une liste `lst` de la forme `[e1 ; e2 ; ... ; en]` ;
3. d'une opération binaire associative $\times$;

et on souhaite calculer la valeur

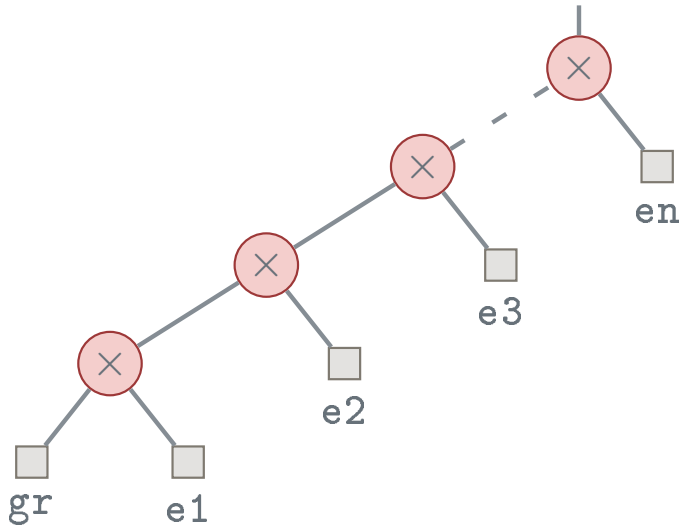
$$\texttt{gr} \times \texttt{e1} \times \texttt{e2} \times \cdots \times \texttt{en}.$$

Ce problème admet de nombreuses instances :

- lorsque `gr = 0`, `lst` est une liste d'entiers et $\times$ est la fonction d'**addition** des entiers, ceci calcule la somme des éléments de `lst` ;
- lorsque `gr = ""`, `lst` est une liste de chaînes de caractères et $\times$ est l'opération de **concaténation**, ceci calcule de gauche à droite la concaténation des chaînes de caractères de `lst` ;
- lorsque `gr = e1`, `lst` est une liste dont les éléments sont comparables et $\times$ est la fonction qui renvoie **le plus grand** de ses deux arguments, ceci calcule la plus grande valeur de `lst`.

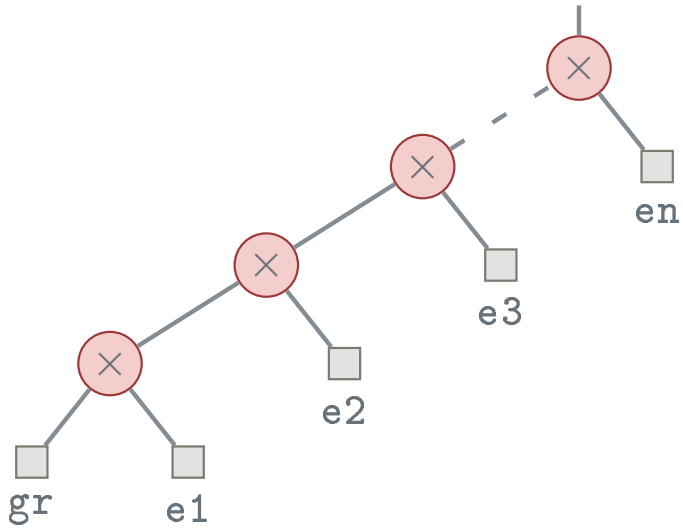
Pliage à gauche

En d'autres termes, étant donnée une liste `[e1 ; e2 ; e3 ; ... ; en]` et une opération $\times$, on souhaite évaluer l'arbre syntaxique



Pliage à gauche

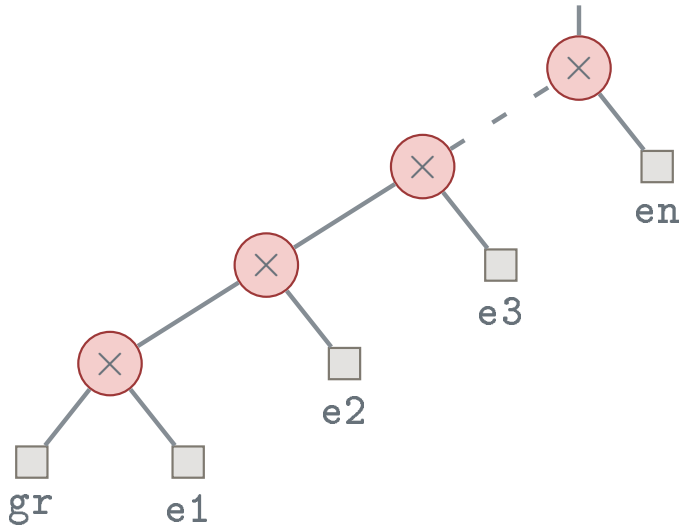
En d'autres termes, étant donnée une liste `[e1 ; e2 ; e3 ; ... ; en]` et une opération $\times$, on souhaite évaluer l'arbre syntaxique



Ici, `gr` est un élément initial pour le calcul. On l'appelle **graine**.

Pliage à gauche

En d'autres termes, étant donnée une liste `[e1 ; e2 ; e3 ; ... ; en]` et une opération $\times$, on souhaite évaluer l'arbre syntaxique



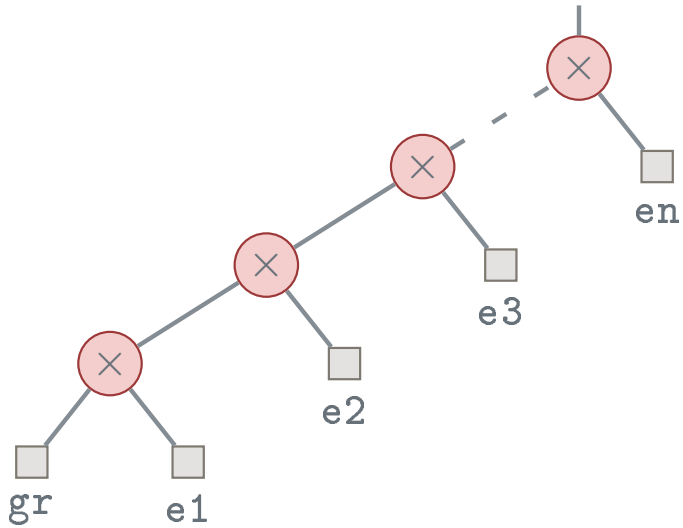
Ici, `gr` est un élément initial pour le calcul. On l'appelle **graine**.

Pour écrire une fonction réalisant cette opération, appelée pliage à gauche, il est nécessaire de transmettre les informations suivantes :

1. l'opération $\times$;

Pliage à gauche

En d'autres termes, étant donnée une liste `[e1 ; e2 ; e3 ; ... ; en]` et une opération $\times$, on souhaite évaluer l'arbre syntaxique



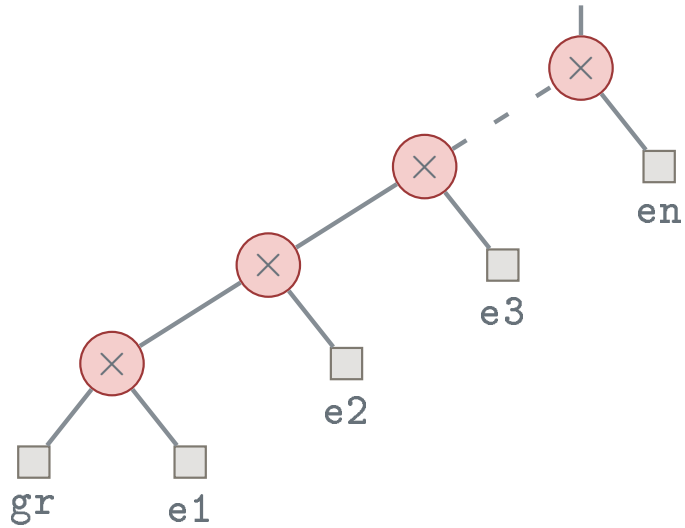
Ici, `gr` est un élément initial pour le calcul. On l'appelle **graine**.

Pour écrire une fonction réalisant cette opération, appelée pliage à gauche, il est nécessaire de transmettre les informations suivantes :

1. l'opération $\times$;
2. la graine `gr` ;

Pliage à gauche

En d'autres termes, étant donnée une liste `[e1 ; e2 ; e3 ; ... ; en]` et une opération $\times$, on souhaite évaluer l'arbre syntaxique



Ici, `gr` est un élément initial pour le calcul. On l'appelle **graine**.

Pour écrire une fonction réalisant cette opération, appelée pliage à gauche, il est nécessaire de transmettre les informations suivantes :

1. l'opération $\times$;
2. la graine `gr` ;
3. la liste `lst = [e1 ; e2 ; e3 ; ... ; en]` des opérandes.

Pliage à gauche

L'opération $\times$ est représentée par une fonction `op` de type `'a -> 'a -> 'a` et la graine `gr` est un élément de type `'a`.

Pliage à gauche

L'opération $\times$ est représentée par une fonction `op` de type `'a -> 'a -> 'a` et la graine `gr` est un élément de type `'a`.

Ainsi, l'opération de pliage à gauche est de type

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

Pliage à gauche

L'opération $\times$ est représentée par une fonction `op` de type `'a -> 'a -> 'a` et la graine `gr` est un élément de type `'a`.

Ainsi, l'opération de pliage à gauche est de type

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

et sa définition est

```
let rec pliage_gauche op gr lst =
    match lst with
        |[] -> gr
        |e :: reste -> (pliage_gauche op (op gr e) reste);;
```

Pliage à gauche

L'opération $\times$ est représentée par une fonction `op` de type `'a -> 'a -> 'a` et la graine `gr` est un élément de type `'a`.

Ainsi, l'opération de pliage à gauche est de type

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

et sa définition est

```
let rec pliage_gauche op gr lst =
    match lst with
        |[] -> gr
        |e :: reste -> (pliage_gauche op (op gr e) reste);;
```

Exemples d'utilisation :

```
# (pliage_gauche (+) 0 [0 ; 2 ; 1 ; 5 ; 2 ; -2 ; 3]);;
- : int = 11
```

Pliage à gauche

L'opération $\times$ est représentée par une fonction `op` de type `'a -> 'a -> 'a` et la graine `gr` est un élément de type `'a`.

Ainsi, l'opération de pliage à gauche est de type

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

et sa définition est

```
let rec pliage_gauche op gr lst =
    match lst with
        |[] -> gr
        |e :: reste -> (pliage_gauche op (op gr e) reste);;
```

Exemples d'utilisation :

```
# (pliage_gauche (+) 0 [0 ; 2 ; 1 ; 5 ; 2 ; -2 ; 3]);;
- : int = 11

# let lst = [0 ; 2 ; 1 ; 5 ; 2 ; -2 ; 3] in
  (pliage_gauche max (List.hd lst) (List.tl lst));;
- : int = 5
```

Pliage à gauche

L'opération $\times$ est représentée par une fonction `op` de type `'a -> 'a -> 'a` et la graine `gr` est un élément de type `'a`.

Ainsi, l'opération de pliage à gauche est de type

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

et sa définition est

```
let rec pliage_gauche op gr lst =
    match lst with
        |[] -> gr
        |e :: reste -> (pliage_gauche op (op gr e) reste);;
```

Exemples d'utilisation :

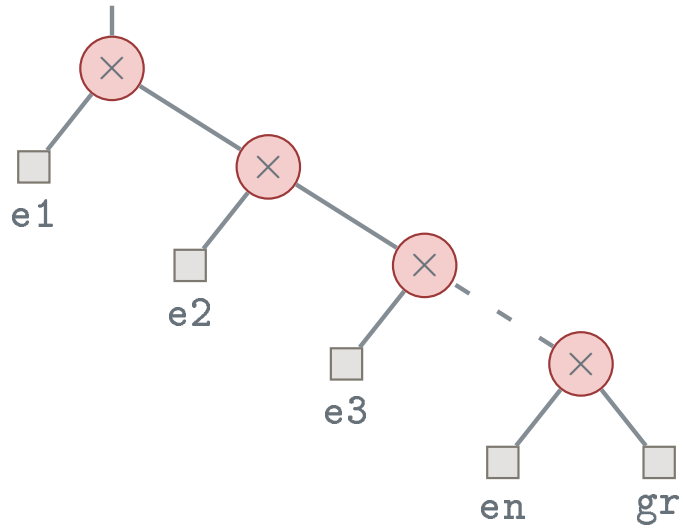
```
# (pliage_gauche (+) 0 [0 ; 2 ; 1 ; 5 ; 2 ; -2 ; 3]);;
- : int = 11

# let lst = [0 ; 2 ; 1 ; 5 ; 2 ; -2 ; 3] in
  (pliage_gauche max (List.hd lst) (List.tl lst));;
- : int = 5

# (pliage_gauche (^) "" ["mi" ; "la" ; "re" ; "sol" ; "do" ; "fa"]);;
- : string = "milaresoldofa"
```

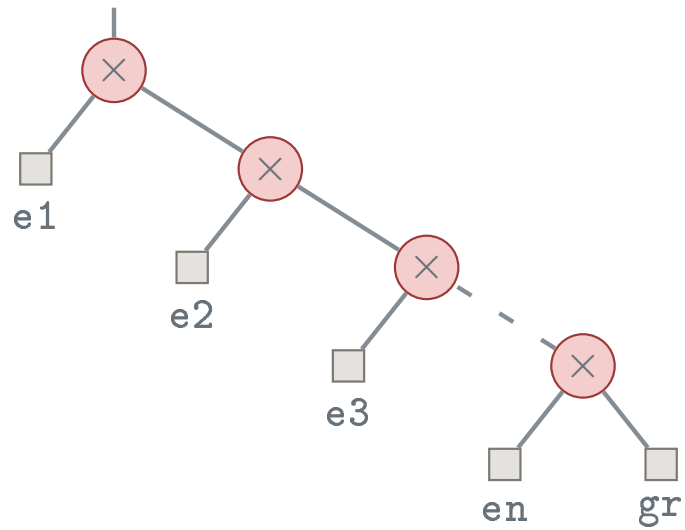
Pliage à droite

Il est possible de faire la même chose mais en considérant plutôt l'arbre syntaxique **peigne droit** (à la place du gauche) :



Pliage à droite

Il est possible de faire la même chose mais en considérant plutôt l'arbre syntaxique **peigne droit** (à la place du gauche) :

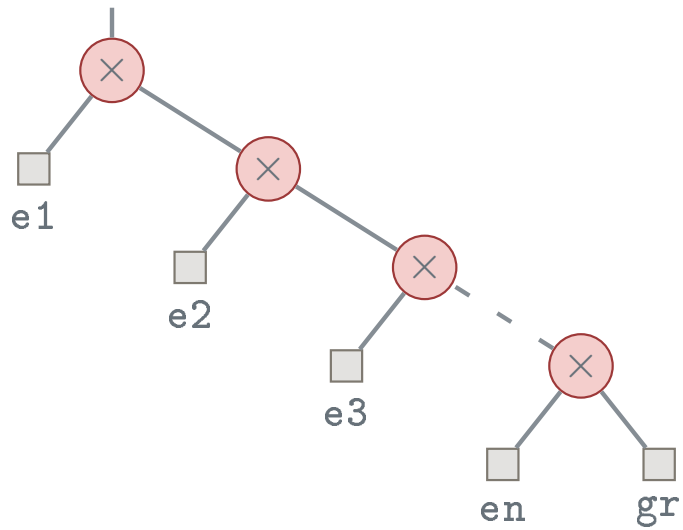


L'opération de pliage à droite est ainsi de type

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

Pliage à droite

Il est possible de faire la même chose mais en considérant plutôt l'arbre syntaxique **peigne droit** (à la place du gauche) :



L'opération de pliage à droite est ainsi de type

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

et sa définition est

```
let rec pliage_droite op lst gr =
    match lst with
        |[] -> gr
        |e :: reste -> (op e (pliage_droite op reste gr));;
```

Pliages à gauche et à droite

Ces deux opérations de pliage existent dans le module `List` sous les noms respectifs de `fold_left` et `fold_right`.

Pliages à gauche et à droite

Ces deux opérations de pliage existent dans le module `List` sous les noms respectifs de `fold_left` et `fold_right`.

Les véritables types (qui nous avons précédemment simplifiés dans un but pédagogique) de ces fonctions sont

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

et

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

Pliages à gauche et à droite

Ces deux opérations de pliage existent dans le module `List` sous les noms respectifs de `fold_left` et `fold_right`.

Les véritables types (qui nous avons précédemment simplifiés dans un but pédagogique) de ces fonctions sont

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

et

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

D'un point de vue sémantique, lorsque l'opération $\times$ est **associative** et que la graine `gr` **commute** avec tous les éléments, les deux pliages donnent le même résultat.

Pliages à gauche et à droite

Ces deux opérations de pliage existent dans le module `List` sous les noms respectifs de `fold_left` et `fold_right`.

Les véritables types (qui nous avons précédemment simplifiés dans un but pédagogique) de ces fonctions sont

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

et

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

D'un point de vue sémantique, lorsque l'opération $\times$ est **associative** et que la graine `gr` **commute** avec tous les éléments, les deux pliages donnent le même résultat.

Il y a une différence d'efficacité : le pliage à gauche est **récursif terminal** alors que le pliage à droite ne l'est pas. En effet, dans le pliage à gauche, la graine est l'accumulateur.

Miroir d'une liste

Pour calculer l'image miroir d'une liste, le type des éléments qu'elle contient n'est pas important.

Il est préférable de proposer une solution récursive terminale :

Miroir d'une liste

Pour calculer l'image miroir d'une liste, le type des éléments qu'elle contient n'est pas important.

Il est préférable de proposer une solution récursive terminale :

```
let miroir lst =
    let rec aux lst acc =
        match lst with
            |[] -> acc
            |e :: reste -> (aux reste (e :: acc))
    in
    (aux lst []);;
```

Miroir d'une liste

Pour calculer l'image miroir d'une liste, le type des éléments qu'elle contient n'est pas important.

Il est préférable de proposer une solution récursive terminale :

```
let miroir lst =
    let rec aux lst acc =
        match lst with
            |[] -> acc
            |e :: reste -> (aux reste (e :: acc))
    in
    (aux lst []);;
```

Exemples d'utilisation :

```
# (miroir ['a' ; 'b' ; 'c']);;
- : char list = ['c'; 'b'; 'a']
```

Miroir d'une liste

Pour calculer l'image miroir d'une liste, le type des éléments qu'elle contient n'est pas important.

Il est préférable de proposer une solution récursive terminale :

```
let miroir lst =
    let rec aux lst acc =
        match lst with
            |[] -> acc
            |e :: reste -> (aux reste (e :: acc))
    in
    (aux lst []);;
```

Exemples d'utilisation :

```
# (miroir ['a' ; 'b' ; 'c']);;
- : char list = ['c'; 'b'; 'a']

# (miroir [1 ; 1 ; 2 ; 2 ; 2 ; 1]);;
- : int list = [1; 2; 2; 2; 1; 1]
```

Miroir d'une liste

Pour calculer l'image miroir d'une liste, le type des éléments qu'elle contient n'est pas important.

Il est préférable de proposer une solution récursive terminale :

```
let miroir lst =
    let rec aux lst acc =
        match lst with
            |[] -> acc
            |e :: reste -> (aux reste (e :: acc))
    in
    (aux lst []);;
```

Exemples d'utilisation :

```
# (miroir ['a' ; 'b' ; 'c']);;
- : char list = ['c'; 'b'; 'a']

# (miroir [1 ; 1 ; 2 ; 2 ; 2 ; 1]);;
- : int list = [1; 2; 2; 2; 1; 1]
```

Nous avons réimplanté ici la fonction `rev` du module `List`.

Fonctions avancées

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

Fonction	Type
`map`	`('a -> 'b) -> 'a list -> 'b list`
`filter`	`('a -> bool) -> 'a list -> 'a list`
`for_all`	`('a -> bool) -> 'a list -> bool`
`exists`	`('a -> bool) -> 'a list -> bool`
`fold_left`	`('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`
`fold_right`	`('a -> 'b -> 'b) -> 'a list -> 'b -> 'b`

Fonctions avancées

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

Fonction	Type
`map`	`('a -> 'b) -> 'a list -> 'b list`
`filter`	`('a -> bool) -> 'a list -> 'a list`
`for_all`	`('a -> bool) -> 'a list -> bool`
`exists`	`('a -> bool) -> 'a list -> bool`
`fold_left`	`('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`
`fold_right`	`('a -> 'b -> 'b) -> 'a list -> 'b -> 'b`

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
`map`	`('a -> 'b) -> 'a list -> 'b list`
`filter`	`('a -> bool) -> 'a list -> 'a list`
`for_all`	`('a -> bool) -> 'a list -> bool`
`exists`	`('a -> bool) -> 'a list -> bool`
`fold_left`	`('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`
`fold_right`	`('a -> 'b -> 'b) -> 'a list -> 'b -> 'b`

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 (1er paramètre).

Plan

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 1re créé une liste (trois cellules) en mémoire :

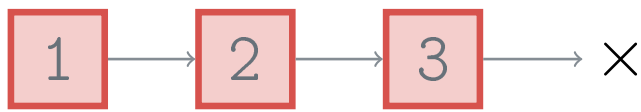
Le cas des listes

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 1re créé une liste (trois cellules) en mémoire :



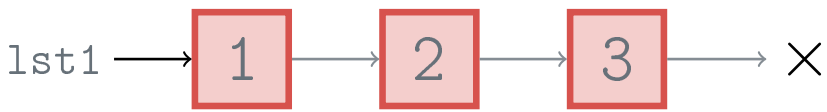
Le cas des listes

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 1re créé une liste (trois cellules) en mémoire :



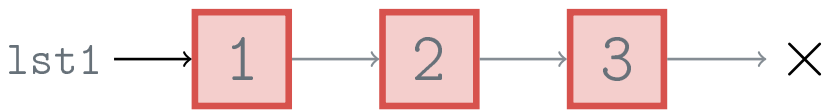
Le cas des listes

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 1re créé une liste (trois cellules) en mémoire :



La 2e ne créé qu'une seule cellule et partage les trois précédentes :

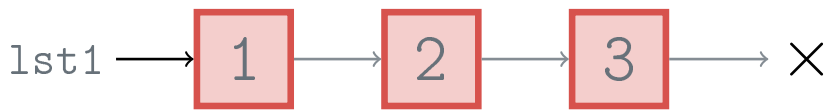
Le cas des listes

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

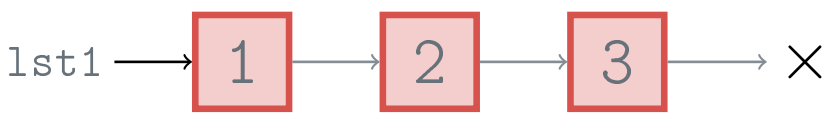
Considérons les phrases

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

La 1re créé une liste (trois cellules) en mémoire :



La 2e ne créé qu'une seule cellule et partage les trois précédentes :



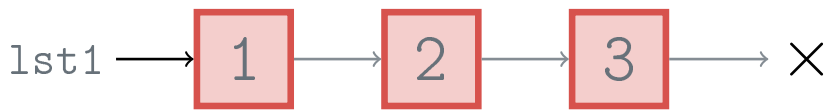
Le cas des listes

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

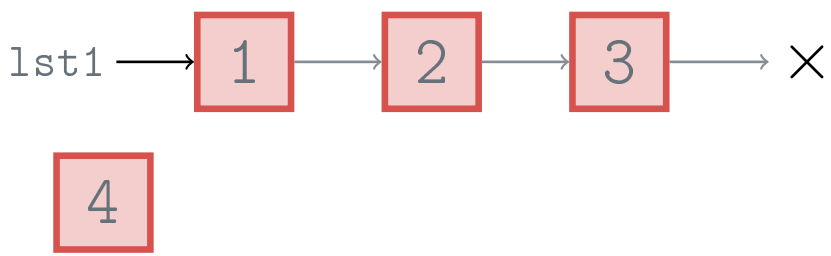
Considérons les phrases

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

La 1re créé une liste (trois cellules) en mémoire :



La 2e ne créé qu'une seule cellule et partage les trois précédentes :



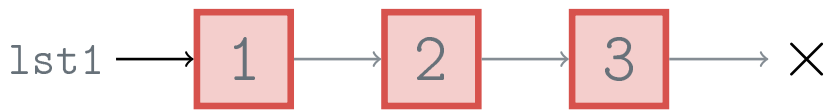
Le cas des listes

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

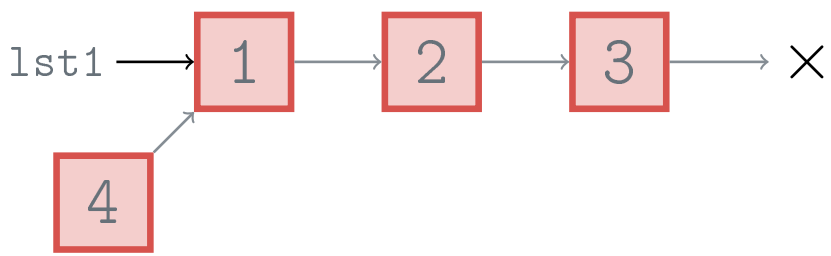
Considérons les phrases

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

La 1re créé une liste (trois cellules) en mémoire :



La 2e ne créé qu'une seule cellule et partage les trois précédentes :



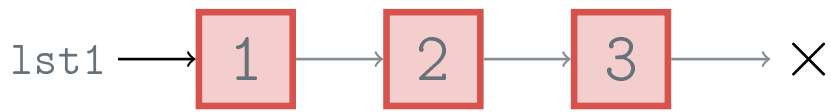
Le cas des listes

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

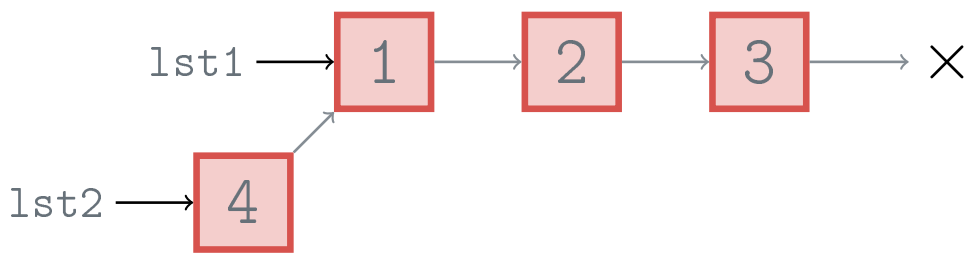
Considérons les phrases

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

La 1re créé une liste (trois cellules) en mémoire :



La 2e ne créé 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

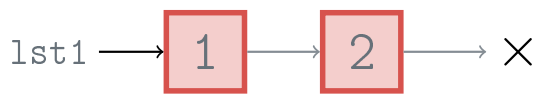
```
let rec concatener lst1 lst2 =
    match lst1 with
        |[] -> lst2
        |e :: reste -> e :: (concatener reste lst2);;
```

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

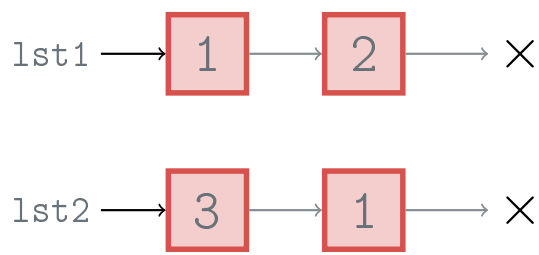
```
let rec concatener lst1 lst2 =
    match lst1 with
        |[] -> lst2
        |e :: reste -> e :: (concatener reste lst2);;
```

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);;
```

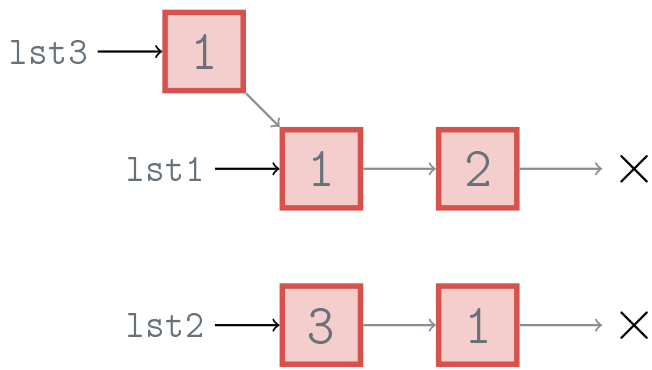
Elle permet de concaténer deux listes (fonction `append` du module `List`).

Considérons l'effet des phrases suivantes :

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

```
# let lst3 = 1 :: lst1;;
```

On obtient en mémoire la configuration de partage suivante :



Le cas des listes

Considérons la fonction

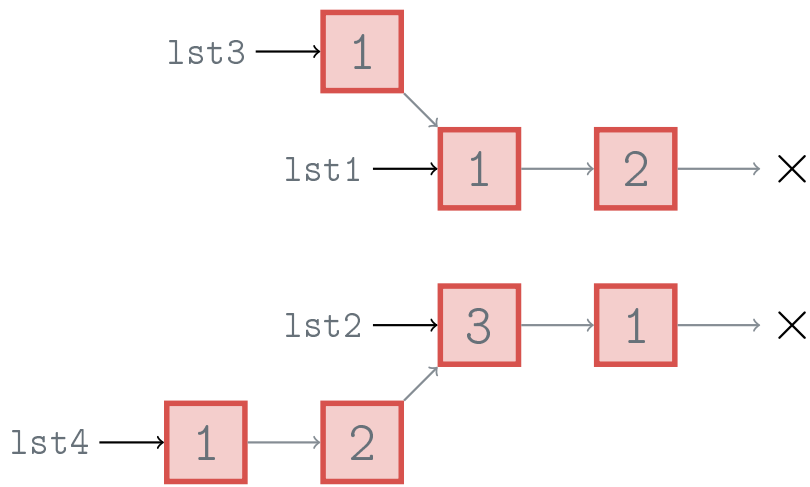
```
let rec concatener lst1 lst2 =
    match lst1 with
        |[] -> lst2
        |e :: reste -> e :: (concatener reste lst2);;
```

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

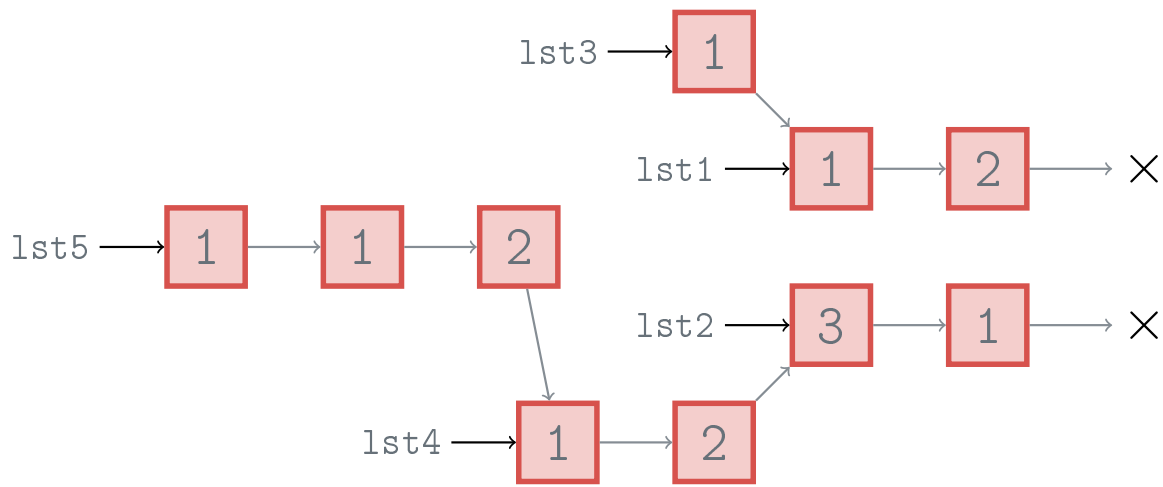
```
let rec concatener lst1 lst2 =
    match lst1 with
        |[] -> lst2
        |e :: reste -> e :: (concatener reste lst2);;
```

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

Considérons l'effet des phrases suivantes :

```
# let lst1 = [1 ; 2];;
# let lst2 = [3 ; 1];;
# let lst3 = 1 :: lst1;;
# let lst4 = (concatener lst1 lst2);;
# let lst5 = (concatener lst3 lst4);;
```

On obtient en mémoire la configuration de partage suivante :



Plan

Files

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

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`;

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 ;

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 ;

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;

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.

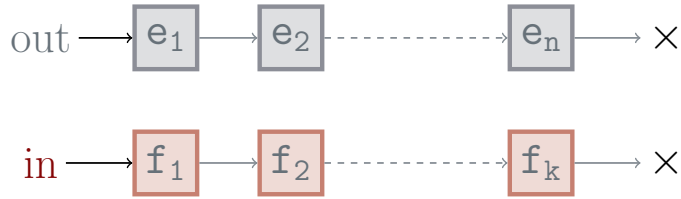
Les opérations sur cette structure de données se décrivent ainsi :

- 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.

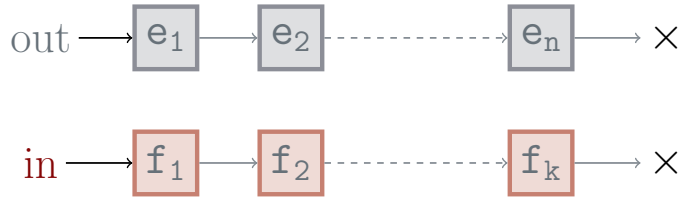
Implantation astucieuse

Voici une file dans une situation générique :

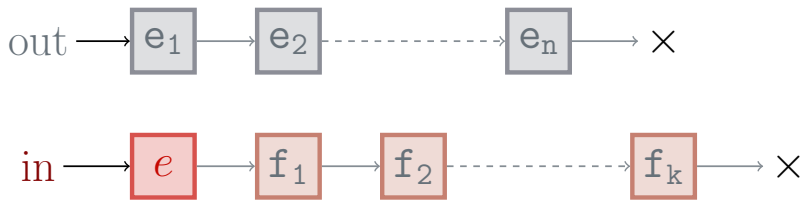


Implantation astucieuse

Voici une file dans une situation générique :

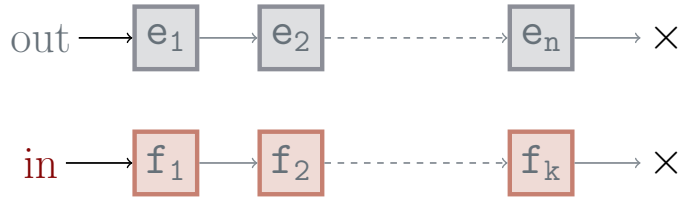


Après l'**ajout** d'un élément e, elle devient

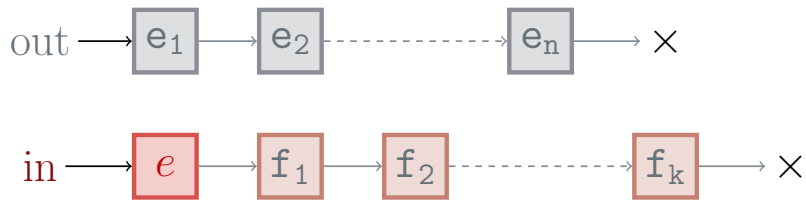


Implantation astucieuse

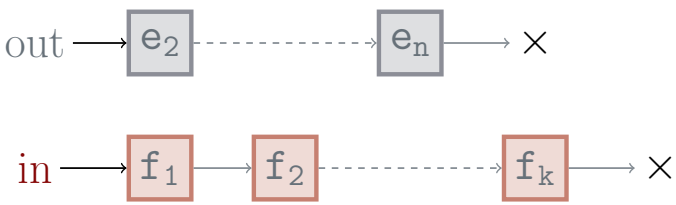
Voici une file dans une situation générique :



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



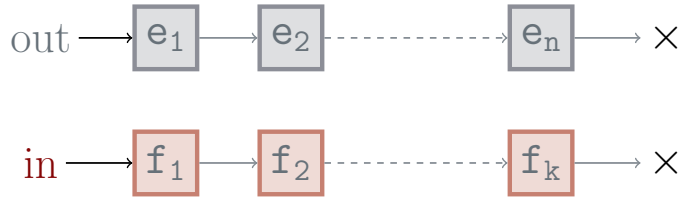
Après une **suppression** depuis la situation générique, elle devient



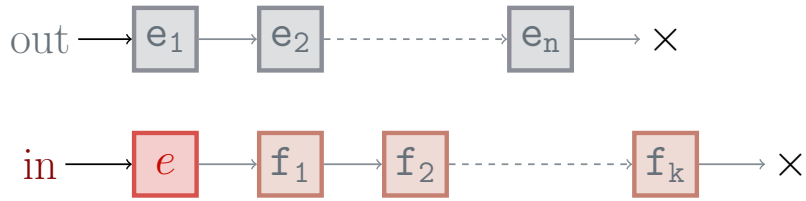
et renvoie e_1 si out n'est pas vide,

Implantation astucieuse

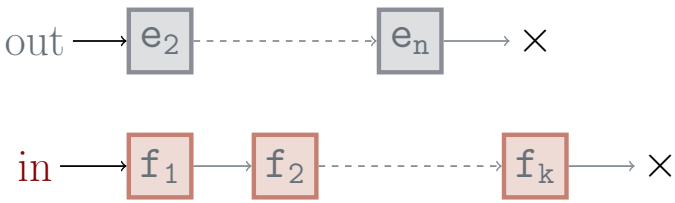
Voici une file dans une situation générique :



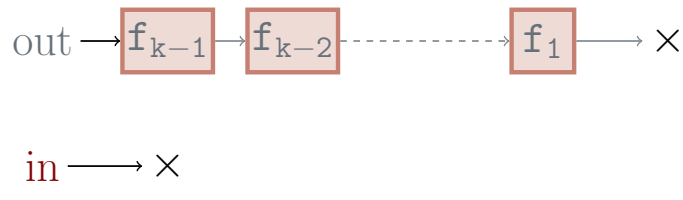
Après l'**ajout** d'un élément e, elle devient



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

- λ-calcul
  - λ-termes
  - Codage
  - Implantation

Généralités

Le λ-calcul a été introduit par Church en 1936.

Généralités

Le λ-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 λ-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 λ-calcul, n'a pas un pouvoir d'expression suffisant.

Généralités

Le λ-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 λ-calcul, n'a pas un pouvoir d'expression suffisant.

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

Généralités

Le λ-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 λ-calcul, n'a pas un pouvoir d'expression suffisant.

En revanche, le λ-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 λ-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 λ-calcul, n'a pas un pouvoir d'expression suffisant.

En revanche, le λ-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 λ-calcul constitue le cœur de tous les **langages de programmation fonctionnels**.

Plan

- λ-calcul
  - λ-termes
  - Codage
  - Implantation

λ-termes

Un λ-terme est

λ-termes

Un λ-terme est

1. une **variable** x, y, z, t, *etc.*,

λ-termes

Un λ-terme est

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

λ-termes

Un λ-terme est

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

λ-termes

Un λ-terme est

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

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

λ-termes

Un λ-terme est

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

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

P.ex.,

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

est un λ-terme.

λ-termes

Un λ-terme est

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

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

P.ex.,

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

est un λ-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 λ-terme t peut se représenter par son arbre syntaxique de la manière suivante :

Arbres syntaxiques

Tout λ-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 λ-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 λx qui possède comme fils l'arbre syntaxique de t' ;

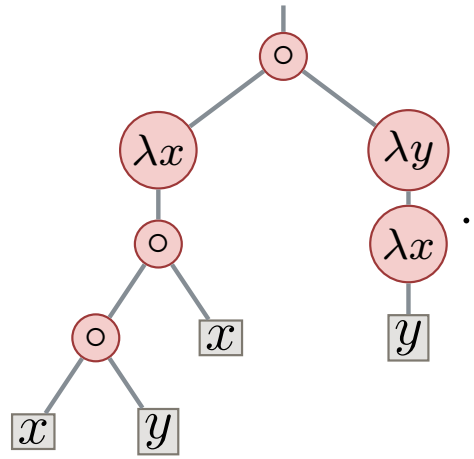
Arbres syntaxiques

Tout λ-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 λx 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

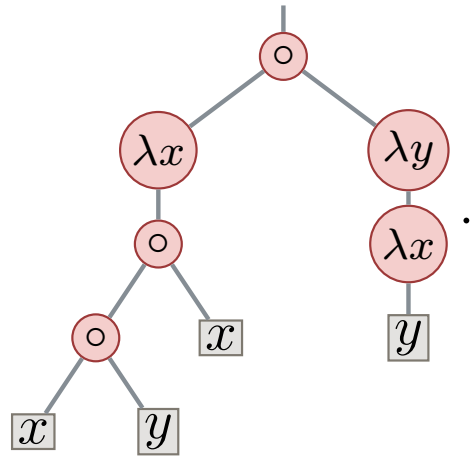
L'arbre syntaxique de $(\lambda x.((x\,y)\,x))\,(\lambda y.\lambda x.y)$ est



.

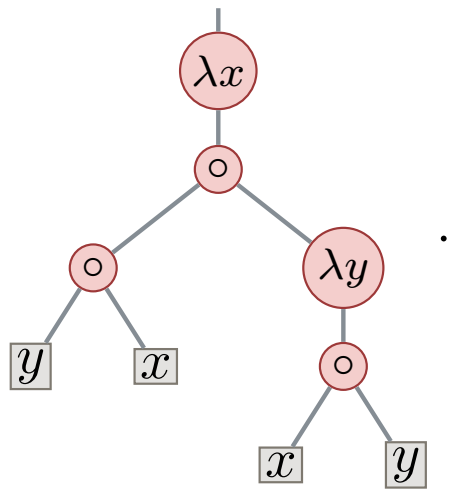
Arbres syntaxiques

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 λ-terme et une occurrence d'une variable x y apparaissant.

Variables libres/liées

Soit t un λ-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 λ-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 λ-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 λx, on s'arrête et on relie f à ce nœud ;

Variables libres/liées

Soit t un λ-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 λx, 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 λ-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 λx, 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 λ-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 λx, 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 λx cible) ; sinon, on dit qu'elle est libre.

Variables libres/liées

Soit le λ-terme

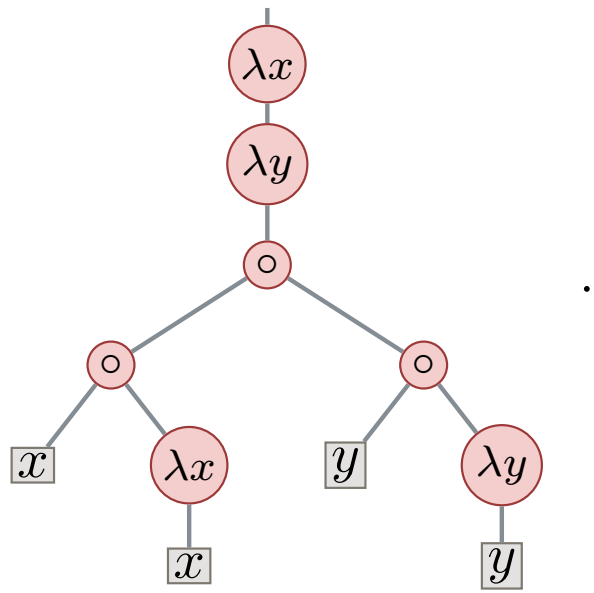
$$\lambda x.\lambda x.((x\,(\lambda x.x))\,(y\,(\lambda y.y))).$$

Variables libres/liées

Soit le λ-terme

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

Son arbre syntaxique est



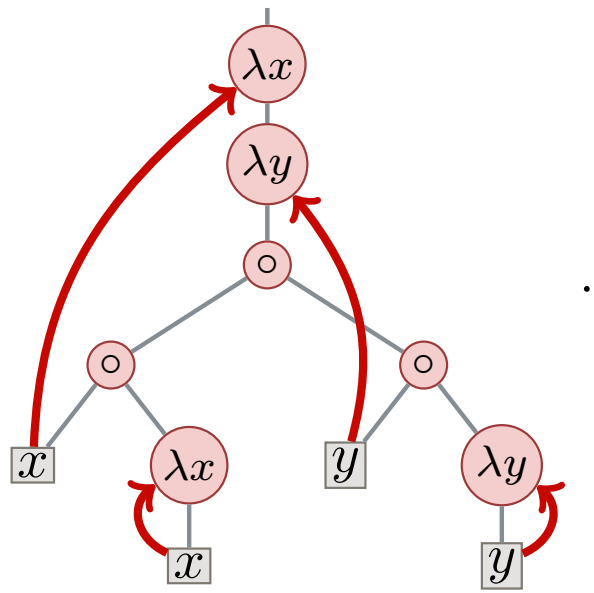
.

Variables libres/liées

Soit le λ-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 λ-terme, x une variable et s un autre λ-terme.

Substitutions

Soit t un λ-terme, x une variable et s un autre λ-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 λ-terme ainsi obtenu.

Substitutions

Soit t un λ-terme, x une variable et s un autre λ-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 λ-terme ainsi obtenu.

P.ex.,

- $\lambda z.(x\, y) \star_x z\, z$

Substitutions

Soit t un λ-terme, x une variable et s un autre λ-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 λ-terme ainsi obtenu.

P.ex.,

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

Substitutions

Soit t un λ-terme, x une variable et s un autre λ-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 λ-terme ainsi obtenu.

P.ex.,

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

Substitutions

Soit t un λ-terme, x une variable et s un autre λ-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 λ-terme ainsi obtenu.

P.ex.,

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

Substitutions

Soit t un λ-terme, x une variable et s un autre λ-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 λ-terme ainsi obtenu.

P.ex.,

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

Substitutions

Soit t un λ-terme, x une variable et s un autre λ-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 λ-terme ainsi obtenu.

P.ex.,

- $\lambda z.(x\,y) \;\star_x\; z\,z = \lambda z.((z\,z)\,y)$,
- $\lambda x.(x\,y) \;\star_x\; z\,z = \lambda x.(x\,y)$,
- $(z\,(\lambda z.(y\,z)))\,z \;\star_z\; \lambda x.(x\,x) = ((\lambda x.(x\,x))\,(\lambda z.(y\,z)))\,(\lambda x.(x\,x))$.

α-renommages et α-équivalence

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

α-renommages et α-équivalence

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

L'α-renommage de x en y dans t consiste à remplacer dans t chaque λx par λy et chaque occurrence de x liée à un λx par y.

α-renommages et α-équivalence

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

L'α-renommage de x en y dans t consiste à remplacer dans t chaque λx par λy et chaque occurrence de x liée à un λx par y.

P.ex.,

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

α-renommages et α-équivalence

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

L'α-renommage de x en y dans t consiste à remplacer dans t chaque λx par λy et chaque occurrence de x liée à un λx par y.

P.ex.,

- l'α-renommage de x en y du λ-terme $\lambda x.x$ est $\lambda y.y$;
- l'α-renommage de x en y du λ-terme $(x\,x)\,(\lambda x.(x\,z))$ est $(x\,x)\,(\lambda y.(y\,z))$.

α-renommages et α-équivalence

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

L'α-renommage de x en y dans t consiste à remplacer dans t chaque λx par λy et chaque occurrence de x liée à un λx par y.

P.ex.,

- l'α-renommage de x en y du λ-terme $\lambda x.x$ est $\lambda y.y$;
- l'α-renommage de x en y du λ-terme $(x\,x)\,(\lambda x.(x\,z))$ est $(x\,x)\,(\lambda y.(y\,z))$.

Deux λ-termes t et t' sont α-équivalents s'il est possible de transformer t en t' par une suite d'α-renommages.

α-renommages et α-équivalence

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

L'α-renommage de x en y dans t consiste à remplacer dans t chaque λx par λy et chaque occurrence de x liée à un λx par y.

P.ex.,

- l'α-renommage de x en y du λ-terme $\lambda x.x$ est $\lambda y.y$;
- l'α-renommage de x en y du λ-terme $(x\,x)\,(\lambda x.(x\,z))$ est $(x\,x)\,(\lambda y.(y\,z))$.

Deux λ-termes t et t' sont α-équivalents s'il est possible de transformer t en t' par une suite d'α-renommages.

P.ex.,

- $\lambda x.x$ et $\lambda y.y$ sont α-équivalents;

α-renommages et α-équivalence

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

L'α-renommage de x en y dans t consiste à remplacer dans t chaque λx par λy et chaque occurrence de x liée à un λx par y.

P.ex.,

- l'α-renommage de x en y du λ-terme $\lambda x.x$ est $\lambda y.y$;
- l'α-renommage de x en y du λ-terme $(x\,x)\,(\lambda x.(x\,z))$ est $(x\,x)\,(\lambda y.(y\,z))$.

Deux λ-termes t et t' sont α-équivalents s'il est possible de transformer t en t' par une suite d'α-renommages.

P.ex.,

- $\lambda x.x$ et $\lambda y.y$ sont α-équivalents;
- $\lambda x.\lambda y.(x\,y)$ et $\lambda y.\lambda x.(y\,x)$ sont α-équivalents;

α-renommages et α-équivalence

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

L'α-renommage de x en y dans t consiste à remplacer dans t chaque λx par λy et chaque occurrence de x liée à un λx par y.

P.ex.,

- l'α-renommage de x en y du λ-terme $\lambda x.x$ est $\lambda y.y$;
- l'α-renommage de x en y du λ-terme $(x\,x)\,(\lambda x.(x\,z))$ est $(x\,x)\,(\lambda y.(y\,z))$.

Deux λ-termes t et t' sont α-équivalents s'il est possible de transformer t en t' par une suite d'α-renommages.

P.ex.,

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

β-réductions en racine

Soit t un λ-terme de la forme

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

β-réductions en racine

Soit t un λ-terme de la forme

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

La β-réduction en racine appliquée sur t consiste à transformer t en le λ-terme

$$u \star_x v.$$

β-réductions en racine

Soit t un λ-terme de la forme

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

La β-réduction en racine appliquée sur t consiste à transformer t en le λ-terme

$$u \star_x v.$$

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

β-réductions en racine

Soit t un λ-terme de la forme

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

La β-réduction en racine appliquée sur t consiste à transformer t en le λ-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 α-renommer y en y' dans u au préalable, où y' est une nouvelle variable qui n'admet pas d'occurrence libre dans v.

β-réductions en racine

Soit t un λ-terme de la forme

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

La β-réduction en racine appliquée sur t consiste à transformer t en le λ-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 α-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 $t \xrightarrow{\mathrm{r}}_\beta t'$ le fait qu'un λ-terme t' puisse être obtenu à partir du λ-terme t par une β-réduction en racine.

β-réductions en racine — exemples

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

β-réductions en racine — exemples

- $(\lambda y.y)(\lambda x.(x\ z)) \xrightarrow{\mathrm{r}}_{\beta} \lambda x.(x\ z)$

β-réductions en racine — exemples

- $(\lambda y.y)(\lambda x.(x\ z)) \xrightarrow{\mathrm{r}}_{\beta} \lambda x.(x\ z)$

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

β-réductions en racine — exemples

- $(\lambda y.y)(\lambda x.(x\ z)) \xrightarrow{\mathrm{r}}_{\beta} \lambda x.(x\ z)$

- $(\lambda x.(x\ y))\ z \xrightarrow{\mathrm{r}}_{\beta} z\ y$

β-réductions en racine — exemples

- $(\lambda y.y)(\lambda x.(x\ z)) \xrightarrow{\mathrm{r}}_\beta \lambda x.(x\ z)$
- $(\lambda x.(x\ y))\ z \xrightarrow{\mathrm{r}}_\beta z\ y$
- $(\lambda x.\lambda y.((x\ x)\ (\lambda x.x)))\ (\lambda y.(y\ x))$

β-réductions en racine — exemples

- $(\lambda y.y)(\lambda x.(x\ z)) \overset{\mathrm{r}}{\to}_\beta \lambda x.(x\ z)$
- $(\lambda x.(x\ y))\ z \overset{\mathrm{r}}{\to}_\beta z\ y$
- $(\lambda x.\lambda y.((x\ x)\ (\lambda x.x)))\ (\lambda y.(y\ x)) \overset{\mathrm{r}}{\to}_\beta \lambda y.(((\lambda y.(y\ x))\ (\lambda y(y\ x)))\ (\lambda x.x))$

β-réductions en racine — exemples

- $(\lambda y.y)(\lambda x.(x\ z)) \overset{\mathrm{r}}{\to}_\beta \lambda x.(x\ z)$
- $(\lambda x.(x\ y))\ z \overset{\mathrm{r}}{\to}_\beta z\ y$
- $(\lambda x.\lambda y.((x\ x)\ (\lambda x.x)))\ (\lambda y.(y\ x)) \overset{\mathrm{r}}{\to}_\beta \lambda y.(((\lambda y.(y\ x))\ (\lambda y(y\ x)))\ (\lambda x.x))$
- $(\lambda x.\lambda y.x)\ (z\ y)$

β-réductions en racine — exemples

- $(\lambda y.y)(\lambda x.(x\ z)) \overset{\mathrm{r}}{\to}_\beta \lambda x.(x\ z)$
- $(\lambda x.(x\ y))\ z \overset{\mathrm{r}}{\to}_\beta z\ y$
- $(\lambda x.\lambda y.((x\ x)\ (\lambda x.x)))\ (\lambda y.(y\ x)) \overset{\mathrm{r}}{\to}_\beta \lambda y.(((\lambda y.(y\ x))\ (\lambda y(y\ x)))\ (\lambda x.x))$
- $(\lambda x.\lambda y.x)\ (z\ y) \overset{\mathrm{r}}{\not\to}_\beta \lambda y.(z\ y)$

β-réductions en racine — exemples

- $(\lambda y.y)(\lambda x.(x\ z)) \overset{\mathrm{r}}{\to}_\beta \lambda x.(x\ z)$
- $(\lambda x.(x\ y))\ z \overset{\mathrm{r}}{\to}_\beta z\ y$
- $(\lambda x.\lambda y.((x\ x)\ (\lambda x.x)))\ (\lambda y.(y\ x)) \overset{\mathrm{r}}{\to}_\beta \lambda y.(((\lambda y.(y\ x))\ (\lambda y(y\ x)))\ (\lambda x.x))$
- $(\lambda x.\lambda y.x)\ (z\ y) \overset{\mathrm{r}}{\not\to}_\beta \lambda y.(z\ y)$

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

β-réductions en racine — exemples

- $(\lambda y.y)(\lambda x.(x\ z)) \overset{\mathrm{r}}{\to}_\beta \lambda x.(x\ z)$
- $(\lambda x.(x\ y))\ z \overset{\mathrm{r}}{\to}_\beta z\ y$
- $(\lambda x.\lambda y.((x\ x)\ (\lambda x.x)))\ (\lambda y.(y\ x)) \overset{\mathrm{r}}{\to}_\beta \lambda y.(((\lambda y.(y\ x))\ (\lambda y(y\ x)))\ (\lambda x.x))$
- $(\lambda x.\lambda y.x)\ (z\ y) \overset{\mathrm{r}}{\not\to}_\beta \lambda y.(z\ y)$

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

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

$$(\lambda x.\lambda y.x)\ (z\ y) = (\lambda x.\lambda y'.x)\ (z\ y) \overset{\mathrm{r}}{\to}_\beta \lambda y'.(z\ y).$$

β-réductions

Soit t un λ-terme.

β-réductions

Soit t un λ-terme.

On appelle sous-terme de t tout λ-terme dont l'arbre syntaxique est un sous-arbre de l'arbre syntaxique de t.

β-réductions

Soit t un λ-terme.

On appelle sous-terme de t tout λ-terme dont l'arbre syntaxique est un sous-arbre de l'arbre syntaxique de t.

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

β-réductions

Soit t un λ-terme.

On appelle sous-terme de t tout λ-terme dont l'arbre syntaxique est un sous-arbre de l'arbre syntaxique de t.

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

$$x,$$

β-réductions

Soit t un λ-terme.

On appelle sous-terme de t tout λ-terme dont l'arbre syntaxique est un sous-arbre de l'arbre syntaxique de t.

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

$$x, \quad y,$$

β-réductions

Soit t un λ-terme.

On appelle sous-terme de t tout λ-terme dont l'arbre syntaxique est un sous-arbre de l'arbre syntaxique de t.

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

$$x,\quad y,\quad y\,y,$$

β-réductions

Soit t un λ-terme.

On appelle sous-terme de t tout λ-terme dont l'arbre syntaxique est un sous-arbre de l'arbre syntaxique de t.

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

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

β-réductions

Soit t un λ-terme.

On appelle sous-terme de t tout λ-terme dont l'arbre syntaxique est un sous-arbre de l'arbre syntaxique de t.

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

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

β-réductions

Soit t un λ-terme.

On appelle sous-terme de t tout λ-terme dont l'arbre syntaxique est un sous-arbre de l'arbre syntaxique de t.

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

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

β-réductions

Soit t un λ-terme.

On appelle sous-terme de t tout λ-terme dont l'arbre syntaxique est un sous-arbre de l'arbre syntaxique de t.

P.ex., le λ-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 β-réduction appliquée sur t consiste à appliquer une β-réduction en racine sur l'un de ses sous-termes.

β-réductions

Soit t un λ-terme.

On appelle sous-terme de t tout λ-terme dont l'arbre syntaxique est un sous-arbre de l'arbre syntaxique de t.

P.ex., le λ-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 β-réduction appliquée sur t consiste à appliquer une β-réduction en racine sur l'un de ses sous-termes.

On note $t \to_\beta t'$ le fait qu'un λ-terme t' puisse être obtenu à partir de t par une β-réduction.

β-réductions

Soit t un λ-terme.

On appelle sous-terme de t tout λ-terme dont l'arbre syntaxique est un sous-arbre de l'arbre syntaxique de t.

P.ex., le λ-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 β-réduction appliquée sur t consiste à appliquer une β-réduction en racine sur l'un de ses sous-termes.

On note $t \to_\beta t'$ le fait qu'un λ-terme t' puisse être obtenu à partir de t par une β-réduction.

On note $t \overset{*}{\to}_\beta t'$ le fait qu'un λ-terme t' puisse être obtenu à partir de t par une suite de β-réductions.

β-réductions — exemples

- Voici une suite de β-réductions :

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

β-réductions — exemples

- Voici une suite de β-réductions :

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

β-réductions — exemples

- Voici une suite de β-réductions :

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

β-réductions — exemples

- Voici une suite de β-réductions :

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

β-réductions — exemples

- Voici une suite de β-réductions :

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

- En partant d'un même λ-terme, on peut appliquer plusieurs suites de β-réductions différentes :

β-réductions — exemples

- Voici une suite de β-réductions :

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

- En partant d'un même λ-terme, on peut appliquer plusieurs suites de β-réductions différentes :

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

β-réductions — exemples

- Voici une suite de β-réductions :

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

- En partant d'un même λ-terme, on peut appliquer plusieurs suites de β-réductions différentes :

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

β-réductions — exemples

- Voici une suite de β-réductions :

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

- En partant d'un même λ-terme, on peut appliquer plusieurs suites de β-réductions différentes :

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

β-réductions — exemples

- Voici une suite de β-réductions :

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

- En partant d'un même λ-terme, on peut appliquer plusieurs suites de β-réductions différentes :

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

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

β-réductions — exemples

- Voici une suite de β-réductions :

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

- En partant d'un même λ-terme, on peut appliquer plusieurs suites de β-réductions différentes :

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

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

β-réductions — exemples

- Voici une suite de β-réductions :

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

- En partant d'un même λ-terme, on peut appliquer plusieurs suites de β-réductions différentes :

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

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

β-réductions — exemples

- Voici une suite de β-réductions :

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

- En partant d'un même λ-terme, on peut appliquer plusieurs suites de β-réductions différentes :

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

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

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

Règle de calcul

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

Règle de calcul

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

Règle de calcul

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

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

Règle de calcul

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

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

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

Règle de calcul

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

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

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

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

Règle de calcul

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

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

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

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

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

Règle de calcul

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

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

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

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

Question : est-ce que les différentes évaluations d'un même λ-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 λ-terme aboutit toujours à une valeur ?

Terminaison

Question : est-ce que l'évaluation de tout λ-terme aboutit toujours à une valeur ?

Réponse : considérons le λ-terme

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

Terminaison

Question : est-ce que l'évaluation de tout λ-terme aboutit toujours à une valeur ?

Réponse : considérons le λ-terme

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

Nous avons

$$\Delta\, \Delta$$

Terminaison

Question : est-ce que l'évaluation de tout λ-terme aboutit toujours à une valeur ?

Réponse : considérons le λ-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 λ-terme aboutit toujours à une valeur ?

Réponse : considérons le λ-terme

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

Nous avons

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

Terminaison

Question : est-ce que l'évaluation de tout λ-terme aboutit toujours à une valeur ?

Réponse : considérons le λ-terme

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

Nous avons

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

Terminaison

Question : est-ce que l'évaluation de tout λ-terme aboutit toujours à une valeur ?

Réponse : considérons le λ-terme

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

Nous avons

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

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

Plan

Programmer en λ-calcul

Voici quelques correspondances entre le λ-calcul et la programmation (fonctionnelle).

Programmer en λ-calcul

Voici quelques correspondances entre le λ-calcul et la programmation (fonctionnelle).

Tout λ-terme t code un programme.

Programmer en λ-calcul

Voici quelques correspondances entre le λ-calcul et la programmation (fonctionnelle).

Tout λ-terme t code un programme.

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

Programmer en λ-calcul

Voici quelques correspondances entre le λ-calcul et la programmation (fonctionnelle).

Tout λ-terme t code un programme.

Étant donné que toutes les différentes évaluations d'un même λ-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 λ-calcul

Voici quelques correspondances entre le λ-calcul et la programmation (fonctionnelle).

Tout λ-terme t code un programme.

Étant donné que toutes les différentes évaluations d'un même λ-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 λ-calcul

Voici quelques correspondances entre le λ-calcul et la programmation (fonctionnelle).

Tout λ-terme t code un programme.

Étant donné que toutes les différentes évaluations d'un même λ-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 λ-calcul

Voici quelques correspondances entre le λ-calcul et la programmation (fonctionnelle).

Tout λ-terme t code un programme.

Étant donné que toutes les différentes évaluations d'un même λ-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 λ-termes.

Programmer en λ-calcul

Voici quelques correspondances entre le λ-calcul et la programmation (fonctionnelle).

Tout λ-terme t code un programme.

Étant donné que toutes les différentes évaluations d'un même λ-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 λ-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 :

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

Booléens

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

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

Nous avons, si u et v sont deux λ-termes,

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

Booléens

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

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

Nous avons, si u et v sont deux λ-termes,

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

Booléens

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

$$\begin{aligned} \text{vrai} &:= \lambda x.\lambda y.x, \\ \text{faux} &:= \lambda x.\lambda y.y, \end{aligned}$$

Nous avons, si u et v sont deux λ-termes,

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

Booléens

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

$$\begin{aligned} \text{vrai} &:= \lambda x.\lambda y.x, \\ \text{faux} &:= \lambda x.\lambda y.y, \end{aligned}$$

Nous avons, si u et v sont deux λ-termes,

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

Booléens

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

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

Nous avons, si u et v sont deux λ-termes,

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

Booléens

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

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

Nous avons, si u et v sont deux λ-termes,

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

De même,

$$(\text{faux}\, u)\, v$$

Booléens

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

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

Nous avons, si u et v sont deux λ-termes,

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

De même,

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

Booléens

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

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

Nous avons, si u et v sont deux λ-termes,

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

De même,

$$\begin{aligned} (\text{faux}\, u)\, v &= ((\lambda x.\lambda y.y)\, u)\, v \\ &\to_\beta (\lambda y.y)\, v \end{aligned}$$

Booléens

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

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

Nous avons, si u et v sont deux λ-termes,

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

De même,

$$\begin{aligned} (\text{faux}\, u)\, v &= ((\lambda x.\lambda y.y)\, u)\, v \\ &\to_\beta (\lambda y.y)\, v \\ &\to_\beta v. \end{aligned}$$

Booléens

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

$$\begin{aligned} \text{vrai} &:= \lambda x.\lambda y.x, && \text{renvoie son 1}^{\text{er}}\text{ argument,} \\ \text{faux} &:= \lambda x.\lambda y.y, && \text{renvoie son 2}^{\text{e}}\text{ argument.} \end{aligned}$$

Nous avons, si u et v sont deux λ-termes,

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

De même,

$$\begin{aligned} (\text{faux}\, u)\, v &= ((\lambda x.\lambda y.y)\, u)\, v \\ &\to_\beta (\lambda y.y)\, v \\ &\to_\beta v. \end{aligned}$$

Booléens

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

$$\begin{aligned} \text{vrai} &:= \lambda x.\lambda y.x, && \text{renvoie son 1}^{\text{er}}\text{ argument,} \\ \text{faux} &:= \lambda x.\lambda y.y, && \text{renvoie son 2}^{\text{e}}\text{ argument.} \end{aligned}$$

Nous avons, si u et v sont deux λ-termes,

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

De même,

$$\begin{aligned} (\text{faux}\, u)\, v &= ((\lambda x.\lambda y.y)\, u)\, v \\ &\to_\beta (\lambda y.y)\, v \\ &\to_\beta v. \end{aligned}$$

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 :

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

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,

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

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 :

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

Par exemple,

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

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 :

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

Par exemple,

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

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 :

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

Par exemple,

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

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 :

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

Par exemple,

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

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 :

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

Par exemple,

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

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 :

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

Par exemple,

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

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 :

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

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

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

Opérateurs booléens

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

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

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

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

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

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

Constructions conditionnelles

En s'appuyant sur le codage précédent des booléens, on code la construction conditionnelle sas (« **s**i **a**lors **s**inon ») de la manière suivante :

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

Constructions conditionnelles

En s'appuyant sur le codage précédent des booléens, on code la construction conditionnelle sas (« **s**i **a**lors **s**inon ») de la manière suivante :

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

Nous avons, si u et v sont deux λ-termes,

$$((\mathrm{sas}\ \mathrm{vrai})\,u)\,v$$

Constructions conditionnelles

En s'appuyant sur le codage précédent des booléens, on code la construction conditionnelle sas (« **s**i **a**lors **s**inon ») de la manière suivante :

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

Nous avons, si u et v sont deux λ-termes,

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

Constructions conditionnelles

En s'appuyant sur le codage précédent des booléens, on code la construction conditionnelle sas (« **s**i **a**lors **s**inon ») de la manière suivante :

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

Nous avons, si u et v sont deux λ-termes,

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

Constructions conditionnelles

En s'appuyant sur le codage précédent des booléens, on code la construction conditionnelle sas (« **s**i **a**lors **s**inon ») de la manière suivante :

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

Nous avons, si u et v sont deux λ-termes,

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

Constructions conditionnelles

En s'appuyant sur le codage précédent des booléens, on code la construction conditionnelle sas (« **s**i **a**lors **s**inon ») de la manière suivante :

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

Nous avons, si u et v sont deux λ-termes,

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

Constructions conditionnelles

En s'appuyant sur le codage précédent des booléens, on code la construction conditionnelle sas (« **s**i **a**lors **s**inon ») de la manière suivante :

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

Nous avons, si u et v sont deux λ-termes,

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

Constructions conditionnelles

En s'appuyant sur le codage précédent des booléens, on code la construction conditionnelle sas (« **s**i **a**lors **s**inon ») de la manière suivante :

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

Nous avons, si u et v sont deux λ-termes,

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

De même, on obtient

$$((\mathrm{sas}\,\mathrm{faux})\,u)\,v \to_\beta v.$$

Constructions conditionnelles

En s'appuyant sur le codage précédent des booléens, on code la construction conditionnelle sas (« **s**i **a**lors **s**inon ») de la manière suivante :

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

Nous avons, si u et v sont deux λ-termes,

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

De même, on obtient

$$((\mathrm{sas}\,\mathrm{faux})\,u)\,v \to_\beta v.$$

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

Entiers naturels

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

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

Entiers naturels

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

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

Voici les 1$^{\text{ers}}$ λ-termes codant les petits entiers :

Entiers naturels

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

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

Voici les 1$^{\text{ers}}$ λ-termes codant les petits entiers :

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

Entiers naturels

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

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

Voici les 1$^{\text{ers}}$ λ-termes codant les petits entiers :

$$\mathrm{ent}(0) = \lambda f.\lambda x.x,$$
$$\mathrm{ent}(1) = \lambda f.\lambda x.(f\ x),$$

Entiers naturels

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

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

Voici les 1$^{\text{ers}}$ λ-termes codant les petits entiers :

$$\begin{aligned} \mathrm{ent}(0) &= \lambda f.\lambda x.x, \\ \mathrm{ent}(1) &= \lambda f.\lambda x.(f\ x), \\ \mathrm{ent}(2) &= \lambda f.\lambda x.(f\ (f\ x)), \end{aligned}$$

Entiers naturels

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

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

Voici les 1$^{\text{ers}}$ λ-termes codant les petits entiers :

$$\begin{aligned} \mathrm{ent}(0) &= \lambda f.\lambda x.x, \\ \mathrm{ent}(1) &= \lambda f.\lambda x.(f\ x), \\ \mathrm{ent}(2) &= \lambda f.\lambda x.(f\ (f\ x)), \\ \mathrm{ent}(3) &= \lambda f.\lambda x.(f\ (f\ (f\ x))). \end{aligned}$$

Entiers naturels

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

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

Voici les 1$^{\text{ers}}$ λ-termes codant les petits entiers :

$$\begin{aligned} \mathrm{ent}(0) &= \lambda f.\lambda x.x, \\ \mathrm{ent}(1) &= \lambda f.\lambda x.(f\ x), \\ \mathrm{ent}(2) &= \lambda f.\lambda x.(f\ (f\ x)), \\ \mathrm{ent}(3) &= \lambda f.\lambda x.(f\ (f\ (f\ x))). \end{aligned}$$

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 $\mathrm{ent}(n)$ défini de la manière suivante :

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

Voici les 1$^{\text{ers}}$ λ-termes codant les petits entiers :

$$\begin{aligned} \mathrm{ent}(0) &= \lambda f.\lambda x.x, \\ \mathrm{ent}(1) &= \lambda f.\lambda x.(f\ x), \\ \mathrm{ent}(2) &= \lambda f.\lambda x.(f\ (f\ x)), \\ \mathrm{ent}(3) &= \lambda f.\lambda x.(f\ (f\ (f\ x))). \end{aligned}$$

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

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

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

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

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

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

Arithmétique sur les entiers de Church

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

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

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

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

Arithmétique sur les entiers de Church

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

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

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

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

Arithmétique sur les entiers de Church

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

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

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

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

Arithmétique sur les entiers de Church

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

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

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

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

Arithmétique sur les entiers de Church

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

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

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

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

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

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

Arithmétique sur les entiers de Church

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

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

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

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

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

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

Arithmétique sur les entiers de Church

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

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

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

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

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

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

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

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

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

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

Récursivité

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

Récursivité

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 ρ défini par

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

Récursivité

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 ρ défini par

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

Il vérifie la propriété fondamentale suivante : pour tout λ-terme u,

$$\rho\,u \;\to_\beta\; u\,(\rho\,u).$$

Récursivité

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 ρ défini par

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

Il vérifie la propriété fondamentale suivante : pour tout λ-terme u,

$$\rho\,u \;\to_\beta\; u\,(\rho\,u).$$

Ceci implique

$$\rho\,u \;\to_\beta\; u\,(\rho\,u) \;\to_\beta\; u\,(u\,(\rho\,u)) \;\to_\beta\; u\,(u\,(u\,(\rho\,u))) \;\to_\beta\; \cdots,$$

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

Plan

Représentation de λ-termes

On commence par définir un type pour représenter les variables :

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

Représentation de λ-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'α-renommage en particulier).

Représentation de λ-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'α-renommage en particulier).

Le type pour représenter les λ-termes est un type somme :

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

Représentation de λ-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'α-renommage en particulier).

Le type pour représenter les λ-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 λ-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 λ-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]
```