Programmation fonctionnelle CM8

Antonio E. Porreca

aeporreca.org/progfonc

Évaluation paresseuse en Haskell = réduction par l’extérieur en weak head normal form

Partage de sous-expressions

Inspiré par https://apfelmus.nfshost.com/articles/lazy-eval-intro.html

Exemple d’évaluation

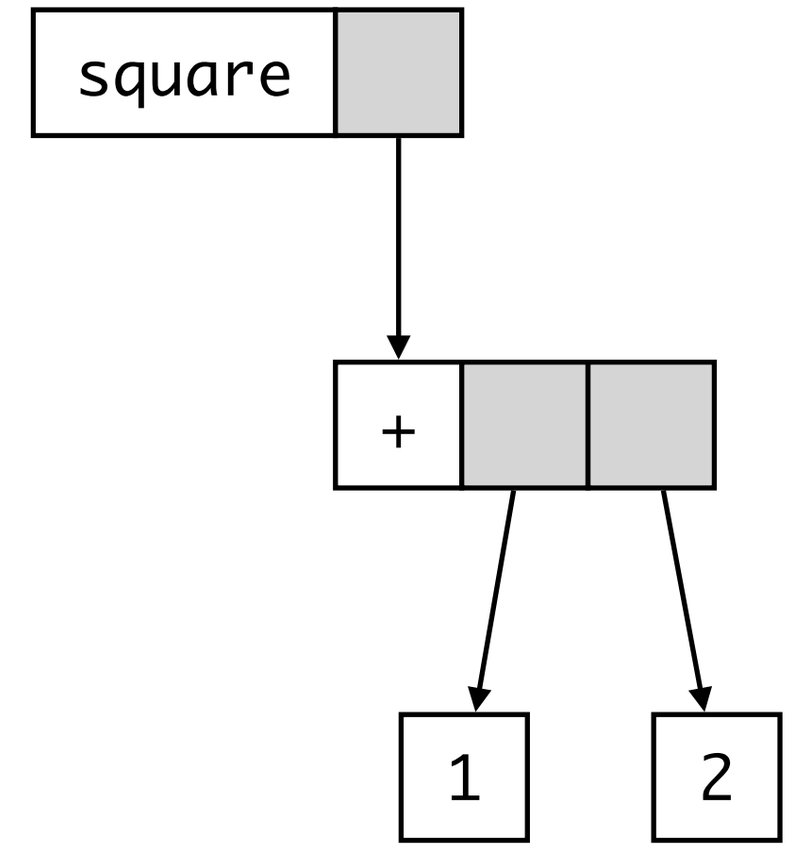
```
double (square (3 + 2))
= {application de double}
square (3 + 2) + square (3 + 2)
= {application de square}
((3 + 2) * (3 + 2)) + square (3 + 2)
= {application de (+)}
(5 * (3 + 2)) + square (3 + 2)
= {application de (+)}
(5 * 5) + square (3 + 2)
= {application de (*)}
25 + square (3 + 2)
= {application de square}
25 + ((3 + 2) * (3 + 2))
= {application de (+)}
25 + (5 * (3 + 2))
= {application de (+)}
25 + (5 * 5)
= {application de (*)}
25 + 25
= {application de (+)}
50
```

beaucoup de sous-expressions identiques réévaluées !

Réduction de graphes

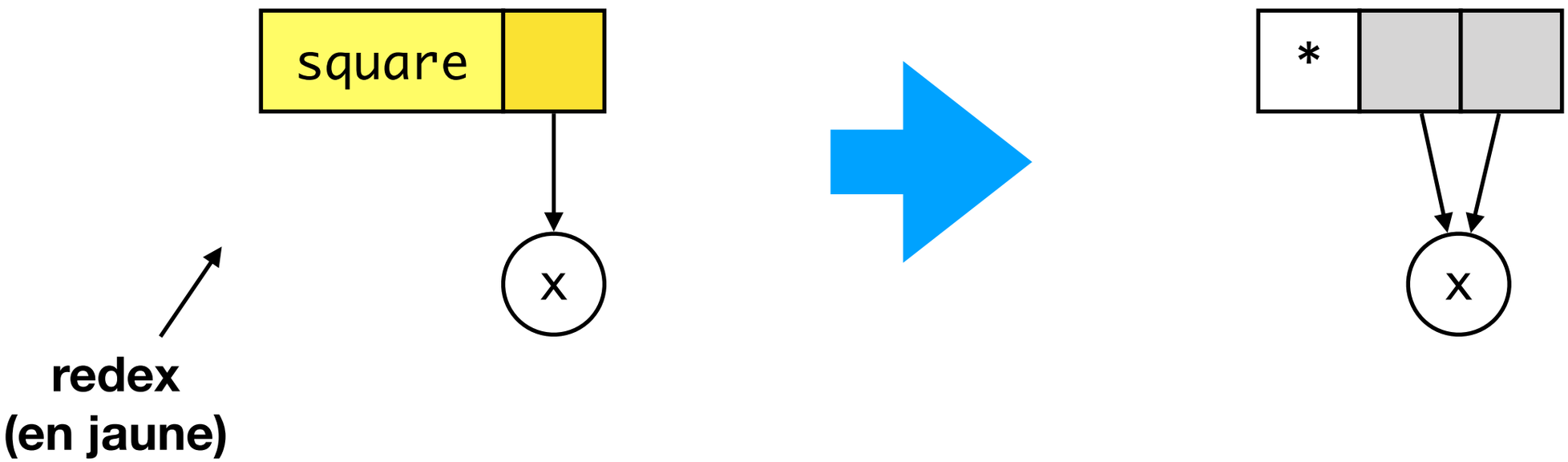
Représentation d'expressions par des graphes

```
square (1 + 2)
```

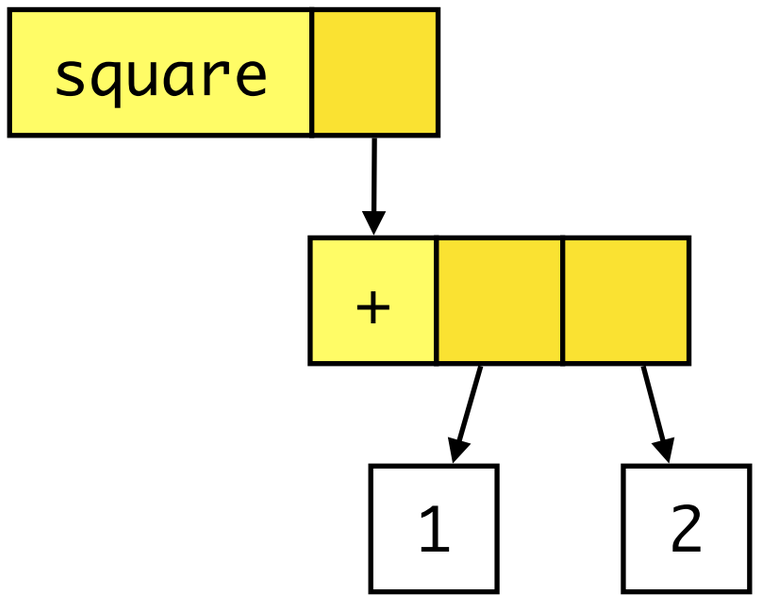


Définition de fonctions = règles de réduction

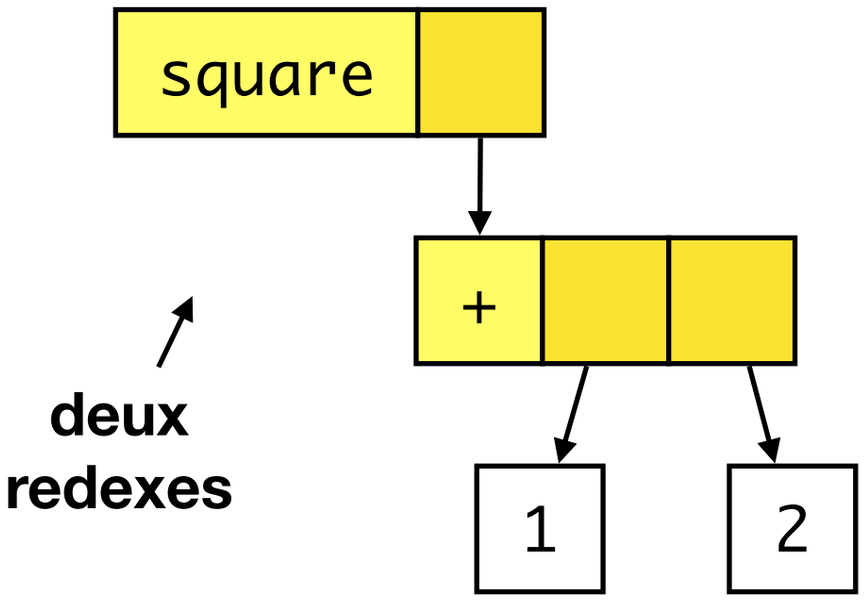
```
square x = x * x
```



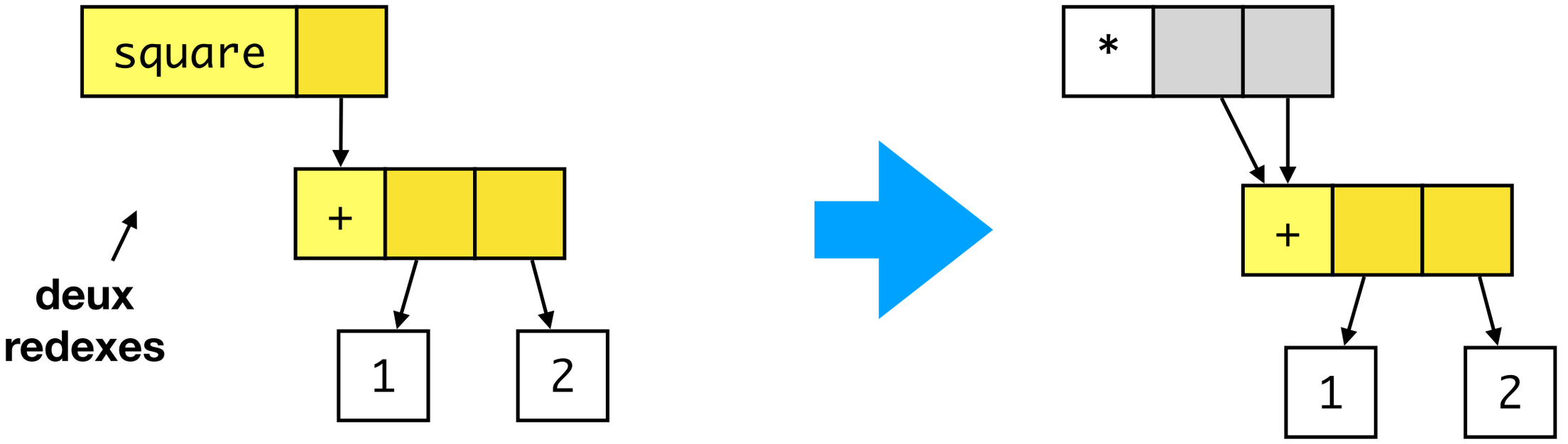
Évaluation paresseuse



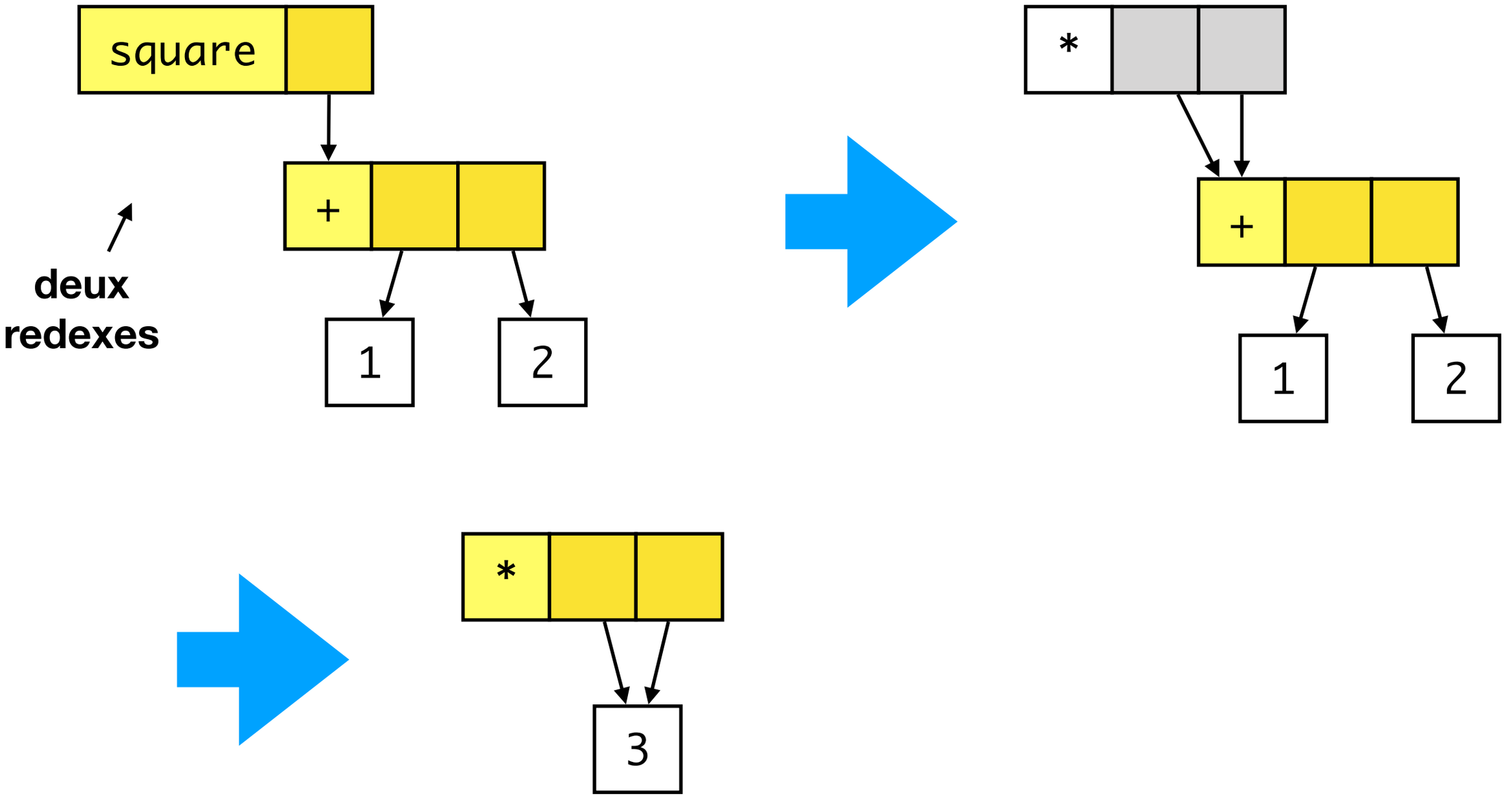
Évaluation paresseuse



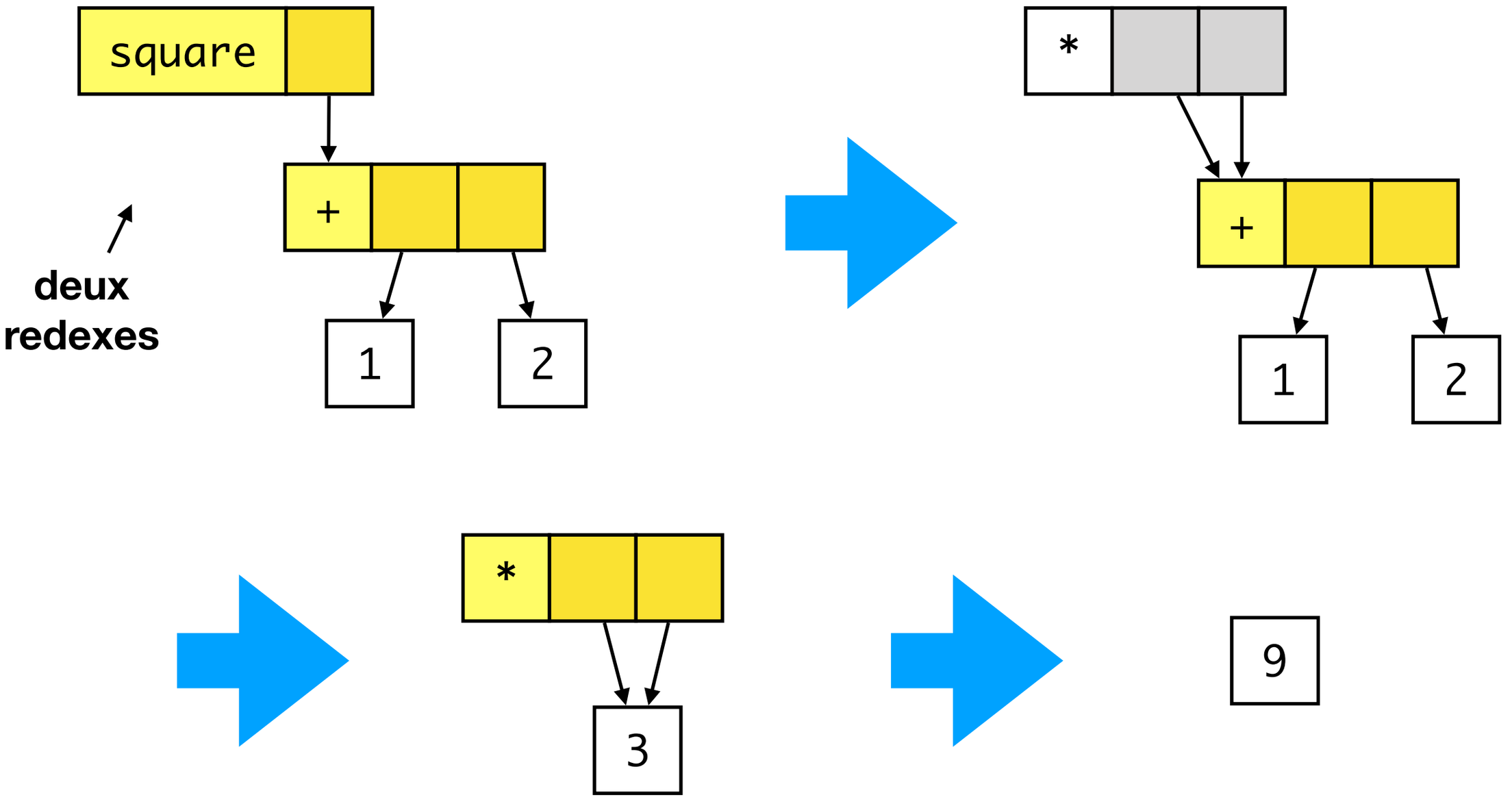
Évaluation paresseuse



Évaluation paresseuse



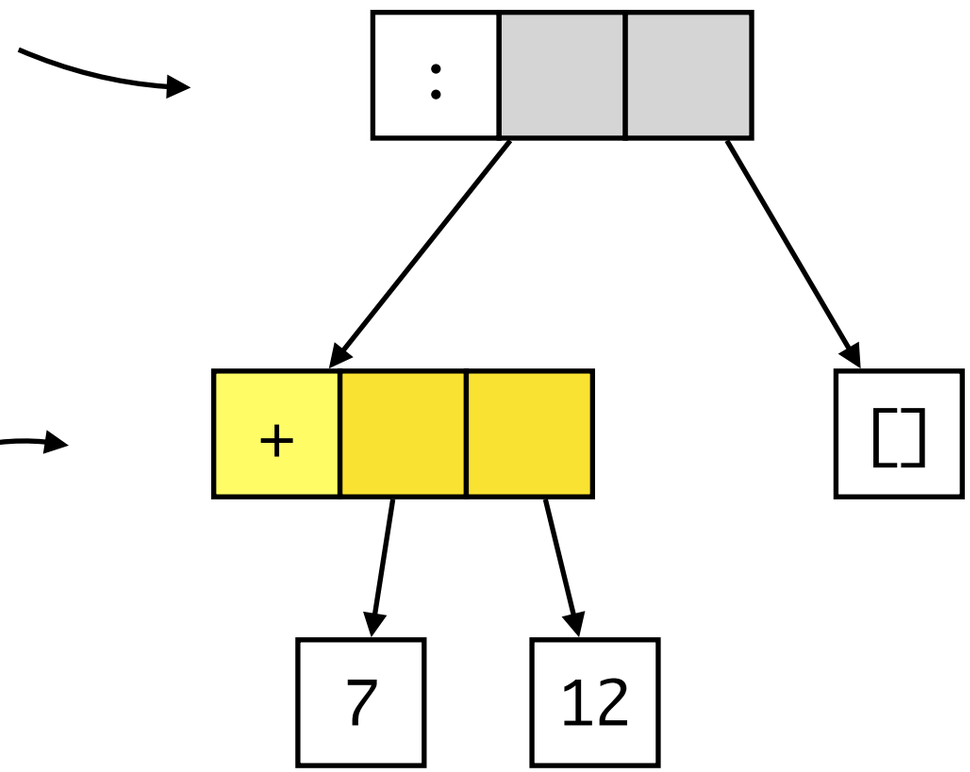
Évaluation paresseuse



Weak head normal form

`(:)` **ne donne pas un redex, donc on arrête l'évaluation**

`(7 + 12):[]`



un redex, mais on ne le réduit pas

Representation textuelle

```
double x = x + x
square x = x * x
```

Representation textuelle

```
double x = x + x
square x = x * x


double (square (3 + 2))
```

Representation textuelle

```
double x = x + x
square x = x * x


double (square (3 + 2))
= {application de double}
```

Representation textuelle

```
double x = x + x
square x = x * x
```

```
double (square (3 + 2))
= {application de double}
let x = square (3 + 2) in x + x
```

Representation textuelle

```
double x = x + x
square x = x * x
```

```
double (square (3 + 2))
= {application de double}
let x = square (3 + 2) in x + x
= {application de square}
```

Representation textuelle

```
double x = x + x
square x = x * x
```

```
double (square (3 + 2))
= {application de double}
let x = square (3 + 2) in x + x
= {application de square}
let x' = 3 + 2 in let x = x' * x' in x + x
```

Representation textuelle

```
double x = x + x
square x = x * x
```

```
double (square (3 + 2))
= {application de double}
let x = square (3 + 2) in x + x
= {application de square}
let x' = 3 + 2 in let x = x' * x' in x + x
= {application de (+)}
```

Representation textuelle

```
double x = x + x
square x = x * x
```

```
double (square (3 + 2))
= {application de double}
let x = square (3 + 2) in x + x
= {application de square}
let x' = 3 + 2 in let x = x' * x' in x + x
= {application de (+)}
let x' = 5 in let x = x' * x' in x + x
```

Representation textuelle

```
double x = x + x
square x = x * x
```

```
double (square (3 + 2))
= {application de double}
let x = square (3 + 2) in x + x
= {application de square}
let x' = 3 + 2 in let x = x' * x' in x + x
= {application de (+)}
let x' = 5 in let x = x' * x' in x + x
= {substitution}
```

Representation textuelle

```
double x = x + x
square x = x * x
```

```
double (square (3 + 2))
= {application de double}
let x = square (3 + 2) in x + x
= {application de square}
let x' = 3 + 2 in let x = x' * x' in x + x
= {application de (+)}
let x' = 5 in let x = x' * x' in x + x
= {substitution}
let x = 5 * 5 in x + x
```

Representation textuelle

```
double x = x + x
square x = x * x
```

```
double (square (3 + 2))
= {application de double}
let x = square (3 + 2) in x + x
= {application de square}
let x' = 3 + 2 in let x = x' * x' in x + x
= {application de (+)}
let x' = 5 in let x = x' * x' in x + x
= {substitution}
let x = 5 * 5 in x + x
= {application de (*)}
```

Representation textuelle

```
double x = x + x
square x = x * x
```

```
double (square (3 + 2))
= {application de double}
let x = square (3 + 2) in x + x
= {application de square}
let x' = 3 + 2 in let x = x' * x' in x + x
= {application de (+)}
let x' = 5 in let x = x' * x' in x + x
= {substitution}
let x = 5 * 5 in x + x
= {application de (*)}
let x = 25 in x + x
```

Representation textuelle

```
double x = x + x
square x = x * x
```

```
double (square (3 + 2))
= {application de double}
let x = square (3 + 2) in x + x
= {application de square}
let x' = 3 + 2 in let x = x' * x' in x + x
= {application de (+)}
let x' = 5 in let x = x' * x' in x + x
= {substitution}
let x = 5 * 5 in x + x
= {application de (*)}
let x = 25 in x + x
= {substitution}
```

Representation textuelle

```
double x = x + x
square x = x * x
```

```
double (square (3 + 2))
= {application de double}
let x = square (3 + 2) in x + x
= {application de square}
let x' = 3 + 2 in let x = x' * x' in x + x
= {application de (+)}
let x' = 5 in let x = x' * x' in x + x
= {substitution}
let x = 5 * 5 in x + x
= {application de (*)}
let x = 25 in x + x
= {substitution}
25 + 25
```

Representation textuelle

```
double x = x + x
square x = x * x
```

```
double (square (3 + 2))
= {application de double}
let x = square (3 + 2) in x + x
= {application de square}
let x' = 3 + 2 in let x = x' * x' in x + x
= {application de (+)}
let x' = 5 in let x = x' * x' in x + x
= {substitution}
let x = 5 * 5 in x + x
= {application de (*)}
let x = 25 in x + x
= {substitution}
25 + 25
= {application de (+)}
```

Representation textuelle

```
double x = x + x
square x = x * x
```

```
double (square (3 + 2))
= {application de double}
let x = square (3 + 2) in x + x
= {application de square}
let x' = 3 + 2 in let x = x' * x' in x + x
= {application de (+)}
let x' = 5 in let x = x' * x' in x + x
= {substitution}
let x = 5 * 5 in x + x
= {application de (*)}
let x = 25 in x + x
= {substitution}
25 + 25
= {application de (+)}
50
```

✏️ Exercice 8.1

- Soit

```
repeat2 :: [a] -> [a]
repeat2 xs = xs ++ xs
```

- Évaluer en weak head normal form par réécriture de graphes l’expression

```
repeat2 (map (+1) [1,2,3])
```

- Évaluer l’expression en WHNF sous forme textuelle

Raisonner par équations sur les programmes

La fonction `reverse`

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

La fonction `reverse`

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

Propriété : `reverse [x] = [x]`

La fonction `reverse`

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

Propriété : `reverse [x] = [x]`

```
reverse [x]
```

La fonction `reverse`

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

Propriété : `reverse [x] = [x]`

```
reverse [x]
= {notation}
```

La fonction `reverse`

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

Propriété : `reverse [x] = [x]`

```
reverse [x]
= {notation}
reverse (x:[])
```

La fonction `reverse`

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

Propriété : `reverse [x] = [x]`

```
reverse [x]
= {notation}
reverse (x:[])
= {application de reverse}
```

La fonction `reverse`

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

Propriété : `reverse [x] = [x]`

```
reverse [x]
= {notation}
reverse (x:[])
= {application de reverse}
reverse [] ++ [x]
```

La fonction `reverse`

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

Propriété : `reverse [x] = [x]`

```
reverse [x]
= {notation}
reverse (x:[])
= {application de reverse}
reverse [] ++ [x]
= {application de reverse}
```

La fonction `reverse`

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

Propriété : `reverse [x] = [x]`

```
reverse [x]
= {notation}
reverse (x:[])
= {application de reverse}
reverse [] ++ [x]
= {application de reverse}
[] ++ [x]
```

La fonction `reverse`

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

Propriété : `reverse [x] = [x]`

```
reverse [x]
= {notation}
reverse (x:[])
= {application de reverse}
reverse [] ++ [x]
= {application de reverse}
[] ++ [x]
= {application de (++)}
```

La fonction `reverse`

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

Propriété : `reverse [x] = [x]`

```
reverse [x]
= {notation}
reverse (x:[])
= {application de reverse}
reverse [] ++ [x]
= {application de reverse}
[] ++ [x]
= {application de (++)}
[x]
```

Induction sur les listes finies

La fonction `reverse`

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

Propriété : `reverse (xs ++ ys) = reverse ys ++ reverse xs`

La fonction `reverse`

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

Propriété : `reverse (xs ++ ys) = reverse ys ++ reverse xs`

cas de base

La fonction `reverse`

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

Propriété : `reverse (xs ++ ys) = reverse ys ++ reverse xs`

cas de base

`reverse ([] ++ ys)`

La fonction `reverse`

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

Propriété : `reverse (xs ++ ys) = reverse ys ++ reverse xs`

cas de base

```
reverse ([] ++ ys)
= {application de (++)}
```

La fonction `reverse`

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

Propriété : `reverse (xs ++ ys) = reverse ys ++ reverse xs`

cas de base

```
reverse ([] ++ ys)
= {application de (++)}
reverse ys
```

La fonction `reverse`

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

Propriété : `reverse (xs ++ ys) = reverse ys ++ reverse xs`

cas de base

```
reverse ([] ++ ys)
= {application de (++)}
reverse ys
= {désapplication de (++)}
```

La fonction `reverse`

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

Propriété : `reverse (xs ++ ys) = reverse ys ++ reverse xs`

cas de base

```
reverse ([] ++ ys)
= {application de (++)}
reverse ys
= {désapplication de (++)}
reverse ys ++ []
```

La fonction `reverse`

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

Propriété : `reverse (xs ++ ys) = reverse ys ++ reverse xs`

cas de base

```
reverse ([] ++ ys)
= {application de (++)}
reverse ys
= {désapplication de (++)}
reverse ys ++ []
= {désapplication de reverse}
```

La fonction `reverse`

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

Propriété : `reverse (xs ++ ys) = reverse ys ++ reverse xs`

cas de base

```
reverse ([] ++ ys)
= {application de (++)}
reverse ys
= {désapplication de (++)}
reverse ys ++ []
= {désapplication de reverse}
reverse ys ++ reverse []
```

La fonction `reverse`

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

Propriété : `reverse (xs ++ ys) = reverse ys ++ reverse xs`

cas de base

```
reverse ([] ++ ys)
= {application de (++)}
reverse ys
= {désapplication de (++)}
reverse ys ++ []
= {désapplication de reverse}
reverse ys ++ reverse []
```

cas récursif

La fonction `reverse`

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

Propriété : `reverse (xs ++ ys) = reverse ys ++ reverse xs`

cas de base

```
reverse ([] ++ ys)
= {application de (++)}
reverse ys
= {désapplication de (++)}
reverse ys ++ []
= {désapplication de reverse}
reverse ys ++ reverse []
```

cas récursif

```
reverse ((x:xs) ++ ys)
```

La fonction `reverse`

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

Propriété : `reverse (xs ++ ys) = reverse ys ++ reverse xs`

cas de base

```
reverse ([] ++ ys)
= {application de (++)}
reverse ys
= {désapplication de (++)}
reverse ys ++ []
= {désapplication de reverse}
reverse ys ++ reverse []
```

cas récursif

```
reverse ((x:xs) ++ ys)
= {application de (++)}
```

La fonction `reverse`

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

Propriété : `reverse (xs ++ ys) = reverse ys ++ reverse xs`

cas de base

```
reverse ([] ++ ys)
= {application de (++)}
reverse ys
= {désapplication de (++)}
reverse ys ++ []
= {désapplication de reverse}
reverse ys ++ reverse []
```

cas récursif

```
reverse ((x:xs) ++ ys)
= {application de (++)}
reverse (x:(xs ++ ys))
```

La fonction `reverse`

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

Propriété : `reverse (xs ++ ys) = reverse ys ++ reverse xs`

cas de base

```
reverse ([] ++ ys)
= {application de (++)}
reverse ys
= {désapplication de (++)}
reverse ys ++ []
= {désapplication de reverse}
reverse ys ++ reverse []
```

cas récursif

```
reverse ((x:xs) ++ ys)
= {application de (++)}
reverse (x:(xs ++ ys))
= {application de reverse}
```

La fonction `reverse`

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

Propriété : `reverse (xs ++ ys) = reverse ys ++ reverse xs`

cas de base

```
reverse ([] ++ ys)
= {application de (++)}
reverse ys
= {désapplication de (++)}
reverse ys ++ []
= {désapplication de reverse}
reverse ys ++ reverse []
```

cas récursif

```
reverse ((x:xs) ++ ys)
= {application de (++)}
reverse (x:(xs ++ ys))
= {application de reverse}
reverse (xs ++ ys) ++ [x]
```

La fonction `reverse`

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

Propriété : `reverse (xs ++ ys) = reverse ys ++ reverse xs`

cas de base

```
reverse ([] ++ ys)
= {application de (++)}
reverse ys
= {désapplication de (++)}
reverse ys ++ []
= {désapplication de reverse}
reverse ys ++ reverse []
```

cas récursif

```
reverse ((x:xs) ++ ys)
= {application de (++)}
reverse (x:(xs ++ ys))
= {application de reverse}
reverse (xs ++ ys) ++ [x]
= {hypothèse d'induction}
```

La fonction `reverse`

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

Propriété : `reverse (xs ++ ys) = reverse ys ++ reverse xs`

cas de base

```
reverse ([] ++ ys)
= {application de (++)}
reverse ys
= {désapplication de (++)}
reverse ys ++ []
= {désapplication de reverse}
reverse ys ++ reverse []
```

cas récursif

```
reverse ((x:xs) ++ ys)
= {application de (++)}
reverse (x:(xs ++ ys))
= {application de reverse}
reverse (xs ++ ys) ++ [x]
= {hypothèse d'induction}
(reverse ys ++ reverse xs) ++ [x]
```

La fonction `reverse`

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

Propriété : `reverse (xs ++ ys) = reverse ys ++ reverse xs`

cas de base

```
reverse ([] ++ ys)
= {application de (++)}
reverse ys
= {désapplication de (++)}
reverse ys ++ []
= {désapplication de reverse}
reverse ys ++ reverse []
```

cas récursif

```
reverse ((x:xs) ++ ys)
= {application de (++)}
reverse (x:(xs ++ ys))
= {application de reverse}
reverse (xs ++ ys) ++ [x]
= {hypothèse d'induction}
(reverse ys ++ reverse xs) ++ [x]
= {associativité de (++)}
```

La fonction `reverse`

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

Propriété : `reverse (xs ++ ys) = reverse ys ++ reverse xs`

cas de base

```
reverse ([] ++ ys)
= {application de (++)}
reverse ys
= {désapplication de (++)}
reverse ys ++ []
= {désapplication de reverse}
reverse ys ++ reverse []
```

cas récursif

```
reverse ((x:xs) ++ ys)
= {application de (++)}
reverse (x:(xs ++ ys))
= {application de reverse}
reverse (xs ++ ys) ++ [x]
= {hypothèse d'induction}
(reverse ys ++ reverse xs) ++ [x]
= {associativité de (++)}
reverse ys ++ (reverse xs ++ [x])
```

La fonction `reverse`

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

Propriété : `reverse (xs ++ ys) = reverse ys ++ reverse xs`

cas de base

```
reverse ([] ++ ys)
= {application de (++)}
reverse ys
= {désapplication de (++)}
reverse ys ++ []
= {désapplication de reverse}
reverse ys ++ reverse []
```

cas récursif

```
reverse ((x:xs) ++ ys)
= {application de (++)}
reverse (x:(xs ++ ys))
= {application de reverse}
reverse (xs ++ ys) ++ [x]
= {hypothèse d'induction}
(reverse ys ++ reverse xs) ++ [x]
= {associativité de (++)}
reverse ys ++ (reverse xs ++ [x])
= {désapplication de reverse}
```

La fonction `reverse`

```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

Propriété : `reverse (xs ++ ys) = reverse ys ++ reverse xs`

cas de base

```
reverse ([] ++ ys)
= {application de (++)}
reverse ys
= {désapplication de (++)}
reverse ys ++ []
= {désapplication de reverse}
reverse ys ++ reverse []
```

cas récursif

```
reverse ((x:xs) ++ ys)
= {application de (++)}
reverse (x:(xs ++ ys))
= {application de reverse}
reverse (xs ++ ys) ++ [x]
= {hypothèse d'induction}
(reverse ys ++ reverse xs) ++ [x]
= {associativité de (++)}
reverse ys ++ (reverse xs ++ [x])
= {désapplication de reverse}
reverse ys ++ reverse (x:xs)
```

✏️ Exercice 8.2

- Soit

```
(++) :: [a] -> [a]
[] ++ ys      = ys
(x:xs) ++ ys = x:(xs ++ ys)
```

- Montrer que pour tout xs a

```
xs ++ [] = xs
```

- Montrer la propriété associative de (++), c-à-d que, pour tout xs, ys, zs, on a

```
xs ++ (ys ++ zs) = (xs ++ ys) ++ zs
```

✏️ Exercice 8.3

- En utilisant les propriétés

```
reverse (xs ++ ys) =
    reverse ys ++ reverse xs
reverse [x] = [x]
```

- Montrer que pour tout xs a

```
reverse (reverse xs) = xs
```

✏️ Exercice 8.4

- En utilisant les définitions

```
map f [] = []
map f (x:xs) = f x : map f xs
(f . g) x = f (g x)
```

- Montrer que

```
map f (map g xs) = map (f . g) xs
```