

Programmation Fonctionnelle

Adrien Durier

13 septembre 2023

Langages dans le Monde

- **Langages naturel** : Environ 7 000 dans le monde



Langages dans le Monde

- **Langages naturel** : Environ 7 000 dans le monde
→ Probablement overkill ?



Langages dans le Monde

- **Langages naturel** : Environ 7 000 dans le monde
→ Probablement overkill ?
- **Langages de programmation** : 700+ utilisés aujourd'hui



Langages dans le Monde

- **Langages naturel** : Environ 7 000 dans le monde
→ Probablement overkill ?
- **Langages de programmation** : 700+ utilisés aujourd'hui
→ Probablement overkill ?





```
let rec fibonacci n =  
  match n with  
  | 0 -> 0  
  | 1 -> 1  
  | _ -> fibonacci (n - 1) + fibonacci (n - 2)
```

Correct + Efficace + Éléphant

C (*impératif*)

```
int fibonacci(int n) {  
    int a = 0;  
    int b = 1;  
    int c;  
    int i;  
    if (n == 0) return 0;  
    for (i = 2; i <= n; i++) {  
        c = a + b;  
        a = b;  
        b = c;  
    }  
    return b;  
}
```

fibonacci = 1, 1, 2, 3, 5, 8, 13, 21...

```
int fibonacci(int n) {  
    int a = 0;  
    int b = 1;  
    int c;  
    int i;  
    if (n == 0) return 0;  
    for (i = 2; i <= n; i++) {  
        c = a + b;  
        a = b;  
        b = c;  
    }  
    return b;  
}
```

```
let rec fibonacci n =  
    match n with  
    | 0 -> 0  
    | 1 -> 1  
    | _ -> fibonacci (n - 1)  
           + fibonacci (n - 2)
```

fibonacci = 1, 1, 2, 3, 5, 8, 13, 21...

Un peu de code !

```
let rec fibonacci n =  
  if n = 0 then 0  
  else if n = 1 then 1  
  else fibonacci (n - 1) + fibonacci (n - 2)
```

- **let** : déclaration de "variables" ou de fonctions.

Un peu de code !

```
let rec fibonacci n =  
  if n = 0 then 0  
  else if n = 1 then 1  
  else fibonacci (n - 1) + fibonacci (n - 2)
```

- **let** : déclaration de "variables" ou de fonctions.

→ Comme en maths !

Soit x tel que ...

let x such that ...

Un peu de code !

```
let rec fibonacci n =  
  if n = 0 then 0  
  else if n = 1 then 1  
  else fibonacci (n - 1) + fibonacci (n - 2)
```

- **let** : **déclaration** de "variables" ou de fonctions.
- **rec** : Indique que la fonction est **récursive**.

Un peu de code !

```
let rec fibonacci n =  
  if n = 0 then 0  
  else if n = 1 then 1  
  else fibonacci (n - 1) + fibonacci (n - 2)
```

- **let** : déclaration de "variables" ou de fonctions.
- **rec** : Indique que la fonction est **récursive**.
- **fibonacci** : Appel **récuratif**
→ **fibonacci** n utilise **fibonacci** $(n - 1)$ et **fibonacci** $(n - 2)$

Un peu de code !

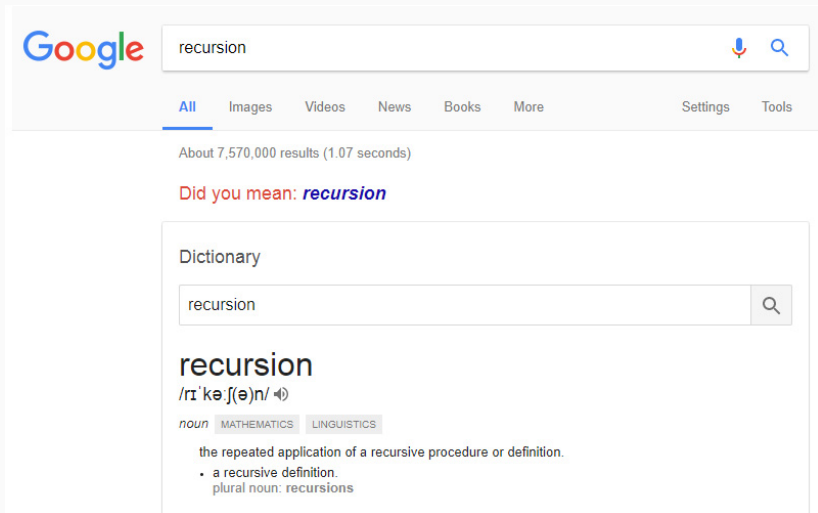
```
let rec fibonacci n =  
  if n = 0 then 0  
  else if n = 1 then 1  
  else fibonacci (n - 1) + fibonacci (n - 2)
```

- **let** : déclaration de "variables" ou de fonctions.
- **rec** : Indique que la fonction est **récursive**.
- **fibonacci** : Appel **récuratif**
→ **fibonacci** n utilise **fibonacci** $(n - 1)$ et **fibonacci** $(n - 2)$
- **if then else** : Conditionnelle classique. **Vous connaissez !**

Un peu de code !

```
let rec fibonacci n =  
  match n with  
  | 0 -> 0  
  | 1 -> 1  
  | _ -> fibonacci (n - 1) + fibonacci (n - 2)
```

- **let** : déclaration de "variables" ou de fonctions.
- **rec** : Indique que la fonction est **récursive**.
- **fibonacci** : Appel **récuratif**
→ **fibonacci** n utilise **fibonacci** $(n - 1)$ et **fibonacci** $(n - 2)$
- ~~**if then else**~~ : Conditionnelle classique. ~~Vous connaissez !~~
- **match ...with** : Filtrage.
→ Plus souple, plus concis, plus puissant !



Google

recursion

About 7,570,000 results (1.07 seconds)

Did you mean: *recursion*

Dictionary

recursion

recursion
/rɪˈkæʃ(ə)n/ ⓘ

noun MATHEMATICS LINGUISTICS

the repeated application of a recursive procedure or definition.

- a recursive definition.

plural noun: recursions

C (*impératif*)

vs.

OCaml

```
int factorial(int n) {  
    int result = 1;  
    for (int i = 1; i <= n; ++i) {  
        result *= i;  
    }  
    return result;  
}
```

```
let rec factorial n =  
    if n = 0 then 1  
    else n * factorial (n - 1)
```

Ca marche aussi avec la fonction factorielle !

Pourquoi la programmation fonctionnelle ?

Pourquoi Différents Langages de Programmation ?

- Objectifs différents :
 - **C (1972)** : Contrôle bas niveau, performance, compilé
 - **Python (1991)** : Facilité d'utilisation, polyvalence, hybride
 - **SQL (1974)** : Manipulation de bases de données, interprété
 - **R (1993)** : Analyse statistique, interprété
 - **JavaScript (1995)** : Développement web, interprété

Pourquoi Différents Langages de Programmation ?

- Objectifs différents :
 - **C (1972)** : Contrôle bas niveau, performance, compilé
 - **Python (1991)** : Facilité d'utilisation, polyvalence, hybride
 - **SQL (1974)** : Manipulation de bases de données, interprété
 - **R (1993)** : Analyse statistique, interprété
 - **JavaScript (1995)** : Développement web, interprété
- Caractéristiques variées :
 - **Typage statique vs dynamique**
 - **Langages compilés vs interprétés**
 - **Paradigmes :**
 - **Impératif (1950s)** : Fortran, C
 - **Fonctionnel (1950s)** : Lisp, Haskell, OCaml
 - **Orienté-Objet (1960s)** : Simula, Smalltalk, Java

Pourquoi Différents Langages de Programmation ?

- Objectifs différents :
 - **C (1972)** : Contrôle bas niveau, performance, compilé
 - **Python (1991)** : Facilité d'utilisation, polyvalence, hybride
 - **SQL (1974)** : Manipulation de bases de données, interprété
 - **R (1993)** : Analyse statistique, interprété
 - **JavaScript (1995)** : Développement web, interprété
- Caractéristiques variées :
 - **Typage statique vs dynamique**
 - **Langages compilés vs interprétés**
 - **Paradigmes :**
 - **Impératif (1950s)** : Fortran, C
 - **Fonctionnel (1950s)** : Lisp, Haskell, OCaml
 - **Orienté-Objet (1960s)** : Simula, Smalltalk, Java

Quels langages avez-vous déjà utilisés et pour quels types de projets ?

- Origine :
 - λ -calcul (Church, 1930s)
 - ML (1973)
 - **But** : Preuve de programmes

Un peu d'histoire

- **Origine :**

- λ -calcul (Church, 1930s)
- ML (1973)

→ **But** : Preuve de programmes

- **OCaml :**

- Compilé et Interprété
- Presque aussi rapide que C
- Pas de gestion mémoire
- Haut niveau, Abstrait
- Fortement typé
- **O** de **Ocaml** :
Orienté Objet



Robin Milner

Inventeur de ML, Prix Turing



Coq



Facebook
(Messenger)



WOLFRAM



F*



Frama-C

AIRBUS

Principales applications : *vérification* de programme, manipulation de *code*, applications *critiques*

OCaml

<https://ocaml.org>

Développé par **INRIA**

OCaml est avant tout un **langage compilé**.

Mais il peut être **testé** grâce à un interpréteur.

Pour lancer l'interpréteur :

- Une fois OCaml installé, avec la commande '**ocaml**'
- Avec n'importe quel navigateur à l'adresse :

try.ocamlpro.com

Installation

- Guides d'installation pour **Windows** et **MacOS** :
 - ocaml.org/docs/up-and-running
 - cs3110.github.io/textbook/chapters/preface/install.html
- **Installation sous Linux** :
 1. Installer le paquet opam depuis les dépôts de votre distro :
`pacman -S opam`
(remplacer pacman par le gestionnaire de paquets de votre distro, apt, dnf...)
 2. Initialiser opam :
`opam init`
`eval $(opam env)`

Et voilà !



Du mauvais code (en C)

- OCaml utilise un système de *Garbage Collection*
- Pas d'allocation mémoire
- Libérée automatiquement
- Réduit les risques de fuites
- *(léger) impact sur les performances*

Évaluation d'une expression en OCaml

Étapes de l'évaluation

L'évaluation d'une expression se fait en trois temps :

1. Je saisis l'expression et j'indique que j'ai terminé en tapant ; ;
2. L'interpréteur évalue mon expression
3. Puis il affiche son type et sa valeur

```
> ocaml
OCaml version 4.01.0
# 1 + 2 ; ;
- : int = 3
#
```

Récursion

Recursion is recursion



Puissance Récursive

```
let rec power x n =  
  if n = 0 then 1  
  else x * power x (n - 1)
```

```
let rec power x n =  
  if n = 0 then 1  
  else x * power x (n - 1)
```

Exécution de la fonction power 2 3

- $\text{power } 2 \ 3 = 2 \times \text{power } 2 \ 2$


```
let rec power x n =  
  if n = 0 then 1  
  else x * power x (n - 1)
```

Exécution de la fonction power 2 3

- $\text{power } 2 \ 3 = 2 \times \text{power } 2 \ 2$
- $\text{power } 2 \ 2 = 2 \times \text{power } 2 \ 1$

```
let rec power x n =  
  if n = 0 then 1  
  else x * power x (n - 1)
```

Exécution de la fonction power 2 3

- $\text{power } 2 \ 3 = 2 \times \text{power } 2 \ 2$
- $\text{power } 2 \ 2 = 2 \times \text{power } 2 \ 1$
- $\text{power } 2 \ 1 = 2 \times \text{power } 2 \ 0$

```
let rec power x n =  
  if n = 0 then 1  
  else x * power x (n - 1)
```

Exécution de la fonction power 2 3

- $\text{power } 2 \ 3 = 2 \times \text{power } 2 \ 2$
- $\text{power } 2 \ 2 = 2 \times \text{power } 2 \ 1$
- $\text{power } 2 \ 1 = 2 \times \text{power } 2 \ 0$
- $\text{power } 2 \ 0 = 1$

```
let rec power x n =  
  if n = 0 then 1  
  else x * power x (n - 1)
```

Exécution de la fonction power 2 3

- $\text{power } 2 \ 3 = 2 \times \text{power } 2 \ 2$
- $\text{power } 2 \ 2 = 2 \times \text{power } 2 \ 1$
- $\text{power } 2 \ 1 = 2 \times \text{power } 2 \ 0$
- $\text{power } 2 \ 0 = 1$
- $\text{power } 2 \ 1 = 2$

```
let rec power x n =  
  if n = 0 then 1  
  else x * power x (n - 1)
```

Exécution de la fonction power 2 3

- $\text{power } 2 \ 3 = 2 \times \text{power } 2 \ 2$
- $\text{power } 2 \ 2 = 2 \times \text{power } 2 \ 1$
- $\text{power } 2 \ 1 = 2 \times \text{power } 2 \ 0$
- $\text{power } 2 \ 0 = 1$
- $\text{power } 2 \ 1 = 2$
- $\text{power } 2 \ 2 = 4$

```
let rec power x n =  
  if n = 0 then 1  
  else x * power x (n - 1)
```

Exécution de la fonction power 2 3

- $\text{power } 2 \ 3 = 2 \times \text{power } 2 \ 2$
- $\text{power } 2 \ 2 = 2 \times \text{power } 2 \ 1$
- $\text{power } 2 \ 1 = 2 \times \text{power } 2 \ 0$
- $\text{power } 2 \ 0 = 1$
- $\text{power } 2 \ 1 = 2$
- $\text{power } 2 \ 2 = 4$
- $\text{power } 2 \ 3 = 8$

Attention aux boucles infinies !

```
let rec f x =  
  if x = 0 then 0  
  else x + f (x - 1)
```

Évaluation

Que se passe-t-il si on évalue la déclaration suivante ?

```
let () = Printf.printf "%d\n" (f (-1))
```

Résultat : **Stack overflow during evaluation.**

Déclarations locales et globales

Déclarations globales et portée

```
let x = 4
let y = 20
let z = 0
let f z = x + z * y
let g z a = (f z) + 3 (* g z a = 4 + z * 20 + 3 *)
let g z _ = (f z) + 3 (* même fonction! *)
let x = 2
let y = x + x
let f = g (z + 3) y
let _ = if f > 65
    then Printf.printf "%d\n" (g 3 (f * y))
    else Printf.printf "ERROR"
```

Déclarations globales et portée

```
let x = 4
let y = 20
let z = 0
let f z = x + z * y
let g z a = (f z) + 3 (* g z a = 4 + z * 20 + 3 *)
let g z _ = (f z) + 3 (* même fonction! *)
let x = 2
let y = x + x
let f = g (z + 3) y
let _ = if f > 65
    then Printf.printf "%d\n" (g 3 (f * y))
    else Printf.printf "ERROR"
```

Déclarations globales et portée

```
let x = 4
let y = 20
let z = 0
let f z = x + z * y
let g z a = (f z) + 3 (* g z a = 4 + z * 20 + 3 *)
let g z _ = (f z) + 3 (* même fonction! *)
let x = 2
let y = x + x
let f = g (z + 3) y
let _ = if f > 65
    then Printf.printf "%d\n" (g 3 (f * y))
    else Printf.printf "ERROR"
```

Déclarations globales et portée

```
let x = 4
let y = 20
let z = 0
let f z = x + z * y
let g z a = (f z) + 3 (* g z a = 4 + z * 20 + 3 *)
let g z _ = (f z) + 3 (* même fonction! *)
let x = 2
let y = x + x
let f = g (z + 3) y
let _ = if f > 65
    then Printf.printf "%d\n" (g 3 (f * y))
    else Printf.printf "ERROR"
```

Déclarations globales et portée

```
let x = 4
let y = 20
let z = 0
let f z = x + z * y
let g z a = (f z) + 3 (* g z a = 4 + z * 20 + 3 *)
let g z _ = (f z) + 3 (* même fonction! *)
let x = 2
let y = x + x
let f = g (z + 3) y
let _ = if f > 65
    then Printf.printf "%d\n" (g 3 (f * y))
    else Printf.printf "ERROR"
```

Qu'est-ce qui s'affiche ?

Immutabilité

- les "variables" sont **immuables**
- Une succession de `let` au top-level **n'est pas une mutation**
- Car `let` ouvre un nouveau contexte jusqu'à la fin du fichier.

```
let x = 10
let print_old_x () = Printf.printf "%d\n" x

let x = x + 10
let () = Printf.printf "%d\n" x  (* prints 20 *)
let () = print_old_x ()          (* prints 10 *)
```

Immutabilité

- les "variables" sont **immuables**
- Une succession de `let` au top-level **n'est pas une mutation**
- Car `let` ouvre un nouveau contexte jusqu'à la fin du fichier.

```
let x = 10
let print_old_x () = Printf.printf "%d\n" x

let x = x + 10
let () = Printf.printf "%d\n" x  (* prints 20 *)
let () = print_old_x ()          (* prints 10 *)
```

→ Il n'y a pas d'effets de bord !

```
(* OCaml interprété *)  
let x = 10 ;;  
let y = x + 5 ;;  
let f z = x + z ;;
```

- En OCaml interprété, on utilise ; ; pour signaler la fin de l'input.


```
(* OCaml compilé *)  
let x = 10  
let y = x + 5  
let f z = x + z
```

- En dehors de l'interpréteur, pas besoin de ; ; pour signaler la fin de l'input.

```
(* OCaml compilé *)  
let x = 10  
let y = x + 5  
let f z = x + z
```

- En dehors de l'interpréteur, pas besoin de ; ; pour signaler la fin de l'input.
- **NE JAMAIS UTILISER ; ; EN OCAML COMPILE**

Syntaxe

- Un programme est une séquence de déclarations globales, séparées par des retours à la ligne.
- Pour donner la main à l'interpréteur il faut terminer par ; ;
- Pour un programme compilé, certaines de ces déclarations devront avoir des effets collatéraux, sinon rien d'observable ne se passera.

```
let x = 4
let () =
  x+2;
Printf.printf "%d\n" x;
Printf.printf "OK\n" ;;
```

Notez que `e1; e2; e3` rend la valeur `e3`, le reste est 'jeté' et ne sert que pour ses effets collatéraux.

let in et expressions

let en tant qu'expression

```
let x = 42 in x + 1 ;;  
- : int = 43
```

C'est une **définition locale**.

let interdit à gauche de +, il faut une **valeur** :

```
(let x = 42) + 1 ;;  
Error: Syntax error
```

En revanche, une *expression let* fonctionne bien :

```
(let x = 42 in x) + 1 ;;  
- : int = 43
```

Déclarations locales et portée des variables

```
let f x y =  
    let z = x * x in  
    y * (z + z)  
  
let x = 10  
  
let y =  
    let x = "bonjour" in  
    x ^ x  
  
let z = x + 3
```

Déclarations locales et portée des variables

```
let f x y =  
    let z = x * x in  
    y * (z + z)  
  
let x = 10  
  
let y =  
    let x = "bonjour" in  
    x ^ x  
  
let z = x + 3
```

Déclarations locales et portée des variables

```
let x y =  
    let z = x * x in  
    y * (z + z)  
  
let x = 10  
  
let y =  
    let x = "bonjour" in  
    x ^ x  
  
let z = x + 3
```

Fonctions

Ordre supérieur : fonctions en arguments, fonctions anonymes

```
let rec somme f n =  
  if n <= 0 then 0 else f n + somme f (n-1)  
  
let g = somme (fun x -> x * x)  
  
let () = Printf.printf "%d\n" (g 10)
```

Notez que `(fun x -> x * x)` est une **fonction anonyme** et qu'elle est passée **en argument**.

Ordre supérieur et curryfication

Par défaut Ocaml fait de la curryfication : les fonctions à plusieurs arguments y sont implémentés comme des fonctions qui rendent des **fonctions en résultat** :

```
# let plus x y = x+y;;  
val plus : int -> int -> int
```

Ordre supérieur et curryfication

Par défaut Ocaml fait de la curryfication : les fonctions à plusieurs arguments y sont implémentés comme des fonctions qui rendent des **fonctions en résultat** :

```
# let plus x y = x+y;;  
val plus : int -> int -> int
```

Il faut lire le type de cette fonction comme étant :

`int -> (int -> int)`

Ordre supérieur et curryfication

Par défaut Ocaml fait de la curryfication : les fonctions à plusieurs arguments y sont implémentés comme des fonctions qui rendent des **fonctions en résultat** :

```
# let plus x y = x+y;;  
val plus : int -> int -> int
```

Il faut lire le type de cette fonction comme étant :

`int -> (int -> int)`

De manière équivalente, on aurait pu écrire la fonction plus de la façon suivante :

```
# let plus x = (fun y -> x+y);;  
val plus : int -> int -> int
```

Ordre supérieur et curryfication

Les fonctions rendant des fonctions en résultats peuvent être appliquées partiellement :

```
# let plus2 = plus 2;;  
val plus2 : int -> int = <fun>
```

Ordre supérieur et curryfication

Les fonctions rendant des fonctions en résultats peuvent être **appliquées partiellement** :

```
# let plus2 = plus 2;;  
val plus2 : int -> int = <fun>
```

```
# plus2 10;;  
- : int = 12
```

Le fait de pouvoir manipuler des fonctions comme s'il s'agissait de données (ne pas avoir à les nommer, les passer en argument, les rendre en résultat...) s'appelle **l'ordre supérieur**.

Encore du code !

Puissance Rapide

```
let rec fast_power x n =  
  if n = 0 then 1  
  else if n mod 2 = 0 then  
    let half_power = fast_power x (n / 2) in  
    half_power * half_power  
  else x * fast_power x (n - 1)
```


Puissance Rapide

```
let rec fast_power x n =  
  if n = 0 then 1  
  else if n mod 2 = 0 then  
    let half_power = fast_power x (n / 2) in  
    half_power * half_power  
  else x * fast_power x (n - 1)
```

- $\text{fast_power } 2 \ 5 = 2 \times \text{fast_power } 2 \ 4$

Puissance Rapide

```
let rec fast_power x n =  
  if n = 0 then 1  
  else if n mod 2 = 0 then  
    let half_power = fast_power x (n / 2) in  
    half_power * half_power  
  else x * fast_power x (n - 1)
```

- $\text{fast_power } 2 \ 5 = 2 \times \text{fast_power } 2 \ 4$
- $\text{fast_power } 2 \ 4 = \text{fast_power } 2 \ 2 \times \text{fast_power } 2 \ 2$

Puissance Rapide

```
let rec fast_power x n =  
  if n = 0 then 1  
  else if n mod 2 = 0 then  
    let half_power = fast_power x (n / 2) in  
    half_power * half_power  
  else x * fast_power x (n - 1)
```

- $\text{fast_power } 2 \ 5 = 2 \times \text{fast_power } 2 \ 4$
- $\text{fast_power } 2 \ 4 = \text{fast_power } 2 \ 2 \times \text{fast_power } 2 \ 2$
- $\text{fast_power } 2 \ 2 = \text{fast_power } 2 \ 1 \times \text{fast_power } 2 \ 1$

Puissance Rapide

```
let rec fast_power x n =  
  if n = 0 then 1  
  else if n mod 2 = 0 then  
    let half_power = fast_power x (n / 2) in  
    half_power * half_power  
  else x * fast_power x (n - 1)
```

- $\text{fast_power } 2 \ 5 = 2 \times \text{fast_power } 2 \ 4$
- $\text{fast_power } 2 \ 4 = \text{fast_power } 2 \ 2 \times \text{fast_power } 2 \ 2$
- $\text{fast_power } 2 \ 2 = \text{fast_power } 2 \ 1 \times \text{fast_power } 2 \ 1$
- $\text{fast_power } 2 \ 1 = 2 \times \text{fast_power } 2 \ 0$

Puissance Rapide

```
let rec fast_power x n =  
  if n = 0 then 1  
  else if n mod 2 = 0 then  
    let half_power = fast_power x (n / 2) in  
    half_power * half_power  
  else x * fast_power x (n - 1)
```

- $\text{fast_power } 2 \ 5 = 2 \times \text{fast_power } 2 \ 4$
- $\text{fast_power } 2 \ 4 = \text{fast_power } 2 \ 2 \times \text{fast_power } 2 \ 2$
- $\text{fast_power } 2 \ 2 = \text{fast_power } 2 \ 1 \times \text{fast_power } 2 \ 1$
- $\text{fast_power } 2 \ 1 = 2 \times \text{fast_power } 2 \ 0$
- $\text{fast_power } 2 \ 0 = 1$

Puissance Rapide

```
let rec fast_power x n =  
  if n = 0 then 1  
  else if n mod 2 = 0 then  
    let half_power = fast_power x (n / 2) in  
    half_power * half_power  
  else x * fast_power x (n - 1)
```

- $\text{fast_power } 2 \ 5 = 2 \times \text{fast_power } 2 \ 4$
- $\text{fast_power } 2 \ 4 = \text{fast_power } 2 \ 2 \times \text{fast_power } 2 \ 2$
- $\text{fast_power } 2 \ 2 = \text{fast_power } 2 \ 1 \times \text{fast_power } 2 \ 1$
- $\text{fast_power } 2 \ 1 = 2 \times \text{fast_power } 2 \ 0$
- $\text{fast_power } 2 \ 0 = 1$
- **Résultat : 32**