

Elixir



le poly

langages

machines

apprentissage

processus

réplication

Comprendre un calcul, construire une machine, faire apprendre et coopérer.

Décoller · Le pattern matching · Collections et le pipe · Fonctions, clauses, récursion · Construire un langage
· Des octets aux instructions · Des tenseurs et des graphes · Des calculs qui apprennent · Processus,
messages et état · Confier un état à un serveur · Répliquer un état et retrouver l'accord

Sommaire

Avant de commencer	5
Comment lire ce poly	5
Le poly et les séances	6
PARTIE I · ÉCRIRE EN ELIXIR	
1 Décoller	8
1.1 Le laboratoire	8
1.2 Les nombres	9
1.3 Les chaînes	10
1.4 Les atomes et les tuples	11
1.5 L’immuabilité	12
1.6 Vérifier sa compréhension	13
2 Le pattern matching	14
2.1 <code>=</code> n’est pas une affectation	14
2.2 Déstructurer : mettre une forme à gauche	15
2.3 Le pin <code>^</code> : comparer plutôt que lier	17
2.4 <code>{:ok, valeur}</code> : planter tôt, planter clair	17
2.5 Essayer plusieurs motifs : <code>case</code>	18
2.6 Vérifier sa compréhension	20
3 Collections et le pipe	21
3.1 Le tuple : quelques cases, des rôles connus	21
3.2 La liste : une chaîne de maillons	21
3.3 Les maps : retrouver par le nom	24
3.4 Les options et les intervalles	25
3.5 Le pipe : écrire un traitement comme une phrase	25
3.6 Vérifier sa compréhension	27
4 Fonctions, clauses, récursion	28
4.1 Un module, des fonctions	28
4.2 Les fonctions anonymes et la capture	29
4.3 Plusieurs clauses : la forme décide	29
4.4 La récursion, la respiration du langage	31
4.5 La calculatrice à pile d’AP1	33
4.6 <code>Enum</code> est la boucle	35
4.7 Vérifier sa compréhension	37
PARTIE II · LANGAGES ET MACHINES	
5 Construire un langage	39
5.1 Du texte aux tokens	39
5.2 Un arbre pour garder le calcul	41
5.3 Évaluer un arbre	42
5.4 Grammaires et notation BNF	44
5.5 De la grammaire à l’arbre	46
5.6 Noms et environnements	50
5.7 Fonctions et fermetures	52
5.8 Décider : booléens et <code>case</code>	57
5.9 Le langage complet	58

5.10	Où le souci devient-il visible ?	60
5.11	Ce que l'on peut savoir avant d'exécuter	62
5.12	Vérifier sa compréhension	63
6	Des octets aux instructions	65
6.1	Ce qu'est un processeur	65
6.2	Octets et représentations	68
6.3	Une machine à pile	73
6.4	Compiler un arbre en instructions	77
6.5	Variables et cases	80
6.6	Sauts et contrôle	82
6.7	Quand refuser un programme ?	85
6.8	Appels de fonction : un prolongement	86
6.9	Vérifier sa compréhension	87
PARTIE III · CALCULER ET APPRENDRE		
7	Des tenseurs et des graphes	91
7.1	Transformer des exemples avant de calculer	92
7.2	Lire la forme d'un tenseur	93
7.3	Décrire un calcul pour pouvoir le dériver	97
7.4	Vérifier sa compréhension	100
8	Des calculs qui apprennent	102
8.1	Une perte donne un sens à « mieux »	102
8.2	Une limite de représentation ne disparaît pas en attendant	110
8.3	Garder des exemples pour poser une autre question	112
8.4	Apprendre une transformation avec Axon	113
8.5	Construire une comparaison que l'on peut expliquer	120
8.6	Vérifier sa compréhension	122
8.7	Ce que les projets d'AP9 vont prolonger	123
PARTIE IV · FAIRE COOPÉRER DES PROCESSUS		
9	Processus, messages et état	125
9.1	Plusieurs calculs à la fois	125
9.2	Partager la mémoire, ou s'envoyer des messages ?	128
9.3	Qui exécute cette fonction ?	131
9.4	Envoyer n'est pas attendre une réponse	133
9.5	Comment l'état reste-t-il vivant ?	138
9.6	Observer une panne	143
9.7	Laisser planter, puis relancer	146
9.8	Sortir du carnet avec Mix	149
9.9	Vérifier sa compréhension	150
10	Confier un état à un serveur	152
10.1	Un total, plusieurs clients	152
10.2	Agent : une valeur gardée par un processus	155
10.3	Une transition et une boucle	157
10.4	GenServer : le contrat entre client et serveur	160
10.5	call, cast ou send ?	164
10.6	Nommer le serveur	169
10.7	Superviser le serveur	170
10.8	Un serveur unique, un goulot d'étranglement	173
10.9	Tester un serveur	174
10.10	Vérifier sa compréhension	176
11	Répliquer un état et retrouver l'accord	178

11.1	Pourquoi garder plusieurs copies ?	178
11.2	Quand les copies divergent	179
11.3	Cohérence forte ou cohérence à terme ?	182
11.4	Un état qui se fusionne : le G-Counter	183
11.5	Trois lois pour résister au désordre	187
11.6	Faire vivre trois répliques	190
11.7	Converger n'est pas sauvegarder	194
11.8	Compter dans les deux sens : le PN-Counter	195
11.9	Des ensembles qui se fusionnent	196
11.10	Éditer un texte à plusieurs	198
11.11	Vérifier sa compréhension	201
	Relier ce que nous avons construit	203
	Aide-mémoire	204
	Les valeurs	204
	Les opérateurs	204
	Les motifs	205
	Les fonctions, dans l'ordre du chapitre 4	205
	Choisir et enchaîner	207
	Lire une erreur	207
	Les fonctions de bibliothèque à connaître	207
	Processus et OTP	208
	Nx et Axon	209
	Glossaire	211
	Écrire en Elixir	211
	Langages et machines	212
	Calculer et apprendre	213
	Faire coopérer des processus	214
	Lectures pour continuer	216

Avant de commencer

Ce poly est le livre du cours d'AP : il prolonge les séances, il ne les remplace pas. Les carnets Livebook vous guident pendant les TP ; ici, nous prenons le temps d'expliquer les mécanismes qui se cachent derrière chaque bout de code, avec des exemples, leurs résultats et des schémas. Ses onze chapitres sont regroupés en quatre parties, qui suivent les quatre temps du cours :

- **Écrire en Elixir**, pour AP1, chapitres 1 à 4 ;
- **Langages et machines**, pour AP2 et AP3, chapitres 5 et 6 ;
- **Calculer et apprendre**, pour AP4 et AP5, chapitres 7 et 8 ;
- **Faire coopérer des processus**, pour AP6 à AP8, chapitres 9 à 11.

Les projets d'AP9 s'appuient sur l'ensemble. Chaque partie se lit aussi bien avant qu'après les TP qu'elle accompagne : elle introduit ses notions, de la grammaire d'un langage au gradient d'une fonction en passant par les messages entre processus, au moment où elle en a besoin. En fin de volume, un aide-mémoire rassemble la syntaxe d'Elixir et les fonctions à connaître ; c'est par lui qu'il faut commencer pour relire un programme, pendant le cours comme longtemps après.

Quelques bases de Python suffisent pour commencer. Elixir va vous paraître dépaysant pendant une séance ou deux, puis les mêmes gestes reviendront partout : décrire la forme d'une donnée, écrire une fonction par cas, laisser une récursion parcourir une liste. Ne vous découragez pas devant les premières erreurs, elles font partie de l'apprentissage, et le poly vous montre à chaque fois comment les lire.

Comment lire ce poly

Chaque exemple est fait pour être tapé dans Livebook. Avant de regarder le résultat, prédisiez-le, puis comparez : c'est ainsi que l'on apprend le plus vite. Sous les blocs de code, trois zones distinguent ce que le code a *affiché* de ce qu'il *vaut* :

- *console* désigne le texte écrit à l'écran, typiquement par `IO.puts` ;
- *valeur* désigne le résultat rendu par la dernière expression, celui que Livebook affiche sous la cellule ;
- *erreur* signale un échec volontaire, que nous provoquons pour apprendre à lire les messages d'Elixir.

Quand plusieurs petites expressions se suivent, elles sont rangées l'une sous l'autre, chacune avec sa valeur en face, sur fond gris : tapez chaque ligne dans sa propre cellule.

Les encarts colorés ont chacun un rôle. **Piège** signale une erreur que je vois revenir chaque année ; **À retenir** résume l'idée d'une section ; **En pratique** donne un conseil d'usage ; **Propriété** énonce un fait précis et dit en quelques lignes pourquoi il est vrai ; **Pour aller plus loin**, en gris, propose une piste facultative avec une lecture. En fin de chapitre, **Pour en parler** rassemble quelques affirmations que vous devez savoir soutenir, et **Vérifier sa compréhension** vous propose un petit exercice de prédiction avec sa réponse.

Les schémas ne sont pas décoratifs : ils montrent ce que le texte décrit, une pile qui grandit, une liste qui partage ses maillons, un message qui attend dans une boîte. Prenez le temps de les lire avec leur légende.

Le poly et les séances

Au début de la plupart des séances, un quiz de dix minutes porte sur la séance précédente. Le chapitre à relire est annoncé à l'avance ; ses encarts **Pour en parler** sont votre liste de révision. Refaites aussi les essais du chapitre dans un carnet vierge : une prédiction juste vaut mieux qu'une relecture passive.

À partir du chapitre 5, quelques passages marqués « Pour aller plus loin » proposent des pistes d'approfondissement, souvent avec une lecture ou un lien. Ils sont facultatifs : ni les TP ni les quiz n'en dépendent. Les liens du PDF sont cliquables ; dans la version A4, faite pour l'impression, leur adresse figure aussi en note de bas de page.

Pour toute question, vous pouvez me contacter par email : aurelien.noce@minesparis.psl.eu.

PARTIE I

Écrire en Elixir

SÉANCE AP1 · CHAPITRES 1 À 4

Comment décrire un calcul sans modifier de valeur ni écrire de boucle ?

Cette première partie accompagne la séance AP1 et ses cinq carnets. Elle pose les bases du langage que nous utiliserons pendant tout le cours : des valeurs qui ne changent jamais, des motifs qui décrivent la forme des données, des fonctions écrites cas par cas, et la récursion, qui remplace les boucles. Elle se termine sur la calculatrice à pile d'AP1, qui réunit toutes ces idées.

La syntaxe rappelle souvent Python ; ce sont surtout les habitudes qui changent, et ces quatre chapitres les installent une à une. Ils servent ensuite de référence : les parties suivantes y renvoient chaque fois qu'elles réemploient une forme du langage.

1	Décoller	8
2	Le pattern matching	14
3	Collections et le pipe	21
4	Fonctions, clauses, récursion	28

CHAPITRE 1

Décoller

SÉANCE AP1 · LE CARNET, LES VALEURS ET L'IMMUTABILITÉ

Pourquoi apprendre Elixir après Python ? Parce qu'il oblige à programmer autrement : sans modifier une valeur en place, sans boucle `for`, en décrivant la *forme* des données plutôt qu'en les interrogeant pas à pas. Ces habitudes nous serviront tout le cours, pour écrire un langage, une machine, puis des programmes qui coopèrent par messages.

Elixir date de 2012, mais il tourne sur la *BEAM*, la machine virtuelle d'Erlang, conçu chez Ericsson à partir de 1986 pour des centraux téléphoniques qui ne devaient jamais s'arrêter ; la [keynote d'AP1](#)¹ raconte cette histoire. Ce chapitre pose les valeurs de base, la différence entre ce qu'un code affiche et ce qu'il vaut, puis l'idée centrale : l'immutabilité.

1.1 Le laboratoire

Livebook est un carnet interactif, cousin des notebooks Jupyter : une page faite de cellules de texte et de cellules de code. Ctrl+Entrée (Cmd+Entrée sur Mac) évalue une cellule, et sa valeur s'affiche juste en dessous. Ce cycle, lire une expression, l'évaluer, afficher le résultat, puis recommencer, s'appelle un *REPL* (*read-eval-print loop*).

1.1.1 Afficher n'est pas rendre

L'équivalent du `print` de Python s'appelle `IO.puts` :

```
1 IO.puts("Bonjour, monde !")
```

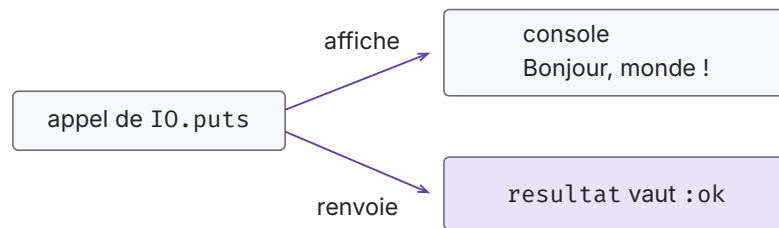
```
CONSOLE  Bonjour, monde !
```

```
VALEUR   :ok
```

Deux choses se sont produites. `IO.puts` a *affiché* le texte dans la console, puis il a *rendu* une valeur, l'atome `:ok`, qui signifie simplement « c'est fait ». Or c'est la valeur, et elle seule, que la suite du programme peut réutiliser. Si nous écrivons `resultat = IO.puts(...)`, que contient `resultat` ?

¹ <https://ap-2026.noce.fr/slides/AP1.html>

```
resultat = IO.puts("Bonjour, monde !")
```



Les deux flèches ont des rôles distincts : l'une écrit dans la console, l'autre rend une valeur au code appelant. `resultat` ne contient pas le texte affiché.

1.1.2 Tout est expression

En Elixir, tout a une valeur, y compris un `if` :

```
1 age = 20
2 if age >= 18, do: "majeur", else: "mineur"
```

VALEUR "majeur"

Le `if` n'a rien affiché : il *vaut* "majeur", et `statut = if ...` garderait ce résultat sous un nom. De même, un bloc de plusieurs lignes vaut sa dernière expression. Il n'y a donc pas de `return` en Elixir : la dernière ligne d'une fonction est son résultat.

À RETENIR

Pour lire un bout de code, cherchez sa valeur finale et distinguez-la de ce qu'il affiche.

1.1.3 Les vérifications des carnets

Les tests des carnets s'écrivent avec `assert`, disponible après `import ExUnit.Assertions`. Une assertion vraie rend `true`, et l'on passe à la suite ; une assertion fausse arrête la cellule :

```
1 assert 2 + 2 == 5
```

```
ERREUR ** (ExUnit.AssertionError)
Assertion with == failed
code:  assert 2 + 2 == 5
left:  4
right: 5
```

`left` est ce que votre code a calculé, `right` ce que le test attendait : c'est là qu'il faut regarder en premier. Les cellules partagent enfin leurs variables dans l'ordre d'évaluation : avant de rendre un carnet, relancez-le de haut en bas dans une session neuve.

1.2 Les nombres

Comme Python, Elixir est *typé dynamiquement* : on n'écrit pas le type d'une variable, ce sont les valeurs qui ont un type, vérifié au moment du calcul. Les entiers grandissent autant que la mémoire le permet : `2 ** 100` vaut exactement `1267650600228229401496703205376`, et le tiret bas aère les grands nombres, `1_000_000` valant `1000000`.

La division `/` rend toujours un flottant, même entre deux entiers. Le quotient et le reste entiers, `//` et `%` en Python, s'écrivent `div` et `rem` :

<code>10 / 2</code>	<code>5.0</code>
<code>div(10, 3)</code>	<code>3</code>
<code>rem(10, 3)</code>	<code>1</code>

Sur un nombre négatif, `div` tronque vers zéro : `div(-10, 3)` vaut `-3`, là où Python donne `-4`. Un flottant est, comme en Python, une approximation binaire sur 64 bits, si bien que certains décimaux n'ont pas de représentation exacte :

<code>0.1 + 0.2</code>	<code>0.30000000000000004</code>
<code>0.1 + 0.2 == 0.3</code>	<code>false</code>

On retrouve les comparaisons de Python, `==`, `!=`, `<`, `<=`, `>` et `>=`. Elixir en ajoute une, `===`, qui exige en plus le même type : `1 == 1.0` est vrai, mais `1 === 1.0` est faux.

Comparer deux valeurs de types différents ne plante pas : Elixir range les nombres avant les atomes, puis les tuples, les maps, les listes et les chaînes. C'est donc à vous de vérifier que vos données ont le type attendu.

1.3 Les chaînes

Une chaîne s'écrit entre guillemets doubles. Les apostrophes ne sont pas une variante comme en Python : `'abc'` désigne une liste de codes de caractères héritée d'Erlang, que nous n'utiliserons pas. Pour coller deux chaînes, on écrit `<>`, et non `+`, réservé aux nombres : `"Ada " <> "Lovelace"` vaut `"Ada Lovelace"`.

L'*interpolation* `#{...}` insère la valeur d'une expression dans une chaîne, comme une f-string de Python :

```
1 nom = "Ada"
2 age = 36
3 "#{nom} aura #{age + 1} ans"
```

```
VALEUR "Ada aura 37 ans"
```

Les opérations sur le texte vivent dans le module `String`, et reçoivent toujours la chaîne en premier argument. Le point d'interrogation final de `contains?` est une convention : il signale une fonction qui répond par `true` ou `false`.

<code>String.length("Ada")</code>	<code>3</code>
<code>String.upcase("Ada")</code>	<code>"ADA"</code>
<code>String.contains?("Ada Lovelace", "Love")</code>	<code>true</code>

Les opérateurs arithmétiques, eux, exigent des nombres :

```
1 1 + "2"
```

```
ERREUR ** (ArithmeticError) bad argument in
arithmetic expression: 1 + "2"
:erlang.+(1, "2")
```

La première ligne nomme l'erreur ; les suivantes situent l'appel. Une conversion, au besoin, se demande explicitement :

```
1 1 + String.to_integer("2")
```

```
VALEUR 3
```

1.4 Les atomes et les tuples

1.4.1 L'atome, une étiquette

Un *atome* est une constante dont le nom est la valeur : `:ok` ne contient rien d'autre que le mot `ok`. Il sert d'étiquette pour nommer les cas d'un programme, un peu comme les constantes en majuscules de Python, mais sans rien déclarer.

Pourquoi pas la chaîne `"ok"` ? Parce qu'un atome est fixé par le programme et se compare en une opération, tandis qu'une chaîne porte du texte, qui peut venir d'un utilisateur et contenir n'importe quoi. Ce sont deux types différents :

```
:ok == "ok"      false
is_atom(:ok)     true
is_atom(nil)     true
```

`true`, `false` et `nil` sont des atomes, écrits sans `:` par commodité. `nil` joue le rôle de `None` : c'est ce que rendent les opérations qui n'ont rien à rendre, comme la lecture d'une clé absente.

Les opérateurs logiques forment deux familles. `and`, `or` et `not` exigent un booléen à gauche : `1 and true` lève une `BadBooleanError`. `&&`, `||` et `!` acceptent toute valeur, et ne tiennent pour faux que `nil` et `false`. `||` fournit ainsi une valeur de repli :

```
true and false   false
nil || "défaut"   "défaut"
0 && 1            1
```

`0 && 1` vaut `1`, car `0` est vrai en Elixir, à la différence de Python. À l'usage, écrivez `and` et `or` entre booléens, et `||` pour un repli.

1.4.2 Le tuple et la convention `{:ok, ...}`

Un *tuple* regroupe quelques valeurs entre accolades, chacune à une position fixe qui a un rôle connu. `elem/2` lit une position, en comptant à partir de zéro, et `tuple_size/1` donne la taille. Le `/2` qui suit un nom indique le nombre d'arguments de la fonction, cf. chapitre 4.

```
reponse = {:ok, 200, "Bonjour"}
tuple_size(reponse)      3
elem(reponse, 2)         "Bonjour"
```

De ces deux briques naît une convention fréquente : une fonction qui peut échouer rend `{:ok, résultat}` si tout s'est bien passé, et `{:error, raison}` sinon. `File.read/1` l'illustre, sur un fichier `notes.txt` qui contient `12 15 9` et sur un fichier qui n'existe pas :

```
File.read("notes.txt")
```

:ok	"12 15 9\n"
l'étiquette	le contenu du fichier

```
File.read("nulle-part.txt")
```

:error	:enoent
l'étiquette	la raison de l'échec

Les deux valeurs rendues par `File.read`. L'atome de tête dit ce qui s'est passé, le second élément porte la charge utile : le contenu du fichier, ou la raison de l'échec (`:enoent` est le code Unix historique pour « ce fichier n'existe pas »).

L'échec est ici une valeur comme une autre, que le programme examine ; le chapitre 2 apprendra à la démonter avec des motifs. Lisez tout de même chaque contrat : `Integer.parse/1`, qui servira à la calculatrice, rend plutôt `{nombre, reste}` ou `:error`.

1.5 L'immuabilité

Nous arrivons à l'idée centrale de la séance. En Elixir, les valeurs sont **immuables** : une fois créée, une valeur ne change jamais. Aucune opération ne modifie une case d'un tuple ou un élément d'une liste ; on fabrique une valeur neuve à partir de l'ancienne. Python, lui, modifie une liste en place : après `point[0] = 10`, c'est l'objet lui-même qui a changé.

1.5.1 Un nom se déplace, la valeur reste

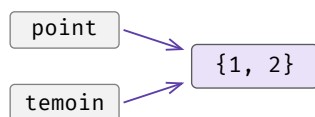
Que fait alors `x = x + 5` en Elixir ? La ligne calcule d'abord une valeur neuve, puis déplace l'étiquette `x` sur elle : c'est le *rebinding*, le fait de relier un nom existant à une autre valeur. L'ancienne valeur n'a pas changé ; elle n'est simplement plus désignée par `x`.

Pour observer qu'elle subsiste, gardons un témoin. `put_elem/3` construit un tuple neuf, avec une case remplacée :

```
1 point = {1, 2}
2 temoin = point
3 point = put_elem(point, 0, 10)
4 {point, temoin}
```

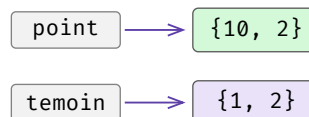
VALEUR `{{10, 2}, {1, 2}}`

```
point = {1, 2}
temoin = point
```



deux noms, une seule valeur

```
point = put_elem(point, 0, 10)
```



une valeur neuve, l'ancienne intacte

Le rebinding déplace une étiquette, il ne modifie pas une valeur. `put_elem` fabrique un tuple neuf, et `=` fait pointer `point` vers lui ; `temoin` désigne toujours l'original. En Python, `point[0] = 10` modifierait l'objet lui-même, et `temoin` « verrait » le changement.

Quel est l'intérêt de cette contrainte ? Une valeur qui ne change jamais peut être lue par plusieurs morceaux du programme, ou partagée entre processus, sans que personne

ne craigne une modification dans son dos. Toute une famille de bugs disparaît, ceux où une fonction modifie sans prévenir la liste qu'on lui a confiée. Le prix est une habitude à prendre : au lieu de « modifier x », on écrit « calculer un nouveau x ».

À RETENIR

Une valeur ne change jamais : on lie des noms à des valeurs, et l'on fabrique des valeurs neuves à partir des anciennes.

POUR ALLER PLUS LOIN

Puisque rien ne change, une valeur neuve peut réutiliser sans risque des morceaux de l'ancienne au lieu de les recopier : c'est le *partage de structure*, que le chapitre 3 montre sur les listes. Chaque version reste intacte : la structure est dite *persistante*². Git procède ainsi : un commit désigne les fichiers inchangés du précédent, sans les recopier. Pour des files ou des arbres persistants efficaces, cf. la thèse de Chris Okasaki, *Purely Functional Data Structures*³ (1996).

POUR EN PARLER

`IO.puts("Bonjour")` affiche le texte sur la console *et* rend `:ok` : ce que l'on voit et ce que l'expression vaut sont deux choses distinctes.

`x = x + 5` ne modifie pas l'ancienne valeur : il lie `x` à une valeur neuve, et un témoin lié avant garde l'ancienne.

`assert 2 + 2 == 5` arrête la cellule : `left: 4` est ce que votre code a calculé.

1.6 Vérifier sa compréhension

Avant de lancer ce code, annoncez séparément la console et la valeur finale :

```
1 price = 12
2 copy = price
3 price = price + 3
4 IO.puts(price)
5 {price, copy}
```

La console affiche 15, puis la valeur du bloc est `{15, 12}`. `IO.puts/1` a bien rendu `:ok`, mais ce n'est pas la dernière expression. La liaison `copy` conserve l'ancien nombre ; aucun entier n'a été modifié en place.

Changez maintenant la dernière ligne en `IO.puts(copy)`. La console affiche successivement 15 et 12, et la valeur du bloc devient `:ok` : l'affichage et le résultat sont bien deux choses distinctes.

² https://fr.wikipedia.org/wiki/Structure_de_donn%C3%A9es_persistante

³ <https://www.cs.cmu.edu/~rwh/students/okasaki.pdf>

CHAPITRE 2

Le pattern matching

SÉANCE AP1 · MOTIFS, CLAUSES ET CASE

Le *pattern matching*, ou filtrage par motif, est le mécanisme que vous utiliserez le plus souvent en Elixir, bien plus que `if`. On écrit un *motif*, c'est-à-dire la forme attendue d'une valeur, avec des trous à remplir, et le langage le confronte à une valeur réelle. Si la forme correspond, les trous se remplissent ; sinon, le programme s'arrête net, ou essaie le motif suivant.

Pensez à un pochoir posé sur une donnée : ce qui passe par les trous devient visible, le reste est vérifié ou ignoré. Les motifs nous serviront partout : lire des tuples, choisir une clause, parcourir des arbres en AP2, reconnaître des messages en AP6.

2.1 = n'est pas une affectation

En Python, `x = 40` se lit « range 40 dans x ». En Elixir, le même signe se lit « fais correspondre le motif `x` à la valeur 40 ». À droite, Elixir calcule une valeur ; à gauche, le *motif* décrit ce qu'il attend.

Une variable seule accepte toute valeur et s'y lie : le résultat ressemble alors à une affectation. Un nombre écrit à gauche, en revanche, exige ce nombre précis. Dans les deux cas, le match vaut la valeur de droite :

<code>x = 40</code>	40
<code>40 = 40</code>	40

Le motif 40 accepte 40. Essayons une autre valeur :

```
1 40 = 41
```

```
ERREUR ** (MatchError) no match of right hand side value:  
41
```

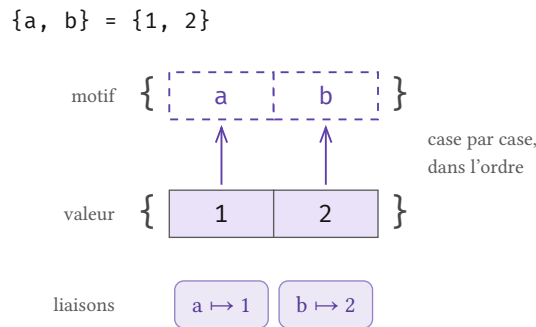
À RETENIR

Un match réussit et lie les variables du motif, ou lève `MatchError`. Une variable seule accepte tout ; une valeur écrite dans le motif impose une égalité.

2.2 Déstructurer : mettre une forme à gauche

L'intérêt du filtrage commence quand le motif a une forme : un tuple, une liste ou une map, avec des variables à l'intérieur. La correspondance se vérifie alors élément par élément, et chaque variable récupère ce qui lui fait face. On dit que l'on *déstructure* la valeur.

2.2.1 Un tuple



Le match confronte les deux formes case par case : les trous du motif (en pointillés) se remplissent de ce qui leur fait face, et c'est tout ce que produit un match, des liaisons.

Comme le *unpacking* de Python, $\{a, b\} = \{1, 2\}$ nomme les éléments. Un motif peut aussi mêler des variables à lier et des valeurs imposées :

```
1 { :ok, valeur } = { :ok, 42 }
2 valeur
```

VALEUR 42

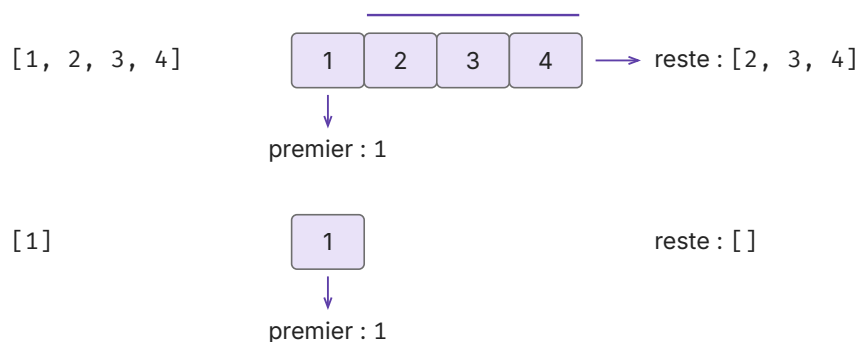
La première case doit valoir `:ok` ; la seconde est liée à `valeur`. Face à `{:error, "fichier introuvable"}`, le motif ne colle plus et lève `MatchError`. Le nombre de cases est imposé, lui aussi : $\{a, b\} = \{1, 2, 3\}$ échoue.

2.2.2 Une liste : la tête et le reste

Une liste non vide a une *tête*, son premier élément, et une *queue*, tout ce qui suit, qui est encore une liste, éventuellement vide. Le motif `[tete | reste]`, avec sa barre verticale, les sépare :

```
1 [premier | reste] = [1, 2, 3, 4]
2 {premier, reste}
```

VALEUR {1, [2, 3, 4]}



La barre de `[premier | reste]` sépare un élément d'une liste. Sur `[1]`, elle laisse la liste vide ; sur `[]`, aucune tête ne peut être prélevée.

[premier, deuxieme | reste] prend deux éléments en tête. La liste vide, elle, n'a aucune tête à fournir :

```
1 [premier | reste] = []
```

```
ERREUR ** (MatchError) no match of right hand side value:
[]
```

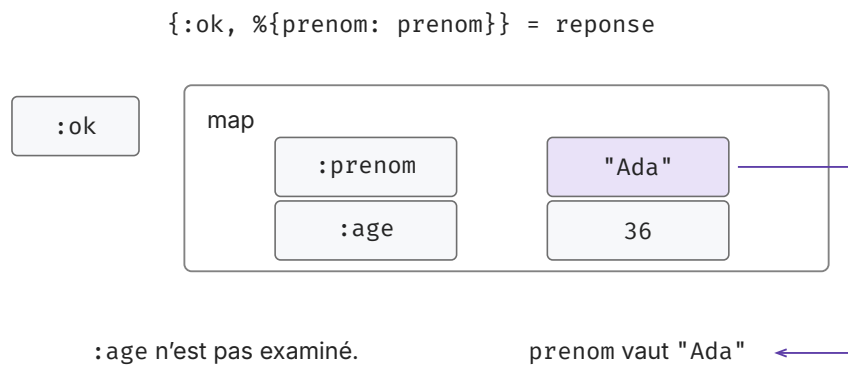
C'est pourquoi les fonctions récursives du chapitre 4 traiteront [] dans une clause à part.

2.2.3 Une map : « au moins ces clés-là »

Le motif d'une map, le dictionnaire d'Elixir présenté au chapitre 3, n'exige que les clés qu'il mentionne : les autres sont permises, et l'on ne nomme que les champs dont on a besoin. Les motifs s'emboîtent par ailleurs, comme ici un tuple qui contient une map :

```
1 reponse = {:ok, %{prenom: "Ada", age: 36}}
2 {:ok, %{prenom: prenom}} = reponse
3 prenom
```

```
VALEUR "Ada"
```



Le motif vérifie :ok, puis cherche :prenom dans la map. Une clé demandée mais absente ferait échouer le match.

2.2.4 Ignorer une case : le _

Le motif _ accepte toute valeur sans la lier à un nom. Ainsi le motif {_, message, _} ne lie que message, le deuxième élément d'un tuple de trois. Un nom qui commence par _, comme _reste, est en revanche une variable ordinaire, qui annonce seulement qu'on ne la relira pas.

2.2.5 Une chaîne : reconnaître un préfixe

Dans un motif, <> reconnaît un préfixe connu et lie la suite :

```
1 "ok:" <> rest = "ok:12"
2 rest
```

```
VALEUR "12"
```

Sur "ok:", rest vaut "" : un reste vide est une réussite. Sur "error:12", le match échoue. En tête d'une fonction, "A" <> rest reconnaît ainsi un A et conserve la suite : le lecteur d'ADN d'AP1 avance d'un caractère à chaque appel, jusqu'à son cas d'arrêt "".

En revanche, `first <> rest = "ab"` est refusé : les deux tailles seraient inconnues. Le préfixe doit être un littéral, ou une valeur déjà liée et épinglée avec le `^` de la section suivante.

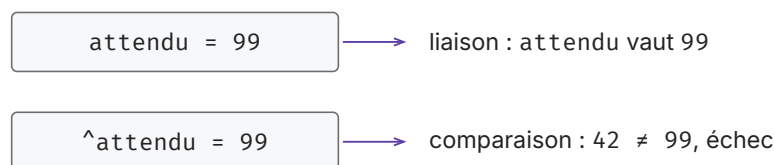
Tous ces motifs sont rassemblés dans l'aide-mémoire en fin de volume, avec ce qu'ils exigent et ce qu'ils lient.

2.3 Le pin `^` : comparer plutôt que lier

Il arrive que l'on veuille vérifier qu'une valeur égale une valeur déjà connue, par ex. le numéro de la question posée. Or une variable dans un motif crée toujours une liaison neuve : après `attendu = 42`, le motif `attendu = 99` réussit, et `attendu` vaut désormais 99.

Le pin `^` fait au contraire de la valeur déjà liée une contrainte du motif : après `attendu = 42`, le motif `^attendu = 99` se lit `42 = 99`, et lève `MatchError`.

départ dans les deux essais : `attendu = 42`



Sans pin, le motif lie un nom ; avec le pin, il exige l'ancienne valeur. Les deux essais repartent de `42`.

2.3.1 Une clé de map connue d'avance

Dans un motif de map, une clé doit être connue avant le match : écrite telle quelle, `%{"Ada" => note}`, ou fournie par une variable épinglée. Pour lire la note d'un nom reçu dans `cle`, on écrit donc :

```

1 notes = %{"Ada" => 18, "Alan" => 17}
2 cle = "Ada"

```

<code>%{^cle => note} = notes</code>	<code>%{"Ada" => 18, "Alan" => 17}</code>
<code>note</code>	<code>18</code>

Sans le `^`, Elixir refuse de compiler `%{cle => note} = notes` : il ne saurait pas quelle clé chercher.

À RETENIR

Sans `^`, une variable à gauche d'un `=` est une *capture* ; avec `^`, elle est une *comparaison* à sa valeur actuelle, liée avant ce motif. Une répétition dans un même motif impose aussi une égalité : `{x, x}` accepte `{3, 3}` et refuse `{3, 4}`.

2.4 `{:ok, valeur}` : planter tôt, planter clair

Écrire `{:ok, contenu} = File.read("notes.txt")` exige une réussite avant de continuer : avec le fichier du chapitre 1, `contenu` vaut `"12 15 9\n"`. Sur un fichier absent, en revanche :

```
1 {:ok, contenu} = File.read("absent.txt")
```

```
ERREUR ** (MatchError) no match of right hand side value:
{:error, :enoent}
```

L'exécution s'arrête là où l'hypothèse est fausse. Un tel match se lit ainsi : « je *sais* que cet appel réussit, et s'il échoue, je veux l'apprendre ici même ». Sinon, il faut plusieurs motifs : c'est le rôle de `case`.

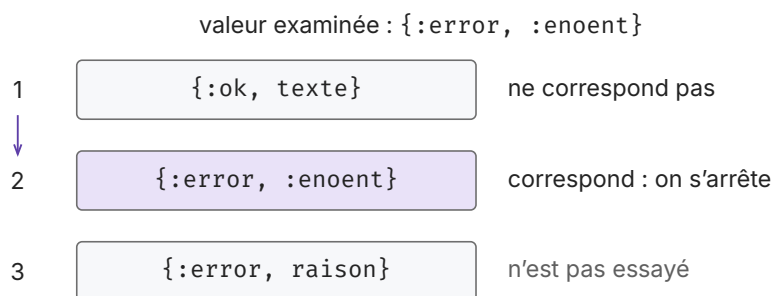
2.5 Essayer plusieurs motifs : `case`

`case` présente une valeur à plusieurs motifs, dans l'ordre, et exécute le code du premier qui correspond. Chaque ligne `motif -> expression` est une *clause* ; retenez ce mot, les fonctions du chapitre 4 en auront aussi.

```
1 case File.read("absent.txt") do
2   {:ok, texte} -> texte
3   {:error, :enoent} -> "le fichier n'existe pas"
4   {:error, raison} -> "échec : #{inspect(raison)}"
5 end
```

```
VALEUR "le fichier n'existe pas"
```

`inspect/1` transforme n'importe quelle valeur en texte lisible, pour l'insérer dans une chaîne. Suivons la réponse `{:error, :enoent}` :



Le `case` vaut « le fichier n'existe pas ».

On descend dans l'ordre écrit jusqu'au premier motif qui correspond. Même si une clause ultérieure pourrait accepter la valeur, son expression n'est pas exécutée.

2.5.1 L'ordre des clauses, et les oublis

Placez donc les cas précis avant un fourre-tout comme `_ -> ...`. Si aucune clause ne correspond, par ex. avec la seule clause `{:ok, texte}` sur un fichier absent, `case` lève `CaseClauseError`, en affichant la valeur refusée, ici `{:error, :enoent}`. Une clause peut enfin porter une condition, introduite par `when` : c'est une *garde*, que le chapitre 4 détaille.

PIÈGE — UNE VARIABLE LIÉE DANS UNE BRANCHE N'EN SORT PAS

En Python, `y` défini dans un `if` reste utilisable après. En Elixir, le compilateur le refuse (`undefined variable`) : la branche doit *rendre* la valeur, et l'on nomme le résultat, `y = case ... end` ou `y = if x > 1, do: 10, else: 0`.

2.5.2 Lire un nombre sans perdre la suite

`Integer.parse/1` rend `{nombre, reste}`, ou `:error` si aucun entier ne commence la chaîne :

<code>Integer.parse("12")</code>	<code>{12, ""}</code>
<code>Integer.parse("12x")</code>	<code>{12, "x"}</code>
<code>Integer.parse("chat")</code>	<code>:error</code>
<code>Integer.parse("-12")</code>	<code>{-12, ""}</code>

Un lecteur de langage peut continuer sur le reste. Notre calculatrice, elle, exige un morceau lu en entier : `"12x"` doit être refusé.

```
1 case Integer.parse("12x") do
2   {number, ""} -> {:ok, number}
3   {_number, rest} -> {:error, {:trailing_text, rest}}
4   :error -> {:error, :expected_integer}
5 end
```

VALEUR `{:error, {:trailing_text, "x"}}`

Gardez `{number, ""}` avant `{_number, rest}` : le second motif accepte aussi un reste vide, et masquerait le premier.

2.5.3 if et cond, et pourquoi on les croise moins

Elixir possède bien un `if`, qui choisit entre deux branches, et un `cond`, qui retient la première condition vraie, comme une chaîne de `elif`. Mais dès qu'il s'agit de distinguer des *formes* de données, un `{:ok, _}` d'un `{:error, _}`, `case` et ses motifs le disent plus sûrement. `cond` garde son intérêt pour des seuils numériques :

```
1 note = 12
2 cond do
3   note >= 16 -> :tres_bien
4   note >= 10 -> :recu
5   true -> :ajourne
6 end
```

VALEUR `:recu`

Attention, pour `if` comme pour `cond`, seuls `false` et `nil` sont faux : `0`, `""` et `[]` sont vrais. Le `if` *liste* de Python ne se traduit donc pas ; on regarde la *forme* de la liste, avec un motif.

POUR ALLER PLUS LOIN

Le filtrage d'Elixir ne regarde que dans un sens : la valeur de droite est entièrement connue, seul le motif a des trous. Prolog, dont Erlang a hérité une partie de sa syntaxe, va plus loin avec l'*unification*⁴ : les deux côtés peuvent contenir des inconnues, et l'on cherche des valeurs qui rendent les deux termes égaux, comme $x = 1$ et $y = 2$ pour `{x, 2}` et `{1, y}`. L'unification est au cœur des typeurs qui devinent les types d'un programme, comme celui de la piste Types des projets d'AP9.

⁴ <https://fr.wikipedia.org/wiki/Unification>

POUR EN PARLER

`[tete | reste] = []` échoue, parce qu'une liste vide n'a pas de tête : ce refus deviendra le cas de base des fonctions récursives.

Sans `^`, une variable à gauche capture ; `^attendu = 99` compare à la valeur actuelle d'`attendu` et échoue si elle diffère.

`case` essaie ses motifs dans l'ordre et retient le premier qui correspond ; seuls `false` et `nil` sont faux, si bien que `if 0` prend la branche vraie.

2.6 Vérifier sa compréhension

Nous avons déjà lié `expected = 3`. Prédisez le résultat de chaque motif, essayé séparément avec cette même liaison de départ :

```
1 { :ok, expected } = { :ok, 4 }
2 { :ok, ^expected } = { :ok, 4 }
3 [ head | tail ] = [ 3 ]
```

Le premier lie à nouveau `expected`, qui vaut désormais `4`. Le second compare avec `3` et lève `MatchError`. Le troisième lie `head` à `3` et `tail` à `[]` : une liste d'un élément possède bien une queue, vide.

Quelle entrée ferait échouer ce dernier motif ? `[]`, qui n'a pas de tête : d'où le cas de base explicite des fonctions du chapitre 4.

Collections et le pipe

LISTES DÈS AP1 · MAPS EN AP3 · PIPE ET `Enum` SYSTÉMATIQUES DÈS AP4

Python vous a habitués à la liste, au tuple et au dictionnaire. Elixir en propose des cousins, le tuple, la liste et la map, mais la ressemblance s'arrête vite : la liste d'Elixir, par exemple, n'est pas un tableau. Ce chapitre montre comment chaque structure est rangée en mémoire, parce que c'est cette forme qui décide de ce qui est rapide ou coûteux.

Nous terminerons par le *pipe*, l'opérateur `|>`, qui écrit une suite de transformations dans l'ordre où elles se produisent.

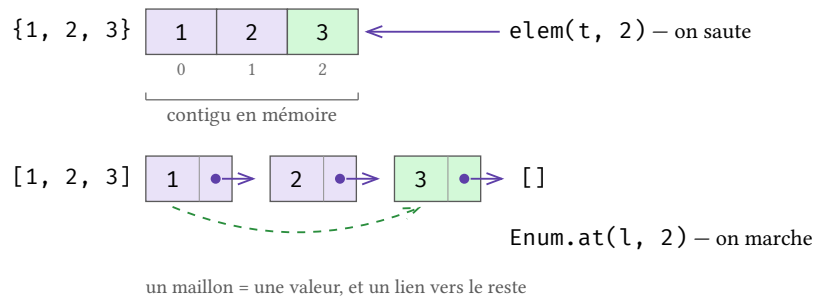
3.1 Le tuple : quelques cases, des rôles connus

Le chapitre 1 a présenté le tuple, lu par `elem/2`. Ses cases sont contiguës en mémoire : `elem` calcule directement l'adresse de la position demandée. En contrepartie, un tuple ne grandit pas, et `put_elem/3` fabrique un tuple neuf en recopiant toutes ses cases. On le réserve donc aux petits groupes de taille fixe, comme `{:ok, valeur}`.

Un tuple n'est pas non plus une collection que l'on parcourt : les fonctions du module `Enum`, que nous retrouverons plus loin, le refusent, et `Enum.sum({1, 2, 3})` lève une `Protocol.UndefinedError`. Des valeurs à parcourir se rangent dans une liste ; `Tuple.to_list/1` convertit au besoin.

3.2 La liste : une chaîne de maillons

La liste Python est un tableau : ses éléments sont rangés côte à côte, et `liste[2]` saute directement à la troisième case. La liste Elixir est *chaînée* : chaque maillon contient une valeur et un lien vers le maillon suivant, comme les wagons d'un train, et `[]`, la liste vide, marque la fin.



Deux collections de trois valeurs, deux formes en mémoire. Le tuple se saute, `elem` calculant l'adresse de la case ; la liste se marche, et atteindre la troisième valeur demande de suivre deux liens.

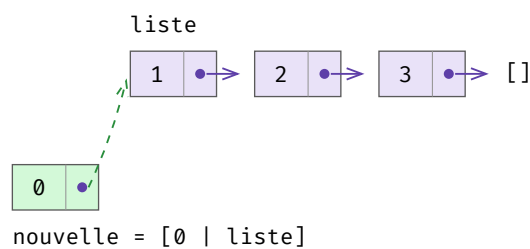
Cette forme paraît moins pratique, et pourtant elle a un avantage décisif : ajouter un élément en tête ne coûte rien et ne recopie rien. Comme en Python, une liste peut mêler des types et répéter une valeur : `[3.14, :pi, "pomme", 3.14]` est une liste valide de quatre éléments.

3.2.1 Travailler par la tête

`hd/1` lit la tête et `tl/1` rend le reste. L'opérateur `|`, le *cons* (de l'anglais *construct*), fait l'inverse : il fabrique un maillon neuf devant une liste existante.

<code>liste = [1, 2, 3]</code>	<code>[1, 2, 3]</code>
<code>hd(liste)</code>	<code>1</code>
<code>tl(liste)</code>	<code>[2, 3]</code>
<code>[0 liste]</code>	<code>[0, 1, 2, 3]</code>

La même forme construit ou décompose : `[0 | liste]` ajoute une tête, et le motif `[tete | reste] = liste` du chapitre 2 extrait celle qui existe. L'ajout ne touche d'ailleurs pas à `liste` : le maillon neuf pointe vers elle.



un seul maillon neuf : la queue de `nouvelle` est `liste`, non recopiée

`[0 | liste]` ne coûte qu'un maillon, quelle que soit la longueur de `liste` : deux noms, une seule structure partagée, aucune copie.

Trois questions reviennent sans cesse : un élément est-il présent, combien y en a-t-il, quel est le *i*-ème ? `in`, `length/1` et `Enum.at/2` y répondent, mais toutes trois *marchent* le long des maillons :

<code>3 in [1, 2, 3]</code>	<code>true</code>
<code>length([1, 2, 3])</code>	<code>3</code>
<code>Enum.at([1, 2, 3], 1)</code>	<code>2</code>

Une liste se parcourt donc de tête en tête : la tête est immédiate, le *i*-ème élément demande *i* pas. Cette décomposition guidera les clauses récursives du chapitre 4.

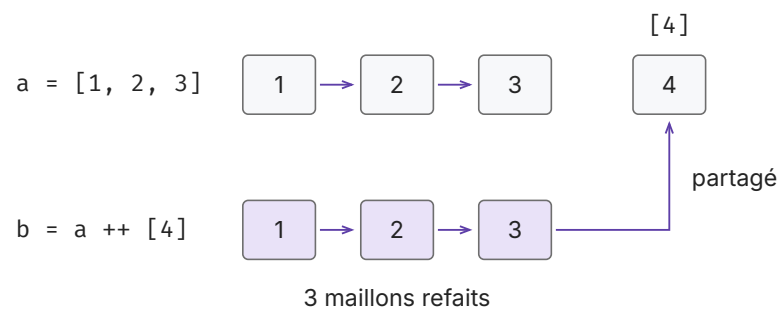
3.2.2 Ce que la forme rend cher

Le coût d'une opération suit le nombre de maillons à parcourir ou à refaire. `++`, qui concatène deux listes, est le cas le plus instructif. Prédisez avant de lire : après `b = a ++ [4]`, que vaut `a` ?

```
1 a = [1, 2, 3]
2 b = a ++ [4]
3 {a, b}
```

VALEUR { [1, 2, 3], [1, 2, 3, 4] }

`a` garde sa valeur, comme toujours. Pour que le dernier maillon de `a` mène à `[4]`, il faut cependant le refaire, puis refaire celui qui le précède, et ainsi de suite : tous les maillons de gauche sont recopiés, ceux de droite sont partagés.



Les maillons neufs de `b` recopient ceux de `a`, puis rejoignent la liste `[4]`, partagée telle quelle. `a` reste intacte.

Si l'on construit un résultat avec `acc ++ [x]` à chaque élément, chaque tour recopie donc tout ce qui a déjà été construit. On ajoute plutôt en tête, ce qui donne l'ordre inverse, et l'on retourne la liste une seule fois à la fin :

```
1 acc = [3 | [2 | [1 | []]]]
2 {acc, Enum.reverse(acc)}
```

VALEUR { [3, 2, 1], [1, 2, 3] }

L'opérateur `--`, enfin, retire de la liste de gauche la première occurrence de chaque élément de droite : `[3, 1, 2, 1] -- [1]` vaut `[3, 2, 1]`.

POUR ALLER PLUS LOIN

Comptons les maillons recopiés pour construire une liste de n éléments avec `acc ++ [x]`. Au tour k , `acc` a déjà k maillons, tous refaits : le total vaut $0 + 1 + \dots + (n - 1) = n(n - 1)/2$. Le coût est *quadratique* : pour $n = 10\,000$, près de cinquante millions de maillons. En ajoutant en tête, chaque tour coûte un maillon, puis `Enum.reverse/1` en refait n : $2n$ au total, un coût *linéaire*. Le guide d'efficacité d'Erlang compare ces deux constructions, cf. [List Handling](https://www.erlang.org/doc/system/listhandling.html)⁵.

⁵ <https://www.erlang.org/doc/system/listhandling.html>

3.3 Les maps : retrouver par le nom

Pour retrouver une valeur par un nom plutôt que par une position, on prend une *map*, l'équivalent du `dict` de Python : un ensemble de paires clé/valeur, sans ordre garanti. Elle s'écrit `%{clé => valeur}`, et les crochets lisent une clé :

```
1 notes = %{ "Ada" => 18, "Alan" => 17 }
2 notes["Ada"]
```

```
VALEUR 18
```

Les clés peuvent être de tout type, et une même map peut les mélanger. Les cases d'une grille, par ex., se rangent sous leurs coordonnées : `%{{0, 0} => :mur, {0, 1} => :vide}`.

3.3.1 Des champs nommés par des atomes

Avec des clés atomes, `%{prenom: "Ada"}` abrège `%{:prenom => "Ada"}`, et le point lit un champ. Une clé absente donne `nil` avec les crochets, mais une erreur avec le point :

```
1 etudiant = %{prenom: "Ada", nom: "Lovelace", age: 36}
```

<code>etudiant.prenom</code>	"Ada"
<code>etudiant[:taille]</code>	nil

```
1 etudiant.taille
```

```
ERREUR ** (KeyError) key :taille not found in:
%{age: 36, nom: "Lovelace", prenom: "Ada"}
```

Choisissez le point si la clé doit exister, les crochets si son absence est normale. `Map.get(etudiant, :taille, 0)` donne une valeur de repli, et le module `Map` fournit le reste : `Map.keys/1`, `Map.has_key?/2` ... (cf. l'aide-mémoire en fin de volume).

EN PRATIQUE

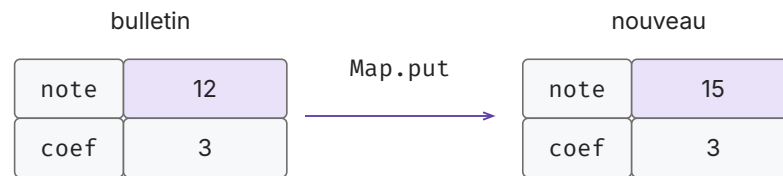
Atomes ou chaînes comme clés ? Les atomes servent aux champs connus d'avance, les chaînes aux données venues du dehors. Un atome n'est en effet jamais récupéré par le ramasse-miettes : le parser du chapitre 5 conserve donc les noms écrits par l'utilisateur sous forme de chaînes.

3.3.2 Mettre à jour, c'est fabriquer une map neuve

`Map.put/3` rend une nouvelle map, avec une clé ajoutée ou changée ; l'ancienne reste telle quelle :

```
1 bulletin = %{note: 12, coef: 3}
2 nouveau = Map.put(bulletin, :note, 15)
3 {bulletin.note, nouveau.note}
```

```
VALEUR {12, 15}
```



`bulletin.note` vaut encore 12 après le calcul.

La flèche indique un calcul de nouvelle map : seule la note diffère. Le dessin décrit les valeurs observables, sans imposer une copie physique de toutes les cases.

`%{bulletin | note: 15}` est plus court, mais exige que la clé existe déjà ; une clé absente lève alors `KeyError`.

Notez enfin qu'une map ne garantit aucun ordre. L'affichage semble stable sur de petites maps, mais ne fondez jamais un test sur l'ordre des clés : si vous voulez un ordre, demandez-le, par ex. `Map.keys(m) |> Enum.sort()`.

3.4 Les options et les intervalles

Une *keyword list* est une liste de couples `{atome, valeur}`, avec une écriture courte ; elle garde l'ordre et accepte les clés répétées. Un *intervalle* retient deux bornes et un pas, sans construire les éléments :

```
[couleur: "bleu", taille: 42] ==
 [{:couleur, "bleu"}, {:taille, 42}]
Enum.to_list(1..10//3)           true
                                [1, 4, 7, 10]
```

Les keyword lists servent surtout à passer des options. Quand elles sont le dernier argument, les crochets peuvent tomber : `IO.inspect(v, label: "x")` s'écrit en réalité `IO.inspect(v, [label: "x"])`. Le `if` du chapitre 1 en utilisait déjà une : dans `if age >= 18, do: "majeur", else: "mineur", do: et else:` sont des clés.

À RETENIR

Un tuple pour quelques valeurs de rôles connus ; une liste pour une suite à parcourir et à faire grandir par la tête ; une map pour retrouver par clé ; une keyword list pour des options ; un intervalle pour une suite régulière d'entiers.

3.5 Le pipe : écrire un traitement comme une phrase

Gardons les nombres pairs d'une liste, élevons-les au carré, puis additionnons. Trois fonctions du module `Enum` s'en chargent. Chacune reçoit la liste, et une petite fonction à appliquer à ses éléments, écrite sur place sans nom : une *fonction anonyme*.

Une fonction anonyme s'écrit entre `fn` et `end` : d'abord son paramètre, puis une flèche, puis son résultat. Ainsi `fn x -> x * x end` reçoit `x` et rend `x * x`, comme la `lambda x: x * x` de Python. Le chapitre 4 les présentera en détail ; ici, il suffit de les lire.

```

1 Enum.sum(
2   Enum.map(
3     Enum.filter([1, 2, 3, 4, 5, 6],
4                 fn x -> rem(x, 2) == 0 end),
5     fn x -> x * x end))

```

VALEUR 56

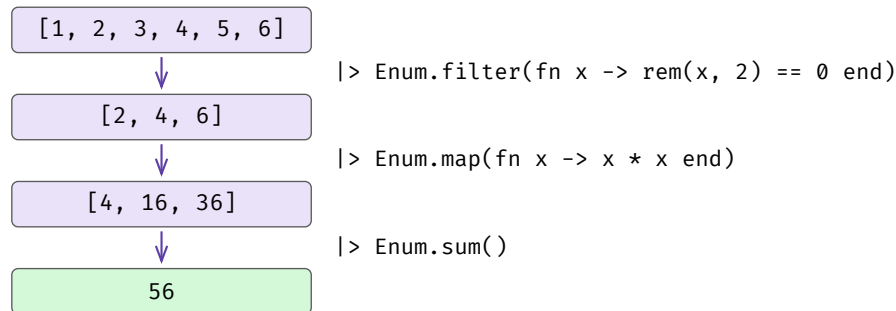
Cette écriture est juste, mais pénible à lire : il faut partir du centre, `Enum.filter`, puis remonter vers l'extérieur, à l'envers de l'ordre des opérations. Le pipe résout ce souci. `a |> f(b)` signifie exactement `f(a, b)` : la valeur de gauche entre en *premier* argument de la fonction de droite, et chaque étape reçoit le résultat de la précédente.

```

1 [1, 2, 3, 4, 5, 6]
2 |> Enum.filter(fn x -> rem(x, 2) == 0 end)
3 |> Enum.map(fn x -> x * x end)
4 |> Enum.sum()

```

VALEUR 56



la valeur d'une étape devient le *premier* argument de la suivante

Le même calcul, lu dans l'ordre où il se produit : la donnée entre par le haut et descend d'étape en étape, chacune produisant la valeur que la suivante consomme.

À RETENIR

`a |> f(b)` signifie `f(a, b)`. C'est pourquoi `Enum`, `String` et `Map` prennent tous la donnée en premier argument : elle peut ainsi traverser un pipeline.

3.5.1 Découper un texte en morceaux

Un texte entre souvent dans un pipeline par `String.split/1`, qui découpe sur les blancs, espaces, tabulations et retours à la ligne, en ignorant les morceaux vides. Avec un séparateur explicite, `String.split/2` garde ces morceaux :

```

1 source = " 12  3 - "
String.split(source)      ["12", "3", "-"]
String.split(source, " ") ["", "12", "", "3", "-", ""]

```

La calculatrice d'AP1 utilise donc `String.split/1`. `String.trim/1`, lui, retire seulement les blancs des extrémités.

3.5.2 Regarder passer les valeurs

Pour voir une étape d'un pipeline, insérez `IO.inspect` : il affiche sa valeur sur la console, puis la renvoie inchangée à l'étape suivante.

```
1 [1, 2, 3, 4, 5, 6]
2 |> Enum.filter(fn x -> rem(x, 2) == 0 end)
3 |> IO.inspect(label: "après le filtre")
4 |> Enum.sum()
```

CONSOLE après le filtre: [2, 4, 6]

VALEUR 12

Dans l'interpréteur d'AP2, les listes conserveront les tokens et les liaisons de l'environnement, et les tuples donneront une forme aux nœuds de l'arbre.

POUR EN PARLER

Un tuple se lit par position mais ne se parcourt pas : `Enum` le refuse, et `put_elem` recopie toutes ses cases.

Ajouter en tête, `[0 | liste]`, coûte un seul maillon. Ajouter en queue, `liste ++ [0]`, recopie tous les maillons de `liste`, et `acc ++ [x]` à chaque tour devient quadratique.

Aucune opération ne modifie une liste : après `b = a ++ [4]`, `a` vaut toujours `[1, 2, 3]`.

`map.cle` lève `KeyError` sur une clé absente, `map[:cle]` rend `nil` : on choisit selon que l'absence est une faute ou un cas normal.

`a |> f(b)` n'est que `f(a, b)` ; `String.split/1` découpe sur tous les blancs et ignore les morceaux vides.

3.6 Vérifier sa compréhension

Prédisez les deux résultats. La liste d'entrée est-elle modifiée ?

```
1 values = [2, 5, 8]
2 kept = Enum.filter(values, fn x -> x > 3 end)
3 {values, Enum.map(kept, fn x -> x * 2 end)}
```

Nous obtenons `{[2, 5, 8], [10, 16]}`. `filter` construit une liste des valeurs retenues, `map` une liste de résultats, et aucune de ces opérations ne change `values`.

Pour constituer une liste de résultats en récursion, pourquoi préférer `[x | acc]` puis un unique `Enum.reverse/1` à `acc ++ [x]` à chaque étape ? Le premier ajoute un maillon en temps constant ; le second reparcourt et recopie le préfixe déjà constitué. L'ordre final est le même après le retournement, mais le coût total diffère.

CHAPITRE 4

Fonctions, clauses, récursion

SÉANCE AP1 · CLAUSES, RÉCURSION ET CALCULATRICE À PILE

Jusqu'ici, nous avons évalué des expressions une à une. Rangeons-les maintenant dans des fonctions. C'est là qu'Elixir se distingue le plus de Python : une fonction peut avoir plusieurs *clauses*, une par forme d'argument, et les boucles n'existent pas, remplacées par la récursion.

Ces deux idées vont ensemble, et la calculatrice à pile d'AP1 les combine ; le module `Enum` fournira enfin les parcours courants déjà écrits.

4.1 Un module, des fonctions

Une fonction nommée appartient toujours à un *module*, un regroupement de fonctions sous un nom, déclaré par `defmodule`. Un module ressemble à un fichier Python, ou à une classe sans attribut : il range des fonctions, et rien d'autre. On appelle une fonction par `Module.fonction(arguments)` :

```
1 defmodule Calculs do
2   def carre(x) do
3     x * x
4   end
5
6   def somme(a, b), do: a + b
7   def somme(a, b, c), do: a + b + c
8 end
```

<code>Calculs.carre(5)</code>	25
<code>Calculs.somme(1, 2)</code>	3
<code>Calculs.somme(1, 2, 3)</code>	6

La dernière expression du corps donne le résultat : il n'y a pas de `return`. Pour un corps d'une seule expression, la forme courte `def somme(a, b), do: a + b` évite le bloc `do ... end`.

4.1.1 L'arité fait partie du nom

L'*arité* est le nombre d'arguments d'une fonction, et la notation `nom/arité` l'identifie : `somme/2` et `somme/3` sont deux fonctions distinctes, qui cohabitent dans le module. Un argument par défaut, noté `\\`, définit d'ailleurs deux arités d'un coup : `def saluer(nom, salut \\ "Bonjour")` donne `saluer/1` et `saluer/2`.

4.1.2 `def` ou `defp` : ce qui sort du module, ce qui reste dedans

`def` expose une fonction ; `defp` la rend *privée*, callable seulement à l'intérieur du module, sans préfixe :

```
1 defmodule Stats do
2   def moyenne(liste), do: total(liste) / Enum.count(liste)
3   defp total(liste), do: Enum.sum(liste)
4 end
5
6 Stats.moyenne([2, 4, 6])
```

VALEUR 4.0

`moyenne` appelle `total` sans préfixe. Depuis l'extérieur, en revanche, `Stats.total/1` est introuvable : l'appel lève une `UndefinedFunctionError`, comme pour une fonction inexistante. N'exposez avec `def` que ce que le reste du programme doit appeler.

4.2 Les fonctions anonymes et la capture

Le chapitre 3 a passé des fonctions anonymes à `Enum.filter` et `Enum.map`. Une fonction anonyme est une valeur, comme une `lambda` de Python : on peut la ranger sous un nom, la passer à une autre fonction ou la renvoyer. `fn` la définit, et un point avant les parenthèses l'appelle :

```
1 double = fn x -> x * 2 end
2 double.(5)
```

VALEUR 10

Attention au point : sans lui, `double(5)` chercherait une fonction *nommée* `double/1`, qui n'existe pas.

L'opérateur `&`, la *capture*, abrège ces écritures de deux façons. `&String.upcase/1` désigne une fonction existante, par son nom et son arité, pour la manipuler comme une valeur. `&(&1 + 1)` abrège `fn x -> x + 1 end` : `&1` y désigne le premier argument, `&2` le deuxième.

```
1 majuscule = &String.upcase/1
2 plus_un = &(&1 + 1)
```

<code>majuscule.("ada")</code>	"ADA"
<code>plus_un.(41)</code>	42
<code>Enum.map(["a", "b"], &String.upcase/1)</code>	["A", "B"]

`fn s -> String.upcase(s) end`, `&String.upcase(&1)` et `&String.upcase/1` sont donc trois écritures du même calcul, de la plus explicite à la plus courte. La dernière exige une fonction qui existe déjà, avec la bonne arité ; pour `x + 3`, on écrit `fn x -> x + 3 end` ou `&(&1 + 3)`.

4.3 Plusieurs clauses : la forme décide

Là où Python enchaîne `if`, `elif` et `else`, Elixir écrit plusieurs *clauses* : plusieurs fois `def categorie(...)`, chacune portant sa condition dans sa tête. Le mot `when` introduit cette condition, que l'on appelle une *garde* :

```

1 defmodule Age do
2   def categorie(age) when age < 3, do: "bebe"
3   def categorie(age) when age < 13, do: "enfant"
4   def categorie(age) when age < 18, do: "ado"
5   def categorie(_age), do: "adulte"
6 end

```

Age.categorie(2)	"bebe"
Age.categorie(15)	"ado"
Age.categorie(42)	"adulte"

À l'appel, Elixir essaie les clauses dans l'ordre écrit, comme celles d'un `case` : la première dont le motif et la garde conviennent fournit le résultat. Pour `Age.categorie(15)`, les gardes `15 < 3` et `15 < 13` sont fausses, `15 < 18` est vraie : la réponse est `"ado"`, et la dernière clause n'est même pas regardée.

Le code se lit alors comme une table, les cas à gauche, les réponses à droite. Une clause qui accepte tout masque en revanche celles qui la suivent : placez le cas général en dernier.

4.3.1 `when` : les gardes et leurs limites

Une garde n'accepte que des opérations simples et sans effet de bord : comparaisons, arithmétique, `and`, `or`, `not`, `in`, tests de type comme `is_integer/1`, et quelques accès comme `length/1`. Une fonction écrite par vous n'y a pas sa place : le module ne compilerait pas. Une condition plus riche se teste dans le corps, avec un `case`.

4.3.2 Brancher sur la forme de la donnée

Les motifs du chapitre 2 peuvent aussi figurer en tête d'une fonction. Trois clauses `bonjour([])`, `bonjour([seul])` et `bonjour([un | _])` distinguent ainsi zéro, un ou plusieurs éléments ; la deuxième doit précéder la troisième, qui accepterait aussi une liste d'un seul élément.

Un motif de `map` en tête de fonction ne retient, comme au chapitre 2, que les clés utiles. Pour garder *aussi* la `map` entière, par ex. afin de la transmettre plus loin, on écrit le motif et un nom, reliés par `=` :

```

1 defmodule Fiche do
2   def presenter(%{prenom: prenom} = personne) do
3     "#{prenom} (#{map_size(personne)} champs)"
4   end
5 end

```

Fiche.presenter(%{prenom: "Ada", age: 36})	"Ada (2 champs)"
--	------------------

L'argument est confronté *séparément* aux deux côtés du `=` : `personne` le reçoit en entier, tandis que le motif y cherche la clé `:prenom`. Sur `%{nom: "Turing"}`, aucune clause ne convient, et l'appel lève une `FunctionClauseError`, erreur que vous croiserez souvent. Elixir affiche l'argument refusé sous le message : comparez-le au motif de chaque clause, ici il manque la clé `:prenom`.

4.3.3 Des clauses dans une fonction anonyme

Une fonction anonyme peut, elle aussi, avoir plusieurs clauses, écrites l'une sous l'autre comme dans un `case`, avec le même nombre d'arguments :

```

1 traiter = fn
2   {:ok, valeur} -> "reçu #{valeur}"
3   {:error, raison} -> "échec : #{raison}"
4 end

```

traiter.({:ok, 42})	"reçu 42"
traiter.({:error, "disque plein"})	"échec : disque plein"

Retenez l'idée de cette section : une clause par forme utile, liste vide ou non, succès ou erreur, feuille ou nœud d'un arbre.

4.4 La récursion, la respiration du langage

Comment parcourir une liste sans boucle `for` ? En écrivant une fonction qui traite la tête, puis qui *s'appelle elle-même* sur le reste : c'est la *récursion*. Le reste raccourcit d'un élément à chaque appel, si bien que l'on finit par atteindre la liste vide.

Une fonction récursive distingue donc au moins deux cas, le plus souvent écrits comme deux clauses : le *cas de base*, qui répond directement sur `[]`, et le *cas récursif*, qui rappelle la fonction sur une entrée plus petite. Voici le produit des éléments d'une liste :

```

1 defmodule Calc do
2   # cas de base : le produit d'une liste vide vaut 1
3   def produit([], do: 1)
4
5   # cas récursif : la tête, fois le produit du reste
6   def produit([tete | reste], do: tete * produit(reste))
7 end
8
9 Calc.produit([1, 2, 3, 4])

```

VALEUR 24

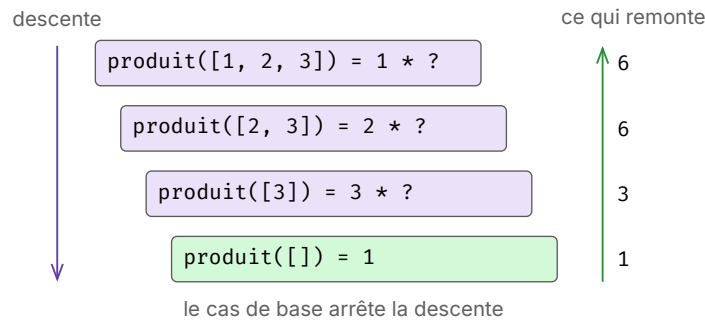
Déroulons `Calc.produit([1, 2, 3])` à la main, ligne à ligne :

```

1 produit([1, 2, 3])
2 = 1 * produit([2, 3])
3 = 1 * (2 * produit([3]))
4 = 1 * (2 * (3 * produit([])))
5 = 1 * (2 * (3 * 1))
6 = 6

```

Chaque appel laisse une multiplication en attente, jusqu'au cas de base ; les attentes se résolvent ensuite, de l'intérieur vers l'extérieur.



La descente empile les appels, chacun laissant une multiplication en attente (le ?) ; le cas de base atteint, la remontée résout ces attentes. C'est la trace ci-dessus, vue de côté.

Le parcours termine, parce que la longueur du reste diminue de un à chaque appel. Sur une liste finie, elle atteint zéro, où la clause `[]` répond sans rappel ; un appel sur la liste entière n'aurait pas cette garantie.

Pour *écrire* le cas récursif, inutile de dérouler : c'est la solution par récurrence de la prépa. Supposez le résultat correct sur le reste, puis vérifiez comment la tête le complète. Le même raisonnement construit une liste de résultats :

```
1 defmodule Carres do
2   def de([], do: [])
3   def de([tete | reste], do: [tete * tete | de(reste)])
4 end
5 Carres.de([2, 3])
```

VALEUR [4, 9]

POUR ALLER PLUS LOIN

Ce raisonnement est une vraie preuve par récurrence, sur la longueur n de la liste. Notons $P(n)$: « pour toute liste l de longueur n , `produit(l)` vaut le produit de ses éléments ». *Initialisation* : la seule liste de longueur 0 est `[]`, et `produit([])` vaut 1, le produit vide. *Hérédité* : une liste de longueur $n + 1$ s'écrit `[t | r]`, avec r de longueur n ; par $P(n)$, `produit(r)` vaut $r_1 \times \dots \times r_n$, donc $t * \text{produit}(r)$ vaut le produit voulu. Le cours *OCaml Programming* de Cornell déroule ces preuves sur les listes et les arbres : [Structural Induction](https://cs3110.github.io/textbook/chapters/correctness/structural_induction.html)⁶.

4.4.1 L'accumulateur et la récursion terminale

Dans `tete * produit(reste)`, la multiplication attend le retour de l'appel récursif. Chaque attente occupe un *cadre d'appel* (*frame*) sur la pile d'appels de la machine, et cette pile grandit avec la liste.

Un *accumulateur* évite ces attentes : c'est un argument de plus, qui transporte le résultat déjà calculé.

⁶ https://cs3110.github.io/textbook/chapters/correctness/structural_induction.html

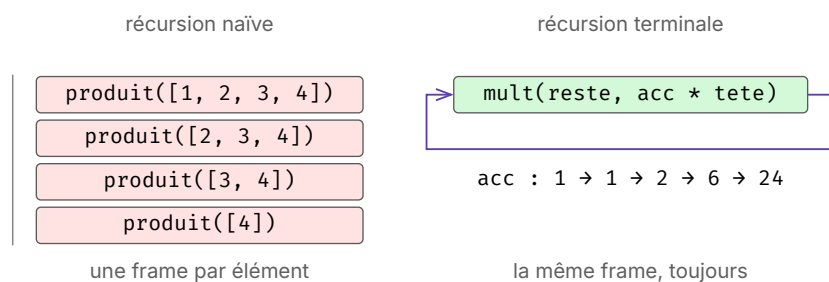
```

1 defmodule Calc do
2   def produit(liste), do: mult(liste, 1)
3
4   defp mult([], acc), do: acc
5   defp mult([tete | reste], acc), do: mult(reste, acc * tete)
6 end
7
8 Calc.produit([1, 2, 3, 4])

```

VALEUR 24

Ce module remplace le précédent et rend le même résultat. Mais `acc * tete` est calculé *avant* l'appel, et rien n'attend son retour : l'appel est *terminal*. La BEAM réutilise alors le même cadre, et la pile d'appels reste bornée, comme pour une boucle `while`.



À gauche, chaque appel en attente coûte une frame, et la pile grandit avec la liste. À droite, l'appel terminal n'attend rien : la BEAM réutilise la même frame, et seul l'accumulateur change.

L'invariant se lit à chaque appel : `acc` contient le produit des éléments déjà consommés. Sur `[]`, il ne reste rien à consommer, et `acc` est donc le résultat.

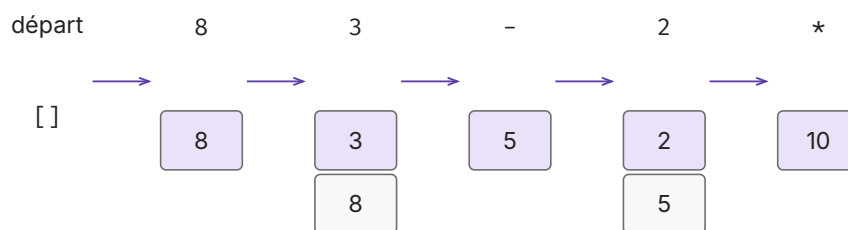
EN PRATIQUE

La récursion terminale borne la pile d'appels, pas toutes les données : une liste ou un grand entier accumulé peut continuer à grandir. Elle ne prouve pas non plus la terminaison : `loop(state)` peut se rappeler sans fin. C'est la diminution du reste jusqu'à `[]` qui fait conclure ; dans un processus d'AP6, continuer à attendre des messages sera au contraire le comportement voulu.

L'accumulateur sert aux longs parcours et aux boucles de processus ; une version sans accumulateur reste correcte.

4.5 La calculatrice à pile d'AP1

L'écriture *postfixe* `"8 3 - 2 *"` place les opérateurs après leurs opérands. Chaque nombre est empilé ; un opérateur retire deux nombres et empile leur résultat. Notre liste représente la pile, sommet en tête :



Sommet en haut : 3 est retiré avant 8, puis on calcule $8 - 3$.

Chaque colonne montre la pile après le token indiqué. La soustraction rend 5, puis la multiplication rend 10.

À la fin de l'entrée, la pile contient exactement une valeur, le résultat. Deux valeurs signaleraient un nombre jamais consommé, aucune un programme vide : la forme de la pile finale fait partie du contrat.

4.5.1 Une transition est une fonction à clauses

La calculatrice sépare deux responsabilités. `Token.read/1` reconnaît un morceau de texte, avec une clause par opérateur puis le `case` sur `Integer.parse` du chapitre 2, et rend par ex. `{:ok, {:number, 12}}` ou `{:ok, :minus}`.

`Stack.step/2` applique ensuite un token déjà reconnu à la pile, et rend la nouvelle pile ou une erreur. Deux de ses clauses suffisent à en montrer la forme :

```
1 def step({:number, number}, stack) do
2   {:ok, [number | stack]}
3 end
4
5 def step(:plus, [right, left | rest]) do
6   {:ok, [left + right | rest]}
7 end
```

La clause nombre ajoute une tête. La clause opérateur retire `right`, puis `left` : avec `[3, 8]`, la soustraction doit donc calculer $8 - 3$, ce que l'addition seule ne permet pas de vérifier. Sur `[]` ou `[8]`, le motif à deux opérandes échoue : une clause générale, placée après les opérateurs, rend alors `{:error, :not_enough_operands}`.

Ne confondez pas cette pile, une liste que vous construisez et qui grandit autant que l'entrée le demande, avec la pile d'appels de la BEAM, gérée par la machine, qu'une récursion terminale garde bornée.

4.5.2 Enchaîner des résultats sans cacher les erreurs

Pour chaque morceau, la boucle lit un token, puis l'applique à la pile. Chaque échec doit être renvoyé tel quel. Dans un `case`, la clause `{:error, _} = error -> error` vérifie la forme et rend le tuple entier, sans rien perdre. Un succès, lui, rappelle la boucle sur le reste avec la nouvelle pile : c'est un appel terminal.

En prolongement, `with` enchaîne des motifs avec `<-`. Chaque succès poursuit l'évaluation ; le premier résultat qui ne correspond pas est rendu tel quel, sans exécuter la suite. Avec le 29 octobre en seconde date, ce `with` vaudrait 2, l'écart en jours ; le 30 février l'arrête avant le calcul :

```

1 with {:ok, first} <- Date.new(2026, 10, 27),
2     {:ok, second} <- Date.new(2026, 2, 30) do
3   Date.diff(second, first)
4 end

```

```
VALEUR  {:error, :invalid_date}
```

Pour la calculatrice, les deux lignes seraient la lecture du morceau puis la transition de pile. Les versions `case` et `with` se valent : choisissez celle dont vous savez dire par où sort chaque erreur.

À RETENIR

Trois contrats : lire un morceau, appliquer un token à la pile, boucler sur le reste. Chacun rend `{:ok, ...}` ou `{:error, ...}`, et une erreur remonte sans être transformée. En AP2, ces trois gestes resteront.

4.6 Enum est la boucle

Faut-il écrire une fonction récursive pour chaque parcours ? Heureusement non : le module `Enum` fournit les parcours courants, qui reçoivent la petite fonction à appliquer. La récursion explicite garde son intérêt pour le parser d'AP2, les arbres et les boucles de processus.

Ces fonctions acceptent toute collection *énumérable* : liste, intervalle ou map. `Enum.map/2` généralise `Carres.de/1`, la transformation devenant un argument ; `Enum.filter/2` garde les éléments pour lesquels la fonction répond vrai.

```

Enum.map([1, 2, 3, 4], fn x -> x * x end)  [1, 4, 9, 16]
Enum.filter(1..6, fn x -> rem(x, 2) == 0 end)  [2, 4, 6]

```

Une map arrive paire par paire, chacune sous la forme d'un tuple `{clé, valeur}`, qu'un motif déstructure dans la tête de la fonction :

```

1 notes = %{"Ada" => 18, "Alan" => 17}
2 Enum.map(notes, fn {nom, note} -> "#{nom} : #{note}" end)

```

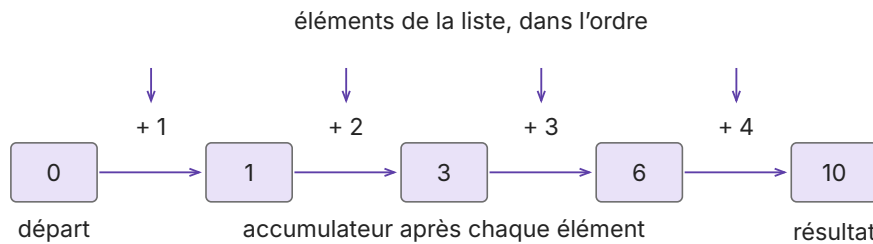
```
VALEUR  ["Ada : 18", "Alan : 17"]
```

4.6.1 reduce, la forme générale

`Enum.reduce/3` reçoit une collection, un accumulateur initial et une fonction qui calcule le nouvel accumulateur à chaque élément. C'est l'accumulateur de la section précédente, sans la récursion à écrire :

```
1 Enum.reduce([1, 2, 3, 4], 0, fn x, acc -> acc + x end)
```

```
VALEUR  10
```



Les flèches horizontales suivent l'accumulateur ; chaque flèche verticale apporte le prochain élément.
 Dans `fn x, acc, x` vient de la liste et `acc` du pas précédent.

Changez la fonction fournie pour obtenir un produit ou un comptage. Les réductions courantes sont même déjà écrites : `Enum.sum/1`, `Enum.max/1`, qui lève une `Enum.EmptyError` sur une liste vide, `Enum.all?/2` ou `Enum.any?/2`. `Enum.each/2`, enfin, appelle une fonction pour son seul effet et rend `:ok`, comme `IO.puts`. Les autres fonctions utiles sont rassemblées dans l'aide-mémoire en fin de volume.

Au chapitre 9, la récursion terminale deviendra la boucle de vie d'un processus, qui transmet son prochain état comme la calculatrice sa pile.

POUR EN PARLER

Une fonction anonyme rangée dans `double` s'appelle avec un point, `double.(5)` ; `&String.upcase/1` capture une fonction existante, et `&(&1 + 1)` abrège une fonction anonyme.

En tête de fonction, `%{prenom: p} = personne` confronte l'argument aux deux côtés : `p` reçoit le prénom, `personne` la map entière ; sans clé `:prenom`, l'appel lève `FunctionClauseError`.

Dans `tete * produit(reste)`, une multiplication attend après l'appel ; dans `mult(reste, acc * tete)`, l'appel est terminal et la pile d'appels reste bornée, ce qui ne prouve pas la terminaison.

Sur la pile `[3, 8]`, l'opérateur `-` calcule `left - right`, soit `5` : le sommet est l'opérande de droite.

4.7 Vérifier sa compréhension

Un calcul reste à faire après l'appel récursif dans `head + sum(tail)`. Au contraire, dans `sum(tail, acc + head)`, l'addition précède l'appel et son résultat devient un argument : cet appel est terminal si sa valeur est directement renvoyée. Suivez ce second parcours sur `[2, 3, 4]` :

Reste	Accumulateur	Somme encore attendue
<code>[2, 3, 4]</code>	0	9
<code>[3, 4]</code>	2	7
<code>[4]</code>	5	4
<code>[]</code>	9	0

À chaque ligne, l'accumulateur plus la somme du reste vaut 9. Cet invariant explique le résultat du cas de base.

Un appel placé à la fin d'une branche de `case` est-il terminal ? Oui, si le `case` est lui-même la dernière expression et qu'aucun calcul n'attend son résultat. C'est la forme de la boucle de la calculatrice, et plus tard celle de la boucle d'un processus.

Langages et machines

SÉANCES AP2 ET AP3 · CHAPITRES 5 ET 6

Comment un programme devient-il un résultat ?

Nous changeons ici de point de vue : au lieu d'écrire des programmes, nous écrivons des programmes qui lisent et exécutent d'autres programmes. En AP2, c'est un interpréteur pour un petit langage de notre invention, que nous découpons en mots, organisons en arbre selon une grammaire, puis calculons en tenant à jour les noms et les fonctions. En AP3, c'est un compilateur, qui traduit ce même arbre en octets pour une petite machine à pile que nous faisons ensuite tourner.

Chemin faisant, nous rencontrerons les grammaires, qui décrivent la forme d'un programme, les environnements et les fermetures, qui donnent un sens aux noms, puis le processeur, les octets et les sauts, qui exécutent le tout. Chaque notion arrive au moment où le programme en a besoin, sur un exemple assez petit pour être refait à la main.

Construire un langage

SÉANCE AP2 · LIRE, ORGANISER ET CALCULER UN PETIT LANGAGE

Qui décide que $1.0 + 2.0 * 3.0$ vaut 7.0 et non 9.0 ? Ni la calculatrice, ni l'ordinateur : quelqu'un a écrit une règle, et un programme l'applique. Dans ce chapitre, c'est nous qui écrivons les règles d'un petit langage, puis le programme qui les applique. Il n'y a pas de meilleure façon de comprendre ce que fait Python ou Elixir quand il lit votre code.

Deux langages se côtoient donc, et il faut les distinguer dès maintenant. Le *langage cible* est celui que notre outil lit et exécute : un petit langage inventé pour le cours, avec des nombres décimaux, `let`, `fn(x) { ... }`, `True` et `False`. Le *langage d'implémentation* est celui dans lequel nous écrivons cet outil, c-à-d. Elixir.

La situation est courante : l'interpréteur Python habituel, celui des carnets Jupyter, est lui-même un programme écrit en C. Python y est le langage cible, et C le langage d'implémentation.

Pour Elixir, le programme `let price = 12.0 price * 3.0` n'est donc qu'une chaîne de caractères ; tapé tel quel dans une cellule Livebook, il serait refusé dès la lecture. C'est notre *interpréteur*, une fonction Elixir ordinaire, qui reçoit cette chaîne et lui donne un sens. Les deux langages se ressemblent parfois sans se confondre : en Elixir, `1 + 2` vaut `3`, alors que notre langage refuse ce texte, car il ne connaît que des nombres décimaux, qui s'écrivent `1.0` et `2.0`.

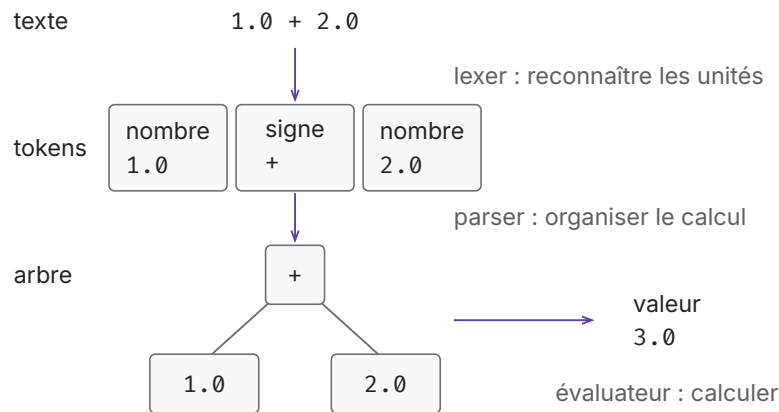
5.1 Du texte aux tokens

5.1.1 Quatre représentations successives

Entre le texte que l'on tape et le nombre qui s'affiche, un interpréteur fabrique plusieurs représentations. Il y en a quatre à bien distinguer :

- le *texte*, une suite de caractères ;
- les *tokens*, les « mots » du langage que l'on en extrait ;
- l'*arbre*, qui dit comment ces mots s'assemblent ;
- la *valeur*, le résultat du calcul.

Chaque passage de l'une à l'autre est l'affaire d'une brique différente. Le *lexer* découpe le texte en tokens, comme on sépare les mots d'une phrase. Le *parser* organise ces tokens en un *arbre de syntaxe abstraite* (AST), qui dit quelle opération s'applique à quoi. Enfin l'*évaluateur* parcourt cet arbre et calcule la valeur.



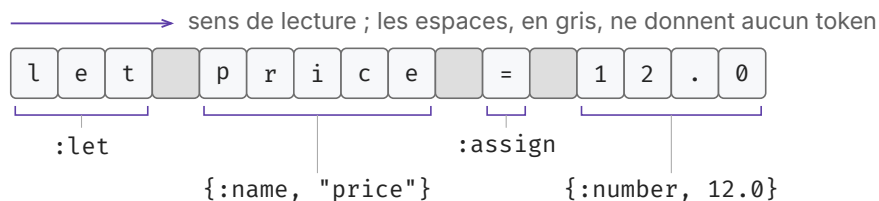
Les flèches nommées transforment une représentation en une autre. Les deux liens de l'arbre désignent les opérandes de l'addition ; ils ne sont pas des étapes d'exécution.

En clair, le lexer découpe, le parser organise et l'évaluateur calcule. Le chapitre suit cet ordre, avant d'ajouter les noms, les fonctions et les choix qui font un vrai petit langage.

5.1.2 Le lexer lit le texte de gauche à droite

Lire un programme pose deux questions, dans cet ordre. Où s'arrêtent les mots : `==` est-il un signe, ou deux signes `=` ? `letter` est-il un nom, ou le mot-clé `let` suivi de `ter` ? Puis, comment ces mots se regroupent-ils ? Le lexer répond à la première question ; le parser, à la seconde.

Pour découper, le lexer regarde le début du texte qui lui reste, y reconnaît un token, le met de côté dans un accumulateur, puis recommence sur la suite. C'est le geste de la calculatrice à pile du chapitre 4, qui lisait ses tokens un par un. Arrivé au texte vide, il rend `{:ok, tokens}`, ses tokens remis dans l'ordre de lecture.



La première ligne de notre programme, caractère par caractère. Chaque accolade regroupe les caractères d'un token ; les espaces séparent les mots, puis sont oubliés.

Deux règles répondent aux questions du début :

- un mot est lu en entier avant que l'on décide s'il est réservé : `letter` est un nom ;
- quand deux signes commencent de la même façon, le plus long est essayé d'abord : `==` avant `=`, `->` avant `-`.

Le lexer ne connaît en revanche aucune règle d'assemblage. Il découpe sans broncher `1.0 +` ou `++ 1.0`, qui ne sont pas des programmes : c'est le parser qui les refusera. Il ne refuse que ce qu'il ne sait pas découper, par ex. le symbole `@`, le nombre `2` sans partie décimale, ou le nom `Price`, qui commence par une majuscule.

5.1.3 Les tokens sont des données Elixir

Voici les tokens du programme complet `let price = 12.0 price * 3.0`, tels que les produit la *référence*, c-à-d. l'interpréteur complet qui fixe le contrat du langage ; toutes les valeurs citées dans ce chapitre sont les siennes :

```
1 [ :let, { :name, "price", :assign, { :number, 12.0 } },
2   { :name, "price", :multiply, { :number, 3.0 } } ]
```

Les mots-clés et les signes deviennent de simples atomes. Les deux sortes de tokens qui portent une donnée, un nom ou un nombre, sont des tuples : le premier élément dit la catégorie, le second la donnée. Attention, `{:name, "price"}` représente un nom, pas son prix ! Le nom reste une chaîne : il est inutile de fabriquer un atome Elixir pour chaque nom tapé.

PIÈGE

Les mots `True` et `False` appartiennent au langage cible. Dans les données de l'interpréteur, nous les représentons par les booléens Elixir `true` et `false`. Cette différence d'écriture ne change pas leur rôle, mais elle compte quand on recopie un exemple.

5.2 Un arbre pour garder le calcul

5.2.1 Calculer au fil de la lecture

La façon la plus simple de calculer ne construit rien : lire un nombre, puis une suite d'opérations, en mettant à jour une *valeur courante*. C'est la récursion avec accumulateur du chapitre 4, appliquée aux tokens. Pour l'entrée `1.0 + 2.0 * 3.0`, prédisez l'accumulateur après chaque opération :

Ce qui est lu	Calcul effectué	Valeur courante
1.0	Aucun encore	1.0
+ 2.0	1.0 + 2.0	3.0
* 3.0	3.0 * 3.0	9.0

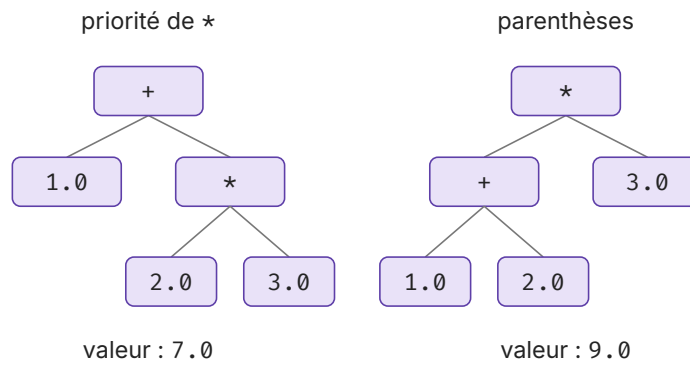
Le résultat est correct pour *cette règle*, mais il diffère de la convention arithmétique habituelle, qui donne 7.0. En effet, une fois 3.0 calculé au milieu de la lecture, l'accumulateur ne dit plus qu'il venait de 1.0 + 2.0 : il conserve la valeur, pas l'organisation du calcul.

À RETENIR

Un algorithme sans arbre peut tout à fait gérer des priorités ou des parenthèses ; notre règle gauche-droite, elle, ne le fait pas. Nous introduisons l'arbre pour garder le calcul comme une donnée, que l'on pourra inspecter, parcourir et transformer.

5.2.2 Les enfants indiquent le groupement

Dans un arbre d'expression, chaque opération est un *nœud* dont les *enfants* sont ses opérands, et chaque nombre est une *feuille*, c-à-d. un nœud sans enfant. Le nœud du haut, qui n'est l'enfant d'aucun autre, est la *racine* : c'est l'opération effectuée en dernier.



Les mêmes nombres et opérateurs, organisés différemment.

Pour $1.0 + 2.0 * 3.0$, la racine est l'addition et son enfant droit est $2.0 * 3.0$. Pour $(1.0 + 2.0) * 3.0$, la racine est la multiplication. Les parenthèses ont modifié la structure ; elles n'ont pas besoin de rester sous la forme d'un nœud supplémentaire. Les priorités sont donc décidées quand l'arbre est construit, et l'évaluateur n'a plus à les deviner.

5.2.3 Un arbre est une donnée récursive

En Elixir, l'arbre de $1.0 + 2.0 * 3.0$ s'écrit avec des tuples imbriqués :

```
1 {:binary, :plus,
2   {:number, 1.0},
3   {:binary, :multiply,
4     {:number, 2.0},
5     {:number, 3.0}}}
```

Le premier élément de chaque tuple indique la catégorie du nœud ; les suivants portent ses données. Une feuille est le token `{:number, ...}` réutilisé tel quel. Le pattern matching du chapitre 2 distingue ainsi une feuille d'une opération, comme il séparait la tête d'une liste de son reste.

L'idée n'est pas neuve. En 1960, John McCarthy décrit le langage Lisp par une fonction `eval`, elle-même écrite en Lisp, qui reçoit le programme sous forme de liste : `(+ 1.0 (* 2.0 3.0))` y dit la même chose que notre tuple. Une fois le programme devenu une valeur, toutes les fonctions sur les valeurs s'appliquent à lui : on peut l'évaluer, l'afficher, le compiler ou le typer.

5.3 Évaluer un arbre

5.3.1 Les valeurs remontent des feuilles vers la racine

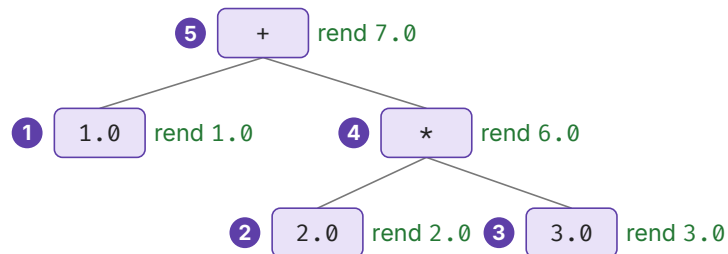
Pour obtenir la valeur d'une addition, il faut d'abord les valeurs de ses deux enfants. L'évaluateur est donc une fonction récursive sur l'arbre : une feuille répond tout de suite, et un nœud d'opération s'appelle d'abord sur ses deux enfants, puis combine leurs réponses.

Suivons ces appels sur $1.0 + 2.0 * 3.0$, en abrégant chaque sous-arbre par son texte. Chaque retrait marque un appel imbriqué, et la flèche donne la valeur rendue :

```

1 eval(1.0 + 2.0 * 3.0)
2   eval(1.0)           → 1.0   une feuille répond aussitôt
3   eval(2.0 * 3.0)
4     eval(2.0)         → 2.0
5     eval(3.0)         → 3.0
6                       → 6.0   2.0 * 3.0, ses enfants connus
7                       → 7.0   1.0 + 6.0, ses enfants connus

```



Le même calcul sur l'arbre. Les pastilles numérotent l'ordre dans lequel les nœuds rendent leur valeur : chacun après ses enfants.

Les appels descendent jusqu'aux feuilles, puis les valeurs remontent : chaque nœud est traité *après* ses enfants. Ce parcours s'appelle le *postordre*. On remarque que la multiplication est calculée avant l'addition sans que l'évaluateur connaisse la moindre règle de priorité : c'est la forme de l'arbre qui l'impose !

Pour vérifier un tel parcours, préférez $8.0 - 3.0$ à $8.0 + 3.0$: l'addition masque une inversion des enfants, alors que la soustraction donne -5.0 au lieu de 5.0 . Un bon exemple doit pouvoir révéler le souci recherché.

5.3.2 Une clause par sorte de nœud

En Elixir, l'évaluateur s'écrit avec une clause par catégorie de nœud. Voici celle des feuilles, qui montre le contrat : un arbre et un environnement, dont nous parlerons plus loin, en entrée ; `{:ok, valeur}` ou une erreur en sortie.

```

1 def eval({:number, number}, _env) do
2   {:ok, number}
3 end

```

La clause d'une opération binaire devra obtenir les résultats de ses deux enfants, propager une éventuelle erreur, puis vérifier que l'opération convient aux valeurs. Ce même parcours servira au chapitre 6 à produire des instructions ; seule la réponse de chaque nœud changera.

POURQUOI CE PARCOURS TERMINE

Sur un arbre fini composé de nombres et d'opérations binaires, chaque appel descend dans un sous-arbre plus petit. Les feuilles répondent directement ; les opérations combinent les réponses ou propagent une erreur. Par récurrence sur la taille de l'arbre, l'évaluation termine. Cette preuve ne couvre pas une récursion arbitraire dans le langage cible.

POUR ALLER PLUS LOIN

Une clause de l'évaluateur s'écrit aussi comme une *règle d'inférence*, où $e \Downarrow v$ se lit « l'expression e s'évalue en la valeur v » :

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{e_1 + e_2 \Downarrow v_1 + v_2}$$

Au-dessus de la barre figurent les *prémisses*, en dessous la *conclusion* : « si e_1 vaut v_1 et e_2 vaut v_2 , alors $e_1 + e_2$ vaut leur somme ». Avec des noms, on ajoute à gauche l'environnement ρ et l'on écrit $\rho \vdash e \Downarrow v$. Cette *sémantique opérationnelle* permet de démontrer des propriétés valables pour tous les programmes, par récurrence sur leurs arbres, comme la terminaison ci-dessus. La page [Sémantique opérationnelle](#)⁷ en donne une première idée ; les chapitres 3 et 5 de *Types and Programming Languages* de Benjamin Pierce la présentent sur de tout petits langages.

5.4 Grammaires et notation BNF

5.4.1 Décrire une infinité de textes en quelques règles

Avant d'écrire le parser qui construit ces arbres, il faut dire précisément quels textes sont des programmes, et comment leurs morceaux s'assemblent. Une liste d'exemples n'y suffit pas, puisqu'il existe une infinité de programmes.

On écrit plutôt une *grammaire*, c-à-d. un petit ensemble de règles qui décrivent toutes les formes admises, comme la grammaire du français dit qu'une phrase simple est formée d'un sujet puis d'un verbe. Notre notation remonte à la description du langage ALGOL 60 par Backus et Naur ; on l'appelle *BNF*, et nous en utilisons une version étendue, avec quelques symboles de répétition.

Découvrons-la sur une grammaire jouet, qui décrit des listes de chiffres comme `[1,2,3]` :

```
1 liste      := "[" elements? "]"
2 elements  := chiffre ("," chiffre)*
3 chiffre   := "0" | "1" | "2" | "3" | "4"
4           | "5" | "6" | "7" | "8" | "9"
```

Chaque ligne est une *règle*. À gauche de `:=`, qui se lit « est formé de », se trouve son nom ; à droite, la forme qu'elle décrit. On y rencontre deux sortes de symboles :

- un *terminal*, écrit entre guillemets comme `"["`, est un morceau à trouver tel quel dans l'entrée ;
- un nom sans guillemets, comme `chiffre`, renvoie à une autre règle : à cet endroit doit figurer quelque chose que cette règle accepte.

Les symboles se combinent ensuite de cinq façons :

⁷ https://fr.wikipedia.org/wiki/S%C3%A9mantique_op%C3%A9rationnelle

Écriture	Sens	Dans l'exemple
a b	une <i>séquence</i> : a , puis b	"[" elements? "]" : un crochet, des éléments, un crochet
a b	une <i>alternative</i> : a ou bien b	un chiffre est "0", ou "1", et ainsi de suite
(...)	un groupe, traité comme un seul élément	("," chiffre) : une virgule suivie d'un chiffre
a*	a répété zéro fois ou davantage	le groupe ("," chiffre) revient autant de fois qu'on veut
a?	a facultatif : zéro ou une fois	elements? autorise la liste vide []

Ainsi la règle `elements` se lit : « un chiffre, suivi d'un nombre quelconque de groupes formés d'une virgule et d'un chiffre ». Cette grammaire accepte `[]`, `[7]` et `[1,2,3]`.

Elle refuse `[1,]`, car une virgule doit être suivie d'un chiffre, et `[,1]`, car une liste non vide commence par un chiffre. Elle refuse même `[12]`, car un `chiffre` est un seul caractère et deux chiffres doivent être séparés par une virgule. Pour simplifier, nous écrivons ces listes sans espace.

PIÈGE

Dans une grammaire, `*`, `?` et `|` appartiennent à la notation : ils disent comment combiner les symboles et ne figurent jamais dans le texte décrit. Quand le langage décrit contient lui-même un astérisque, comme la multiplication de notre langage, celui-ci s'écrit entre guillemets, `"*"`.

5.4.2 Vérifier un texte à la main : la dérivation

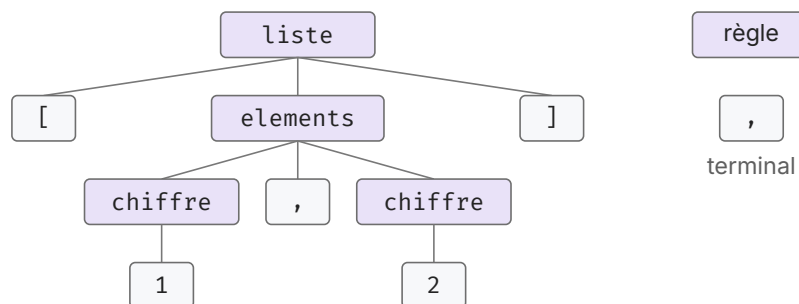
Comment s'assurer qu'un texte appartient au langage décrit ? On part du nom de la première règle et on le remplace, pas à pas, par l'une des formes que la règle autorise, en décidant à chaque `?` et à chaque `*` combien de fois on utilise l'élément.

Ce calcul s'appelle une *dérivation* ; il s'arrête quand il ne reste que des terminaux. Si ceux-ci forment exactement le texte, celui-ci est accepté. Dérivons `[1,2]`, une ligne par remplacement :

1	<code>liste</code>	
2	<code>"[" elements? "]"</code>	règle liste
3	<code>"[" elements "]"</code>	l'option est prise une fois
4	<code>"[" chiffre ("," chiffre)* "]"</code>	règle elements
5	<code>"[" chiffre "," chiffre "]"</code>	le groupe est répété une fois
6	<code>"[" "1" "," chiffre "]"</code>	règle chiffre, alternative "1"
7	<code>"[" "1" "," "2" "]"</code>	règle chiffre, alternative "2"

On retombe sur `[1,2]` : le texte est accepté. Pour `[1,]`, au contraire, aucun choix ne convient, car dès que l'on prend le groupe `("," chiffre)`, la virgule arrive avec un chiffre que le texte ne fournit pas.

Si l'on garde la trace de la règle qui a produit chaque morceau, la dérivation dessine un arbre, l'*arbre de dérivation* : chaque règle utilisée devient un nœud, dont les enfants sont les symboles de la forme choisie.



L'arbre de dérivation de `[1,2]`. Les nœuds violets sont des règles, les cases grises des terminaux ; lues de gauche à droite, les feuilles redonnent le texte.

C'est cet arbre qui explique le mot *abstrait* de l'AST. L'arbre de syntaxe abstraite en est une version épurée : il garde ce qui sert au calcul, comme les nombres et les opérateurs, et laisse de côté les crochets, les virgules et les parenthèses, qui n'ont servi qu'à guider la lecture.

POUR ALLER PLUS LOIN

Chaque règle de nos grammaires remplace un seul nom, quel que soit son entourage : on les dit *non contextuelles*. La hiérarchie de Chomsky les situe au-dessus des langages *réguliers*, ceux que décrivent les expressions régulières, comme `[a-z][a-z0-9_]*` pour nos noms. Les parenthèses imbriquées échappent aux expressions régulières, car il faut se souvenir de chaque (encore ouverte : c'est pourquoi le lexer s'occupe des mots et le parser des imbrications. Une première entrée est la page [Hiérarchie de Chomsky](#)⁸ ; le livre de référence est *Introduction to Automata Theory, Languages, and Computation* de Hopcroft, Motwani et Ullman.

5.5 De la grammaire à l'arbre

5.5.1 Une grammaire peut hésiter entre deux arbres

Revenons à l'arithmétique. Nos grammaires décrivent désormais des suites de tokens, et non plus des caractères : les espaces, retirés par le lexer, n'y figurent pas. Le nom `number`, défini par aucune règle, désigne la catégorie des tokens `{:number, ...}` que fournit le lexer.

La première idée serait d'écrire qu'une expression est un nombre, ou deux expressions reliées par un opérateur :

```
1 expr := number | expr "+" expr | expr "*" expr
```

Cette règle accepte bien `1.0 + 2.0 * 3.0`, mais de deux façons. On peut dériver d'abord `expr "+" expr`, puis développer l'expression de droite en `2.0 * 3.0` ; on peut aussi dériver d'abord `expr "*" expr`, puis développer l'expression de gauche en `1.0 + 2.0`.

⁸ https://fr.wikipedia.org/wiki/Hi%C3%A9rarchie_de_Chomsky

On obtient ainsi les deux arbres de la figure des priorités, et la grammaire ne dit pas lequel retenir : elle est *ambiguë*.

5.5.2 Une règle par niveau de priorité

Pour lever l'ambiguïté, on empile les règles, une par niveau de priorité :

```
1 sum      := product ("+" product)*
2 product := number ("*" number)*
```

Une somme est faite de produits, et un produit de nombres. Dérivons $1.0 + 2.0 * 3.0$: `sum` donne `product "+" product` ; le premier `product` ne répète pas son groupe et donne 1.0 , le second le répète une fois et donne $2.0 * 3.0$.

Aucune autre dérivation n'existe, car un `+` ne peut apparaître qu'au niveau de `sum`, jamais à l'intérieur d'un `product`. La multiplication se retrouve donc forcément sous l'addition. En clair, plus une règle est basse dans la pile, plus son opérateur est prioritaire, c-à-d. plus il « serre » ses opérandes.

Pour qu'une addition passe tout de même sous une multiplication, il faut des parenthèses. On ajoute un niveau *primaire*, qui accepte un nombre ou bien une somme complète entre parenthèses :

```
1 sum      := product ("+" product)*
2 product := primary ("*" primary)*
3 primary := number | "(" sum ")"
```

La règle `primary` fait appel à `sum`, qui fait appel à `product`, qui fait appel à `primary` : une grammaire peut être récursive, comme une fonction. C'est ce qui permet d'imbriquer des parenthèses à n'importe quelle profondeur. Notez enfin qu'une grammaire décrit un ensemble de textes, sans dire comment les lire ni ce qu'ils calculent.

5.5.3 Une fonction de lecture par règle

Le parser regroupe les tokens en suivant la grammaire. Celle de nos calculs ajoute aux règles précédentes la soustraction, la division et le moins unaire :

```
1 expression := sum
2 sum        := product (("+" | "-") product)*
3 product    := unary (("*" | "/" ) unary)*
4 unary      := "-" unary | primary
5 primary    := number | "(" expression ")"
```

Les noms, les fonctions et les comparaisons ajouteront quelques niveaux ; la grammaire complète est imprimée plus loin, dans « Le langage complet ».

Nous écrivons une *descente récursive* : une fonction de lecture, ou *lecteur*, par règle. Chaque lecteur reçoit la liste des tokens qui restent, en consomme le début, et rend l'arbre qu'il a construit avec les tokens qu'il n'a pas consommés, `{:ok, arbre, reste}`, ou bien une erreur. La traduction d'une règle en lecteur suit des correspondances régulières :

Dans la grammaire	Dans le lecteur
une règle, par ex. <code>product</code>	une fonction <code>product(tokens)</code>

Dans la grammaire	Dans le lecteur
un nom de règle dans une forme	un appel au lecteur de cette règle, qui consomme sa part des tokens et rend le reste
un terminal attendu, par ex. <code>)</code>	vérifier que le prochain token est <code>:rparen</code> et le retirer ; sinon, une erreur qui dit ce qui était attendu
une alternative <code> </code>	un choix d'après le prochain token, par ex. une clause par motif
une répétition <code>(...)*</code>	une boucle, c-à-d. une fonction récursive avec un accumulateur, qui s'arrête dès que le prochain token ne peut pas commencer le groupe
une option <code>?</code>	lire l'élément si le prochain token peut le commencer, sinon ne rien consommer

Prenons `primary := number | "(" expression ")"`. Le lecteur regarde le premier token : un nombre donne une feuille ; une parenthèse ouvrante annonce la seconde alternative, et le lecteur consomme `(`, demande une expression entière, puis exige `)` ; tout autre token est une erreur. Notre grammaire est écrite pour que le premier token suffise toujours à choisir l'alternative : c'est ce qui rend la descente récursive si simple.

5.5.4 Suivre la descente sur un exemple

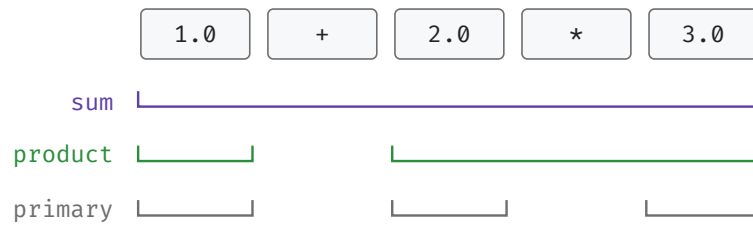
Déroulons la lecture de `1.0 + 2.0 * 3.0`, en omettant le niveau `unary`, qui ne fait ici que passer la main à `primary`. Les tokens sont abrégés par leur texte ; chaque retrait marque un appel imbriqué, et la flèche donne ce que le lecteur rend, l'arbre construit puis le reste :

```

1 sum [1.0 + 2.0 * 3.0]
2   product [1.0 + 2.0 * 3.0]
3     primary [1.0 + 2.0 * 3.0] → 1.0, reste [+ 2.0 * 3.0]
4     + n'est pas de son niveau → 1.0, reste [+ 2.0 * 3.0]
5     + est de son niveau : consommé
6     product [2.0 * 3.0]
7       primary [2.0 * 3.0] → 2.0, reste [* 3.0]
8       * est de son niveau : consommé
9       primary [3.0] → 3.0, reste []
10      fin des tokens → 2.0 * 3.0, reste []
11      fin des tokens → 1.0 + (2.0 * 3.0), reste []

```

Le lecteur de produit lit `2.0 * 3.0` en entier avant de rendre la main au lecteur de somme. Ce dernier obtient donc un arbre de produit comme terme droit de son addition : la priorité de la grammaire est devenue la forme de l'arbre.



Chaque lecteur consomme un intervalle de tokens et rend la main à sa frontière. Le premier `product` s'arrête devant `+`, qui n'est pas de son niveau ; le second absorbe `*`, et c'est pour cela que la multiplication finit sous l'addition.

Remarquez qu'un lecteur rend toujours deux choses : l'arbre construit *et* les tokens qu'il n'a pas consommés. C'est exactement ce que fait `Integer.parse` au chapitre 2 : `Integer.parse("12x")` rend `{12, "x"}`, ce qui a été lu puis ce qui reste à lire. Le lecteur de somme continue ainsi à partir du reste que lui rend le lecteur de produit.

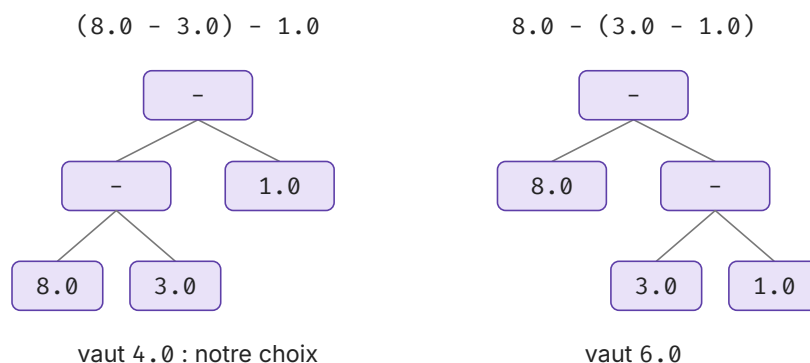
Le lecteur public `parse/1`, lui, exige un reste vide : `1.0 2.0` n'est pas un programme valide parce qu'un préfixe était lisible. C'est le rôle du symbole `EOF` (*end of file*) de la grammaire complète, qui désigne la fin de l'entrée.

À RETENIR

Une grammaire empile une règle par niveau de priorité, du moins prioritaire au plus prioritaire. La descente récursive en fait un lecteur par règle : chacun reçoit des tokens, appelle les lecteurs des règles qu'il contient et rend `{:ok, arbre, reste}` ou une erreur. Le prochain token choisit l'alternative, et une répétition devient une boucle qui accumule l'arbre.

5.5.5 À priorité égale, on groupe à gauche

Que vaut `8.0 - 3.0 - 1.0` ? Tout dépend du groupement : `(8.0 - 3.0) - 1.0` vaut `4.0`, alors que `8.0 - (3.0 - 1.0)` vaut `6.0`. Nos opérateurs binaires s'associent à gauche, et le résultat est donc `4.0`.



Les deux groupements possibles de `8.0 - 3.0 - 1.0`. L'arbre de gauche penche à gauche : la soustraction la plus profonde, `8.0 - 3.0`, est calculée la première.

La grammaire n'en décide pas seule : la règle `sum` voit ici une suite plate, un produit puis deux fois le groupe « signe moins, produit ». C'est la boucle du lecteur de somme qui choisit le groupement. Son accumulateur contient l'arbre déjà construit, et chaque nouvel opérateur en fait l'enfant gauche d'un nœud plus grand :

Tokens restants	Arbre accumulé
- 3.0 - 1.0	8.0
- 1.0	8.0 - 3.0
aucun	(8.0 - 3.0) - 1.0

En Elixir, la première soustraction devient ainsi l'enfant gauche de la seconde : `{:binary, :minus, {:binary, :minus, ...}, {:number, 1.0}}`.

Si, au contraire, le lecteur de somme s'appelait lui-même pour lire tout ce qui suit le premier -, il construirait $8.0 - (3.0 - 1.0)$, l'association à droite, fautive pour la soustraction.

PIÈGE

Le lexer reconnaît le nombre 3.0 et, séparément, le token du moins. Dans -3.0, le parser construit un nœud unaire `{:neg, {:number, 3.0}}`. Il faut aussi distinguer une fin de tokens attendue d'une parenthèse fermante absente : les deux peuvent apparaître pendant la descente récursive.

POUR ALLER PLUS LOIN

Notre parser lit de haut en bas et choisit chaque alternative d'après un seul token : c'est une analyse *LL(1)*. Elle ne supporte pas une règle comme `sum := sum "+" product`, qui ferait rappeler `sum` sans rien consommer ; c'est pourquoi nous écrivons des répétitions. Les analyseurs *LR* lisent de bas en haut et acceptent ces règles, mais leurs tables ne s'écrivent pas à la main : on les génère avec Yacc, Bison ou, pour Erlang et Elixir, [yecc](https://www.erlang.org/doc/apps/parsetools/yecc.html)⁹. Robert Nystrom écrit une descente récursive très proche de la nôtre dans *Crafting Interpreters*¹⁰, librement lisible en ligne ; le chapitre 4 du « Dragon book » d'Aho, Lam, Sethi et Ullman couvre les deux familles.

5.6 Noms et environnements

5.6.1 Une liaison vaut pour la suite

Un nom n'a pas de valeur tout seul : `price` ne vaut 12.0 que parce qu'une ligne précédente l'a dit. Une *liaison* est cette association d'un nom à une valeur. L'*environnement*, que nous appelons aussi *contexte*, est l'ensemble des liaisons connues à un endroit du programme.

Une liste de couples suffit à le représenter, les nouvelles liaisons étant ajoutées en tête :

```
1 # après `let price = 12.0` puis `let quantity = 3.0`
2 [{"quantity", 3.0}, {"price", 12.0}]
```

La recherche parcourt la liste depuis la tête et s'arrête à la première liaison qui porte le nom demandé. Cherchons `price` : le premier couple concerne `quantity`, on passe au suivant ; le second porte le bon nom, et la recherche rend `{:ok, 12.0}`. Si l'on

⁹ <https://www.erlang.org/doc/apps/parsetools/yecc.html>

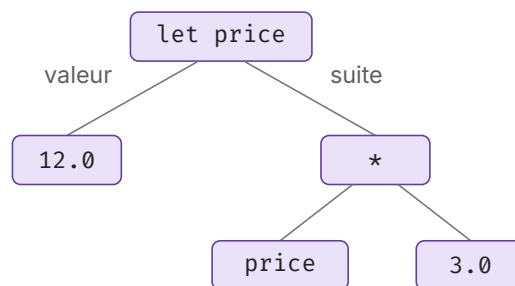
¹⁰ <https://craftinginterpreters.com/parsing-expressions.html>

arrive au bout de la liste sans trouver le nom, par ex. en cherchant `total`, elle rend `{:error, {:undefined_name, "total"}}`.

Pourquoi une liste plutôt qu'une map ? Les deux conviendraient, mais l'ordre de la liste montre quelle liaison est la plus récente, et ajouter une liaison en tête ne recopie rien. Le compilateur du chapitre 6 fera encore autrement : il remplacera chaque nom par un numéro de case.

5.6.2 `let` calcule une valeur, puis évalue la suite

Dans `let price = 12.0 price * 3.0`, on calcule d'abord la valeur `12.0`, puis on évalue la suite dans un environnement enrichi de la liaison. Ce n'est pas une affectation qui transforme une liste existante. L'arbre conserve dans un même nœud, `{:let, "price", {:number, 12.0}, suite}`, le nom, l'expression de valeur et la suite :



L'arbre conserve le nom lié, sa valeur à calculer et la suite du programme.

La *portée* d'une liaison, c-à-d. la partie du programme où elle est visible, se lit donc dans la structure : l'environnement enrichi est passé à la suite, et non aux sous-arbres qui la précèdent. Trois règles de syntaxe découlent de cette forme :

- un programme, ou un bloc, se termine toujours par une expression : `let x = 1.0` seul est refusé par le parser ;
- un `let` n'est admis qu'en tête d'une séquence, jamais comme opérande : `1.0 + let x = 2.0 x` n'a pas de sens ;
- les retours à la ligne ne sont que des espaces : `let x = 8.0 x - 3.0` sur une ligne est le même programme que sur deux.

5.6.3 Masquer n'est pas modifier le passé

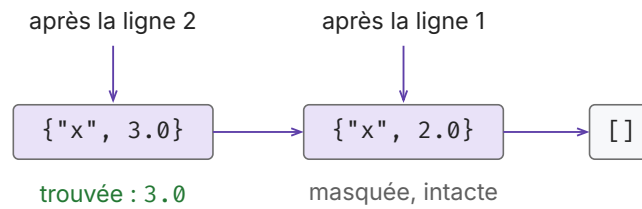
Prédisez la valeur finale, puis écrivez la liste de liaisons visible à chaque ligne :

```

1 let x = 2.0
2 let x = x + 1.0
3 x * 4.0

```

À la deuxième ligne, `x + 1.0` est calculé avant la nouvelle liaison, dans l'environnement `[{"x", 2.0}]` : il vaut `3.0`. La seconde liaison est alors ajoutée en tête, ce qui donne `[{"x", 3.0}, {"x", 2.0}]`. La recherche trouve `3.0` en premier, et le résultat est `12.0`.



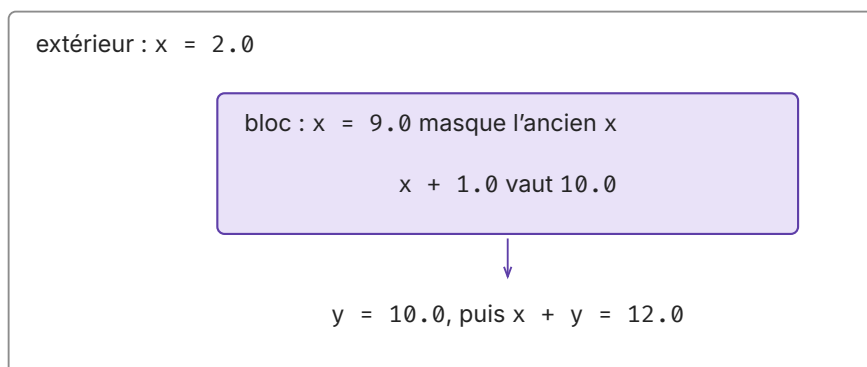
L'environnement après chaque ligne. La liste de la ligne 1 est devenue la queue de celle de la ligne 2 : ajouter une liaison ne recopie rien et ne modifie rien.

On dit que la nouvelle liaison *masque* l'ancienne : celle-ci reste intacte dans la liste, mais la recherche, qui part de la tête, ne l'atteint plus. Rien n'a été effacé, et nous verrons qu'une fonction peut encore la voir.

Les blocs, écrits entre accolades, limitent la portée de leurs liaisons et rendent leur dernière expression. Que vaut cette autre entrée ?

```

1 let x = 2.0
2 let y = { let x = 9.0 x + 1.0 }
3 x + y
  
```



Seule la valeur 10.0 sort du bloc pour être liée à y. Le x intérieur ne remplace pas le x extérieur : le résultat est 12.0.

Le bloc calcule 10.0 dans son propre environnement, où le x intérieur masque l'extérieur. À la sortie, on reprend l'environnement d'avant le bloc, où x vaut toujours 2.0. Un résultat de 19.0 signifierait qu'une liaison a fui hors du bloc.

5.7 Fonctions et fermetures

5.7.1 Une fonction garde un calcul pour plus tard

Notre langage a aussi des fonctions :

```

1 let inc = fn(x) { x + 1.0 }
2 inc(4.0)
  
```

Le parser produit deux nouvelles sortes de nœuds, `{:fn, paramètres, corps}` pour la définition et `{:call, fonction, arguments}` pour l'appel ; le chapitre 6 les reprendra tels quels :

```

1 {:let, "inc",
2   {:fn, ["x"], {:binary, :plus, {:name, "x"}, {:number, 1.0}}}},
3   {:call, {:name, "inc"}, [{:number, 4.0}]}}
  
```

Définir `inc` ne calcule pas `x + 1.0` : aucun argument n'a encore été fourni. Évaluer le nœud `{:fn, ...}` produit donc une valeur qui garde la liste des paramètres et l'arbre du corps, par ex. `{:function, parameters, body}`. Lors de l'appel, `4.0` est calculé puis lié au paramètre `x`, et le corps peut enfin être évalué : le résultat est `5.0`.

Il faut distinguer le *paramètre*, nom écrit dans la définition, de l'*argument*, expression écrite à l'appel. L'*arité* d'une fonction est son nombre de paramètres, et un appel doit fournir autant d'arguments.

Ainsi `inc()` échoue avec `{:error, {:arity, 1, 0}}`, un paramètre attendu, zéro reçu, et `inc(4.0, 5.0)` de même. Appeler un nombre, par ex. `4.0(2.0)`, se lit syntaxiquement mais échoue à l'évaluation : la valeur appelée n'est pas une fonction.

D'où vient `offset` dans `fn(x) { x + offset }` ? Ce nom n'est ni un paramètre de la fonction, ni une liaison créée dans son corps ; on dit que c'est une *variable libre* du corps. Si le corps ne voit que ses paramètres, l'appel échoue avec `{:undefined_name, "offset"}`. Il faut donc conserver davantage que le corps : c'est l'objet de la suite.

5.7.2 Portée lexicale ou portée dynamique ?

Voici l'expérience centrale du chapitre. Avant de lire la suite, choisissez entre `5.0` et `103.0`, et indiquez quelle liaison d'`offset` votre choix utilise :

```
1 let offset = 2.0
2 let f = fn(x) { x + offset }
3 let offset = 100.0
4 f(3.0)
```

Deux règles sont possibles pour trouver la valeur d'une variable libre, et chacune a existé dans de vrais langages :

- la *portée dynamique* la cherche dans l'environnement de l'appelant, au moment de l'appel ;
- la *portée lexicale* la cherche dans l'environnement où la fonction a été définie ; on la dit lexicale parce qu'il suffit de lire le texte du programme pour savoir de quelle liaison il s'agit.

Déroutons les environnements, en abrégant le couple `{"offset", 2.0}` en `offset = 2.0` :

Après la ligne	Environnement, liaison la plus récente à gauche
1	<code>[offset = 2.0]</code>
2	<code>[f = ..., offset = 2.0]</code> , où <code>f</code> a été définie quand l'environnement valait <code>[offset = 2.0]</code>
3	<code>[offset = 100.0, f = ..., offset = 2.0]</code>

À la ligne 4, l'appel `f(3.0)` lie d'abord `x` à `3.0`, puis évalue le corps `x + offset`. La liaison de `x` est placée en tête de l'environnement de l'appelant avec la portée dynamique, et en tête de l'environnement de la définition avec la portée lexicale :

```
1 portée dynamique : [x = 3.0, offset = 100.0, f = ..., offset = 2.0]
2 portée lexicale  : [x = 3.0, offset = 2.0]
```

La recherche d'`offset` part de la gauche. Dans le premier cas, elle trouve `100.0` et le résultat est `103.0` ; dans le second, elle trouve `2.0` et le résultat est `5.0`. Les deux règles ne diffèrent donc que par un seul choix : l'environnement que l'on prolonge pour évaluer le corps.

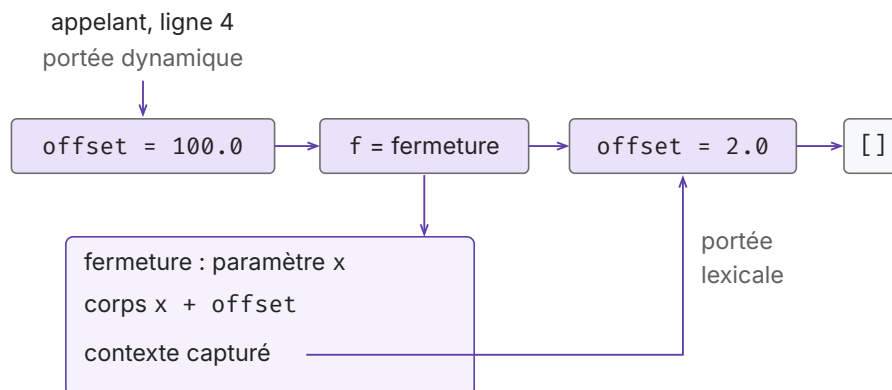
Notre langage choisit la portée lexicale : le résultat est `5.0`. Pourquoi ce choix ? Avec la portée lexicale, le sens d'une fonction se lit là où elle est écrite. Avec la portée dynamique, il dépend de qui l'appelle : il suffirait qu'un appelant possède, par hasard, une liaison nommée `offset` pour changer le résultat de `f`. Elixir, Python et la plupart des langages actuels sont lexicaux.

5.7.3 Une fermeture emporte son environnement

En clair, une fonction se souvient de l'endroit où elle a été écrite. Une *fermeture* (*closure*) est la valeur qui réunit le code de la fonction et cet environnement capturé :

```
1 { :closure, parameters, body, captured_environment }
```

Ce tuple est une donnée ordinaire de notre interpréteur, que nous pouvons ouvrir pour lire le corps, les paramètres et les valeurs capturées. Par rapport à la première version des fonctions, c'est la valeur produite par un nœud `fn` qui change : elle emporte désormais l'environnement du moment.



À la ligne 4, l'appelant et la fermeture désignent la même liste, mais pas au même endroit. La portée dynamique cherche `offset` depuis la tête et trouve `100.0` ; la portée lexicale part du contexte capturé et trouve `2.0`.

Capter un contexte coûte peu. L'environnement est une liste immuable, que la fermeture partage avec le reste du programme sans craindre qu'une liaison ultérieure la modifie, comme deux listes partageaient leur queue au chapitre 3. Le dessin le montre : le contexte capturé n'est pas une copie, mais la queue de la liste de l'appelant.

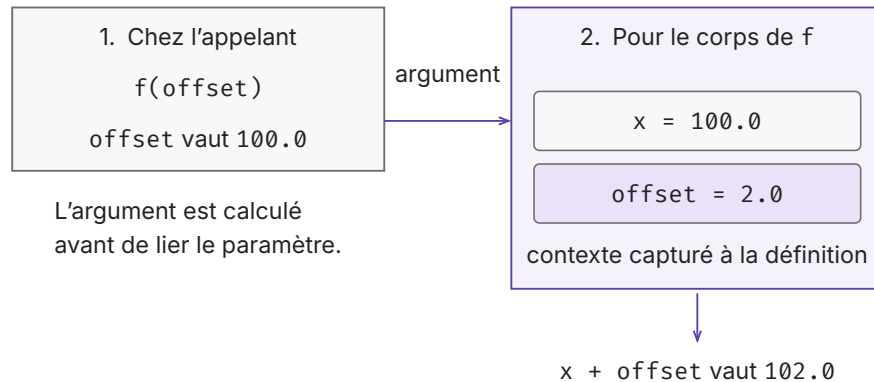
PIÈGE

Essayez le même programme en Python, avec `def f(x): return x + offset` et deux affectations `offset = ...` : vous obtiendrez `103.0` ! Python a pourtant une portée lexicale, mais capturer une liaison n'est pas figer une valeur. En Python, `offset = 100.0` modifie la variable existante, que `f` relit au moment de l'appel. Notre `let`, comme le `=` d'Elixir, crée une *nouvelle* liaison et laisse l'ancienne intacte. En Elixir, `f = fn x -> x + offset end` suivi de `offset = 100.0` donne bien `5.0` pour `f.(3.0)`.

5.7.4 Les arguments se calculent chez l'appelant

Reprenons le même programme, mais remplaçons sa dernière ligne par `f(offset)`. Quel `offset` faut-il lire maintenant ? Il y a deux recherches, à deux moments différents :

Expression	Environnement consulté	Valeur
<code>offset</code> dans l'argument	Celui de l'appelant	100.0
<code>x</code> dans le corps	Paramètre lié à cet argument	100.0
<code>offset</code> dans le corps	Celui capturé par <code>f</code>	2.0



La flèche entre les contextes transporte la valeur de l'argument. Le corps retrouve `offset` = 2.0 dans la fermeture ; il ne relit pas l'`offset` de l'appelant.

Le résultat est 102.0, alors qu'une portée dynamique lirait deux fois `offset` = 100.0 et donnerait 200.0. Il ne suffit donc pas de passer le même environnement à toutes les parties d'un appel.

À RETENIR

Un appel `f(a)` se déroule en trois temps. On évalue `f` et ses arguments dans l'environnement de l'appelant ; `f` doit valoir une fermeture qui attend autant de paramètres. On lie les paramètres aux valeurs obtenues, en tête du contexte capturé par la fermeture. On évalue enfin le corps dans cette nouvelle liste : ses variables libres y prennent la valeur qu'elles avaient à la définition.

5.7.5 Une fonction peut fabriquer une fonction

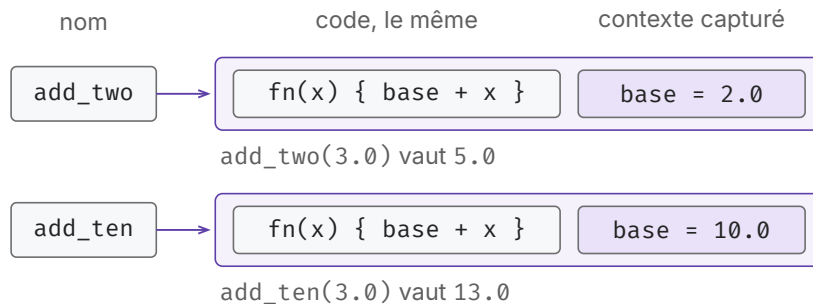
Une fermeture est une valeur comme une autre. Elle peut être liée à un nom, passée à une fonction ou rendue comme résultat :

```

1 let make_adder = fn(base) {
2   fn(x) { base + x }
3 }
4 let add_two = make_adder(2.0)
5 add_two(3.0)
```

L'appel `make_adder(2.0)` évalue le corps de `make_adder` dans l'environnement [`base` = 2.0]. Ce corps est lui-même une définition de fonction : il rend donc une fermeture, qui capture [`base` = 2.0].

Puis l'appel se termine, et le programme principal reprend avec son propre environnement, `[add_two = ..., make_adder = ...]`, où aucune liaison ne porte le nom `base`. Pourtant, `add_two(3.0)` évalue `base + x` dans `[x = 3.0, base = 2.0]`, grâce au contexte capturé, et rend `5.0`.



Avec aussi `let add_ten = make_adder(10.0)`, les deux fermetures partagent le même code mais chacune conserve sa propre `base`.

Sans capture, ce programme ne pourrait pas marcher. Des fonctions qui ne voient que leurs paramètres échouent avec `{:undefined_name, "base"}`, et une portée dynamique échouerait de même : au moment de l'appel, `base` ne figure ni parmi les paramètres de `add_two`, ni chez son appelant.

Dans le corps `base + x`, `base` est une variable libre, et le contexte capturé lui donne une valeur. Une fonction sans variable libre est dite *fermée* ; cette distinction resservira au chapitre 6, pour compiler des appels.

Une dernière précision : à la différence d'un `def` de Python, notre `let` n'est pas récursif. Au moment où l'on calcule la valeur de `let f = fn(x) { f(x) }`, la liaison `f` n'existe pas encore, et la fermeture ne peut pas se capturer elle-même. La définition réussit pourtant ; c'est l'appel `f(1.0)` qui échoue, avec `{:undefined_name, "f"}`.

POUR ALLER PLUS LOIN

Les fermetures viennent du *lambda-calcul* d'Alonzo Church, un langage réduit aux fonctions, à leurs paramètres et à leurs appels. Il suffit pourtant à tout calculer : on y écrit des boucles en passant une fonction à elle-même. Notre langage le permet aussi, sans `let` récursif :

```
1 let fact = fn(self, n) {
2   case n < 1.0 { True -> 1.0 False -> n * self(self, n - 1.0) }
3 }
4 fact(fact, 5.0)
```

vaut `120.0`. Un langage qui peut ainsi simuler n'importe quel calcul est dit *Turing-complet* ; la page [Lambda-calcul](https://fr.wikipedia.org/wiki/Lambda-calcul)¹¹ en donne les bases. Le « [modèle des environnements](#) »¹² de *Structure and Interpretation of Computer Programs* (section 3.2) dessine des environnements très proches de nos listes de liaisons.

¹¹ <https://fr.wikipedia.org/wiki/Lambda-calcul>

5.8 Décider : booléens et `case`

5.8.1 Comparer produit un booléen

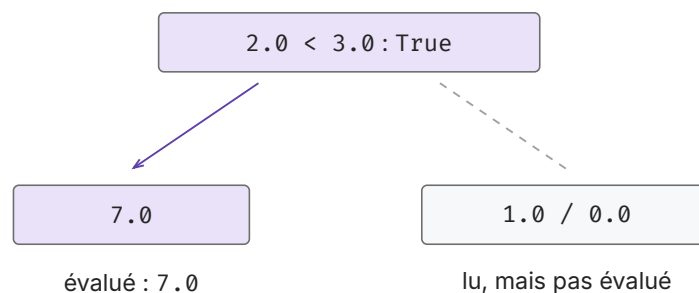
Les comparaisons `<` et `==` produisent des booléens, et ne comparent que deux nombres : `True == True` est une erreur de types, alors que Python l'accepte, et `True` ne se convertit jamais implicitement en `1.0`.

Le contrôle s'écrit avec un `case` à deux branches, `True` puis `False`, dans cet ordre. Attention à ne pas le confondre avec le `case` d'Elixir du chapitre 2 : celui de notre langage cible ne connaît ni motifs ni gardes. Il choisit seulement entre deux branches selon un booléen, et c'est notre interpréteur qui lui donne ce sens.

```
1 case 2.0 < 3.0 {
2   True -> 7.0
3   False -> 1.0 / 0.0
4 }
```

5.8.2 `case` n'évalue qu'une branche

La condition vaut `True`, et l'évaluateur ne calcule que la branche choisie : le résultat est `7.0`, sans division par zéro. Contrairement à une opération arithmétique, `case` ne suit donc pas le postordre : il évalue sa condition, puis un seul de ses deux autres enfants.



Le chemin plein est exécuté ; le chemin pointillé reste inactif. Une erreur de syntaxe y serait néanmoins refusée dès la lecture.

Une branche est une expression, pas une séquence : pour y créer des liaisons, on l'écrit entre accolades, `True -> { let y = 1.0 y }`, et ces liaisons ne sortent pas du bloc.

5.8.3 Une fonction ordinaire ne peut pas remplacer `case`

Pourquoi `case` n'est-il pas une simple fonction ? Nous avons vu *où* les arguments d'un appel sont calculés ; voyons maintenant *quand*. Écrivons une fonction qui imite `case`, puis appelons-la avec une division interdite dans la branche que l'on ne veut pas :

```
1 let choose = fn(test, yes, no) {
2   case test {
3     True -> yes
4     False -> no
5   }
6 }
7 choose(True, 7.0, 1.0 / 0.0)
```

Notre langage est *strict* : lors d'un appel, tous les arguments sont évalués, dans l'ordre et chez l'appelant, avant que le corps ne commence. On parle aussi d'*appel par valeur*, car la fonction reçoit des valeurs, jamais des calculs en attente. Déroulons l'appel :

Pas	Ce qui est évalué	Résultat
1	le nom <code>choose</code>	une fermeture à trois paramètres
2	le premier argument, <code>True</code>	<code>true</code>
3	le deuxième argument, <code>7.0</code>	<code>7.0</code>
4	le troisième argument, <code>1.0 / 0.0</code>	l'erreur <code>:division_by_zero</code>

L'appel s'arrête au quatrième pas : le corps de `choose`, qui aurait choisi `yes`, ne s'exécute jamais. Le programme échoue donc, et `Mini.run/1` rend l'erreur `{:error, {:eval, :division_by_zero}}`.

Le `case`, lui, évalue d'abord sa condition, choisit une branche, puis n'évalue que celle-ci. Une construction qui décide elle-même lesquels de ses sous-arbres évaluer s'appelle une *forme spéciale* ; dans un langage strict, une fonction ordinaire ne peut pas en être une.

Python se comporte de la même façon. Écrite en Python, la fonction `choose` lève `ZeroDivisionError` avant même de commencer. En revanche, l'expression `7.0 if True else 1.0 / 0.0` vaut `7.0`, car `if ... else` est une forme du langage, et non une fonction.

POUR ALLER PLUS LOIN

D'autres langages retardent le calcul d'un argument jusqu'à ce que le corps en ait besoin. L'*appel par nom* d'ALGOL 60 le recalcule à chaque usage ; l'*appel par nécessité* de Haskell le calcule au plus une fois et mémorise le résultat. Avec l'une ou l'autre de ces règles, `choose(True, 7.0, 1.0 / 0.0)` vaudrait `7.0`. La [section 4.2¹³](#) de *Structure and Interpretation of Computer Programs* transforme pas à pas un interpréteur strict en interpréteur paresseux.

5.9 Le langage complet

5.9.1 Un programme qui combine tout

Toutes les formes vues jusqu'ici se lisent maintenant depuis le texte. Voici un programme qui combine liaisons, fonctions, fermeture et choix :

```

1 let discount = 0.9
2 let total = fn(price, quantity) {
3   let amount = price * quantity
4   case amount < 30.0 {
5     True -> amount
6     False -> amount * discount
7   }
8 }
9 total(12.0, 3.0)

```

La fonction capture `discount`. Ses arguments valent `12.0` et `3.0` ; `amount` vaut `36.0` ; la condition est fausse ; le résultat vaut environ `32.4`. Le « environ » rappelle que les nombres sont des flottants binaires, comme en Python : comparer des calculs décimaux demande souvent une tolérance.

Le contrat final contient des nombres décimaux, des booléens, des noms, les opérations, les liaisons, les blocs, les fonctions, les appels et `case`. Il n'a ni chaînes, ni listes, ni mutation, ni entrée-sortie.

5.9.2 La grammaire de référence

Voici la grammaire complète, celle du contrat de référence, où chaque niveau de priorité est une règle. Les noms `number` et `name` désignent des catégories de tokens, et `"True"` le token `{:bool, true}`.

```

1 program    := sequence EOF
2 sequence   := "let" name "=" expression sequence | expression
3 expression := equality
4 equality    := comparison ("==" comparison)*
5 comparison := sum("<" sum)*
6 sum        := product (("+" | "-") product)*
7 product    := unary (("*" | "/" ) unary)*
8 unary      := "-" unary | call
9 call       := primary "(" arguments? ")"*
10 primary   := number | "True" | "False" | name
11           | "(" expression ")" | "{" sequence "}"
12           | "case" expression "{" "True" "->" expression
13             "False" "->" expression "}"
14           | "fn" "(" parameters? ")" "{" sequence "}"
15 arguments := expression ("," expression)*
16 parameters := name ("," name)*

```

Quelques règles méritent une lecture attentive :

- `program := sequence EOF` exige qu'après la séquence, il ne reste plus aucun token ;
- la règle `sequence` s'appelle elle-même à droite : une séquence est soit un `let` suivi d'une autre séquence, soit une simple expression, et c'est pourquoi un programme se termine toujours par une expression ;
- dans `call`, l'option `?` autorise un appel sans argument, `f()`, et la répétition `*` des appels en chaîne : `make_adder(2.0)(3.0)` vaut `5.0` ;
- `arguments` et `parameters` décrivent des listes séparées par des virgules, sur le modèle de la règle `elements` des listes de chiffres.

Chaque règle produit une forme de nœud de l'AST :

Règle	Nœud construit
sequence avec let	<code>{:let, nom, valeur, suite}</code>
equality, comparison, sum, product	<code>{:binary, op, gauche, droite}</code> , un par opérateur lu
unary avec -	<code>{:neg, arbre}</code>
call avec parenthèses	<code>{:call, fonction, arguments}</code>
primary	<code>{:number, n}</code> , <code>{:bool, b}</code> , <code>{:name, nom}</code> , <code>{:case, condition, oui, non}</code> ou <code>{:fn, paramètres, corps}</code>
parenthèses et blocs	aucun nœud : l'arbre de l'intérieur est rendu tel quel

Les priorités, de la plus faible à la plus forte, sont donc `==`, `<`, `+` et `-`, `*` et `/`, le moins unaire, puis l'appel. Tous les opérateurs binaires s'associent à gauche : `1.0 < 2.0 < 3.0` construit `(1.0 < 2.0) < 3.0`, dont la seconde comparaison reçoit un booléen et échoue, à la différence de Python.

Un nom s'écrit en minuscules, chiffres et tirets bas, `[a-z][a-z0-9_]*` : `Price` est refusé dès le lexer. Une fonction peut n'avoir aucun paramètre, `fn() { 7.0 }`, et ses paramètres doivent être distincts.

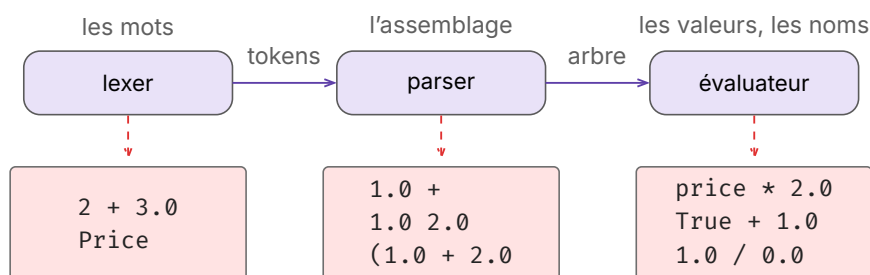
À RETENIR

Une grammaire décrit quelles formes s'assemblent. Une *sémantique* dit ce qu'elles font, et c'est l'évaluateur qui la met en œuvre. Une expression peut respecter la grammaire et échouer à l'exécution : `4.0(2.0)` en est un exemple.

5.10 Où le souci devient-il visible ?

5.10.1 Chaque brique connaît ses erreurs

Face à une erreur, demandez d'abord quelle brique peut la connaître. Le lexer et le parser vérifient la *syntaxe*, c-à-d. la forme du texte : des mots connus, assemblés selon la grammaire. L'évaluateur applique la *sémantique* : il raisonne sur les valeurs et les noms.



Chaque brique arrête les entrées qu'elle est seule à pouvoir juger. Un souci de parenthèse ne se cherche pas dans le calcul, et un programme bien formé peut encore échouer à l'évaluation.

Les briques de la référence rendent `{:ok, ...}` en cas de succès, et sinon une erreur qui dit ce que la brique sait :

- le lexer rend `{:error, {:lex, raison, texte_restant}}` ;
- le parser rend `{:error, {:parse, raison, tokens_restants}}` ;
- l'évaluateur rend `{:error, raison}`.

La fonction `Mini.run/1`, qui enchaîne les trois, étiquette cette dernière en `{:error, {:eval, raison}}`, pour que l'on sache d'où vient l'échec. Voici ce qu'elle rend pour quelques entrées, la valeur en commentaire sous chaque appel :

```
1 Mini.run("1.0 + 2")
2 # {:error, {:lex, :expected_decimal, "2"}}
3 Mini.run("1.0 +")
4 # {:error, {:parse, :expected_expression, []}}
5 Mini.run("(1.0 + 2.0)")
6 # {:error, {:parse, {:expected, :rparen}, []}}
7 Mini.run("1.0 2.0")
8 # {:error, {:parse, :unexpected_tokens, [{:number, 2.0}]}}
9 Mini.run("True + 1.0")
10 # {:error, {:eval, :type_mismatch}}
```

Le lexer joint le texte restant à partir du caractère qu'il n'a pas su lire : pour `1.0 + 2`, c'est `"2"`, ce qui désigne le nombre fautif. Le parser joint les tokens qu'il n'a pas pu placer : la liste vide de `1.0 +` dit que l'entrée s'est terminée trop tôt, celle de `1.0 2.0` montre le token en trop.

5.10.2 Une erreur remonte sans devenir un nombre

Comment une erreur née au fond de l'arbre arrive-t-elle jusqu'à la racine ? Prenons le programme `2.0 * (1.0 / 0.0) + 5.0`. La division rend une erreur au nœud de multiplication, son parent, qui ne calcule pas son produit : il constate l'erreur et la rend telle quelle à l'addition, qui fait de même.

Chaque nœud transmet ainsi l'erreur vers la racine, sans la transformer en nombre, et `Mini.run/1` rend `{:error, {:eval, :division_by_zero}}`.

Entre les briques, le principe est le même : si le lexer échoue, le parser n'est pas appelé, et si le parser échoue, l'évaluateur ne l'est pas. En Elixir, `with` écrit ce chaînage en quelques lignes, car il s'arrête à la première étape qui ne rend pas `{:ok, ...}`.

EN PRATIQUE

Choisissez des contre-exemples qui rendent un mécanisme observable. Pour tester une fermeture, masquez un nom après la définition ; pour tester un choix de branche, mettez un calcul interdit dans la branche inactive ; pour tester la consommation complète, ajoutez un token après une expression valide.

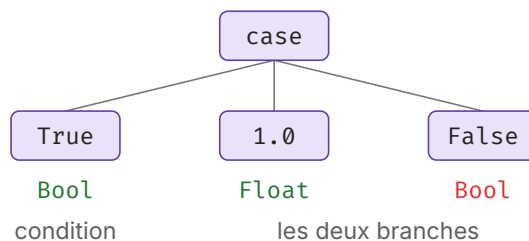
5.11 Ce que l'on peut savoir avant d'exécuter

L'interpréteur demande à chaque nœud une valeur. Le compilateur du chapitre 6 lui demandera des instructions pour une machine, et un *typeur* lui demande un *type*, c-à-d. l'ensemble des valeurs qu'il peut produire : trois outils, une seule représentation.

Dans notre langage, un nombre a le type `Float` et un booléen le type `Bool`. Le typeur applique des règles simples : l'addition exige deux `Float` et rend un `Float`, la condition d'un `case` doit être un `Bool`, et ses deux branches doivent avoir le même type, quelle que soit la condition.

Il examine ainsi tout l'arbre, branches inactives comprises, sans rien exécuter. Le programme `case True { True -> 1.0 False -> False }` rend `1.0` avec l'interpréteur, puisque la branche `False` n'est jamais calculée ; le typeur, lui, le refuse.

branches `Float` et `Bool` : refusé avant d'exécuter



Le typeur annote chaque nœud de son type. La condition est bien un `Bool`, mais les deux branches n'ont pas le même type : le programme est refusé avant toute exécution.

De même, `True + 1.0` est refusé avant de calculer quoi que ce soit, alors que l'interpréteur ne découvre le souci qu'en arrivant à l'addition.

PIÈGE

Bien typé ne veut pas dire correct pour tout objectif. `1.0 / 0.0` a le type `Float`, mais échoue pendant le calcul. Les règles de types ne prouvent ni que le prix facturé est le bon, ni qu'une boucle finira.

POUR ALLER PLUS LOIN

Les règles de typage s'écrivent comme les règles d'évaluation, en fractions, où $e : \tau$ se lit « e a le type τ » :

$$\frac{e_1 : \text{Float} \quad e_2 : \text{Float}}{e_1 + e_2 : \text{Float}} \qquad \frac{c : \text{Bool} \quad a : \tau \quad b : \tau}{\text{case } c \{ \text{True} \rightarrow a \text{ False} \rightarrow b \} : \tau}$$

La seconde dit en clair : si la condition est un booléen et si les deux branches ont un même type τ , le `case` a ce type. Un typeur réel ajoute à gauche un contexte Γ , les types des noms, et écrit $\Gamma \vdash e : \tau$. Il refuse d'ailleurs `fact(fact, 5.0)` : le type de `self` devrait se contenir lui-même. Le projet Hindley–Milner d'AP9 *infère* ces types sans annotation : `fn(x) { x } y` reçoit le type $a \rightarrow a$, où a désigne un type quelconque, choisi à chaque usage, `Float` pour `id(42.0)` et `Bool` pour `id(True)`. La page [Type rule](#)¹⁴ présente la notation ; les chapitres 8 et 22 de *Types and Programming Languages* de Benjamin Pierce, les règles puis l'inférence.

5.12 Vérifier sa compréhension

Commençons par la grammaire des listes de chiffres. Parmi `[]`, `[0,0]`, `[3,]` et `[45]`, lesquelles accepte-t-elle ? Répondez avant de lire la suite.

Elle accepte les deux premières : `[]` en ne prenant pas l'option `elements?`, et `[0,0]` en répétant une fois le groupe `("," chiffre)`. `[3,]` est refusée, car la virgule doit être suivie d'un chiffre, et `[45]` aussi, car deux chiffres doivent être séparés par une virgule. Écrivez la dérivation de `[0,0]` ; elle a la même forme que celle de `[1,2]`.

Dessinez ensuite les deux arbres possibles de `8.0 / 4.0 / 2.0`. Lequel construit notre parser, et que vaut-il ? Le groupement à gauche donne `(8.0 / 4.0) / 2.0`, qui vaut `1.0` ; l'autre arbre vaudrait `4.0`.

Prédisez maintenant ce programme avant de lire la réponse :

```
1 let x = 1.0
2 let f = fn(y) { x + y }
3 let x = 5.0
4 f(1.0)
```

Le résultat est `2.0` : `f` a capturé l'environnement où `x` valait `1.0`, et la liaison `x = 5.0`, créée ensuite, ne change pas ce contexte. Avec une portée dynamique, le corps lirait le `x` de l'appelant et le résultat serait `6.0`.

Remplacez la dernière ligne par `f(x)` : le résultat devient `6.0`, car l'argument, lui, est calculé chez l'appelant, où `x` vaut `5.0`, puis ajouté au `x` capturé.

Que rend `case 1.0 < 2.0 { True -> 1.0 / 0.0 False -> 7.0 }` ? La condition est vraie, et c'est la division qui est choisie : l'évaluateur échoue avec `:division_by_zero`, et `Mini.run/1` rend donc `{:error, {:eval, :division_by_zero}}`. La branche `False` n'a pas été calculée, mais elle a bien été lue.

¹⁴ https://en.wikipedia.org/wiki/Type_rule

Enfin, à quelle brique $(1.0 + 2.0$ et $1.0\ 2.0$ échouent-ils ? Tous deux au parser : le premier attend une parenthèse fermante, le second laisse un token après une expression complète, et `parse/1` refuse ce reste.

POUR EN PARLER

Le même texte $1.0 + 2.0 * 3.0$ vaut 9.0 avec la règle gauche-droite et 7.0 avec l'arbre : la priorité est une décision du parser, pas une propriété du texte.

Dans une grammaire, `|` sépare des alternatives, `*` répète et `?` rend facultatif. Empiler une règle par niveau fixe les priorités, car un `"+"` ne peut pas apparaître à l'intérieur d'un `product` ; lire la répétition de gauche à droite donne l'association à gauche.

Un lecteur rend l'arbre lu *et* le reste des tokens, comme `Integer.parse/1` au chapitre 2 ; `parse/1` exige un reste vide, sans quoi $1.0\ 2.0$ serait accepté.

Une fermeture emporte l'environnement de sa définition : `f(3.0)` vaut 5.0 même après `let offset = 100.0`, alors qu'une portée dynamique donnerait 103.0 , et `f(offset)` vaut 102.0 parce que l'argument est calculé chez l'appelant.

`case` n'évalue qu'une branche ; une fonction `choose` ordinaire calculerait $1.0 / 0.0$ avant même d'entrer dans son corps, car notre langage est strict.

Au chapitre 6, nous repartirons de l'arbre pour descendre vers une machine à pile, avec une nouvelle question à chaque nœud : quelles instructions laisseront sa valeur au bon endroit pour la suite du programme ?

Des octets aux instructions

SÉANCE AP3 · PROCESSEUR, OCTETS, COMPILATION ET MACHINE À PILE

L'interpréteur du chapitre 5 calcule $8.0 - 3.0$ en parcourant l'arbre ; pour évaluer dix fois la même expression, il refait dix fois ce parcours. Un *compilateur* fait autrement. Il traduit l'arbre une fois pour toutes en une suite d'*instructions*, c-à-d. d'ordres élémentaires comme « empiler ce nombre » ou « soustraire les deux derniers ». Une *machine* exécute ensuite ces instructions sans plus jamais regarder l'arbre.

L'idée a l'âge de l'informatique : Grace Hopper nomme *compiler* son système A-0 en 1952, et FORTRAN, chez IBM en 1957, convainc en produisant un code aussi rapide que celui écrit à la main. Le compilateur d'Elixir et sa machine, la BEAM, fonctionnent ainsi, à une tout autre échelle.

Dans ce chapitre, notre compilateur traduira $8.0 - 3.0$ en vingt octets, et notre machine les exécutera pour rendre 5.0 . Que contiennent ces octets, et comment une machine les lit-elle ? Nous partons d'un processeur réel et de l'écriture des nombres en octets, avant de construire notre propre machine et son compilateur.

6.1 Ce qu'est un processeur

Un ordinateur exécute un programme avec deux organes : une *mémoire*, qui conserve des nombres, et un *processeur*, qui les transforme. Prenons-les un par un, sur un exemple assez petit pour être dessiné.

6.1.1 Une adresse désigne une case

Imaginons la mémoire comme une longue rangée de casiers numérotés. Le numéro d'un casier est son *adresse* ; ce qu'il contient est sa *valeur*. Sur nos machines, chaque casier contient un *octet*, c-à-d. un groupe de huit *bits*, huit chiffres valant chacun 0 ou 1 : un octet représente donc un nombre entre 0 et 255 , et c'est tout.

Plusieurs casiers voisins forment un nombre plus grand, une instruction ou un morceau d'image, selon le format que l'on a choisi pour les lire. Nous détaillerons ces écritures dans la section suivante.

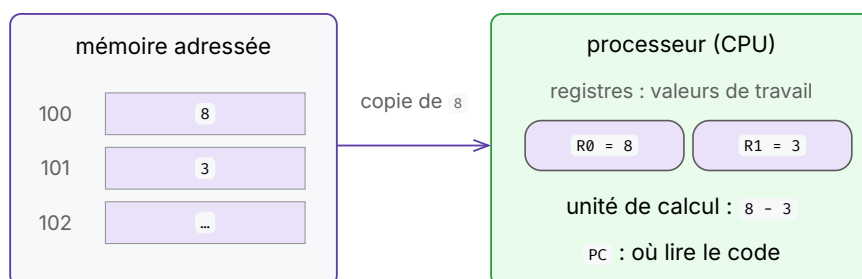
À l'adresse 100 , il peut y avoir la valeur 8 . Demander l'adresse 100 et demander son contenu ne sont donc pas la même chose ! La case peut changer de contenu sans changer d'adresse, et plusieurs adresses peuvent contenir le même octet.

6.1.2 Le processeur calcule sur des registres

Le *processeur*, ou CPU (*central processing unit*), est le circuit qui exécute le programme, une instruction élémentaire après l'autre : copier un nombre, additionner deux nombres, comparer deux valeurs. L'ensemble des instructions qu'il sait exécuter, avec la façon de les écrire en octets, s'appelle son *jeu d'instructions*.

Le processeur possède des *registres*, quelques emplacements internes désignés par un nom fixe, par ex. $R0$, $R1$, $R2$. Ils sont très peu nombreux, quelques dizaines au plus, mais immédiatement disponibles pour calculer. Dans le modèle simple que nous suivons :

- un *chargement* copie une valeur de la mémoire vers un registre ;
- une opération combine des registres ;
- un *rangement* recopie le résultat d'un registre en mémoire.



L'adresse 100 et la valeur 8 sont deux informations distinctes.

Un exemple de machine à registres. Ces noms illustrent le principe ; ce n'est pas encore le jeu d'instructions de notre machine.

Dans la notation illustrative $SUB\ R2, R0, R1$, le processeur prend les valeurs de $R0$ et $R1$, les soustrait dans cet ordre et place le résultat dans $R2$: l'instruction nomme ses emplacements de travail. Notre machine utilisera une autre convention, où les opérandes se trouvent au sommet d'une pile.

6.1.3 Le programme est rangé en mémoire

En 1945, le rapport de John von Neumann sur l'EDVAC décrit une machine dont le programme est rangé en mémoire, comme ses données ; l'EDSAC de Cambridge en est l'une des premières réalisations, en 1949. Nos ordinateurs suivent toujours ce principe : les instructions sont des octets, placés à des adresses comme les autres.

Un registre de contrôle joue alors un rôle particulier : le *compteur ordinal*, souvent appelé *program counter* ou PC. Il contient l'adresse de la prochaine instruction à lire. Dans notre machine, cette adresse se compte en octets depuis le début du programme, la suite d'octets que nous appellerons le *binaire*.

Le PC ne compte ni les lignes du fichier source, ni les tours de boucle. Une instruction qui occupe quatre octets le fait avancer de quatre ; dans notre machine, où certaines instructions en occupent neuf, il avancera de neuf. Une instruction de *saut* peut au contraire lui donner une tout autre valeur, de sorte qu'une même adresse peut être visitée plusieurs fois.

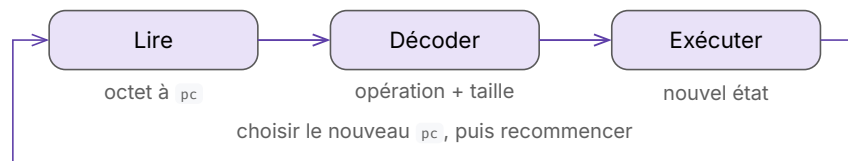
6.1.4 Lire, décoder, exécuter

Le travail du processeur se décrit avec trois verbes, répétés sans fin :

- *lire* les octets situés à l'adresse du PC ;

- *décoder* ces octets, c-à-d. identifier l'opération et ses paramètres ;
- *exécuter* l'opération, qui transforme l'état et choisit le prochain PC.

Le terme anglais *fetch-decode-execute* décrit ce même cycle.



Le cycle recommence jusqu'à l'arrêt, une erreur ou l'épuisement du budget de pas.

Suivons-le sur une machine imaginaire dont chaque instruction occupe quatre octets. Son programme, rangé aux adresses 0 à 19, calcule la différence des deux nombres rangés aux adresses 100 et 101, ceux de la figure précédente, et range le résultat à l'adresse 102. Dans ce listing, [100] désigne *le contenu* de la case 100, et non le nombre 100.

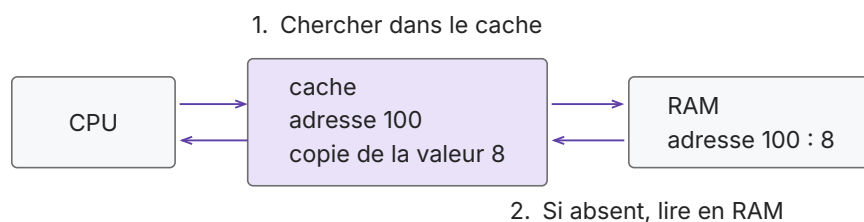
PC	Instruction lue	Effet	PC suivant
0	LOAD R0, [100]	R0 reçoit une copie de 8	4
4	LOAD R1, [101]	R1 reçoit une copie de 3	8
8	SUB R2, R0, R1	R2 reçoit 8 - 3, soit 5	12
12	STORE R2, [102]	la case 102 reçoit 5	16
16	HALT	arrêt de la machine	aucun

À chaque ligne, le processeur lit les quatre octets situés à l'adresse du PC, reconnaît l'instruction, l'exécute, puis ajoute quatre au PC. Remarquez que les cases 100 et 101 valent toujours 8 et 3 à la fin : un chargement copie, il ne déplace pas.

Que faudrait-il changer pour que la machine recommence le calcul au lieu de s'arrêter ? Il suffirait de remplacer HALT par un saut qui donne au PC la valeur 0 ; la machine tournerait alors indéfiniment.

6.1.5 Le cache garde des copies proches du calcul

Sur une machine réelle, lire une donnée dans la mémoire principale, la RAM, prend bien plus de temps qu'une opération sur des registres. Un *cache* est une petite mémoire rapide, placée près du processeur, qui conserve des copies de morceaux de la RAM. C'est le matériel qui choisit ce qu'il y garde : si la donnée demandée y a déjà une copie, la lecture est rapide ; sinon, elle est lue en RAM, et une copie est gardée pour la fois suivante.



Au prochain accès, la copie peut éviter la lecture en RAM.

Les flèches du haut suivent la demande, celles du bas le retour. La RAM n'est consultée que si la donnée manque dans le cache ; le dessin simplifie le chemin de lecture d'une machine réelle.

Notre machine ne simule aucun cache : un cache change la durée d’une lecture, jamais la valeur lue. Retenez surtout la différence avec un registre : un registre est nommé par l’instruction, alors que le cache est invisible pour le programme.

À RETENIR

La mémoire est une rangée de cases d’un octet : l’*adresse* numérote une case, la *valeur* est son contenu. Le processeur calcule sur quelques *registres* nommés ; l’un d’eux, le *PC*, contient l’adresse de la prochaine instruction. Le processeur répète lire, décoder, exécuter : une instruction ordinaire fait avancer le PC de sa taille, un saut lui donne une autre valeur. Le *cache* garde des copies de la mémoire pour aller plus vite, sans que le programme le voie ni que les valeurs changent.

POUR ALLER PLUS LOIN

Un cache copie des blocs entiers, souvent de 64 octets, et les processeurs en empilent plusieurs niveaux. Parcourir des données voisines, comme les cases successives d’un tableau NumPy, profite ainsi des copies déjà faites. Ulrich Drepper détaille ces effets du point de vue du programmeur dans « [What Every Programmer Should Know About Memory](https://www.akkadia.org/drepper/cpumemory.pdf) »¹⁵ (2007). Le cours [Nand2Tetris](https://www.nand2tetris.org)¹⁶ de Nisan et Schocken fait construire un ordinateur complet, des portes logiques jusqu’au compilateur ; la référence classique pour l’architecture est *Computer Organization and Design* de Patterson et Hennessy.

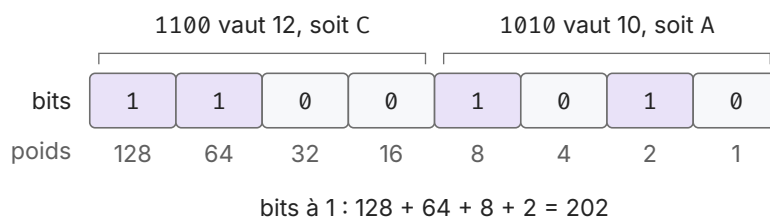
6.2 Octets et représentations

Notre machine lira des programmes faits d’octets. Avant d’en fabriquer, apprenons à écrire un octet de plusieurs façons, à ranger un nombre plus grand sur plusieurs octets, puis à relire ces octets en Elixir.

6.2.1 Un octet, plusieurs écritures

Un bit vaut 0 ou 1. Huit bits forment un *octet*, qui peut représenter $2^8 = 256$ valeurs distinctes, de 0 à 255. Comment lire un octet écrit en bits ? Comme en décimal, chaque chiffre a un *poids* qui dépend de sa position.

En base deux, ces poids sont les puissances de deux, de 128 pour le bit de gauche, dit *de poids fort*, à 1 pour le bit de droite, dit *de poids faible*. On additionne les poids des bits qui valent 1.



L’octet 1100 1010 vaut 202. Coupé en deux moitiés de quatre bits, il s’écrit **CA** en hexadécimal.

¹⁵ <https://www.akkadia.org/drepper/cpumemory.pdf>

¹⁶ <https://www.nand2tetris.org>

Dans l'autre sens, pour écrire 202 en binaire à la main, on retire tour à tour la plus grande puissance de deux possible : $202 - 128 = 74$, puis $74 - 64 = 10$; 32 et 16 sont trop grands ; $10 - 8 = 2$; 4 est trop grand ; $2 - 2 = 0$. Les poids utilisés, 128, 64, 8 et 2, sont les bits à 1 : 1100 1010.

Écrire des octets bit à bit devient vite illisible. On leur préfère l'*hexadécimal*, la base seize, dont les chiffres vont de 0 à 9, puis de A, qui vaut dix, à F, qui vaut quinze. Un chiffre hexadécimal représente exactement quatre bits, puisque $2^4 = 16$; un octet tient donc toujours en deux chiffres hexadécimaux.

Pour passer du binaire à l'hexadécimal, on coupe l'octet en deux moitiés : 1100 vaut $8 + 4 = 12$, soit C, et 1010 vaut $8 + 2 = 10$, soit A. D'où CA, qui se relit en décimal $12 \times 16 + 10 = 202$.

En Elixir, le préfixe 0b annonce un nombre écrit en binaire et 0x un nombre écrit en hexadécimal ; le tiret bas sépare les chiffres pour la lecture et ne change rien à la valeur.

0b1100_1010	202
0xCA	202
202	202

Ces trois écritures désignent le même entier : l'hexadécimal est commode pour lire un binaire, mais ce n'est pas un type numérique différent. Dans les *listings* du chapitre, c-à-d. les listes d'instructions accompagnées de leur adresse, les opcodes sont écrits en hexadécimal et les adresses en décimal, sauf mention contraire.

6.2.2 Un binaire Elixir est une suite d'octets

En Elixir, un *binaire* est une suite immuable d'octets, écrite entre doubles chevrons. Chaque nombre placé entre << et >> occupe un octet. La concaténation <> réunit deux binaires et `byte_size/1` en donne la taille.

<code>bytes = <<1, 2>> <> <<3>></code>	<code><<1, 2, 3>></code>
<code>byte_size(bytes)</code>	3

Une chaîne UTF-8 est aussi un binaire. Par ex. "café" contient cinq octets, alors que `String.length("café")`, qui compte les caractères, vaut quatre : le é s'écrit sur deux octets en UTF-8, C3 A9. La fonction `Base.encode16/1` écrit chaque octet en deux chiffres hexadécimaux : `Base.encode16("café")` rend "636166C3A9". La mémoire ne se découpe donc pas en « caractères » par défaut ; c'est le format que nous lisons qui dit comment interpréter les octets.

EN PRATIQUE

Quand tous ses octets forment du texte lisible, un binaire s'affiche comme une chaîne : `<<72, 105>>` s'affiche "Hi". Pour voir les octets eux-mêmes, `:binary.bin_to_list(bytes)` rend leur liste d'entiers et `Base.encode16(bytes)` leur écriture hexadécimale.

6.2.3 Seize bits sur deux octets

Un octet ne va que jusqu'à 255. Pour un numéro de case ou une adresse de saut, notre machine utilise deux octets voisins, soit seize bits : on parle d'un *entier non signé sur seize bits*, « non signé » voulant dire qu'il ne représente aucun nombre négatif.

Les deux octets ne jouent pas le même rôle : l'octet *de poids fort* compte les paquets de 256, l'octet *de poids faible* les unités restantes. La valeur vaut donc poids fort $\times$ 256 + poids faible, de 0 à $255 \times 256 + 255 = 65535$, soit $2^{16} - 1$.

Pour écrire 1000 à la main, on cherche combien de fois 256 y entre : $1000 = 3 \times 256 + 232$. L'octet de poids fort vaut 3, soit 03 en hexadécimal, et l'octet de poids faible 232, soit E8, puisque $232 = 14 \times 16 + 8$. De même, $258 = 1 \times 256 + 2$ donne les octets 01 et 02.

6.2.4 L'ordre des octets fait partie du format

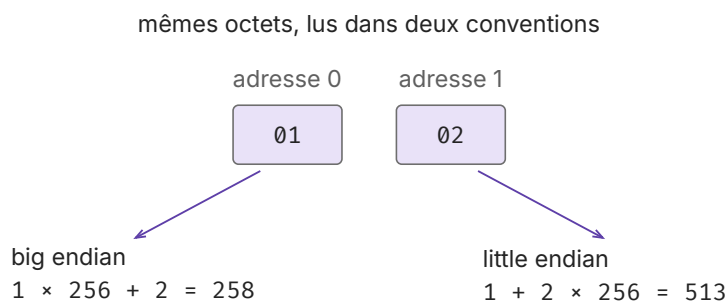
Encore faut-il préciser dans quel ordre ces deux octets sont rangés :

- en *big endian*, l'octet de poids fort vient d'abord, à la plus petite adresse ;
- en *little endian*, l'octet de poids faible vient d'abord.

Les deux conventions existent : les processeurs x86 et ARM de nos ordinateurs rangent à l'usage leurs entiers en little endian, alors que les protocoles réseau, et notre machine, utilisent le big endian.

<pre><<1000::unsigned-big-16>> Base.encode16(<<258::unsigned-big-16>>) Base.encode16(<<258::unsigned-little-16>>)</pre>	<pre><<3, 232>> "0102" "0201"</pre>
---	---

Dans 258::unsigned-big-16, les mots placés après :: décrivent le format du champ : unsigned pour non signé, big pour l'ordre des octets, 16 pour la taille en bits. L'encodage et le décodage doivent employer la même convention ; relus dans l'autre, les mêmes octets donnent un autre nombre :



La position du poids 256 change avec la convention. Le binaire ne porte pas lui-même son choix d'ordre des octets.

```
1 <<as_big::unsigned-big-16>> = <<1, 2>>
2 <<as_little::unsigned-little-16>> = <<1, 2>>
3 {as_big, as_little}
```

VALEUR {258, 513}

Python connaît les mêmes conventions : `int.from_bytes(b"\x01\x02", "big")` vaut aussi 258, et la même lecture avec "little" donne 513.

PIÈGE

Elixir ne vérifie pas qu'un nombre tient dans la taille demandée : il garde seulement les bits de poids faible. Ainsi `<<256>>` vaut `<<0>>`, et `<<65_536::unsigned-big-16>>` vaut `<<0, 0>>`, sans erreur ni avertissement. C'est pourquoi notre encodeur doit refuser lui-même un numéro de case ou une adresse hors de `0..65_535`.

6.2.5 Un flottant occupe huit octets

Un nombre à virgule comme `8.0` n'est pas stocké comme les caractères de son écriture décimale. Le format `Float64` de la norme IEEE 754, celui des `float` de Python et de NumPy, le range sur huit octets, qu'Elixir sait produire et relire pour nous.

```
1 number = 8.0
2 bytes = <<number::float-big-64>>

<<same_number::float-big-64>> = bytes    <<64, 32, 0, 0, 0, 0, 0, 0>>
same_number                               8.0
Base.encode16(bytes)                     "4020000000000000"
```

Les huit octets `40 20 00 00 00 00 00 00`, soit `64, 32` puis six zéros en décimal, sont ceux de `8.0` ; vous les retrouverez tels quels dans les instructions de notre machine. La chaîne `"8.0"`, elle, occupe trois octets UTF-8 : c'est une représentation textuelle différente du même nombre, que notre machine ne lira jamais.

POUR ALLER PLUS LOIN

Les 64 bits d'un `Float64` forment une écriture scientifique en base deux : un bit de signe, onze bits d'exposant, puis cinquante-deux bits de *fraction*. Ainsi $8.0 = 1.0 \times 2^3$: le signe vaut 0, l'exposant est rangé augmenté de 1023, soit $3 + 1023 = 1026$, qui s'écrit `100 0000 0010`, et la fraction est nulle. Les douze premiers bits, `0100 0000 0010`, donnent `402` en hexadécimal, d'où les octets `40 20 00 ...`. Beaucoup de décimaux, comme `0.1`, n'ont pas d'écriture exacte en base deux ; c'est pourquoi `0.1 + 0.2` vaut `0.30000000000000004`, en Elixir comme en Python. Le site [The Floating-Point Guide](https://floating-point-gui.de/)¹⁷ l'explique simplement ; David Goldberg approfondit dans « [What Every Computer Scientist Should Know About Floating-Point Arithmetic](https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html) »¹⁸ (1991).

6.2.6 Un motif binaire relit une structure

Un motif binaire, à gauche d'un `=`, relit les octets avec la notation qui a servi à les écrire. Il peut séparer les champs d'un format et garder les octets restants. Voici un exemple avec un opcode et un entier de seize bits :

¹⁷ <https://floating-point-gui.de/>

¹⁸ https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html

```
1 bytes = <<0x0D, 25::unsigned-big-16, 0>>
2 <<opcode, address::unsigned-big-16,
3   rest::binary>> = bytes
4 {opcode, address, rest}
```

```
VALEUR {13, 25, <<0>>}
```

Lisons ce motif champ par champ :

- `opcode`, sans précision, prend un octet ;
- `address::unsigned-big-16` prend les deux suivants et les lit comme un entier non signé big endian ;
- `rest::binary` prend tout ce qui reste, quelle qu'en soit la longueur.

Les noms sont liés comme dans le pattern matching du chapitre 2, et un motif qui ne correspond pas échoue de la même façon. Notez que le format doit être connu avant de lire : l'octet `0D` ne dit pas à lui seul combien d'octets il faut prendre ensuite.

6.2.7 Un format binaire est un contrat

Nous pouvons maintenant dire ce qu'est un *format binaire* : une convention qui fixe, pour chaque champ, sa position, sa taille, le fait qu'il soit signé ou non, l'ordre de ses octets et le sens de ses valeurs, par ex. que l'opcode `0D` désigne un saut.

Les octets eux-mêmes ne portent aucune de ces informations. Le programme qui écrit et celui qui lit doivent donc s'accorder à l'avance, comme les deux parties d'un contrat. S'ils divergent, la lecture n'échoue pas forcément ; elle peut aussi produire, en silence, des valeurs fausses. Relisons nos trois octets de saut avec la mauvaise convention :

```
1 bytes = <<0x0D, 0, 25>>
```

```
<<_op, target::unsigned-little-16>> = bytes | <<13, 0, 25>>
target                                     | 6400
```

Lus en little endian, les octets `00 19` donnent $25 \times 256 = 6400$: le saut viserait l'adresse 6400, bien au-delà de notre petit programme. Un lecteur robuste traite aussi le binaire tronqué, l'opcode inconnu et l'adresse invalide : `<<0x0D, 0>>` ne fournit qu'un des deux octets de l'adresse, et l'on rend une erreur explicite plutôt que de lancer la machine avec une donnée incomplète.

Le tableau suivant rassemble les écritures de cette section et servira d'aide-mémoire :

Écriture	Sens	Exemple
<code>0b...</code> , <code>0x...</code>	entier écrit en binaire, en hexadécimal	<code>0xCA</code> vaut 202
<code><<a, b>></code>	binaire, un octet par nombre	<code><<1, 2>></code>
<code>a <> b</code>	concaténation de deux binaires	<code><<1>> <> <<2>></code> vaut <code><<1, 2>></code>
<code>byte_size(b)</code>	nombre d'octets	5 pour "café"
<code>n::unsigned-big-16</code>	entier non signé sur 16 bits, poids fort d'abord	258 donne <code><<1, 2>></code>
<code>n::unsigned-little-16</code>	le même, poids faible d'abord	258 donne <code><<2, 1>></code>

Écriture	Sens	Exemple
<code>x::float-big-64</code>	flottant IEEE 754 sur huit octets	<code>8.0</code> donne <code>40 20 00 ...</code>
<code>rest::binary</code>	dans un motif : tous les octets restants	<code><<op, rest::binary>> = b</code>
<code>Base.encode16(b)</code>	les octets en hexadécimal	<code>"0B0102"</code> pour <code><<11, 1, 2>></code>
<code>:binary.bin_to_list(b)</code>	les octets en liste d'entiers	<code>[1, 2]</code> pour <code><<1, 2>></code>

Les guides « [Binaries, strings, and charlists](https://hexdocs.pm/elixir/binaries-strings-and-charlists.html) »¹⁹ d'Elixir et « [Bit Syntax](https://www.erlang.org/doc/system/bit_syntax.html) »²⁰ d'Erlang décrivent les autres options des motifs binaires, comme les entiers signés ou les champs de quelques bits. Cette syntaxe vient d'Erlang, le langage de télécommunications d'Ericsson, où elle sert à décoder des protocoles bit à bit.

6.3 Une machine à pile

6.3.1 Notre machine virtuelle

Notre *machine virtuelle*, ou VM, est un programme Elixir qui joue le rôle d'un processeur : il lit des octets et applique les instructions qu'ils codent, comme le ferait un circuit. C'est une architecture pédagogique, inventée pour ce cours : elle n'exécute ni du code ARM ou x86, ni le bytecode de la BEAM.

Nous avons choisi une *machine à pile*. Au lieu de nommer des registres comme `R0` et `R1`, ses instructions prennent leurs opérandes au sommet d'une *pile*, une liste où l'on n'ajoute et ne retire qu'à un bout, le *sommet*, et elles y déposent leur résultat. C'est la pile de la calculatrice d'AP1, décrite au chapitre 4.

Quatre données décrivent la machine : le code, un binaire immuable, et trois données d'état qui changent à chaque instruction :

- `pc`, l'adresse de la prochaine instruction ;
- `stack`, une pile de valeurs utilisée pour les calculs ;
- `locals`, les *cases locales*, des emplacements numérotés 0, 1, 2... où les liaisons du programme gardent leurs valeurs.

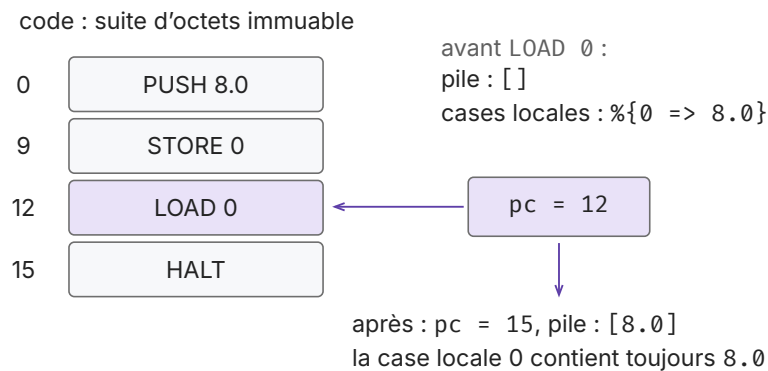
La pile est représentée par une liste Elixir, sommet en tête : empiler `3.0` sur `[8.0]` donne `[3.0, 8.0]`, et dépiler retire cette tête. Les cases locales sont une map, par ex. `%{0 => 8.0}`. Rapprochons ces choix de la machine réelle du début du chapitre :

Machine réelle	Notre VM	En Elixir
programme en mémoire	le code	un binaire immuable
PC	<code>pc</code>	un entier
registres de calcul	la pile de calcul	une liste, sommet en tête
données en mémoire	les cases locales	une map <code>%{0 => 8.0}</code>
cache	aucun	aucune donnée

¹⁹ <https://hexdocs.pm/elixir/binaries-strings-and-charlists.html>

²⁰ https://www.erlang.org/doc/system/bit_syntax.html

Voici l'état complet à un instant donné. Le code commence par `PUSH 8.0`, qui empile `8.0`, puis `STORE 0`, qui le retire de la pile pour le ranger dans la case locale 0 ; `LOAD 0` en remettra une copie sur la pile.

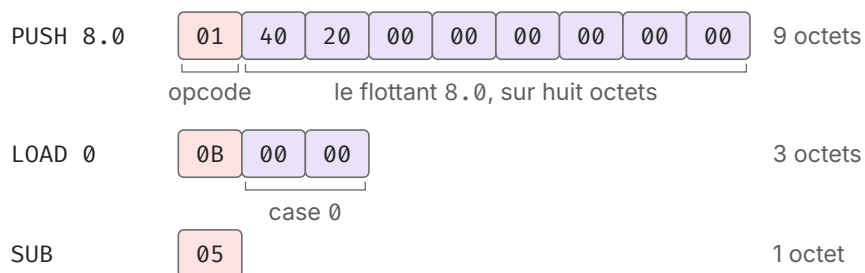


Le PC pointe dans le code ; la case locale 0 conserve une valeur ; la pile reçoit les opérandes du calcul.
 Exécuter `LOAD 0` copie ici la valeur vers la pile et avance le PC de trois octets.

La machine reçoit enfin un *budget de pas*, c-à-d. un nombre maximal d'instructions à exécuter, pour arrêter un programme construit à la main qui bouclerait sans fin. Sans précision, elle accorde dix mille pas, puis rend `{:error, :step_limit}`. Ce budget protège l'expérimentation ; ce n'est pas une preuve que le programme se termine.

6.3.2 Le jeu d'instructions

Une instruction de notre machine est, elle aussi, une suite d'octets. Son premier octet est l'*opcode* (*operation code*), le numéro qui désigne l'opération : c'est celui que la machine lit pour savoir quoi faire. Certaines instructions portent aussi un *opérande*, une donnée qui suit l'opcode dans le binaire.



Trois instructions et leurs octets. L'opcode, en rouge, annonce la taille de ce qui suit : huit octets de flottant pour `PUSH`, deux octets de numéro de case pour `LOAD`, rien pour `SUB`.

Le tableau suivant est le *jeu d'instructions* de notre machine : la liste de ce qu'elle sait faire, avec le codage de chaque instruction. Un programme écrit dans un tel codage pour une machine virtuelle s'appelle du *bytecode*, littéralement du « code en octets ». Les fichiers `.beam` d'Elixir et `.pyc` de Python contiennent ainsi le bytecode de leur propre machine.

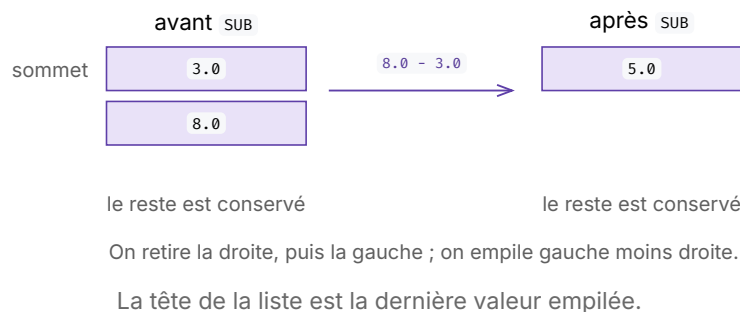
Hexa	Instruction	Octets	Effet
00	HALT	1	Rendre l'unique valeur
01	PUSH nombre	9	Empiler un Float64
02, 03	TRUE, FALSE	1	Empiler un booléen
04 – 07	ADD, SUB, MUL, DIV	1	Combiner deux nombres
08	NEG	1	Opposer le sommet
09, 0A	LT, EQ	1	Comparer deux nombres
0B, 0C	LOAD, STORE	3	Lire ou remplir une case
0D, 0E	JUMP, JUMP_FALSE	3	Choisir le prochain PC

Les opérandes de case et d'adresse sont des entiers non signés sur seize bits, en big endian ; les flottants sont des Float64, également en big endian. Les adresses de saut sont absolues, comptées depuis le début du code, et la taille totale d'un programme est limitée à 65 535 octets.

Les noms lisibles comme `SUB` servent au listing. Dans les données Elixir de la référence, cette opération est l'atome `:minus`, comme dans l'AST, et `TRUE` et `FALSE` sont simplement `true` et `false`. Ne confondez pas le texte de l'instruction, sa représentation Elixir et l'octet 05 écrit dans le binaire.

6.3.3 Une instruction transforme l'état

Chaque instruction a un effet précis sur la pile et les cases. `PUSH 8.0` empile `8.0` ; `PUSH 3.0` empile ensuite `3.0` par-dessus, et la pile vaut `[3.0, 8.0]`. Que fait alors `SUB` ?



La première valeur retirée est celle du dessus, `3.0`, c-à-d. l'opérande de droite ; la seconde est `8.0`, l'opérande de gauche. `SUB` empile donc `8.0 - 3.0`, soit `5.0`. L'addition et la multiplication cacheraient une inversion, puisqu'elles sont commutatives ; la soustraction, la division et la comparaison `<` la révèlent.

Une instruction binaire exige deux valeurs et des types compatibles :

- `SUB` sur une pile vide est une erreur de pile insuffisante ;
- `ADD` sur un booléen et un nombre est une erreur de types.

La machine ne confond pas ces situations avec un calcul réussi : elle rend `{:error, {pc, raison}}`, avec l'adresse de l'instruction fautive. Ainsi le binaire `<<5, 0>>`, c-à-d. `SUB` puis `HALT`, échoue en `{:error, {0, :stack_underflow}}`.

Enfin, `HALT` exige exactement une valeur sur la pile. Une pile vide ne contient pas de résultat, et deux valeurs signalent que le code ne respecte pas le contrat attendu : `HALT` ne « prend pas simplement ce qui traîne au sommet ».

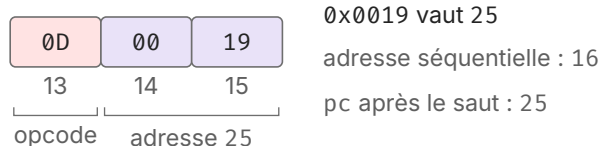
6.3.4 Décoder sans exécuter

La machine sépare deux gestes. *Décodeur* lit l’instruction située à une adresse et calcule l’adresse de l’instruction qui la suit, sans rien exécuter. Lu à l’adresse 0, `PUSH 8.0` se décode ainsi :

```
1 MiniBytecode.decode_at(binary, 0) # {:ok, {:push, 8.0}, 9}
```

Cette réponse décrit une instruction ; elle n’a pas encore empilé le nombre. La même brique sert donc à l’exécution et au *désassemblage*, c-à-d. à la production d’un listing lisible des octets.

JUMP 25 occupe trois octets



Le décodeur connaît l’adresse qui suit ; l’exécution d’un saut peut en choisir une autre.

Quel est le prochain PC si le code contient `JUMP 25` à l’adresse 13 ? Le décodeur rend 16 comme adresse séquentielle ; la machine exécute le saut et choisit 25. Le PC ne vient donc pas d’un simple incrément effectué aveuglément après chaque instruction.

Un binaire ne contient aucun séparateur entre ses instructions, pas plus qu’un message en morse privé de ses silences. C’est l’opcode qui annonce la taille de l’instruction, donc l’endroit où commence la suivante : en partant de l’adresse 0 et en avançant de la taille de chaque instruction, on retrouve toutes les frontières.

Lancé au milieu d’un opérande, en revanche, le décodeur se trompe de lecture. Dans le programme `PUSH 8.0`, `PUSH 3.0`, `SUB`, `HALT`, l’octet d’adresse 10 est le 40 du flottant 3.0, soit 64 : la référence répond `{:error, {:unknown_opcode, 10, 64}}`.

POUR ALLER PLUS LOIN

Un code dont aucun mot complet n’est le début d’un autre s’appelle un *code préfixe*. Notre jeu d’instructions en est un, puisque l’opcode fixe la longueur de l’instruction, et un tel code se décode sans ambiguïté ni retour en arrière, à condition de partir du début. UTF-8 suit ce principe : les premiers bits du premier octet d’un caractère disent s’il en occupe un, deux, trois ou quatre ; la norme est décrite dans la [RFC 3629](https://www.rfc-editor.org/rfc/rfc3629)²¹.

6.3.5 La boucle de la machine

La VM réalise le cycle lire, décodeur, exécuter du début du chapitre. Son état, le PC, la pile et les cases, passe d’un appel récursif au suivant, comme la boucle de la calculatrice transmettait sa pile. À chaque tour, elle vérifie son budget, décode l’instruction située au PC, applique son effet, puis recommence, sauf après `HALT` ou une erreur.

Les réponses citées dans ce chapitre sont celles de la *référence*, une implémentation complète de la machine et de son compilateur. Sa fonction `MiniVM.run/2` reçoit un binaire et un

²¹ <https://www.rfc-editor.org/rfc/rfc3629>

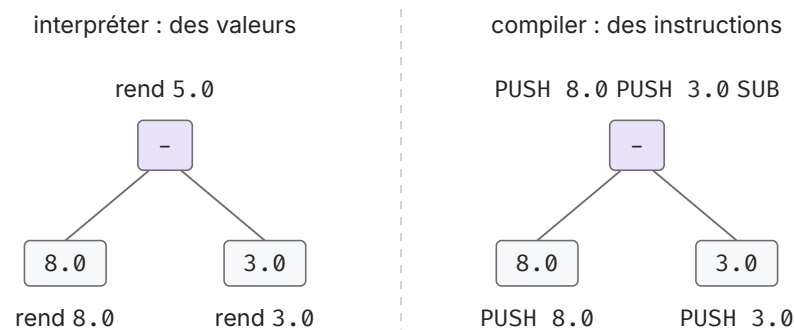
budget, et rend `{:ok, valeur}` ou `{:error, raison}`. Sa voisine `MiniVM.trace/2` rend en plus la *trace* de l'exécution : la liste des états de la machine relevés *avant* chaque instruction, chacun avec son PC, sa pile et ses cases.

6.4 Compiler un arbre en instructions

6.4.1 Le même arbre, une autre question

Repartons de l'arbre de $8.0 - 3.0$: un nœud de soustraction et deux feuilles. L'interpréteur demande à chaque nœud sa *valeur* : la feuille gauche répond 8.0 , la feuille droite 3.0 , et la racine combine ces réponses en 5.0 .

Le compilateur pose une autre question au même arbre : quelles *instructions* laisseront ta valeur sur la pile ? La feuille gauche répond `PUSH 8.0`, la feuille droite `PUSH 3.0`, et la racine met bout à bout les réponses de ses enfants avant d'ajouter `SUB`.



Le même arbre, parcouru de la même façon récursive. Seule la réponse de chaque nœud change : une valeur pour l'interpréteur, une liste d'instructions pour le compilateur.

Les deux parcours ont la même forme ; leur différence tient au moment où le calcul a lieu. L'interpréteur calcule pendant le parcours, et refait ce parcours à chaque exécution. Le compilateur ne calcule rien : il produit une fois pour toutes une liste d'instructions, et c'est la machine qui fera la soustraction plus tard, sans l'arbre.

Notre VM est d'ailleurs elle-même un interpréteur, mais un interpréteur d'octets. Python fonctionne ainsi : il compile votre fichier en bytecode, que sa machine virtuelle exécute ensuite.

6.4.2 Les enfants précèdent l'opérateur

Pour traduire $8.0 - 3.0$, il faut d'abord rendre ses deux valeurs disponibles. Le compilateur parcourt le fils gauche, puis le fils droit, puis produit l'instruction de soustraction. C'est le *postordre* du chapitre 5 : l'opération vient après les instructions de ses enfants, et le programme complet s'écrit `PUSH 8.0, PUSH 3.0, SUB, HALT`.

Cet ordre est celui de l'écriture postfixe de la calculatrice du chapitre 4. L'arbre de $(8.0 - 3.0) * 2.0$, parcouru en postordre, donne `8.0 3.0 - 2.0 *`, exactement la forme de l'entrée `"8 3 - 2 *"` que la calculatrice évaluait avec une pile. Le compilateur réécrit donc l'arbre en notation postfixe, et la VM rejoue la calculatrice sur des octets.

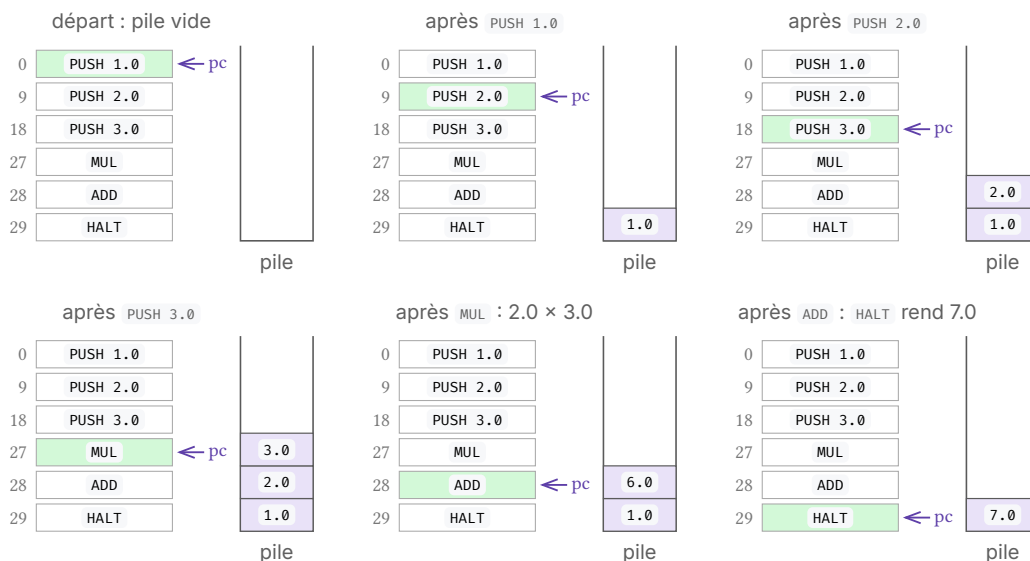
Pourquoi une pile suffit-elle ? Lorsque le postordre arrive à un opérateur, ses deux opérandes viennent d'être calculés, le gauche puis le droit, et toute valeur intermédiaire

calculée entre-temps a déjà été consommée. Ce sont donc les deux dernières valeurs produites et pas encore utilisées : exactement celles qui se trouvent au sommet de la pile.

La machine n'a ainsi jamais à chercher une valeur ailleurs qu'au sommet, et le compilateur n'a jamais à nommer un emplacement. Un compilateur pour une machine à registres doit au contraire choisir, pour chaque résultat intermédiaire, un registre libre.

6.4.3 Suivre la machine pas à pas

Pour $1.0 + 2.0 * 3.0$, le postordre donne `PUSH 1.0`, `PUSH 2.0`, `PUSH 3.0`, `MUL`, puis `ADD` : le produit, enfant droit de l'addition, est calculé en entier avant elle. Suivez le PC et la pile ; les adresses sautent de neuf en neuf tant que la machine lit des `PUSH`.



Les six états de la machine sur le code de $1.0 + 2.0 * 3.0$, le sommet de la pile en haut. Le PC désigne la prochaine instruction ; 1.0 reste au fond pendant tout le calcul du produit.

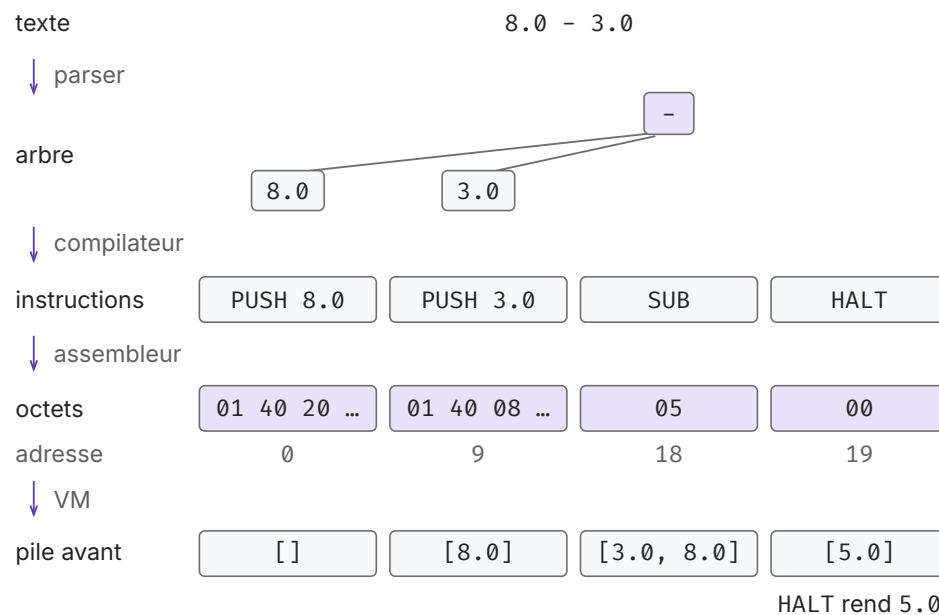
Après `MUL`, la pile vaut $[6.0, 1.0]$: la première valeur attend toujours sous le résultat du sous-arbre droit, et `ADD` les réunit. Ce n'est pas un hasard, mais un *invariant* qui relie compilateur et machine.

PROPRIÉTÉ · UNE EXPRESSION, UNE VALEUR DE PLUS

Le code produit pour une expression ajoute exactement une valeur au sommet de la pile et ne touche pas à ce qui se trouvait dessous. Pourquoi c'est vrai : pour un nombre, `PUSH` empile une valeur ; pour une opération binaire, le code du fils gauche en ajoute une, celui du fils droit une autre par-dessus, puis l'instruction retire ces deux valeurs et en empile une seule, soit une de plus au total, et à aucun moment le fond de la pile n'a été lu. Cette récurrence sur l'arbre est ce qui permet à `HALT` d'exiger exactement une valeur, et au compilateur de n'avoir aucun registre à réserver.

6.4.4 Du texte aux octets

Rassemblons maintenant tout le trajet de $8.0 - 3.0$, du texte jusqu'à la valeur. Chaque colonne de la figure suit une instruction à travers les représentations successives.



Une colonne par instruction : le même calcul sous forme d'arbre, d'instructions, d'octets, puis d'états de la pile. La racine de l'arbre est placée au-dessus de `SUB`, car le postordre émet l'opération après ses deux enfants.

Voici les octets exacts de ce programme, les adresses en décimal et les octets en hexadécimal :

Adresse	Instruction	Octets
0	PUSH 8.0	01 40 20 00 00 00 00 00 00
9	PUSH 3.0	01 40 08 00 00 00 00 00 00
18	SUB	05
19	HALT	00

Chaque `PUSH` commence par son opcode `01`, suivi des huit octets du flottant. Le programme occupe vingt octets. L'adresse de `SUB` n'est pas `2`, même s'il s'agit de la troisième instruction : le décodeur avance de neuf octets à chacun des deux `PUSH`, puis de un pour `SUB`.

La dernière rangée de la figure est la trace : la pile avant chaque instruction, aux PC `0`, `9`, `18` et `19`. `HALT` trouve bien une seule valeur et rend `5.0`.

Ces quatre représentations sont aussi le meilleur outil de diagnostic. Un résultat faux ne dit pas encore quelle étape est fautive : une adresse incorrecte se voit dans le listing, un ordre de dépilement incorrect dans la trace.

POUR ALLER PLUS LOIN

La machine virtuelle de Python est elle aussi une machine à pile. Sur `dis.dis("a - b")`, Python 3.13 affiche notamment `LOAD_NAME` pour `a`, `LOAD_NAME` pour `b`, puis `BINARY_OP` avec la mention `(-)` : empiler, empiler, soustraire, cf. la [documentation du module `dis`](#)²². La machine virtuelle Java et WebAssembly empilent de même ; la BEAM, au contraire, travaille sur des registres, comme le décrit *The BEAM Book*²³ d'Erik Stenman. La partie « A Bytecode Virtual Machine » de *Crafting Interpreters*²⁴ de Robert Nystrom construit en entier une machine à pile et son compilateur.

6.5 Variables et cases

6.5.1 Des noms à la compilation, des cases à l'exécution

Le programme `let x = 8.0 x - 3.0` possède une liaison nommée `x`. Le compilateur associe ce nom à une case locale, par ex. la case `0`. Les instructions n'ont ensuite plus besoin de transporter la chaîne `"x"` : le code devient `PUSH 8.0, STORE 0, LOAD 0, PUSH 3.0, SUB, HALT`. `STORE 0` retire le sommet de la pile et range cette valeur dans la case `0`. `LOAD 0` lit cette case et empile une copie de la valeur ; la case reste initialisée. Ce ne sont donc pas deux noms pour la même opération.

On retrouve le rangement et le chargement de la machine à registres du début du chapitre, la pile tenant lieu de registres : `STORE` range un résultat dans une case, `LOAD` en recopie le contenu vers la zone de calcul.

PC	Instruction	Pile avant	Cases avant
0	<code>PUSH 8.0</code>	<code>[]</code>	<code>%{ }</code>
9	<code>STORE 0</code>	<code>[8.0]</code>	<code>%{ }</code>
12	<code>LOAD 0</code>	<code>[]</code>	<code>%{0 => 8.0}</code>
15	<code>PUSH 3.0</code>	<code>[8.0]</code>	<code>%{0 => 8.0}</code>
24	<code>SUB</code>	<code>[3.0, 8.0]</code>	<code>%{0 => 8.0}</code>
25	<code>HALT</code>	<code>[5.0]</code>	<code>%{0 => 8.0}</code>

Le programme occupe vingt-six octets. Distinguez le nom source `x`, le numéro de case `0` et sa valeur `8.0`. Deux tables interviennent, à deux moments différents :

- pendant la compilation, la *table des noms* associe un nom à un numéro de case, `%{"x" => 0}` ; elle disparaît avec le compilateur ;
- pendant l'exécution, les cases de la VM associent un numéro à une valeur, `%{0 => 8.0}`.

Entre les deux, le binaire ne contient que des numéros.

6.5.2 Un bloc reçoit des cases fraîches

Reprenons le bloc du chapitre 5, dont le résultat vaut `12.0` :

²² <https://docs.python.org/3/library/dis.html>

²³ <https://blog.stenmans.org/theBeamBook/>

²⁴ <https://craftinginterpreters.com/a-bytecode-virtual-machine.html>

```

1 let x = 2.0
2 let y = { let x = 9.0 x + 1.0 }
3 x + y

```

Le compilateur transporte deux choses pendant son parcours : une table des noms, par ex. `%{"x" => 0}`, et un compteur de cases qui ne recule jamais. En entrant dans le bloc, le `x` intérieur reçoit une case fraîche, la 1, et la table devient `%{"x" => 1}`.

En sortant, le compilateur reprend la table d'avant le bloc : `x` désigne de nouveau la case 0, tandis que `y` reçoit la case 2. Le listing produit est :

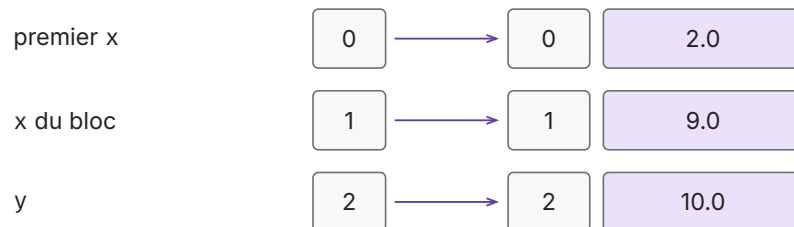
```

1 PUSH 2.0   STORE 0
2 PUSH 9.0   STORE 1   LOAD 1   PUSH 1.0   ADD   STORE 2
3 LOAD 0     LOAD 2     ADD     HALT

```

compilation : noms → numéros

exécution : cases → valeurs



À la sortie du bloc, `x` retrouve la case 0 ; les cases 1 et 2 restent allouées.

La portée décide quel numéro désigne `x`. L'allocation donne une case fraîche à chaque liaison, même après la sortie d'un bloc.

À `HALT`, les cases valent `%{0 => 2.0, 1 => 9.0, 2 => 10.0}` et le résultat est `12.0`. La case 1 existe toujours ; simplement, plus aucun nom ne la désigne.

PIÈGE

La présence d'une case en mémoire ne rend pas son nom visible partout. La portée est une règle du langage, vérifiée pendant la compilation ; la map de valeurs de la VM n'a pas à retrouver toute seule les règles lexicales.

POUR ALLER PLUS LOIN

Notre compilateur ne réutilise jamais une case. Un compilateur réel cherche au contraire, en chaque point du programme, quelles variables seront encore lues, leur *vivacité* ; deux variables qui ne sont jamais vivantes en même temps peuvent partager un emplacement. Sur une machine à registres, il répartit ainsi les variables entre les quelques registres disponibles, si bien qu'une variable du programme source peut occuper un registre, une case mémoire, ou ne plus exister du tout après optimisation. Andrew Appel traite ces étapes dans *Modern Compiler Implementation*.

6.6 Sauts et contrôle

6.6.1 Un saut donne une autre valeur au PC

Jusqu'ici, le PC passait toujours à l'instruction suivante. Un *saut* est une instruction qui lui donne une autre valeur. Notre machine en connaît deux :

- `JUMP adresse` est un saut *inconditionnel* : il donne toujours au PC la valeur de son opérande ;
- `JUMP_FALSE adresse` est un saut *conditionnel* : il retire du sommet de la pile un booléen, saute à l'adresse si ce booléen vaut `false`, et laisse sinon la machine continuer à l'instruction suivante.

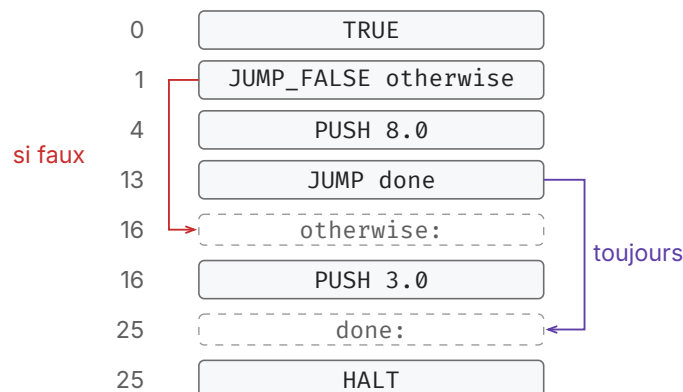
Dans les deux cas, l'adresse est *absolue* : `JUMP 25` désigne l'octet 25 du programme, compté depuis le début, et non « vingt-cinq octets plus loin ».

6.6.2 Un `case` devient deux sauts

Un `case` ne doit effectuer qu'un des deux calculs. Voici le programme que nous allons traduire :

```
1 case True {
2   True -> 8.0
3   False -> 3.0
4 }
```

Après avoir calculé la condition, `JUMP_FALSE` la retire de la pile. Si elle est fausse, la machine saute au début de la branche `False` ; si elle est vraie, elle continue immédiatement dans la branche `True`. À la fin de cette branche, un second saut évite d'exécuter aussi la branche fausse, qui suit dans le binaire.



Le code du `case`, une ligne par élément, avec les adresses que l'assembleur calculera. Les étiquettes, en pointillés, n'occupent aucun octet. Le saut conditionnel, à gauche, n'est pris que si la condition est fausse ; le saut incondiionnel, à droite, l'est toujours.

Les *étiquettes* `otherwise` et `done` nomment des positions ; ce ne sont pas des instructions exécutées. Chaque branche rejoint `done` avec une valeur de résultat. La condition, elle, a déjà été retirée : l'invariant de pile reste donc valable de part et d'autre du choix.

6.6.3 L'assembleur résout les étiquettes en deux passes

Lorsque le compilateur écrit `JUMP_FALSE otherwise`, il ne connaît pas encore l'adresse de `otherwise` : la branche vraie, qui les sépare, n'a pas encore été produite. C'est un saut *en avant*, vers une position qui n'existe pas encore.

Plutôt que de recalculer à la main la longueur de toutes les branches, le compilateur produit des étiquettes symboliques, puis demande à un *assembleur* de les résoudre, c-à-d. de remplacer chaque nom par une adresse et d'encoder le tout en octets. Le souci est aussi vieux que la programmation : sur l'EDSAC, en 1949, David Wheeler écrivait déjà un tel traducteur, l'un des tout premiers assembleurs.

La *première passe* parcourt la liste symbolique en cumulant les tailles des instructions ; une étiquette a une taille nulle et reçoit l'adresse courante. La *seconde passe* remplace les étiquettes des sauts par ces adresses, retire les déclarations d'étiquette et encode les instructions :

Élément	Adresse	Taille	Après la passe 2
TRUE	0	1	02
JUMP_FALSE otherwise	1	3	JUMP_FALSE 16, soit 0E 00 10
PUSH 8.0	4	9	01 40 20 ...
JUMP done	13	3	JUMP 25, soit 0D 00 19
otherwise:	16	0	retirée, otherwise vaut 16
PUSH 3.0	16	9	01 40 08 ...
done:	25	0	retirée, done vaut 25
HALT	25	1	00

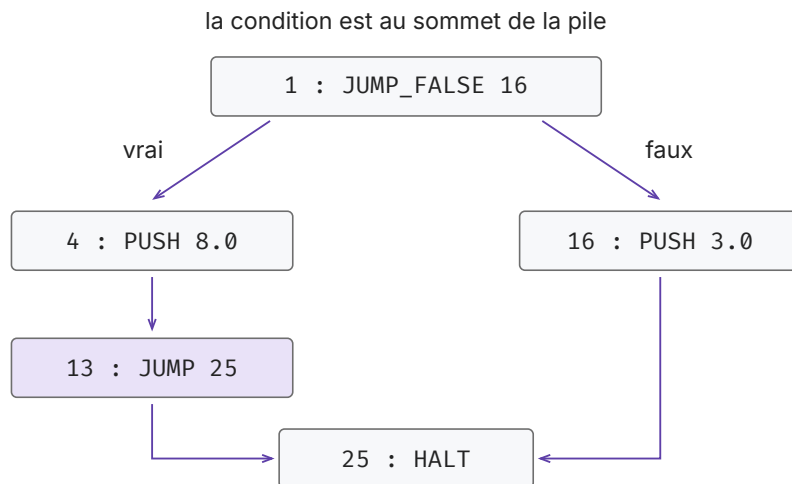
Refaisons la première passe à la main. On part de l'adresse 0 ; `TRUE` occupe un octet, donc l'élément suivant est en $0 + 1 = 1$; `JUMP_FALSE` en occupe trois, d'où $1 + 3 = 4$; `PUSH 8.0` en occupe neuf, d'où $4 + 9 = 13$; `JUMP` en occupe trois, d'où $13 + 3 = 16$.

L'étiquette `otherwise` arrive à cet instant : elle vaut 16, et l'adresse courante ne bouge pas. `PUSH 3.0` mène à $16 + 9 = 25$, où `done` reçoit la valeur 25. Enfin, `HALT` mène à $25 + 1 = 26$, la taille totale du programme.

Dans la seconde passe, `JUMP_FALSE 16` s'encode avec l'opcode 0E, puis 16 sur deux octets big endian, 00 10, puisque $16 = 1 \times 16 + 0$; de même, `JUMP 25` s'encode 0D 00 19, car $25 = 1 \times 16 + 9$. Pourquoi deux passes ? Parce qu'une étiquette peut être utilisée avant d'être définie : il faut avoir tout mesuré une première fois pour connaître toutes les adresses, puis repasser pour les écrire.

6.6.4 Deux chemins dans le même binaire

Avec la condition vraie, les PC visités sont 0, 1, 4, 13, 25. Avec une condition fausse, ils deviennent 0, 1, 16, 25. Une adresse qui existe dans le listing n'est donc pas nécessairement visitée pendant une exécution particulière.



Les deux chemins retirent la condition et produisent chacun une valeur. Le saut à l'adresse 13 permet au chemin vrai de rejoindre `HALT` sans passer par `PUSH 3.0`.

Quelle est la pile juste avant `HALT` ? `[8.0]` sur le premier chemin et `[3.0]` sur le second. Si l'on retire le `JUMP done`, le chemin vrai empile aussi `3.0` et arrive avec deux résultats : le contrôle final de `HALT` révèle le souci.

Un `case` à l'intérieur d'un autre créerait des collisions si toutes les branches partageaient les étiquettes `otherwise` et `done` : le compilateur numérote donc ses étiquettes, par ex. `{:else, 3}`. Une étiquette dupliquée ou inconnue est une erreur d'assemblage, et un saut vers le milieu d'un opérande est refusé par la VM, qui vérifie que chaque cible est un début d'instruction.

6.6.5 Sauter en arrière

Un saut peut aussi viser une adresse *antérieure* : la machine repasse alors par des instructions déjà exécutées. C'est ainsi que se compilent les boucles des langages qui en ont. Notre langage n'en a pas, mais un binaire écrit à la main comme `JUMP 0` boucle indéfiniment, jusqu'à épuiser le budget de pas.

À RETENIR

Le compilateur choisit les opérations et le contrôle à partir de l'arbre. L'assembleur résout les positions et produit les octets. La VM lit ces octets et transforme son état. Chaque brique a un résultat que l'on peut examiner avant de passer à la suivante.

POUR ALLER PLUS LOIN

Il ne faut guère plus que des sauts pour tout calculer. Marvin Minsky a montré dans les années 1960 qu’une machine munie de deux compteurs, qui sait ajouter un, retrancher un, tester zéro et sauter, calcule tout ce que calcule une machine de Turing : elle est *Turing-complète*. La contrepartie est célèbre : aucun programme ne peut décider, pour tout binaire, s’il finira par s’arrêter. C’est le [problème de l’arrêt](#)²⁵, dont Alan Turing a établi l’indécidabilité en 1936 ; notre budget de pas en est la réponse pratique. La page [Machine à compteurs](#)²⁶ donne la construction.

6.7 Quand refuser un programme ?

6.7.1 Chaque brique a ses refus

Le décodeur rejette une instruction tronquée ou un opcode inconnu, l’assembleur une étiquette incorrecte, le compilateur un nom absent de son environnement, et la VM une pile trop courte, une case non initialisée, une division par zéro ou un `HALT` avec le mauvais nombre de valeurs. Quelques essais courts permettent de les distinguer :

Essai	Réponse de la référence
<code><<1, 0>></code>	<code>{:error, {:invalid_float_operand, 0}}</code> : le flottant de <code>PUSH</code> est incomplet
<code><<0xFF>></code>	<code>{:error, {:unknown_opcode, 0, 255}}</code> : aucun opcode ne vaut <code>FF</code>
<code><<5, 0>></code> , soit <code>SUB</code> puis <code>HALT</code>	<code>{:error, {0, :stack_underflow}}</code> : pile insuffisante dès <code>SUB</code>
<code><<0x0D, 0, 0>></code> , soit <code>JUMP 0</code>	<code>{:error, :step_limit}</code> : le budget de pas finit par être épuisé

Le dernier essai est un binaire construit à la main : son format est valide, et pourtant il ne termine pas. Un budget épuisé dit seulement que l’on a arrêté l’observation.

6.7.2 Avant ou pendant l’exécution

On appelle *statique* un contrôle fait avant l’exécution, sur le texte ou l’arbre du programme, et *dynamique* un contrôle fait pendant l’exécution, sur les valeurs réellement calculées.

Notre compilateur fait déjà un contrôle statique : il visite toutes les branches et refuse un nom inconnu, même dans une branche qui ne s’exécutera pas. L’interpréteur du chapitre 5, lui, n’explore jamais une branche inactive ; pour comparer les deux moteurs, choisissez donc des programmes dont tous les noms se résolvent partout.

Les types, en revanche, ne sont vérifiés que dynamiquement, par la VM. Ainsi `True + 1.0` se compile sans difficulté, et c’est la machine qui échoue, en `{:error, {10, :type_mismatch}}` : `10` est l’adresse de l’`ADD`, après le `TRUE` d’un octet et le `PUSH` de neuf. Quant à

²⁵ https://fr.wikipedia.org/wiki/Probl%C3%A8me_de_l%27arr%C3%AAt

²⁶ https://fr.wikipedia.org/wiki/Machine_%C3%A0_compteurs

`case True { True -> 1.0 False -> False }`, il rend `1.0`, car la branche fausse n'est jamais exécutée.

Le typeur du chapitre 5 refuserait ces deux programmes avant même de produire un octet, et laisserait passer `1.0 / 0.0`, qui a bien le type `Float` mais échoue pendant le calcul : un typeur écarte certaines erreurs, pas toutes. C'est le point de départ du sujet « Inférence de types » proposé en AP9.

6.8 Appels de fonction : un prolongement

6.8.1 Une seconde pile pour les retours

Notre compilateur refuse les nœuds de fonction et d'appel. Un prolongement facultatif accepte les fonctions *fermées*, sans variable libre. Il ajoute trois instructions :

- `PUSH_FN` adresse arité, opcode `0F`, quatre octets, empile la valeur fonction `{:fn, adresse, arité}` ;
- `CALL n`, opcode `10`, deux octets, appelle une fonction avec `n` arguments ;
- `RETURN`, opcode `11`, un octet, revient chez l'appelant avec le résultat.

Un appel doit mémoriser où reprendre après la fonction, ainsi que les cases de l'appelant : c'est le *cadre d'appel*, `{adresse_de_retour, cases}`, rangé sur une seconde pile. L'appelant dépose sur la pile de calcul la valeur fonction, puis les arguments.

`CALL` retire la valeur fonction placée sous les arguments, vérifie leur nombre, empile un cadre, repart avec des cases vides et saute à l'entrée de la fonction. Le *prologue* de la fonction range alors les arguments dans ses cases. `RETURN` laisse le résultat sur la pile, rétablit les cases de l'appelant et reprend à l'adresse de retour.

Il faut donc distinguer la pile de *calcul*, qui porte les opérandes d'une addition, de la pile des *appels*, qui garde les retours en attente : deux listes Elixir, jamais la même donnée. Le reste de la pile de calcul n'a pas bougé pendant l'appel, exactement comme pour toute expression : c'est l'invariant « une expression, une valeur de plus » qui protège les valeurs de l'appelant.

6.8.2 Suivre un appel

Prenons une fonction fermée :

```
1 let inc = fn(x) { x + 1.0 }
2 inc(4.0)
```

L'extension la compile en quarante-deux octets et rend `5.0`. Le corps est placé dans le même binaire, derrière un saut qui l'évite quand on crée la valeur fonction, car créer la fonction n'exécute toujours pas son corps :

Adresse	Instruction	Rôle
0	JUMP 20	enjamber le corps
3	STORE 0	prologue : ranger l'argument dans la case 0
6, 9, 18	LOAD 0, PUSH 1.0, ADD	le corps $x + 1.0$
19	RETURN	laisser le résultat, rétablir l'appelant
20	PUSH_FN 3 1	la valeur fonction : entrée 3, arité 1
24, 27	STORE 1, LOAD 1	let inc, puis lire inc
30, 39	PUSH 4.0, CALL 1	l'argument, puis l'appel
41	HALT	résultat : 5.0

Les PC visités sont 0, 20, 24, 27, 30, 39, puis 3, 6, 9, 18, 19 dans le corps, et enfin 41. Voici l'état de la machine autour de l'appel, la valeur fonction étant abrégée en `inc` :

PC	Instruction	Pile avant	Cases avant	Cadres avant
39	CALL 1	[4.0, inc]	{1 => inc}	aucun
3	STORE 0	[4.0]	{}	{41, {1 => inc}}
19	RETURN	[5.0]	{0 => 4.0}	{41, {1 => inc}}
41	HALT	[5.0]	{1 => inc}	aucun

Pendant l'appel, le cadre garde l'adresse de retour 41 et les cases de l'appelant ; RETURN les rétablit. Pour une fonction à deux paramètres $fn(a, b)$, le prologue range les arguments dans l'ordre inverse, STORE 1 puis STORE 0, puisque le dernier argument empilé est au sommet.

Une fonction qui lit un `offset` extérieur exige une capture et reste refusée à ce palier, avec `{:undefined_name, "offset"}` : compiler les fermetures demanderait de ranger aussi le contexte capturé.

6.8.3 Vers une autre machine : CHIP-8

Notre compilateur produit des instructions pour notre VM ; c'est son *backend*. Le projet backend CHIP-8 d'AP9 fait écrire un second backend, pour une petite machine des années 1970 dont l'émulateur est fourni.

Son palier avancé revient sur la récursion terminale du chapitre 4 : après l'appel récursif, il ne reste rien à faire dans la fonction appelante. Plutôt que d'empiler à chaque tour une adresse de retour inutile, un *auto-appel terminal* prépare les nouveaux paramètres, puis saute à l'entrée de la fonction avec JP : la pile de retours ne grandit plus. Encore faut-il calculer tous les nouveaux arguments avant de les ranger : pour un appel qui échange `a` et `b`, ranger `b` dans `a` en premier détruirait l'ancien `a`.

6.9 Vérifier sa compréhension

Commencez par les octets. Que valent `0x2A` et `0b0010_1010` en décimal ? Comment s'écrit l'entier 1000 sur deux octets big endian ?

Le premier vaut $2 \times 16 + 10 = 42$, et le second s'écrit avec les mêmes bits, puisque `0010` correspond au chiffre 2 et `1010` au chiffre A. Pour 1000, on cherche combien de fois 256 y

entre : $1000 = 3 \times 256 + 232$, et $232 = 14 \times 16 + 8$ s'écrit E8 ; l'octet de poids fort vaut donc 03 et l'octet de poids faible E8 .

<code>0x2A</code>	42
<code>0b0010_1010</code>	42
<code>Base.encode16(<<1000::unsigned-big-16>>)</code>	"03E8"

Choisissez ensuite une soustraction, écrivez son arbre, son listing et les états de pile jusqu'à `HALT`. Faites-le d'abord sur $2.0 * 3.0 - 1.0$, avant de lire la suite.

Le compilateur produit `PUSH 2.0`, `PUSH 3.0`, `MUL`, `PUSH 1.0`, `SUB`, `HALT`, aux adresses 0, 9, 18, 19, 28 et 29 : trente octets. La pile avant chaque instruction vaut [], [2.0], [3.0, 2.0], [6.0], [1.0, 6.0] puis [5.0], et `HALT` rend 5.0. Le `MUL` vient avant le `PUSH 1.0` parce que le produit est le fils gauche de la soustraction.

Que se passe-t-il, dans le `case` compilé plus haut, si l'on remplace `JUMP 25` par `JUMP 16` ? Le chemin vrai, après avoir empilé 8.0, saute sur `PUSH 3.0` et arrive à `HALT` avec [3.0, 8.0] : deux valeurs, donc une erreur, exactement comme si le saut manquait.

Pourquoi `LOAD` doit-il laisser la case intacte ? Compilez `let x = 2.0 x + x : STORE 0 range 2.0`, puis deux `LOAD 0` empilent chacun une copie, et `ADD` rend 4.0. Un `LOAD` qui viderait la case ferait échouer le second sur une case non initialisée.

Enfin, refaites à la main les deux passes de l'assembleur pour ce programme, puis comparez avec la suite :

```
1 case 1.0 < 2.0 {
2   True -> 8.0
3   False -> 3.0
4 }
```

La condition occupe maintenant `PUSH 1.0`, `PUSH 2.0` et `LT`, soit dix-neuf octets au lieu d'un seul pour `TRUE` : tout le reste est décalé de dix-huit. `JUMP_FALSE` est en 19, `PUSH 8.0` en 22, `JUMP` en 31, `otherwise` et `PUSH 3.0` en 34, `done` et `HALT` en 43, pour quarante-quatre octets en tout.

Les deux sauts s'encodent `0E 00 22` et `0D 00 2B`, puisque $34 = 2 \times 16 + 2$ et $43 = 2 \times 16 + 11$. Sur le chemin vrai, les PC visités sont 0, 9, 18, 19, 22, 31 et 43, et la pile vaut [true] juste avant le `JUMP_FALSE`, puis [] juste après.

POUR EN PARLER

`0xCA`, `0b1100_1010` et `202` sont le même octet : C vaut 12 et A vaut 10, d'où $12 \times 16 + 10$. Sur seize bits big endian, 258 s'écrit `01 02`, poids fort d'abord ; relu en little endian, le même binaire vaudrait 513.

Une adresse se compte en octets, pas en instructions : dans `PUSH 8.0`, `PUSH 3.0`, `SUB`, `HALT`, le `SUB` est à l'adresse 18, et le programme en occupe vingt.

Le sommet de la pile est l'opérande de droite : sur `[3.0, 8.0]`, `SUB` empile $8.0 - 3.0$, et c'est la soustraction, pas l'addition, qui révèle une inversion.

`STORE` retire le sommet et remplit une case ; `LOAD` empile une copie et laisse la case :
let `x = 2.0` `x + x` vaut 4.0.

Un `case` compilé a deux sauts : `JUMP_FALSE` retire la condition et évite la branche vraie, `JUMP` évite ensuite la branche fausse ; sans le second, `HALT` trouve deux valeurs.

Le PC contient l'adresse de la prochaine instruction : le décodeur rend l'adresse séquentielle, 16 après le `JUMP 25` lu en 13, mais c'est l'exécution qui choisit 25.

Calculer et apprendre

SÉANCES AP4 ET AP5 · CHAPITRES 7 ET 8

Comment un programme apprend-il à partir de données ?

Jusqu'ici, nous écrivions nous-mêmes les règles d'un calcul. Dans cette partie, nous laissons des exemples les ajuster : c'est l'apprentissage automatique, que nous abordons avec Nx, la bibliothèque de calcul numérique d'Elixir, puis avec Axon, qui assemble des réseaux de neurones.

Il nous faudra pour cela quelques outils : des vecteurs et des matrices pour ranger les exemples, une perte pour mesurer l'erreur, des dérivées et un gradient pour savoir dans quel sens corriger. Le chapitre 7, pour AP4, range les exemples et décrit les calculs ; le chapitre 8, pour AP5, fait apprendre les paramètres. Tous deux construisent ces outils pas à pas, avec des calculs faits à la main sur de petites valeurs avant d'être confiés à la machine. Prenez le temps de refaire ces calculs : ce sont eux qui donnent un sens aux courbes que vous verrez dans les carnets.

Des tenseurs et des graphes

SÉANCE AP4 · DONNÉES, TENSEURS ET GRAPHE DE CALCUL

Jusqu'ici, nous avons écrit nous-mêmes toutes les règles de nos programmes. Pour calculer le prix d'une course de taxi, par ex., il suffit de connaître le tarif, disons 5 € de prise en charge et 2 € par kilomètre, et d'écrire $\text{prix} = 5 + 2 \times \text{km}$. Mais comment faire si nous ignorons ce tarif et que nous disposons seulement de quelques tickets de courses passées ?

En *apprentissage supervisé*, nous changeons de méthode. Nous choisissons d'abord une formule dont certains nombres restent libres, les *paramètres* : ici $\text{prix} \approx w \times \text{km} + b$, avec deux paramètres w et b . Nous rassemblons ensuite des *exemples* dont la réponse est connue : un ticket indique 2 km pour 9 €, un autre 5 km pour 15 €, un troisième 10 km pour 25 €. Partons d'un essai quelconque, $w = 1$ et $b = 4$: la formule prédit 6 €, 9 € et 14 €, toujours trop peu, et d'autant plus que la course est longue. Une *mesure d'erreur* résume ces écarts par un seul nombre, et un algorithme d'*ajustement* modifie w et b pour faire baisser cette mesure, des milliers de fois au besoin, en suivant une règle précise. À la fin, w vaut à peu près 2 et b à peu près 5 : le tarif a été retrouvé à partir des seuls tickets.

Qu'est-ce qui a changé par rapport à un programme écrit à la main ? Le code que nous écrivons ne contient plus la règle. Il contient la famille de formules, la mesure d'erreur et la manière d'ajuster ; la règle, elle, se trouve dans les valeurs des paramètres, et elle ne vaut que ce que valent les exemples. Pour un tarif de taxi, la méthode n'a guère d'intérêt, puisque nous savons écrire la formule. Elle devient précieuse quand personne ne sait écrire la règle, par ex. pour reconnaître un vêtement sur une photo, comme dans le prolongement Fashion-MNIST d'AP5.

Ce chapitre et le suivant tiennent en quatre idées, que nous retrouverons dans chaque projet d'AP9 : une *représentation* numérique des exemples, ici des kilomètres et des euros ; un *modèle*, c'est-à-dire la formule à paramètres ; une *perte*, le nom de la mesure d'erreur, qui dit à quel point les prédictions sont fausses ; et une *mise à jour*, qui modifie les paramètres pour réduire cette perte. Ce chapitre-ci, qui accompagne AP4, construit les deux premières : il range des exemples dans des *tenseurs* et décrit le calcul d'un score par un graphe que Nx saura dériver. Le chapitre 8, qui accompagne AP5, ajoutera la perte et la mise à jour. Dans la suite, une grille de points du plan remplace les tickets : elle suffit pour observer les prédictions, puis, au chapitre suivant, mesurer l'erreur et modifier les paramètres.

Nous utiliserons `Nx`²⁷ pour les tableaux numériques et les dérivées, `Axon`²⁸ pour assembler les couches d'un réseau, et `Polaris`²⁹ pour l'optimiseur, dans les versions des carnets du cours ; la première installation demande une connexion.

```
1 Mix.install([
2   {:nx, "1.0.0"},
3   {:axon, "0.9.0"},
4   {:polaris, "0.2.0"}
5 ])
6
7 Nx.default_backend(Nx.BinaryBackend)
8 Nx.Defn.default_options(compiler: Nx.Defn.Evaluator)
```

Le *backend* détermine le stockage et le calcul. Ici, `Nx.BinaryBackend` et l'évaluateur exécutent les petites expériences sur CPU. Aucun GPU n'est nécessaire ; EXLA est une autre option, compilée, hors de ce socle.

7.1 Transformer des exemples avant de calculer

Commençons sans `Nx`. Chaque observation, c'est-à-dire chaque exemple, sera une map avec deux coordonnées et une *étiquette* (en anglais *label*), la réponse attendue : ici une *classe*, 0 ou 1, qui range le point dans l'une de deux catégories. La classe vaut 1 lorsque les coordonnées ont le même signe, 0 sinon ; nous choisissons aussi 0 sur les axes. Pour préparer une petite grille, `for` parcourt les combinaisons, tandis que `Enum.map` transforme chaque observation sans modifier la liste initiale.

```
1 coordinates = [-1.0, -0.5, 0.5, 1.0]
2 rows = for x <- coordinates, y <- coordinates do
3   %{x: x, y: y,
4     label: if(x * y > 0, do: 1, else: 0)}
5 end
6
7 counts = Enum.frequencies_by(rows, & &1.label)
8 features = Enum.map(rows, fn p -> [p.x, p.y] end)
9 labels = Enum.map(rows, fn p -> [p.label] end)
```

Deux écritures sont nouvelles ici. La *compréhension* `for x <- liste, y <- liste do ... end` parcourt toutes les combinaisons de ses générateurs et rend la liste des valeurs du bloc, comme deux boucles imbriquées en Python qui rempliraient une liste. Quant à `& &1.label`, c'est l'abréviation de `fn p -> p.label end`, la capture du chapitre 4 ; `Enum.frequencies_by/2` compte ainsi les observations par étiquette et rend `%{0 => 8, 1 => 8}`.

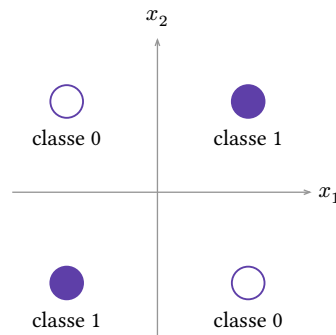
Combien d'exemples avons-nous construits ? Quatre choix pour chaque coordonnée donnent seize points. Les deux classes comptent huit exemples chacune. L'ordre commence par `[-1.0, -1.0]`, `[-1.0, -0.5]`, `[-1.0, 0.5]`, `[-1.0, 1.0]` : le générateur intérieur parcourt toutes ses valeurs avant que le premier change. `features` conserve cet ordre, tout comme `labels`. Les deux listes séparent ce que le modèle recevra, les *caractéristiques*

²⁷ <https://hexdocs.pm/nx/1.0.0/Nx.html>

²⁸ <https://hexdocs.pm/axon/0.9.0/Axon.html>

²⁹ <https://hexdocs.pm/polaris/0.2.0/Polaris.Optimizers.html>

(*features*) de chaque exemple, ici ses deux coordonnées, de ce qu'il devra retrouver, les étiquettes.



Aux quatre coins, les classes alternent. Les deux points de classe 1 sont sur une diagonale, ceux de classe 0 sur l'autre.

Nous connaissons volontairement la règle $x * y > 0$: c'est ce qui nous permettra de juger ce que le modèle apprend, lui qui n'en recevra que des exemples. Pourquoi `labels` contient-il des listes à un élément ? Nous préparons déjà une convention de forme, un exemple par ligne et une réponse par ligne ; si nous triions les coordonnées sans trier les étiquettes de la même manière, les deux tableaux garderaient la bonne taille, mais leurs lignes ne parleront plus des mêmes points.

PIÈGE

On peut obtenir un score parfait en préparant de mauvaises données. Garder uniquement la classe 1 permet à un programme qui répond toujours 1 d'atteindre 100 %. Ajouter `label` aux coordonnées lui fournit directement la réponse recherchée. Dans les deux cas, le calcul s'exécute ; c'est la question posée à l'expérience qui a changé.

7.2 Lire la forme d'un tenseur

`Nx` range les nombres dans des *tenseurs*, l'équivalent des tableaux NumPy (`np.array`). Les mathématiciens ont déjà des noms pour les premiers cas : un nombre seul est un *scalaire* ; une liste de nombres, comme les deux coordonnées $(-1, 0.5)$ d'un point, est un *vecteur* ; un tableau rectangulaire, avec des lignes et des colonnes, est une *matrice*. Un tenseur généralise ces trois cas à un nombre quelconque de directions de rangement, les *axes* : un *lot* de cent images de 28×28 pixels, c'est-à-dire un paquet d'exemples traités ensemble, en a trois.

Objet	Exemple	Axes	Forme Nx	NumPy
scalaire	un biais	0	{}	()
vecteur	deux poids	1	{2}	(2,)
matrice	nos seize points	2	{16, 2}	(16, 2)
tenseur	cent images	3	{100, 28, 28}	(100, 28, 28)

Le *rang* d'un tenseur est son nombre d'axes, et sa *forme* (*shape*) le tuple des tailles de ses axes, dans l'ordre : `Nx` l'écrit avec les accolades d'un tuple Elixir, `{16, 2}`, là où NumPy écrit `(16, 2)`. Tous les éléments d'un tenseur ont le même *type*, par ex. des flottants. Ce

sont donc les mêmes questions qu’avec un tableau NumPy, mais nous allons rendre nos conventions explicites : l’axe 0 énumère les exemples, l’axe 1 leurs coordonnées ou sorties.

```
1 x = Nx.tensor(features, type: :f32)
2 y = Nx.tensor(labels, type: :f32)
3 scalar = Nx.tensor(0.5)
4
5 {Nx.shape(x),          # => {16, 2}
6  Nx.shape(y),          # => {16, 1}
7  Nx.shape(scalar)}     # => {}
```

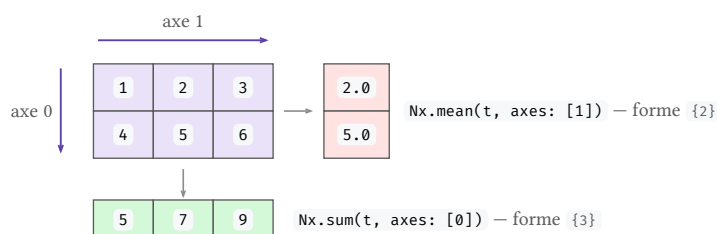
La forme `{16, 2}` contient seize lignes de deux nombres. La forme `{16, 1}` contient seize lignes d’un nombre ; ce n’est pas la forme `{16}`. Enfin un scalaire n’a aucun axe : sa forme est le tuple vide `{}`. Le nombre d’éléments ne détermine donc pas, à lui seul, le sens d’un tenseur.

Le type `:f32` demande des flottants IEEE 754 comme ceux du chapitre 6, mais sur 32 bits, quatre octets, au lieu de 64. `Nx.type(x)` rend `{:f, 32}`. Un entier signé sur 32 bits porte le type `:s32`, un octet non signé `:u8`. On choisit le type en fonction du calcul : les pixels d’une image peuvent être stockés sur des octets, puis convertis en flottants pour l’apprentissage. La précision finie intervient même lorsque la formule mathématique est exacte.

Pour lire une réduction, cherchons quel axe disparaît. Voici un tableau dont les valeurs sont toutes différentes, afin que chaque résultat soit vérifiable.

```
1 a = Nx.tensor([[1, 2, 3], [4, 5, 6]])
2 column_sums = Nx.sum(a, axes: [0])
3 row_means = Nx.mean(a, axes: [1])
```

Une *réduction* remplace tout un axe par une seule valeur. Réduire l’axe 0 additionne les nombres qui ne diffèrent que par leur ligne, c-à-d. chaque colonne de haut en bas : $1 + 4 = 5$, $2 + 5 = 7$ et $3 + 6 = 9$. Il reste `[5, 7, 9]`, une somme par colonne, de forme `{3}`. Réduire l’axe 1 fait la moyenne des trois nombres de chaque ligne : $(1 + 2 + 3)/3 = 2$ et $(4 + 5 + 6)/3 = 5$, d’où `[2.0, 5.0]`, de forme `{2}`. L’axe nommé dans `axes:` est donc celui qui disparaît de la forme ; c’est l’`axis=0` de `np.sum(a, axis=0)` en NumPy. `Nx.sum(a)`, sans préciser d’axe, réduit toutes les cases et rend le scalaire `21`.



Les indices d’axes partent de zéro. Réduire les lignes conserve une valeur par colonne ; réduire les colonnes conserve une valeur par ligne.

L’option `keep_axes: true` conserve une dimension de taille 1 à la place de chaque axe réduit. Ainsi `Nx.sum(a, axes: [1], keep_axes: true)` a la forme `{2, 1}`. Le résultat numérique n’a pas changé ; sa forme prépare une opération ultérieure. Nous nous en servons pour produire une réponse par exemple.

7.2.1 Diffuser une valeur sans perdre la correspondance

Hors d'un `defn`, les opérations sur tenseurs passent par `Nx.add`, `Nx.subtract`, `Nx.multiply`, etc. L'opérateur Elixir `+` ordinaire ne sait pas additionner deux structures `Nx.Tensor`. Une opération `Nx` agit sur des tableaux entiers, sans que nous écrivions une boucle sur chaque nombre.

```
1 shifted = Nx.add(a, Nx.tensor([10, 20, 30]))
2 # [[11, 22, 33], [14, 25, 36]]
```

La ligne de forme `{3}` a été appliquée aux deux lignes de `{2, 3}`. C'est le *broadcasting*, ou diffusion : `Nx` étend un tenseur en répétant ses valeurs pour l'accorder à la forme de l'autre, sans que nous ayons à écrire ces copies. Les formes sont alignées à partir de la droite ; des tailles égales se correspondent, et une taille 1 peut être étendue. Un axe absent à gauche se traite comme une taille 1. Un scalaire peut donc être ajouté à chaque case d'un tableau.

Pour prédire le résultat, écrivez les deux formes l'une sous l'autre en les alignant à droite, complétez la plus courte par des 1 à gauche, puis comparez les tailles colonne par colonne : deux tailles égales se conservent, une taille 1 prend la taille d'en face, et deux tailles différentes dont aucune ne vaut 1 provoquent une erreur.

Formes	Complétées	Colonne par colonne	Résultat
{2, 3} et {3}	{2, 3} et {1, 3}	2 contre 1, puis 3 contre 3	{2, 3}
{3, 1} et {3}	{3, 1} et {1, 3}	3 contre 1, puis 1 contre 3	{3, 3}
{3, 2} et {3}	{3, 2} et {1, 3}	2 contre 3 : aucune ne vaut 1	erreur

La première ligne est notre addition. La troisième échoue : `Nx` lève une `ArgumentError` (« cannot broadcast tensor... »). La deuxième ne lève aucune erreur, et c'est elle qui va nous jouer un tour. Ces règles sont celles de NumPy, dont la [documentation sur le broadcasting](https://numpy.org/doc/stable/user/basics.broadcasting.html)³⁰ donne d'autres exemples dessinés.

Cette commodité peut aussi transformer une comparaison ligne à ligne en toutes les comparaisons possibles. Prédisez la forme de `wrong` avant de lire la suite.

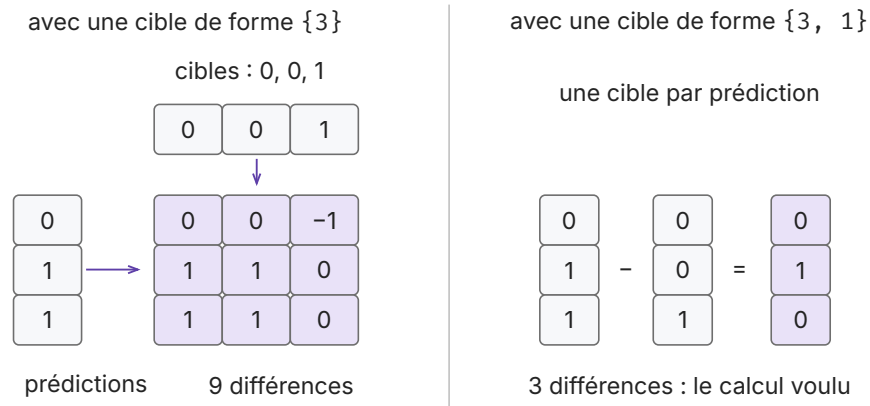
```
1 predictions = Nx.tensor([[0.0], [1.0], [1.0]])
2 expected = Nx.tensor([[0.0], [0.0], [1.0]])
3 right = Nx.subtract(predictions, expected)
4 wrong = Nx.subtract(predictions,
5   Nx.reshape(expected, {3}))
```

`right` a la forme `{3, 1}` et contient `[[0.0], [1.0], [0.0]]` : une seule prédiction est fausse. Pour résumer ces écarts par un seul nombre, on les élève au carré, ce qui les rend tous positifs, puis on en fait la moyenne : c'est l'*erreur quadratique moyenne*, abrégée MSE, que le chapitre 8 étudiera comme perte. Ici, les carrés valent 0, 1 et 0, et leur moyenne vaut $\frac{1}{3}$; en `Nx`, elle s'écrit `right |> Nx.pow(2) |> Nx.mean()`.

Pour `wrong`, les formes `{3, 1}` et `{3}` deviennent compatibles comme `{3, 1}` et `{1, 3}` : la sortie est `{3, 3}`. Sa première ligne compare la première prédiction aux trois étiquettes, et

³⁰ <https://numpy.org/doc/stable/user/basics.broadcasting.html>

vaut $[0.0, 0.0, -1.0]$. La moyenne de ses neuf carrés vaut $\frac{5}{9}$ au lieu de $\frac{1}{3}$: l'erreur mesurée est faussée, et il n'y a pourtant aucune exception à rattraper, puisque le calcul est autorisé. Le dessin permet de compter les comparaisons. À gauche, chaque prédiction rencontre toutes les cibles ; à droite, les lignes restent appariées.



Les nombres sont les différences prédiction moins cible. Une cible aplatie produit un tableau de neuf différences ; conserver sa forme de colonne en produit trois. Les valeurs entières abrègent ici les flottants du code.

À RETENIR

Une forme est un contrat sur les axes. Pour comparer une prédiction à sa cible, vérifier que les deux tableaux ont le même nombre d'éléments ne suffit pas. Donnez le sens de chaque axe, puis exigez des formes identiques lorsque vous voulez une différence case par case.

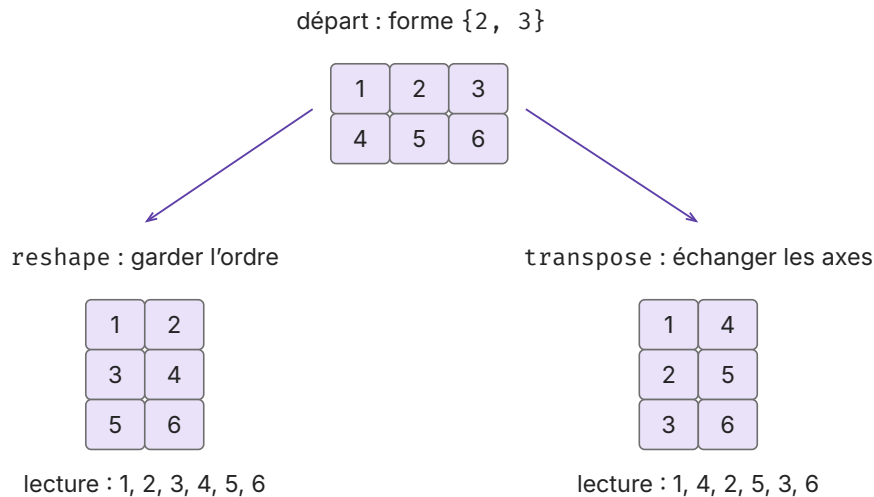
7.2.2 Changer la forme n'est pas changer les axes

`reshape` conserve l'ordre des valeurs et les range dans une nouvelle forme. `transpose` permute les axes et donc l'ordre de parcours des cases.

```
1 reshaped = Nx.reshape(a, {3, 2})
2 transposed = Nx.transpose(a)
```

Les deux sorties ont la forme $\{3, 2\}$. Pourtant, la première contient $[[1, 2], [3, 4], [5, 6]]$, tandis que la seconde contient $[[1, 4], [2, 5], [3, 6]]$. Un tableau carré rempli de zéros aurait masqué la différence : un bon exemple de contrôle doit faire apparaître le souci cherché.

Suivez par exemple la valeur 4 dans les deux transformations.



Les sorties ont toutes deux trois lignes et deux colonnes. `reshape` conserve la suite lue ligne par ligne ; `transpose` fait de chaque ancienne colonne une nouvelle ligne.

Pour centrer des données, on retire à chaque coordonnée sa moyenne sur les exemples. La moyenne a la forme {2} et se diffuse sur toutes les lignes.

```

1 sample = Nx.tensor([[1.0, 10.0], [3.0, 20.0],
2                     [5.0, 30.0]])
3 center = Nx.mean(sample, axes: [0])
4 centered = Nx.subtract(sample, center)

```

Le centre vaut [3.0, 20.0], et la première ligne centrée [-2.0, -10.0]. `sample` reste inchangé : le résultat est un nouveau tenseur. Plus loin, nous réserverons des données pour évaluer le modèle. Le centre devra alors être calculé sur les seules données d'apprentissage, puis réutilisé tel quel sur les autres, afin de reproduire le traitement d'une entrée future.

7.3 Décrire un calcul pour pouvoir le dériver

Nos points sont maintenant des tenseurs. Il reste à décrire le calcul qui leur attribue un *score*, c'est-à-dire un nombre par point, que le chapitre 8 comparera aux réponses attendues. Nous l'écrirons pour tout un lot à la fois, puis nous regarderons la représentation que Nx construit de ce calcul : c'est elle qui lui permettra, au chapitre 8, d'en calculer les dérivées.

7.3.1 Une somme pondérée par point

Un score *affine* de deux coordonnées s'écrit $s(x) = w_1x_1 + w_2x_2 + b$, où x_1 et x_2 sont les deux coordonnées du point x . C'est une fonction affine, comme $ax + b$, mais de deux variables au lieu d'une. Les *poids* w_1 et w_2 disent combien chaque coordonnée compte, et dans quel sens ; le *biais* b décale le résultat. Poids et biais sont les paramètres, que l'apprentissage modifiera ; les coordonnées sont les données, qu'il ne touche pas. Pour le point $x = (1, 2)$, les poids $(2, -1)$ et le biais 0.5 :

$$s(x) = 2 \times 1 + (-1) \times 2 + 0.5 = 0.5.$$

Chaque coordonnée est multipliée par son poids, puis les produits sont additionnés : on parle de *somme pondérée*. Avec davantage de coordonnées, on l'abrège par le symbole $\sum$, la lettre grecque sigma majuscule :

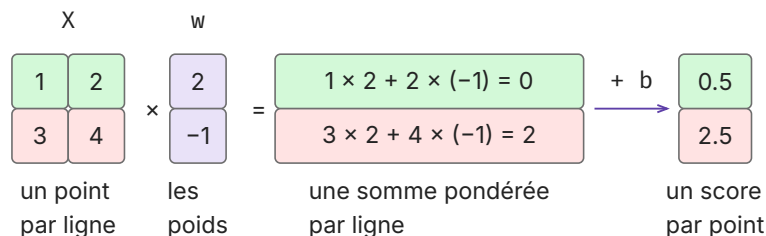
$$w_1x_1 + w_2x_2 = \sum_{j=1}^2 w_jx_j.$$

Lisez « somme, pour j allant de 1 à 2, de w_jx_j » : on remplace j par 1, puis par 2, et l'on additionne les termes obtenus. L'indice j n'existe qu'à l'intérieur de la somme ; avec cinq coordonnées, il irait de 1 à 5. Cette somme des produits terme à terme s'appelle aussi le *produit scalaire* des vecteurs w et x , `np.dot(w, x)` en NumPy.

Pour un lot, rangeons les points dans une matrice X , un point par ligne, comme dans nos tenseurs $\{n, 2\}$. Le *produit matrice-vecteur* Xw calcule le produit scalaire de chaque ligne de X avec w :

$$Xw = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} 2 \\ -1 \end{pmatrix} = \begin{pmatrix} 1 \times 2 + 2 \times (-1) \\ 3 \times 2 + 4 \times (-1) \end{pmatrix} = \begin{pmatrix} 0 \\ 2 \end{pmatrix}.$$

C'est le `x @ w` de NumPy : plusieurs sommes pondérées, une par exemple, calculées d'un coup. Il reste à ajouter le biais à chacune pour obtenir les scores 0.5 et 2.5.



Chaque ligne de X rencontre les mêmes poids et donne sa propre somme pondérée ; les lignes restent dans l'ordre des points. Le biais s'ajoute ensuite à chaque somme.

Nx propose `Nx.dot(x, w)` pour ce produit, qui rend un vecteur de forme $\{n\}$. Le carnet d'AP4 suit plutôt un trajet qui fait apparaître chaque forme et rend une colonne $\{n, 1\}$, une sortie par exemple, comme les étiquettes. Pour les poids $[2.0, -1.0]$, le biais 0.5 et les points $[[1.0, 2.0], [3.0, 4.0]]$, suivons ce trajet avant de le programmer. Le produit terme à terme, que le broadcasting applique à chaque ligne, donne $[[2.0, -2.0], [6.0, -4.0]]$. La somme sur l'axe 1, avec `keep_axes: true`, rend $[[0.0], [2.0]]$; l'ajout du biais rend $[[0.5], [2.5]]$. La forme finale est $\{2, 1\}$. Une somme sur l'axe 0 aurait additionné les contributions d'exemples différents. Le carnet d'AP4 vous demande de traduire ce trajet en une fonction numérique, `Score.predict`, qui reçoit les paramètres et le lot et rend ces scores.

7.3.2 Une fonction numérique devient un graphe

Pour découvrir `defn` sur un calcul indépendant, prenons la parabole $q(u) = (u - 3)^2$. Le mot-clé `defn` définit une *fonction numérique* : elle s'écrit comme un `def`, mais son corps est confié à Nx, qui y traduit les opérateurs ordinaires `+`, `-`, `*` ou `**` en opérations sur tenseurs et peut ensuite le dériver ; `Nx.pow` ci-dessous rend simplement l'opération visible. Cette petite formule devient elle aussi un graphe.

```

1 defmodule Parabola do
2   import Nx.Defn
3
4   defn value(position), do: Nx.pow(position - 3, 2)
5 end
6
7 parabola_start = Nx.tensor(0.0)
8 parabola_value = Parabola.value(parabola_start)

```

À la position `0.0`, le résultat est le scalaire `9.0`. Quelle valeur attendez-vous à la position `3.0` ? Nous connaissons déjà le minimum de cette courbe, atteint en $u = 3$: c'est ce qui en fait un bon terrain d'essai pour observer, au chapitre 8, le mécanisme du gradient.

Dans les chapitres 5 et 6, un programme devenait un arbre puis des instructions. L'arbre de `1.0 + 2.0 * 3.0` avait l'addition pour racine et la multiplication pour enfant ; l'évaluateur calculait les enfants avant leur parent. Nx fait de même avec le corps d'un `defn` : il en construit une représentation, un *graphe* de calcul, dont les nœuds sont les opérations et dont les liens disent quelle valeur alimente quelle opération. Une seule différence avec un arbre : une même valeur peut alimenter plusieurs nœuds. On peut inspecter ce graphe :

```

1 graph = Nx.Defn.jit(&Parabola.value/1,
2   compiler: Nx.Defn.Debug)
3 parabola_graph = graph.(parabola_start)

```

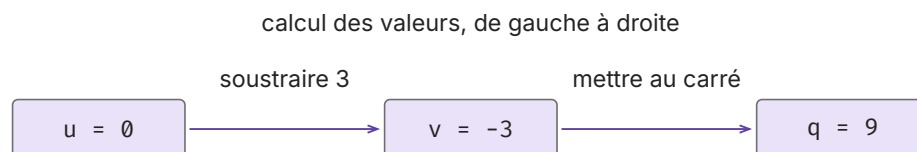
`Nx.Defn.jit` prépare une fonction numérique pour un compilateur donné. Avec le compilateur d'inspection `Nx.Defn.Debug`, un module interne que le carnet AP4 utilise aussi, l'appel ne calcule rien : il rend le graphe, que Livebook affiche ainsi, sous l'en-tête `Nx.Defn.Expr` :

```

1 parameter a:0      f32
2 b = subtract a, 3   f32
3 c = pow b, 2        f32

```

Chaque ligne nomme un résultat par une lettre, puis donne l'opération, ses opérandes et le type du résultat ; une forme non scalaire s'afficherait après le type, par ex. `f32[2][1]`. La première ligne déclare l'entrée, le paramètre numéro 0 de la fonction ; `b` est cette entrée moins 3 ; `c` est `b` au carré. Si nous écrivions plutôt `v = position - 3` puis `v * v`, la dernière ligne deviendrait `c = multiply b, b` : le même nœud `b` alimenterait deux fois la multiplication, ce qu'un arbre ne sait pas représenter. Ce sont ces opérations et leurs dépendances qui nous intéressent ; différents moteurs peuvent exécuter cette description différemment.



Le calcul suit les dépendances de gauche à droite : chaque opération attend la valeur de celle qui l'alimente. Les noms `v` et `q` désignent les résultats intermédiaires, comme `b` et `c` dans l'affichage de Nx.

Pourquoi construire ce graphe au lieu de calculer tout de suite ? Parce qu'il devient une donnée que Nx peut parcourir et transformer, comme notre compilateur transformait

l'arbre au chapitre 6. La transformation qui nous intéresse est la *différentiation automatique* : en parcourant le graphe, Nx peut calculer la dérivée du résultat par rapport à une entrée. Le chapitre 8 explique ce qu'est une dérivée et comment ce parcours la calcule.

EN PRATIQUE

La préparation des listes, les noms de fichiers et les graphiques restent du code Elixir ordinaire. `defn` sert au calcul numérique avec ses propres règles, dont des contrôles de flux numériques soumis à des contraintes de formes. Il ne transforme pas tout programme Elixir en code GPU.

Voici les opérations Nx de ces premières sections, telles que nous les écrivons hors d'un `defn` :

Écriture	Effet
<code>Nx.tensor(liste, type: :f32)</code>	construit un tenseur de flottants sur 32 bits
<code>Nx.shape(t)</code> , <code>Nx.type(t)</code>	forme, par ex. <code>{16, 2}</code> , et type, par ex. <code>{:f, 32}</code>
<code>Nx.sum(t, axes: [1])</code>	somme le long de l'axe 1, qui disparaît ; <code>Nx.mean</code> de même
<code>keep_axes: true</code>	garde l'axe réduit avec la taille 1 : <code>{2, 1}</code> au lieu de <code>{2}</code>
<code>Nx.add</code> , <code>Nx.subtract</code> , <code>Nx.multiply</code>	opérations case par case, avec broadcasting
<code>Nx.reshape(t, {3, 2})</code>	range les mêmes valeurs, dans le même ordre, sous une autre forme
<code>Nx.transpose(t)</code>	échange les axes : chaque colonne devient une ligne
<code>Nx.dot(x, w)</code>	produit matrice-vecteur, ou entre matrices : des sommes pondérées
<code>Nx.to_number</code> , <code>Nx.to_list</code>	ramènent un résultat sous forme de nombre ou de listes
<code>defn</code>	fonction numérique : <code>+</code> , <code>-</code> , <code>*</code> y opèrent sur des tenseurs

7.4 Vérifier sa compréhension

Refaites sur papier les calculs du chapitre avec d'autres nombres. Prédisez chaque réponse avant de lire celle qui suit, puis vérifiez dans un carnet.

Premier exercice, les formes. `Nx.iota({5, 3})` remplit un tenseur de forme `{5, 3}` avec les entiers 0, 1, 2... lus ligne par ligne. Quelle est la forme de chacun des trois résultats ci-dessous ?

```
1 t = Nx.iota({5, 3}, type: :f32)
2 column_totals = Nx.sum(t, axes: [0])
3 row_averages = Nx.mean(t, axes: [1], keep_axes: true)
4 all_pairs = Nx.subtract(row_averages,
5   Nx.iota({5}, type: :f32))
```

`column_totals` a la forme `{3}` : l'axe 0 disparaît, il reste une somme par colonne, `[30.0, 35.0, 40.0]`. `row_averages` a la forme `{5, 1}` : une moyenne par ligne, l'axe réduit étant gardé avec la taille 1. `all_pairs` a la forme `{5, 5}` : alignées à droite, `{5, 1}` et `{5}` deviennent `{5, 1}` et `{1, 5}`, qui s'étendent l'une l'autre, sans erreur. C'est le piège de la section sur les formes.

Second exercice, des sommes pondérées. Deux points, $(1, 2)$ et $(3, 4)$, rencontrent les poids $w = (0.5, -1)$ et le biais $b = 1$. Calculez à la main leurs deux scores, puis écrivez le calcul pour le lot entier.

```
1 points = Nx.tensor([[1.0, 2.0], [3.0, 4.0]])
2 point_weights = Nx.tensor([0.5, -1.0])
3 point_scores = Nx.add(Nx.dot(points, point_weights), 1.0)
```

Le premier score vaut $0.5 \times 1 - 1 \times 2 + 1 = -0.5$, le second $0.5 \times 3 - 1 \times 4 + 1 = -1.5$. `Nx.dot` fait les deux sommes pondérées d'un coup, une par ligne : la matrice `{2, 2}` rencontre le vecteur `{2}` et rend un vecteur `{2}`, auquel le biais s'ajoute par broadcasting. `point_scores` vaut donc `[-0.5, -1.5]`. Le chapitre 8 se demandera comment choisir ces poids.

POUR EN PARLER

Une forme est un contrat sur les axes : `{16, 2}` range seize exemples de deux coordonnées, l'axe nommé dans `axes` : disparaît, et `{3, 1}` moins `{3}` donne `{3, 3}` sans aucune erreur.

Le broadcasting étend un tenseur en répétant ses valeurs ; il rend service pour ajouter un biais à tout un lot, mais il ne prévient pas quand deux formes ne se correspondent pas comme on le croyait.

Une somme pondérée multiplie chaque entrée par son poids et ajoute un biais ; `Nx.dot` en calcule une par ligne, pour tout un lot à la fois.

Le corps d'un `defn` devient un graphe de calcul, comme le texte d'un programme devenait un arbre au chapitre 5 ; une même valeur peut y alimenter plusieurs opérations, et c'est sur ce graphe que `Nx` calculera des dérivées.

Des calculs qui apprennent

SÉANCE AP5 · PERTE, GRADIENT, RÉSEAU ET EXPÉRIENCES

Au chapitre 7, nous avons rangé des exemples dans des tenseurs et calculé un score à partir de paramètres, en décrivant ce calcul par un graphe. Une question restait ouverte : comment choisir ces paramètres ? Revenons aux tickets de taxi. Partis de $w = 1$ et $b = 4$, nous voulions arriver à $w \approx 2$ et $b \approx 5$ sans connaître le tarif. Il nous manque deux des quatre idées annoncées : une *perte*, qui dit par un seul nombre à quel point le modèle se trompe, et une *mise à jour*, qui modifie les paramètres pour faire baisser cette perte.

Ce chapitre les construit pas à pas. Nous mesurons d'abord l'erreur, puis nous voyons ce qu'est une dérivée et comment le gradient indique dans quel sens corriger ; nous répétons ensuite ces corrections, nous rencontrons une limite qu'aucune répétition ne lève, et nous la dépassons avec un petit réseau construit par Axon. Nous terminons sur la façon de comparer honnêtement des essais.

Les blocs de code se tapent dans un carnet qui commence par la même installation qu'au chapitre 7, Nx, Axon et Polaris, avec le même backend. Ils ne reprennent rien d'autre de ce chapitre, sauf la parabole du graphe de calcul, que nous rappellerons au moment d'en avoir besoin.

8.1 Une perte donne un sens à « mieux »

Pour améliorer un modèle, il faut d'abord pouvoir dire en quoi il est mauvais, avec un nombre : c'est le rôle de la perte. Avant de choisir des classes, prenons une règle affine que notre modèle peut représenter exactement. Aux quatre coins du carré, les réponses attendues seront données par $2x_1 + x_2 + 0.5$. Quels paramètres devraient donc être retrouvés ? Nous attendons deux poids 2.0 et 1.0 , avec un biais 0.5 .

```
1 corners = Nx.tensor([
2     [-1.0, -1.0], [-1.0, 1.0],
3     [1.0, -1.0], [1.0, 1.0]
4 ])
5 affine_targets = Nx.tensor([
6     [-2.5], [-0.5], [1.5], [3.5]
7 ])
8 initial = %{weights: Nx.tensor([0.3, -0.2]),
9     bias: Nx.tensor(0.0)}
```

Les paramètres initiaux sont rangés dans une map, un poids par coordonnée et un biais. Le score du premier coin, $(-1, -1)$, vaut $0.3 \times (-1) + (-0.2) \times (-1) + 0 = -0.1$; de même pour les autres coins, les prédictions valent $[-0.1, -0.5, 0.5, 0.1]$, une par ligne, loin des réponses attendues. Comment résumer ces quatre erreurs par un seul nombre ? Construisons la mesure pas à pas.

Coin	Prédiction s	Réponse y	Écart $s - y$	Carré
$(-1, -1)$	-0.1	-2.5	2.4	5.76
$(-1, 1)$	-0.5	-0.5	0.0	0.00
$(1, -1)$	0.5	1.5	-1.0	1.00
$(1, 1)$	0.1	3.5	-3.4	11.56
moyenne			-0.5	4.58

Première idée : la moyenne des écarts. Elle vaut ici -0.5 , comme si le modèle se trompait à peine, alors qu'un écart atteint -3.4 : les écarts positifs et négatifs se compensent. Nous prenons donc le carré de chaque écart, toujours positif ou nul. Le carré a un autre effet : il pénalise davantage les grosses erreurs, puisqu'un écart de 3.4 coûte 11.56 , plus de onze fois le coût d'un écart de 1 . La moyenne de ces carrés est l'erreur quadratique moyenne du chapitre 7, la MSE (*mean squared error*) : $18.32/4 = 4.58$. Avec la notation $\sum$ introduite au chapitre 7 pour les sommes pondérées, elle s'écrit

$$L(w, b) = \frac{1}{n} \sum_{i=1}^n (s(x_i) - y_i)^2.$$

Lisons-la symbole par symbole. n est le nombre d'exemples, ici 4 ; l'indice i parcourt les exemples ; x_i désigne le i -ième point et y_i sa réponse attendue ; $s(x_i) - y_i$ est l'écart de la ligne i du tableau, que l'on met au carré ; enfin $\frac{1}{n} \sum$ fait la moyenne. L'écriture $L(w, b)$ rappelle que la perte dépend des paramètres : les points et les réponses sont fixés, ce sont les poids et le biais que nous allons changer.

Ce nombre résume les erreurs avec la convention choisie. Une autre perte poserait une autre question ; nous ne comparons pas directement deux valeurs obtenues avec des définitions différentes.

Comment savoir dans quel sens modifier chaque paramètre ? Si j'augmente ce paramètre d'un tout petit peu, la perte monte-t-elle ou descend-elle, et à quelle vitesse ? C'est exactement la question à laquelle répond une dérivée. Commençons avec un seul nombre à régler.

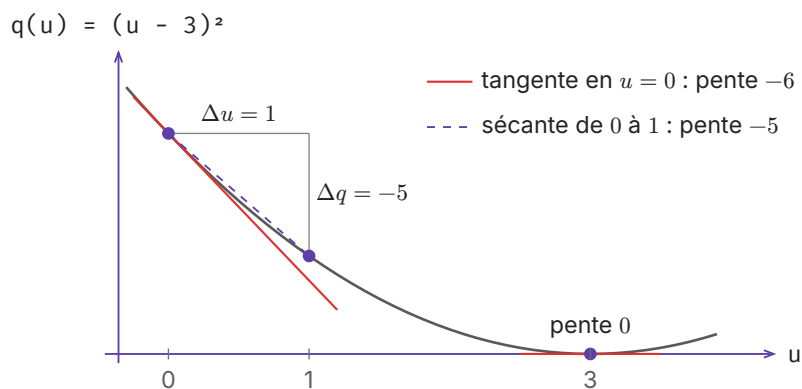
8.1.1 La dérivée mesure une sensibilité

Revenons à la parabole $q(u) = (u - 3)^2$ du graphe de calcul du chapitre 7. En $u = 0$, elle vaut 9 . Que se passe-t-il si l'on augmente u d'un petit peu ? Notons h ce petit pas et calculons de combien q change, rapporté à h :

Pas h	$q(h)$	$q(h) - q(0)$	$(q(h) - q(0))/h$
1	4	-5	-5
0.1	8.41	-0.59	-5.9
0.01	8.9401	-0.0599	-5.99

Le rapport se rapproche de -6 à mesure que h diminue. C'est la définition de la *dérivée* : $q'(0)$ est la limite de ce rapport quand h tend vers 0. Ici, le calcul est même exact : $q(h) - q(0) = (h - 3)^2 - 9 = h^2 - 6h$, donc le rapport vaut $h - 6$, qui tend bien vers -6 . Les règles de dérivation usuelles donnent directement $q'(u) = 2(u - 3)$, et $q'(0) = -6$.

Deux lectures de ce nombre nous serviront. Sur le dessin, c'est la *pente* de la tangente à la courbe ; le rapport du tableau est la pente d'une sécante, qui se rapproche de la tangente quand h diminue. Concrètement, c'est une *sensibilité* : si j'augmente u d'une petite quantité, q varie d'environ -6 fois cette quantité. Augmenter u de 0.01 fait donc baisser q d'environ 0.06. Le signe dit dans quel sens q varie, la valeur absolue à quelle vitesse.



La lettre Δ (delta) note une variation. La sécante de $u = 0$ à $u = 1$ a pour pente $\Delta q / \Delta u = -5$; quand le second point se rapproche du premier, elle tend vers la tangente, de pente $q'(0) = -6$. Au minimum, en $u = 3$, la tangente est horizontale.

En $u = 5$, la dérivée $2(5 - 3) = 4$ est positive : augmenter u ferait monter q . En $u = 3$, elle est nulle : la courbe est plate au fond de la vallée. Pour faire baisser q , une dérivée négative invite donc à augmenter u , et une dérivée positive à le diminuer.

Nx permet de vérifier les deux points de vue. Reprenons la fonction numérique du chapitre 7, qui calcule cette parabole, et sa valeur en $u = 0$:

```
1 defmodule Parabola do
2   import Nx.Defn
3
4   defn value(position), do: Nx.pow(position - 3, 2)
5 end
6
7 parabola_start = Nx.tensor(0.0)
8 parabola_value = Parabola.value(parabola_start)
```

Le rapport du tableau s'appelle une *différence finie* ; il se calcule avec cette fonction. `Nx.Defn.grad(point, f)` rend, lui, la dérivée de la fonction f au point donné ; `&Parabola.value/1` est la capture du chapitre 4, qui désigne notre fonction sans l'appeler.

```

1 h = 0.01
2 nudged = Parabola.value(Nx.tensor(h))
3 finite_slope = Nx.divide(
4     Nx.subtract(nudged, parabola_value), h)
5 exact_slope = Nx.Defn.grad(parabola_start,
6     &Parabola.value/1)

```

`finite_slope` vaut environ -5.99, aux arrondis des flottants sur 32 bits près, et `exact_slope` vaut -6.0. Nx n'a pourtant essayé aucune petite valeur de h : nous verrons plus bas comment il obtient la valeur exacte. Une différence finie reste un bon moyen de contrôler une dérivée.

8.1.2 Deux paramètres, deux dérivées partielles

Revenons aux tickets de taxi de l'introduction, avec un seul ticket pour garder des calculs courts : 2 km pour 9 €. Le modèle prédit $2w + b$; partons de $w = 1$ et $b = 4$. La prédiction vaut 6, l'écart $6 - 9 = -3$, et la perte, ici le carré de cet unique écart, vaut 9. Il y a maintenant deux nombres à régler. L'idée est de les examiner un par un : on fait varier l'un, on gèle l'autre.

On change	Prédiction	Écart	Perte	Variation / 0.01
rien	6	-3	9	
w : de 1 à 1.01	6.02	-2.98	8.8804	-11.96
b : de 4 à 4.01	6.01	-2.99	8.9401	-5.99

Gelons $b = 4$: la perte devient une fonction de w seul, $L(w) = (2w + 4 - 9)^2 = (2w - 5)^2$. La dérivée de $(aw + c)^2$ est $2a(aw + c)$, une règle de dérivation usuelle ; ici elle vaut $4(2w - 5)$, soit -12 en $w = 1$. Gelons maintenant $w = 1$: $L(b) = (b - 7)^2$, de dérivée $2(b - 7)$, soit -6 en $b = 4$. Ce sont les *dérivées partielles* de la perte, notées

$$\frac{\partial L}{\partial w} = -12 \quad \text{et} \quad \frac{\partial L}{\partial b} = -6.$$

Le symbole ∂ , un « d rond », rappelle que l'on a dérivé par rapport à un seul paramètre en gelant les autres. Le tableau les retrouve à peu près, par différence finie.

Les deux dérivées sont négatives : augmenter w ou b fait baisser la perte. Mais w agit deux fois plus vite, parce qu'il est multiplié par la distance, 2 km, alors que b s'ajoute tel quel. On range ces sensibilités dans un vecteur, le *gradient*, noté ∇L (le symbole ∇ se lit « nabla ») :

$$\nabla L(1, 4) = \left(\frac{\partial L}{\partial w}, \frac{\partial L}{\partial b} \right) = (-12, -6).$$

Le gradient a autant de composantes que de paramètres : trois pour le score affine de nos coins, 33 pour le réseau de la fin du chapitre.

8.1.3 Des dérivées qui se multiplient le long du calcul

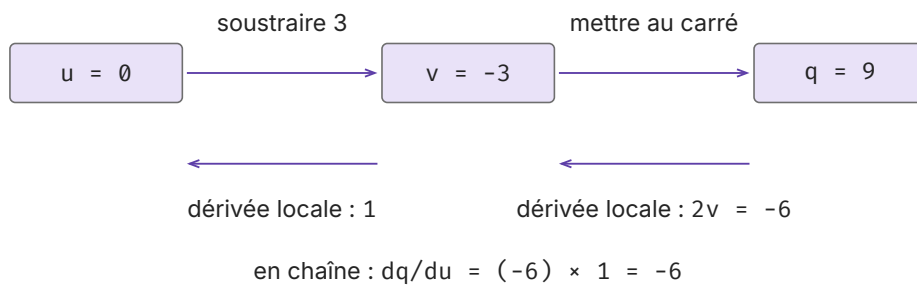
Comment Nx calcule-t-il ces dérivées sans que nous ayons écrit de formule ? Revenons au graphe de la parabole. Il enchaîne deux opérations simples : $v = u - 3$, puis $q = v^2$. Chacune a une dérivée facile, sa *dérivée locale*. Quand u augmente d'un petit peu, v

augmente d'autant : $\frac{dv}{du} = 1$. Quand v augmente d'un petit peu, q varie de $2v$ fois autant : $\frac{dq}{dv} = 2v$, soit -6 en $v = -3$. L'écriture $\frac{dq}{dv}$ se lit « dérivée de q par rapport à v ».

Les sensibilités se multiplient le long de la chaîne. Si une voiture va deux fois plus vite qu'un vélo, et le vélo quatre fois plus vite qu'un piéton, la voiture va huit fois plus vite que le piéton. De même, si v bouge autant que u , et si q bouge de -6 fois ce que bouge v , alors q bouge de -6×1 fois ce que bouge u . C'est la *règle de dérivation des fonctions composées*, ou *règle de la chaîne* :

$$\frac{dq}{du} = \frac{dq}{dv} \times \frac{dv}{du} = (-6) \times 1 = -6.$$

calcul des valeurs, de gauche à droite



Les valeurs se calculent de gauche à droite. Les dérivées locales se multiplient ensuite de droite à gauche, en remontant les mêmes opérations.

grad automatise exactement ce parcours. Nx calcule d'abord les valeurs dans le sens du graphe, puis il remonte les opérations en multipliant leurs dérivées locales, qu'il connaît pour chaque opération de sa bibliothèque. Il ne cherche pas une formule par essais successifs et n'utilise pas de différence finie : le résultat est exact, aux arrondis près.

La même règle explique la formule du gradient de la MSE. Pour une seule observation, la perte est le carré de l'écart $e = s(x_i) - y_i$; la dérivée locale du carré vaut $2e$. L'écart, lui, augmente de $x_{i,j}$ quand le poids w_j augmente de 1, puisque w_j multiplie $x_{i,j}$ dans le score. Ce double indice désigne la coordonnée numéro j du point numéro i : par ex., $x_{3,1} = 1$ est la première coordonnée du troisième coin, $(1, -1)$. En multipliant, la dérivée de e^2 par rapport à w_j vaut $2ex_{i,j}$, et celle par rapport à b vaut $2e$, puisque le biais s'ajoute tel quel. La moyenne sur les n observations apporte le facteur $2/n$:

$$\frac{\partial L}{\partial w_j} = \frac{2}{n} \sum_{i=1}^n (s(x_i) - y_i) x_{i,j}, \quad \frac{\partial L}{\partial b} = \frac{2}{n} \sum_{i=1}^n (s(x_i) - y_i).$$

Au départ, avec les écarts du tableau et $2/n = 1/2$, les trois dérivées valent :

Dérivée	Calcul	Valeur
$\frac{\partial L}{\partial w_1}$	$\frac{1}{2}(2.4 \times (-1) + 0 \times (-1) + (-1) \times 1 + (-3.4) \times 1)$	-3.4
$\frac{\partial L}{\partial w_2}$	$\frac{1}{2}(2.4 \times (-1) + 0 \times 1 + (-1) \times (-1) + (-3.4) \times 1)$	-2.4
$\frac{\partial L}{\partial b}$	$\frac{1}{2}(2.4 + 0 - 1 - 3.4)$	-1.0

Chaque dérivée indique comment la perte varie localement lorsque ce paramètre augmente, les autres étant fixés. Leurs signes négatifs indiquent qu'une petite augmentation des paramètres diminuera la perte.

POUR ALLER PLUS LOIN

Ce parcours à rebours du graphe s'appelle la *rétropropagation*. Quand une valeur alimente plusieurs nœuds, il faut additionner les contributions des différents chemins : dans $v * v$, chaque chemin apporte v , et la dérivée totale vaut $v + v = 2v$, comme celle de v^2 . Le projet micrograd d'AP9 fait construire ce mécanisme ; Chris Olah le dessine en une page dans *Calculus on Computational Graphs: Backpropagation*³¹ (2015), et Andrej Karpathy le programme pas à pas dans *The spelled-out intro to neural networks and backpropagation*³² (2022).

8.1.4 Le signe moins et la taille du pas

Imaginez que vous descendez une vallée dans le brouillard : vous ne voyez pas le fond, mais vous sentez la pente sous vos pieds, et vous faites un pas vers le bas. La *descente de gradient* fait exactement cela. Avec un seul paramètre, un pas s'écrit

$$u' = u - \eta q'(u).$$

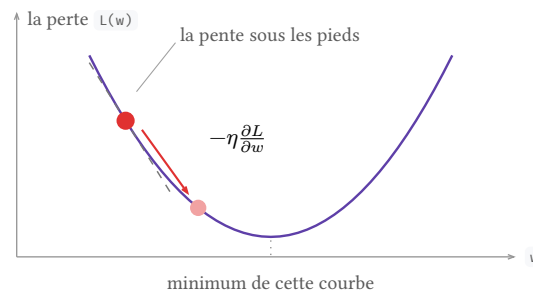
Le nombre η , la lettre grecque êta, est le *taux d'apprentissage* : un nombre positif, choisi par nous, qui règle la longueur du pas. Le signe moins fait toujours descendre. Si la dérivée est négative, comme en $u = 0$, le terme $-\eta q'(u)$ est positif et u augmente ; si elle est positive, u diminue. Avec $\eta = 0.1$, le premier pas donne $u' = 0 - 0.1 \times (-6) = 0.6$, où la parabole vaut $(0.6 - 3)^2 = 5.76$ au lieu de 9. Re commençons depuis 0.6, puis depuis chaque nouveau point :

Pas	u	$q(u)$	$q'(u)$	$u - 0.1 q'(u)$
1	0	9	-6	0.6
2	0.6	5.76	-4.8	1.08
3	1.08	3.6864	-3.84	1.464
4	1.464	2.359296	-3.072	1.7712

La valeur de q baisse à chaque pas, et les pas raccourcissent : 0.6, puis 0.48, puis 0.384. Le taux est pourtant resté le même ; c'est la pente qui s'adoucit à l'approche du fond. Chaque pas multiplie ici la distance au minimum par 0.8 : elle passe de 3 à 2.4, 1.92, puis 1.536. En effet, $u' - 3 = (u - 3) - 0.1 \times 2(u - 3) = 0.8(u - 3)$.

³¹ <https://colah.github.io/posts/2015-08-Backprop/>

³² <https://www.youtube.com/watch?v=VMj-3S1tku0>



Une coupe schématique de la perte en fonction d'un paramètre. Le gradient donne une direction locale ; le taux règle l'ampleur du déplacement.

Confions maintenant ce pas à Nx. La fonction suivante, écrite dans un `defn`, demande la dérivée avec `grad`, le même calcul que `Nx.Defn.grad` écrit à l'intérieur de la fonction numérique, puis fait un pas ; nous n'avons écrit la formule $2(u - 3)$ nulle part. Prédisez la nouvelle position avec un taux `0.1`.

```
1 defmodule Descent do
2   import Nx.Defn
3
4   defn step(position, rate) do
5     slope = grad(position, fn u -> Parabola.value(u) end)
6     position - rate * slope
7   end
8 end
9
10 parabola_next = Descent.step(parabola_start, 0.1)
11 parabola_next_value = Parabola.value(parabola_next)
```

Le pas de taux `0.1` donne `0.6`, où la perte vaut `5.76` au lieu de `9.0`, comme dans le tableau. La position initiale reste inchangée : `step` rend une nouvelle valeur.

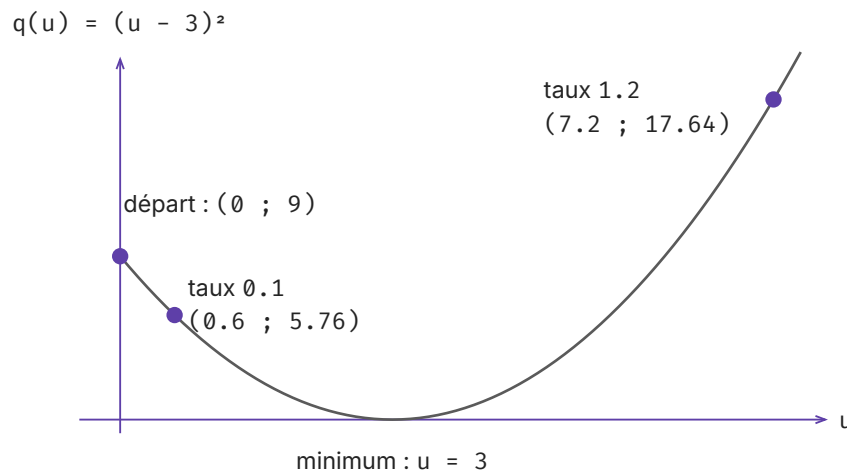
Avec plusieurs paramètres, chacun fait son propre pas avec sa dérivée partielle. En notant θ (la lettre thêta) la liste de tous les paramètres, la mise à jour s'écrit $\theta' = \theta - \eta \nabla L(\theta)$: le gradient indique la montée locale, nous le retranchons pour prendre l'autre direction. Pour le ticket de taxi, avec $\eta = 0.05$, $w' = 1 - 0.05 \times (-12) = 1.6$ et $b' = 4 - 0.05 \times (-6) = 4.3$. La nouvelle prédiction vaut $1.6 \times 2 + 4.3 = 7.5$, l'écart -1.5 , et la perte `2.25` au lieu de `9`.

Pour nos quatre coins, θ est le triplet (w_1, w_2, b) et ∇L ses trois dérivées. Avec $\eta = 0.1$, w_1 passe de `0.3` à $0.3 - 0.1 \times (-3.4) = 0.64$; les nouveaux poids sont `[0.64, 0.04]` et le biais `0.1`. Ils se sont rapprochés des paramètres attendus. Dans le TP, `grad` dérive la perte par rapport à la map `%{weights: ..., bias: ...}` tout entière et rend une map de même structure, qui contient les dérivées du poids et du biais ; coordonnées et cibles restent fixes. Ce premier pas fait passer la MSE de `4.58` à `2.9312`.

Repartons de la parabole, à la même position, avec un taux douze fois plus grand :

```
1 parabola_too_far = Descent.step(parabola_start, 1.2)
2 parabola_too_far_value = Parabola.value(parabola_too_far)
```

Le taux `1.2` donne `7.2`, où la perte monte à `17.64` :



Points calculés sur la même parabole, depuis $u = 0$. Le grand pas dépasse le minimum et augmente la perte.

Le calcul de la distance au minimum l'explique. Avec un taux η , le même raisonnement que plus haut donne $u' - 3 = (1 - 2\eta)(u - 3)$: chaque pas multiplie cette distance par $1 - 2\eta$. Pour $\eta = 0.1$, ce facteur vaut 0.8. Pour $\eta = 1.2$, il vaut -1.4 : le signe moins signale que le pas franchit le minimum, et la valeur 1.4 qu'il atterrit plus loin qu'au départ, à 4.2 du minimum au lieu de 3.

Sur le modèle affine du TP, ce taux fait aussi monter la MSE, de 4.58 à 8.9768. Le taux utile dépend du problème et de la règle de mise à jour, l'*optimiseur*, dont nous reparlerons avec Axon.

PIÈGE

Un pas trop grand peut augmenter la perte ; un pas très petit peut ne pas suffire dans le budget disponible. La descente ne garantit pas en général le meilleur modèle : comparez les mesures.

8.1.5 Répéter en transportant les paramètres

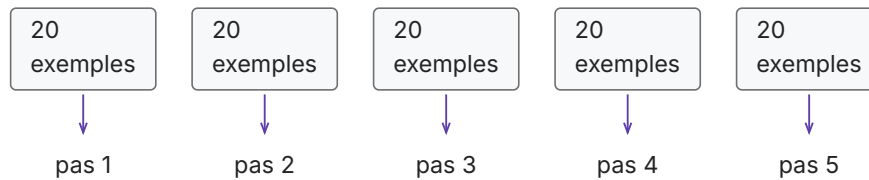
Pourquoi faut-il répéter ? Chaque pas n'utilise qu'une information locale, la pente à l'endroit où l'on se trouve. Elle n'indique pas où est le minimum, seulement dans quel sens descendre ; il faut donc recalculer la pente à chaque nouvelle position. L'accumulateur du chapitre 4 revient : à chaque tour, il contient la position courante. Reprenons notre parabole depuis la position initiale.

```
1 parabola_final = Enum.reduce(1..80, parabola_start,
2   fn _, position -> Descent.step(position, 0.1) end)
3 parabola_final_value = Parabola.value(parabola_final)
```

Après quatre-vingts pas, la position est proche de 3.0 : le facteur 0.8, appliqué quatre-vingts fois, réduit la distance initiale à moins de 10^{-7} , et la valeur de la parabole est inférieure à 10^{-8} . Le TP répète le même principe avec ses trois paramètres : les poids approchent $[2.0, 1.0]$, le biais 0.5, et la MSE passe aussi sous 10^{-8} . Cette réussite attendue donne un point de comparaison lorsque nous changerons les cibles.

Un *pas* met à jour les paramètres après un lot d'exemples ; une *époque* parcourt tous les exemples d'apprentissage. Dans notre TP, le lot contient tous les exemples, donc un pas fait une époque. Sur de grands jeux de données, on découpe les exemples en plusieurs lots : chaque pas coûte moins cher à calculer, et l'on fait plusieurs mises à jour par passage.

une époque : parcourir les 100 exemples



Un pas modifie les paramètres après un lot. Ici, une époque fait cinq pas.

Les cinq lots appartiennent au même parcours des données ; chaque flèche mène à une mise à jour des paramètres.

À RETENIR

Apprendre, c'est répéter quatre étapes : prédire avec les paramètres courants, mesurer la perte, calculer son gradient, puis retrancher η fois ce gradient. La dérivée de la perte par rapport à un paramètre dit comment la perte réagit quand ce paramètre augmente un peu ; le signe moins fait descendre, et le taux η règle la longueur du pas, qu'un excès fait passer par-dessus le minimum.

Les notations de cette section et de la précédente se répondent ainsi :

Notation	Sens
$\sum_{i=1}^n a_i$	somme $a_1 + a_2 + \dots + a_n$; l'indice i parcourt les exemples
$x_i, y_i, x_{i,j}$	le i -ième exemple, sa réponse attendue, sa coordonnée numéro j
$s(x), L$	score d'un point ; perte, qui dépend des paramètres
$q'(u), \frac{dq}{du}$	dérivée de q par rapport à u : pente, sensibilité
$\frac{\partial L}{\partial w}$	dérivée partielle : seul w varie, les autres paramètres sont gelés
∇L	gradient : le vecteur des dérivées partielles, une par paramètre
θ, η	tous les paramètres ; taux d'apprentissage, la longueur du pas
$\theta' = \theta - \eta \nabla L(\theta)$	un pas de descente de gradient

8.2 Une limite de représentation ne disparaît pas en attendant

Changeons seulement les réponses attendues : les mêmes quatre points doivent maintenant porter les classes 1, 0, 0, 1, celles de la figure du début du chapitre. Les paramètres initiaux, la formule, la perte, le taux et les quatre-vingts pas restent identiques.

Les quatre scores tendent vers 0.5, et la perte vers 0.25. La boucle ne s'est pas arrêtée par erreur. Additionnons en effet les scores des deux coins de classe 1, puis ceux des deux coins de classe 0 :

$$s(-1, -1) + s(1, 1) = (-w_1 - w_2 + b) + (w_1 + w_2 + b) = 2b,$$

$$s(-1, 1) + s(1, -1) = (-w_1 + w_2 + b) + (w_1 - w_2 + b) = 2b.$$

Les deux paires ont toujours la même somme, quels que soient les paramètres ! Or, pour bien classer avec un seuil t , il faudrait que les deux scores de classe 1 dépassent t et que les deux autres restent en dessous : la première somme dépasserait $2t$, la seconde resterait sous $2t$. C'est impossible. Avec $w_1 = w_2 = 1$ et $b = 0$, par ex., les coins de classe 1 reçoivent les scores -2 et 2 , ceux de classe 0 reçoivent 0 et 0 : aucun seuil ne place -2 et 2 d'un côté, et les deux 0 de l'autre.

Sur le dessin, la situation est tout aussi claire. Les points où le score vaut exactement t vérifient $w_1 x_1 + w_2 x_2 + b = t$: c'est l'équation d'une droite, la *frontière de décision*, et un seuil sur un score affine coupe toujours le plan en deux demi-plans. Or, sur la figure, les deux points pleins sont les extrémités d'une diagonale du carré, les deux cercles celles de l'autre, et ces diagonales se coupent au centre. Une droite qui laisse les deux points pleins d'un même côté y laisse aussi tout le segment qui les joint, donc le centre ; pour la même raison, le centre devrait se trouver du côté des cercles. Aucune droite ne convient.

Un calcul plus précis explique la valeur 0.25 . Sur ces quatre points symétriques, développer la MSE donne exactement

$$L(w_1, w_2, b) = w_1^2 + w_2^2 + (b - 0.5)^2 + 0.25.$$

Un carré n'est jamais négatif : la perte vaut donc au moins 0.25 , et elle atteint cette valeur en $w_1 = w_2 = 0$ et $b = 0.5$, où les quatre scores valent 0.5 . Davantage de pas ne feraient pas disparaître le dernier terme, qui ne dépend d'aucun paramètre. Ce calcul isole une limite de la famille de fonctions choisie, alors que le grand taux de la section précédente illustrait un problème d'optimisation.

POUR ALLER PLUS LOIN

Le développement de la perte tient en trois faits. Sur les quatre coins, $\sum x_1 = \sum x_2 = \sum x_1 x_2 = 0$: les termes croisés disparaissent et $\sum s(x_i)^2 = 4(w_1^2 + w_2^2 + b^2)$. Les cibles non nulles sont aux coins $(-1, -1)$ et $(1, 1)$, où s vaut $-w_1 - w_2 + b$ puis $w_1 + w_2 + b$, d'où $\sum y_i s(x_i) = 2b$; enfin $\sum y_i^2 = 2$. En divisant par quatre, $L = w_1^2 + w_2^2 + b^2 - b + \frac{1}{2}$, ce qui est bien la forme annoncée. Cette limite d'un seuil sur une somme pondérée a été rendue célèbre par Marvin Minsky et Seymour Papert dans *Perceptrons* (MIT Press, 1969).

À RETENIR

Un résultat médiocre peut venir d'une erreur de code, d'un entraînement mal réglé, d'exemples insuffisants ou d'un modèle qui ne peut pas représenter la règle. Ces explications appellent des essais différents. La comparaison affine/classes alternées permet ici de séparer deux de ces soucis.

Nous pourrions ajouter à la main la coordonnée $x_1 x_2$: un seuil sur cette nouvelle entrée résoudrait le problème. Ce serait une transformation choisie avec notre connaissance de

la règle. Nous allons plutôt apprendre une transformation intermédiaire en ajoutant une couche cachée.

8.3 Garder des exemples pour poser une autre question

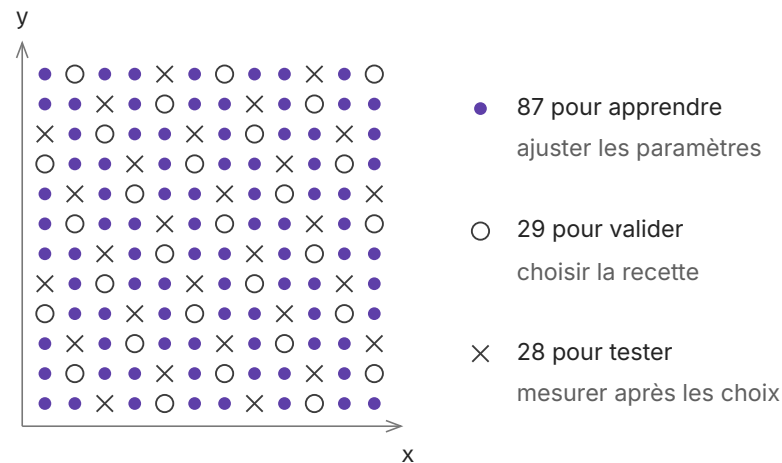
Les quatre coins étaient utiles pour contrôler les calculs à la main. Ils ne disent rien, en revanche, de la capacité du modèle à *généraliser*, c'est-à-dire à bien répondre sur des exemples qu'il n'a jamais vus pendant l'apprentissage. C'est pourtant la seule chose qui compte à l'usage : le tarif de taxi ne sert que pour les courses à venir. Pour la suite, nous préparons une grille de 144 points, dont les coordonnées évitent les axes. Une répartition fixée avant les essais réserve des observations distinctes pour trois usages.

```
1 grid = for i <- 0..11, j <- 0..11 do
2   x = (i + 0.5) / 6 - 1
3   y = (j + 0.5) / 6 - 1
4   %{x: x, y: y,
5     label: if(x * y > 0, do: 1, else: 0),
6     split: rem(7 * i + 11 * j, 5)}
7 end
8
9 train_rows = Enum.filter(grid, &(61.split < 3))
10 validation_rows = Enum.filter(grid, &(61.split == 3))
11 test_rows = Enum.filter(grid, &(61.split == 4))
```

Le reste modulo 5 d'une combinaison des deux indices répartit les points de façon déterministe et entrelacée, sans tirage aléatoire : trois restes sur cinq vont à l'apprentissage, soit environ 60 %, un à la validation et un au test, et chaque jeu garde les deux classes, 44 et 43, 14 et 15, 14 et 14. Les trois jeux contiennent respectivement 87, 29 et 28 exemples. Leurs rôles sont différents, même si les observations viennent de la même grille. Le jeu d'*apprentissage*, souvent appelé le *train*, sert à ajuster les paramètres ; le jeu de *validation* sert à comparer des *recettes*, c'est-à-dire des ensembles de choix faits avant l'entraînement, comme la taille du modèle ou le taux ; le jeu de *test* mesure une seule fois la recette retenue.

Jeu	Usage dans l'expérience
Apprentissage	Calculer les gradients et modifier les paramètres.
Validation	Comparer les recettes et décider des choix d'entraînement.
Test	Mesurer la recette retenue, après avoir figé les choix.

Chaque point appartient à un seul jeu. Voici la répartition exacte du code ci-dessus ; les symboles indiquent l'usage des points, et non leur classe à prédire.



Les points pleins servent à ajuster, les cercles à choisir, les croix à mesurer après les choix. Les trois jeux sont distincts mais voisins dans la grille : cette expérience évalue une interpolation dans le même domaine.

Pensez aux annales d'un concours. Un élève qui apprendrait par cœur les corrigés des annales obtiendrait des notes parfaites sur ces annales, sans rien savoir du sujet de l'année. Pour choisir sa façon de réviser, il passe plutôt un concours blanc, sur des sujets qu'il n'a pas travaillés : c'est la validation. Le concours lui-même, passé une seule fois, joue le rôle du test.

Un modèle peut tomber dans le même travers. Imaginons un « modèle » qui retient simplement les 87 points du train avec leur classe : il donne 100 % de bonnes réponses sur le train, mais que répond-il pour un point qu'il n'a jamais vu ? Ce cas extrême porte un nom, le *surapprentissage* : le modèle ajuste de mieux en mieux les particularités des exemples d'apprentissage, sans bénéfice, voire avec une perte, sur les exemples réservés. Seul un score mesuré sur des exemples réservés permet de le voir.

Un score d'apprentissage répond donc à « retrouve-t-on les réponses des exemples utilisés pour ajuster les paramètres ? » ; un score sur des observations réservées répond à une autre question. Pourquoi ne pas choisir la meilleure recette directement sur le test ? Parce que dès qu'un jeu sert à choisir, il joue le rôle d'une validation, quel que soit le nom de son fichier. Nous réservons donc le test jusqu'à la fin de la comparaison.

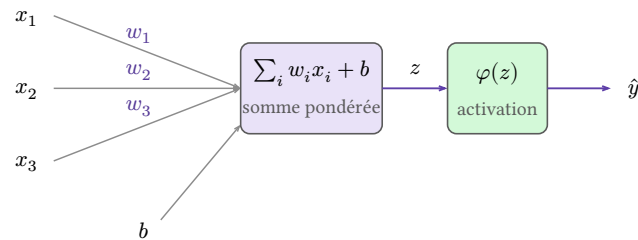
PIÈGE

La séparation concerne aussi la préparation. Un centre, une échelle ou un dictionnaire appris sur l'ensemble des données peut faire entrer une information du test dans la recette. Calculez ces transformations sur le train, puis appliquez-les aux autres jeux. Des copies presque identiques du même exemple, ou des mesures répétées d'une même personne, peuvent également rendre une séparation ligne par ligne trop optimiste.

8.4 Apprendre une transformation avec Axon

Un *réseau de neurones* n'est rien d'autre qu'un empilement de petites formules affines comme celle que nous venons d'entraîner, séparées par des fonctions non linéaires. Chaque *neurone*, ou *unité*, reçoit des valeurs, en fait une somme pondérée à laquelle il ajoute un biais,

exactement comme notre score, puis applique au besoin une *activation*, c'est-à-dire une fonction d'un seul nombre, choisie d'avance. La somme seule est affine ; c'est l'activation non linéaire qui change la famille de fonctions représentables, et c'est précisément ce qui manquait à notre modèle pour séparer les classes alternées.



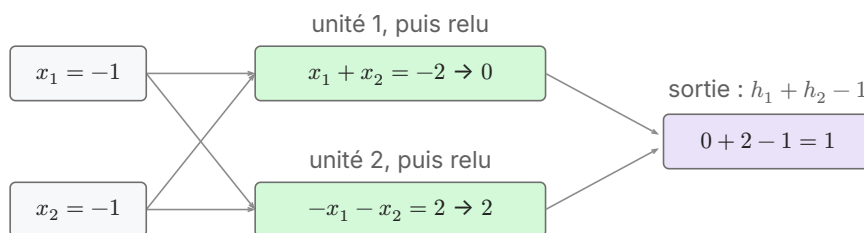
Une unité reçoit des entrées pondérées et un biais. Une activation peut transformer la somme obtenue. Chaque unité a ses propres paramètres.

Le dessin reprend les notations du chapitre 7 pour le score : z désigne la somme pondérée $\sum_i w_i x_i + b$, la lettre grecque φ (phi) une activation quelconque, et $\hat{y} = \varphi(z)$, lu « y chapeau », la prédiction, que le chapeau distingue de la réponse attendue y .

8.4.1 Une couche dense calcule plusieurs sommes pondérées

Une *couche dense* de k unités calcule k sommes pondérées des mêmes entrées, chacune avec ses propres poids et son propre biais. Rangeons les poids de chaque unité dans une colonne : pour deux entrées, nous obtenons une matrice W de forme $\{2, k\}$. Le produit d'un lot X de forme $\{n, 2\}$ par W calcule alors d'un coup les k sommes de chaque exemple, dans une matrice $\{n, k\}$: c'est le produit matrice-vecteur du chapitre 7, répété pour chaque colonne de W . On l'appelle *produit matriciel* ; il s'écrit $x @ w$ en NumPy et $Nx.dot(x, w)$ en Nx.

Construisons à la main une couche de deux unités qui résout les quatre coins. La première unité calcule $z_1 = x_1 + x_2$, avec les poids $(1, 1)$, la seconde $z_2 = -x_1 - x_2$, avec les poids $(-1, -1)$, toutes deux sans biais. Chacune applique ensuite l'activation la plus simple, *relu*, qui remplace un nombre négatif par 0 et garde un nombre positif : $\text{relu}(z) = \max(0, z)$. Une unité de sortie additionne enfin les deux valeurs obtenues, h_1 et h_2 , et retranche 1 : ses poids valent $(1, 1)$ et son biais -1 .



Le calcul pour le coin $(-1, -1)$. Chaque unité cachée fait sa somme pondérée, puis relu ; la sortie combine les deux résultats avec les poids 1 et 1 et le biais -1 .

Coin	z_1	z_2	h_1	h_2	Sortie	Classe
$(-1, -1)$	-2	2	0	2	1	1
$(-1, 1)$	0	0	0	0	-1	0
$(1, -1)$	0	0	0	0	-1	0
$(1, 1)$	2	-2	2	0	1	1

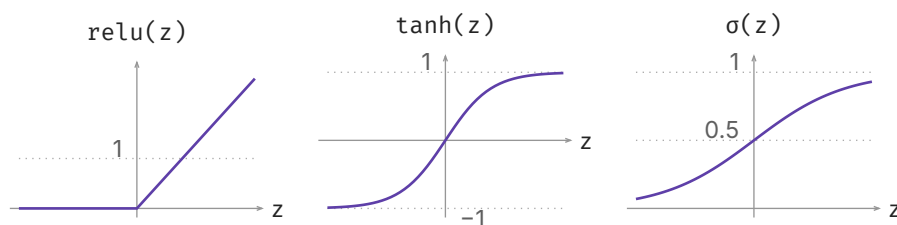
En décidant la classe 1 lorsque la sortie est positive, les quatre coins sont bien classés ! La couche cachée a calculé $h_1 + h_2 = |x_1 + x_2|$, la valeur absolue de $x_1 + x_2$, qui n'est pas une fonction affine. Vérifions avec Nx, en rangeant les poids des deux unités en colonnes :

```
1 w1 = Nx.tensor([[1.0, -1.0], [1.0, -1.0]])
2 hidden = Nx.max(Nx.dot(corners, w1), 0)
3 w2 = Nx.tensor([[1.0], [1.0]])
4 scores = Nx.add(Nx.dot(hidden, w2), -1.0)
5 linear_scores =
6     Nx.add(Nx.dot(Nx.dot(corners, w1), w2), -1.0)
```

`Nx.dot(corners, w1)` a la forme $\{4, 2\}$: deux sommes pondérées par coin. `Nx.max(..., 0)` applique relu case par case, et `scores` vaut $[[1.0], [-1.0], [-1.0], [1.0]]$, la colonne du tableau. Sans relu, en revanche, `linear_scores` vaut -1.0 sur les quatre lignes : les deux unités se compensent exactement, puisque $z_1 + z_2 = 0$, et la couche cachée n'apporte plus rien.

Ce n'est pas un hasard. Sans activation, deux couches affines se composent en une seule. Notons W_1, b_1 les poids et les biais de la première couche, W_2, b_2 ceux de la seconde : $W_2(W_1x + b_1) + b_2$ se développe en $(W_2W_1)x + (W_2b_1 + b_2)$, qui reste affine en x . Les mathématiciens écrivent ainsi le point x en colonne, à droite des matrices ; Nx range le lot en lignes et multiplie à droite par W , mais le raisonnement est le même. Empiler des couches sans activation ne fait donc jamais sortir de la famille affine, et le seuil final reste une droite.

L'activation doit donc être non linéaire. Trois activations reviendront souvent :



relu garde les nombres positifs et remplace les négatifs par 0 ; $\tanh$ écrase tout nombre entre -1 et 1 ; la sigmoïde σ écrase tout nombre entre 0 et 1, et vaut 0.5 en 0.

$\tanh$, la tangente hyperbolique, vaut à peu près z près de 0, puis s'arrondit sans jamais dépasser -1 ou 1 : $\tanh(0) = 0$, $\tanh(1) \approx 0.762$, $\tanh(2) \approx 0.964$. Notre couche à la main ne vaut que pour les quatre coins : sur la grille complète, $|x_1 + x_2| > 1$ n'est pas la règle $x_1x_2 > 0$, puisque le point $(0.1, 0.1)$ est de classe 1 avec $|x_1 + x_2| = 0.2$. Plutôt que de chercher des poids à la main, nous allons les faire apprendre, avec davantage d'unités et l'activation $\tanh$.

POUR ALLER PLUS LOIN

Avec assez d'unités, une seule couche cachée suivie d'une sortie affine peut approcher d'aussi près que l'on veut toute fonction continue sur un domaine borné : c'est le *théorème d'approximation universelle*, démontré par George Cybenko en 1989 pour la sigmoïde. Il garantit l'existence de bons poids, pas qu'un entraînement les trouve. Voir Goodfellow, Bengio et Courville, *Deep Learning*³³ (MIT Press, 2016), section 6.4.1.

La *sigmoïde* (`sigmoid` dans Axon) sert à la sortie. Elle vaut $\sigma(z) = \frac{1}{1+e^{-z}}$, où σ est un sigma minuscule, à ne pas confondre avec le $\sum$ des sommes. Elle transforme un score quelconque en un nombre entre 0 et 1 :

z	-4	-2	0	2	4
$\sigma(z)$	0.018	0.119	0.5	0.881	0.982

Nous lisons ce nombre comme la probabilité que le point soit de classe 1, et nous décidons la classe 1 lorsqu'il atteint 0.5. Ce seuil correspond à un score nul, puisque $\sigma(0) = 0.5$ et que la sigmoïde est croissante : la sigmoïde ne déplace pas la frontière, elle la rend lisible comme une probabilité. Sans activation cachée, la sigmoïde finale ne suffit donc pas à obtenir une frontière non linéaire : son seuil revient encore à celui d'une combinaison affine.

8.4.2 Décrire le réseau avec Axon

Nous pourrions écrire tout le réseau en Nx, avec sa perte et son pas de gradient, comme nous venons de le faire pour deux unités. Axon, une bibliothèque construite sur Nx, s'en charge : il crée les matrices de poids et les biais avec les bonnes formes, tire leurs valeurs initiales au hasard, enchaîne prédiction, perte, gradient et mise à jour pour chaque lot et chaque époque, puis fournit une fonction de prédiction. Le calcul reste un calcul Nx ; Axon l'organise.

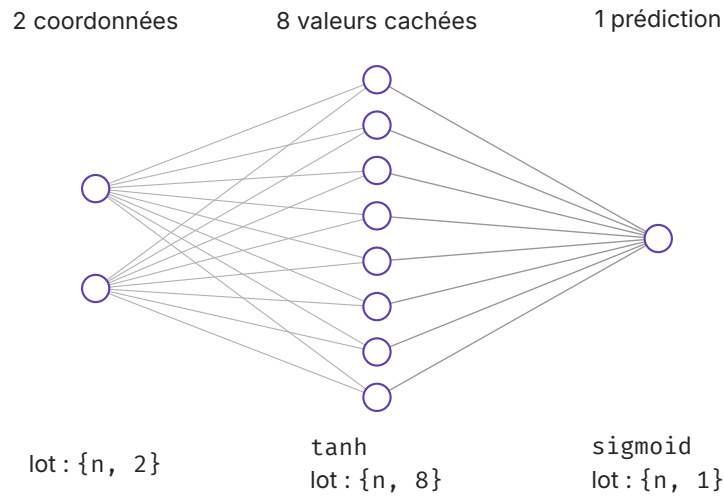
Pour huit unités cachées, chaque point de deux coordonnées devient un vecteur de huit valeurs. L'activation `tanh` transforme chacune de ces valeurs, puis une autre couche dense, d'une seule unité, produit un score ; la sigmoïde le transforme enfin en probabilité de classe 1. Avec Axon, ce réseau s'écrit comme un pipe de couches :

```
1 model = Axon.input("points", shape: {nil, 2})
2   |> Axon.dense(8, activation: :tanh)
3   |> Axon.dense(1, activation: :sigmoid)
```

`Axon.input` déclare l'entrée, nommée "points" ; dans sa forme, `nil` laisse varier le nombre d'exemples. `Axon.dense(couche, k, activation: ...)` ajoute une couche dense de `k` unités et applique l'activation à ses sorties. Les formes successives sont `{n, 2}`, `{n, 8}`, puis `{n, 1}`. Comptons les paramètres : la première couche a une matrice de poids de forme `{2, 8}`, soit 2×8 poids, et huit biais ; la seconde, une matrice `{8, 1}`, soit huit poids, et un biais. Le total vaut 33. Axon nomme `kernel` chaque matrice de poids et `bias` chaque vecteur de biais. Les valeurs des activations ne sont pas des paramètres : elles sont recalculées pour chaque entrée.

³³ <https://www.deeplearningbook.org/>

Suivons un seul point à travers le réseau. Un cercle est une valeur calculée, et chaque lien porte un poids ; les mêmes paramètres servent à toutes les lignes du lot.



Deux coordonnées alimentent huit unités, puis une sortie. Les 24 liens représentent les poids ; les neuf biais, un par unité cachée ou de sortie, sont omis pour garder le dessin lisible. n est le nombre d'exemples du lot.

Le mot *cachée* signifie simplement que cette couche est intermédiaire : nous ne fournissons pas huit cibles pour ses huit unités, et les deux couches sont ajustées ensemble à partir de la perte finale.

8.4.3 Une perte de classification et un optimiseur

Avec des classes 0/1, nous utilisons désormais l'*entropie croisée binaire* (BCE, pour *binary cross-entropy*). Pour une probabilité p de classe 1 et une étiquette y , elle vaut

$$\ell(p, y) = -(y \ln(p) + (1 - y) \ln(1 - p)).$$

La lettre ℓ , un l arrondi, désigne la perte d'un seul exemple. La formule paraît compliquée, mais une seule de ses deux moitiés sert à chaque fois, puisque y vaut 0 ou 1. Si la vraie classe est 1, il reste $-\ln(p)$; si c'est 0, il reste $-\ln(1 - p)$, où $1 - p$ est la probabilité annoncée pour la classe 0. Dans les deux cas, la perte vaut moins le logarithme de la probabilité donnée à la bonne réponse. Rappelons que $\ln$, le logarithme népérien, vaut 0 en 1 et devient de plus en plus négatif quand on s'approche de 0 ; $-\ln$ est donc nul pour une réponse sûre et juste, et grandit sans limite quand la bonne réponse reçoit une probabilité proche de 0. Un modèle qui répond 0.5 partout, sans rien distinguer, a une perte de $\ln 2 \approx 0.693$: c'est un repère utile pour lire une perte BCE.

proba. de la bonne classe	0.99	0.9	0.6	0.5	0.1	0.01
perte $-\ln$	0.010	0.105	0.511	0.693	2.303	4.605

Si la vraie classe est 1, annoncer $p = 0.9$ coûte environ 0.105, annoncer $p = 0.6$ environ 0.511, et annoncer $p = 0.1$ environ 2.303. Les deux premières prédictions conduisent à la même décision, et la perte les distingue pourtant d'un facteur cinq : elle mesure la confiance, pas seulement la décision. Surtout, elle punit durement une erreur sûre d'elle : annoncer $p = 0.99$ pour un point de classe 0 ne laisse que 0.01 à la bonne réponse et coûte environ

4.6, alors que la MSE de cette même réponse, $0.99^2 \approx 0.98$, ne dépasserait jamais 1. Nous n'aurons pas à dériver cette formule : Axon s'en charge, comme `grad` le faisait pour la MSE. La perte d'un lot est la moyenne de ℓ sur ses exemples, comme pour la MSE. Nous utilisons la fonction fournie par Axon, qui évite les logarithmes de zéro. L'optimiseur, la règle de mise à jour, change lui aussi : *Adam*, fourni par la bibliothèque Polaris, conserve un historique des gradients pour adapter la longueur de chaque pas, au lieu de notre taux fixe. Il a donc un état en plus des poids, sans changer l'ordre des opérations : prédire, mesurer une perte, dériver, mettre à jour.

POUR ALLER PLUS LOIN

L'entropie croisée n'est pas un choix arbitraire : c'est moins le logarithme de la probabilité que le modèle attribue aux étiquettes observées, et la minimiser revient à rendre ces étiquettes les plus probables possible, le principe du *maximum de vraisemblance*. Composée avec la sigmoïde, sa dérivée par rapport au score z vaut simplement $p - y$ (Goodfellow, Bengio et Courville, *Deep Learning*, section 6.2). Adam, lui, est décrit par Diederik Kingma et Jimmy Ba dans *Adam: A Method for Stochastic Optimization*³⁴ (2014).

Préparons les tenseurs à partir des observations déjà séparées. L'ordre des lignes est conservé pour les entrées comme pour les étiquettes.

```
1 tensors = fn rows ->
2   x = Enum.map(rows, fn p -> [p.x, p.y] end)
3   y = Enum.map(rows, fn p -> [p.label] end)
4   {Nx.tensor(x, type: :f32), Nx.tensor(y, type: :f32)}
5 end
6 {train_x, train_y} = tensors.(train_rows)
7 {validation_x, validation_y} = tensors.(validation_rows)
```

La boucle d'entraînement suivante reçoit seulement les 87 observations du train. La liste `[data]` contient un unique lot : ici encore, une époque correspond à un pas. Avec plusieurs lots, Axon les parcourrait avant de passer à l'époque suivante.

```
1 fit = fn model, data, rate ->
2   optimizer = Polaris.Optimizers.adam(learning_rate: rate)
3   model
4   |> Axon.Loop.trainer(:binary_cross_entropy,
5     optimizer, seed: 42, log: 0)
6   |> Axon.Loop.run([data], Axon.ModelState.empty(),
7     epochs: 200)
8 end
9 trained = fit.(model, {train_x, train_y}, 0.03)
```

`Axon.Loop.trainer` assemble le modèle, la perte et l'optimiseur en une *boucle* d'entraînement ; `log: 0` coupe l'affichage de sa progression. `Axon.Loop.run` exécute cette boucle sur la liste de lots, pendant 200 époques, à partir de l'état vide

³⁴ <https://arxiv.org/abs/1412.6980>

`Axon.ModelState.empty()`. `model` décrit le calcul ; `trained` contient l'état produit par la boucle : chaque essai repart de nouveaux paramètres, et non de la fin de l'entraînement précédent.

Axon tire en effet les poids initiaux au hasard, et la *graine* (`seed`) fixe ce tirage pour repartir du même point à chaque essai ; avec une autre largeur de couche, les poids changent de forme, et la même graine ne donne plus les mêmes paramètres. Distinguez bien deux familles de nombres : les poids et les biais sont les paramètres, ajustés par l'entraînement ; la largeur, le choix de l'activation, le taux, le nombre d'époques et la graine sont des *hyperparamètres*, choisis avant et consignés dans la recette de chaque essai, comme le font les carnets d'AP5.

8.4.4 Prédire n'est plus entraîner

Une fois les poids appris, calculer une réponse n'utilise ni étiquette ni mise à jour.

`Axon.build(model)` rend deux fonctions, l'une qui initialise les paramètres et l'autre qui prédit ; nous gardons la seconde, à laquelle nous passons les paramètres appris, rangés dans `trained.step_state.model_state`, et les nouvelles entrées.

```
1 {_, predict} = Axon.build(model)
2 probabilities = predict.(trained.step_state.model_state,
3   validation_x)
```

Le résultat a la forme `{29, 1}`. Chaque valeur est la sortie de la sigmoïde pour un point de validation. Pour afficher une classe, nous appliquons le seuil `0.5`. L'*exactitude* (*accuracy*) compte la proportion de décisions correctes ; la perte tient aussi compte des probabilités qui ont conduit à ces décisions. Dans la fonction ci-dessous, `Nx.greater_equal(p, 0.5)` rend un tenseur de 1 et de 0, la classe décidée pour chaque ligne ; `Nx.equal` le compare aux étiquettes, case par case, et rend 1 pour chaque bonne réponse ; la moyenne de ces 1 et de ces 0 est la proportion de bonnes réponses.

```
1 accuracy = fn probabilities, expected ->
2   probabilities
3   |> Nx.greater_equal(0.5)
4   |> Nx.equal(expected)
5   |> Nx.mean()
6   |> Nx.to_number()
7 end
8 validation_accuracy = accuracy.(probabilities,
9   validation_y)
10 validation_loss = Axon.Losses.binary_cross_entropy(
11   validation_y, probabilities, reduction: :mean)
12 |> Nx.to_number()
```

Sur ces 29 exemples, corriger une seule erreur change l'exactitude d'environ 3,4 points de pourcentage : un score se lit toujours avec son effectif.

Voici les pièces d'Axon et de Polaris que nous avons assemblées :

Écriture	Rôle
<code>Axon.input("points", shape: {nil, 2})</code>	l'entrée : un lot de points à deux coordonnées
<code>Axon.dense(couche, k, activation: :tanh)</code>	une couche dense de <code>k</code> unités, puis une activation
<code>Polaris.Optimizers.adam(learning_rate: r)</code>	l'optimiseur Adam, de taux de départ <code>r</code>
<code>Axon.Loop.trainer(model, perte, optimiseur)</code>	assemble la boucle d'entraînement
<code>Axon.Loop.run(boucle, lots, état, epochs: e)</code>	exécute <code>e</code> époques et rend l'état final
<code>état.step_state.model_state</code>	les paramètres appris, poids (<code>kernel</code>) et biais (<code>bias</code>)
<code>Axon.build(model)</code>	rend les fonctions d'initialisation et de prédiction
<code>Axon.Losses.binary_cross_entropy</code>	l'entropie croisée binaire d'un lot

À RETENIR

Un réseau dense enchaîne des couches ; chaque unité fait une somme pondérée de ses entrées, ajoute son biais, puis applique une activation. Sans activation non linéaire entre les couches, le réseau reste affine et ne sépare pas les classes alternées. La sigmoïde de sortie donne une probabilité de classe 1, que l'entropie croisée compare à l'étiquette en punissant surtout les erreurs sûres d'elles. Axon crée les paramètres et orchestre la boucle ; le calcul reste celui de Nx.

8.5 Construire une comparaison que l'on peut expliquer

Nous avons d'abord vérifié un gradient sur une règle affine ; nous mesurons maintenant un classifieur à 33 paramètres. Pour comprendre ce qui a changé, fixons le protocole : mêmes données séparées, même graine, même budget de 200 époques, même BCE et même optimiseur Adam. Nous comparons ensuite une seule modification à la fois.

Le premier contre-exemple supprime l'activation cachée tout en conservant sa largeur. Le second garde `tanh`, mais diminue le taux d'apprentissage. Prédisez la nature du souci attendu dans chaque cas avant de calculer.

```

1 affine_model = Axon.input("points", shape: {nil, 2})
2   |> Axon.dense(8)
3   |> Axon.dense(1, activation: :sigmoid)
4
5 affine_state = fit.(affine_model,
6   {train_x, train_y}, 0.03)
7 slow_state = fit.(model, {train_x, train_y}, 0.0001)
```

Voici les mesures obtenues avec les versions et la graine de ce chapitre. Le nombre de réponses correctes accompagne le pourcentage afin de garder l'effectif visible. Ces résultats servent à lire la comparaison, pas à fixer un score minimal pour vos expériences.

Recette	Validation	Exactitude
tanh, taux 0.03	27 / 29	93,1 %
Sans activation, taux 0.03	17 / 29	58,6 %
tanh, taux 0.0001	14 / 29	48,3 %

Sans activation cachée, le modèle ne dispose toujours que d'une frontière affine : deux couches affines se composent en une seule, et le seuil de la sigmoïde reste alors une droite. Avec un taux très petit, la famille est capable de représenter des frontières plus riches, mais le budget peut être insuffisant pour l'apprendre. Il faut distinguer la capacité de la famille, l'état particulier atteint par l'optimiseur et la qualité des exemples.

Pour chaque essai, nous conservons la recette et quatre mesures : perte et exactitude sur le train, puis sur la validation. Un score train élevé ne remplace pas une mesure de validation, et comparer la MSE du premier modèle à la BCE du second n'aiderait pas à décider lequel est meilleur.

Le surapprentissage se repère précisément à ces quatre mesures : la perte du train continue de baisser, celle de la validation stagne ou remonte. Sur notre grille propre et régulière, il n'a rien de garanti ; pour le provoquer, on peut réduire le train ou y ajouter du bruit, en gardant la validation inchangée.

Une graine reproduit un départ dans un environnement donné ; elle ne dispense pas de plusieurs essais pour conclure sur une tendance, et les versions des bibliothèques font partie de la recette conservée.

8.5.1 Regarder une erreur particulière

Une moyenne indique où chercher ; elle ne raconte pas chaque cas. Associons les réponses du modèle aux points de validation.

```

1 errors = Enum.zip(validation_rows,
2   Nx.to_flat_list(probabilities))
3   |> Enum.filter(fn {point, probability} ->
4     predicted = if probability >= 0.5, do: 1, else: 0
5     predicted != point.label
6   end)

```

Chaque paire indique les coordonnées, la vraie classe et la probabilité prédite. Les erreurs sont-elles près d'un axe, là où la classe change ? Le modèle hésite-t-il, ou se trompe-t-il avec aplomb ? Ces observations aident à formuler la prochaine hypothèse.

Dessiner les prédictions sur une grille plus fine montre la frontière estimée ; c'est de l'*inférence*, le calcul de prédictions avec des paramètres figés, et non un score. Quand la comparaison est terminée, nous choisissons une recette selon un critère annoncé, puis nous la mesurons une seule fois sur le test réservé.

À RETENIR

Un compte rendu utile relie une hypothèse à une recette, des mesures et une interprétation. « Sans $\tanh$, les deux couches restent affines ; dans ces essais, la validation reste mauvaise » est une conclusion que l'on peut discuter. « Le réseau marche » laisse de côté les données, le budget et les limites de ce qui a été observé.

8.6 Vérifier sa compréhension

Refaites sur papier les calculs du chapitre avec d'autres nombres. Prédisez chaque réponse avant de lire celle qui suit, puis vérifiez dans un carnet.

Premier exercice, un pas de gradient. Un ticket de taxi indique 3 km pour 11 €, et le modèle $w \times \text{km} + b$ part de $w = 2$ et $b = 4$. Calculez la prédiction, la perte, les deux dérivées partielles, puis un pas avec $\eta = 0.02$. Que se passerait-il avec $\eta = 0.1$?

La prédiction vaut $3 \times 2 + 4 = 10$, l'écart -1 et la perte 1. Les dérivées valent $2 \times (-1) \times 3 = -6$ pour w et $2 \times (-1) = -2$ pour b . Le pas donne $w = 2 + 0.02 \times 6 = 2.12$ et $b = 4 + 0.02 \times 2 = 4.04$; la prédiction devient 10.4, l'écart -0.6 et la perte 0.36. Avec $\eta = 0.1$, on obtiendrait $w = 2.6$ et $b = 4.2$, une prédiction de 12 et un écart de $+1$: la perte resterait à 1, car le pas aurait franchi le minimum.

Deuxième exercice, un réseau. Combien de paramètres compte le réseau ci-dessous ? Que pourrait-il encore représenter si l'on retirait relu ?

```
1 small_model = Axon.input("points", shape: {nil, 2})
2   |> Axon.dense(3, activation: :relu)
3   |> Axon.dense(1, activation: :sigmoid)
```

La couche cachée a une matrice de poids $\{2, 3\}$ et trois biais, soit $2 \times 3 + 3 = 9$ paramètres ; la sortie a trois poids et un biais, soit 4. Le total vaut 13. Sans relu, les deux couches se composent en une seule transformation affine : la frontière reste une droite, et les quatre coins restent inséparables, quel que soit l'entraînement.

Enfin, une question sans calcul : vous essayez trois largeurs de couche, gardez celle qui obtient la meilleure exactitude sur la validation, puis mesurez le test. Un camarade choisit directement la largeur qui réussit le mieux sur le test. Lequel des deux scores de test estime honnêtement le résultat sur de nouveaux points ? Le vôtre : le test de votre camarade a servi à choisir, il a joué le rôle d'une validation.

POUR EN PARLER

Une dérivée est une sensibilité : $q'(0) = -6$ signifie qu'augmenter u de 0.01 fait baisser q d'environ 0.06, ce que confirme la différence finie, -5.99 . Le gradient range une telle dérivée par paramètre, et `grad` l'obtient en multipliant les dérivées locales le long du graphe.

Un pas de descente retranche η fois le gradient : depuis $u = 0$, le taux 0.1 mène à 0.6, où la parabole vaut 5.76 au lieu de 9 ; le taux 1.2 franchit le minimum jusqu'à 7.2, où elle vaut 17.64.

Davantage de pas ne corrigent pas une famille trop pauvre : sur les quatre coins, les deux scores affines de classe 1 ont toujours la même somme, $2b$, que les deux autres. Une couche cachée avec une activation non linéaire lève cette limite ; sans activation, deux couches restent affines.

L'entropie croisée punit l'assurance dans l'erreur : donner 0.01 à la bonne classe coûte 4.6. Le train ajuste les paramètres, la validation choisit la recette, et le test, consulté une seule fois, mesure le choix figé.

8.7 Ce que les projets d'AP9 vont prolonger

Nous disposons maintenant de quatre briques réutilisables : une représentation numérique, un calcul paramétré, une perte et une manière de modifier les paramètres. Les quatre projets d'apprentissage d'AP9 les gardent et changent une seule chose à la fois. [micrograd](https://ap-2026.noce.fr/seances/9/micrograd/)³⁵ remplace N_x par un petit graphe de calculs scalaires dont vous écrivez les dérivées ; [l'image par coordonnées](https://ap-2026.noce.fr/seances/9/image-coordonnees/)³⁶ fait d'un réseau une fonction qui donne la couleur d'un pixel à partir de sa position ; [l'autoencodeur](https://ap-2026.noce.fr/seances/9/autoencodeur/)³⁷ comprime une entrée en un petit code puis la reconstruit ; le [petit Transformer](https://ap-2026.noce.fr/seances/9/petit-transformer/)³⁸ apprend à prédire le caractère suivant d'une séquence avec une tête d'attention. Chaque sujet introduit sa nouveauté au moment où elle devient nécessaire ; les questions de ces deux chapitres, formes, perte, train et validation, y reviennent toutes. Pour toute question, vous pouvez me contacter par email avec les formes des tenseurs et un exemple minimal.

³⁵ <https://ap-2026.noce.fr/seances/9/micrograd/>

³⁶ <https://ap-2026.noce.fr/seances/9/image-coordonnees/>

³⁷ <https://ap-2026.noce.fr/seances/9/autoencodeur/>

³⁸ <https://ap-2026.noce.fr/seances/9/petit-transformer/>

Faire coopérer des processus

SÉANCES AP6, AP7 ET AP8 · CHAPITRES 9 À 11

Comment faire coopérer des programmes qui s'exécutent en même temps ?

Un programme réel fait plusieurs choses à la fois : il attend le réseau, sert plusieurs utilisateurs et doit survivre à une panne. Cette dernière partie revient à ce pour quoi Erlang a été conçu, des processus légers et isolés, qui ne partagent pas leur mémoire et communiquent par messages.

Nous y distinguerons la concurrence du parallélisme, nous verrons ce que l'on gagne à ne rien partager, comment un processus garde un état et comment un superviseur le redémarre, puis comment plusieurs copies d'un même état, séparées par une coupure, finissent par se mettre d'accord. Les chapitres 9 à 11, un par séance, construisent ces idées pas à pas, avec des diagrammes à lire et des traces à suivre ; plusieurs projets d'AP9 les prolongent.

9	Processus, messages et état	125
10	Confier un état à un serveur	152
11	Répliquer un état et retrouver l'accord	178

Processus, messages et état

SÉANCE AP6 · PROCESSUS, MESSAGES, PANNES ET SUPERVISION

Jusqu'ici, nos programmes faisaient une seule chose à la fois. Or un serveur web répond à des centaines de visiteurs en même temps, et un central téléphonique, le problème d'origine d'Erlang, tient des milliers d'appels sans qu'une panne sur l'un coupe les autres.

La réponse de la BEAM est de découper un programme en *processus*, très nombreux et très légers, qui ne partagent aucune mémoire et ne communiquent que par *messages*. Un processus peut ainsi garder un état et répondre aux demandes des autres : c'est le principe du tableau de contributions des deux séances suivantes, et plus généralement de toute application Elixir.

Ce chapitre construit ces échanges à la main, pour que vous en voyiez le mécanisme : pourquoi mener plusieurs calculs à la fois, ce que l'on gagne à ne rien partager, comment créer un processus, lui écrire et attendre sa réponse, comment il garde un état, et ce qui se passe quand il tombe en panne. Le chapitre 10 confiera ensuite ces échanges à OTP, la bibliothèque standard d'Erlang pour les systèmes qui durent.

9.1 Plusieurs calculs à la fois

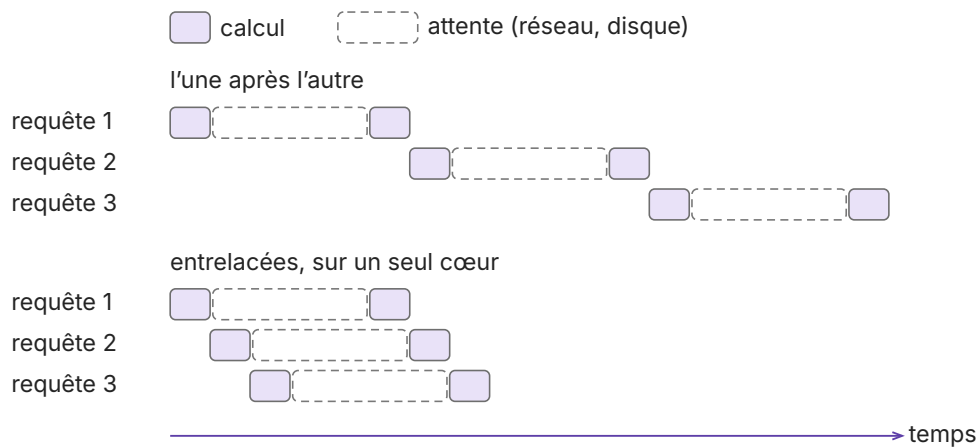
Avant de créer le moindre processus, demandons-nous pourquoi un programme voudrait mener plusieurs calculs à la fois. Il y a deux raisons, très différentes : ne pas rester inactif pendant une attente, et calculer plus vite en occupant plusieurs cœurs du processeur.

9.1.1 Pendant qu'une tâche attend

Un programme passe une grande partie de son temps à *attendre* : la réponse d'un serveur distant, la lecture d'un fichier sur le disque, la frappe d'un utilisateur. Pendant ces attentes, le processeur n'a rien à faire pour lui.

Imaginons un service web qui répond à trois visiteurs. Chaque réponse demande un court calcul, puis une longue attente de la base de données, puis un dernier court calcul. Si le service traite les trois demandes l'une après l'autre, le troisième visiteur attend la fin des deux premières, attentes comprises.

S'il commence au contraire la deuxième demande pendant que la première attend, puis la troisième, les attentes se recouvrent. Les trois réponses partent peu après la première, et pourtant le service n'utilise qu'un seul cœur !



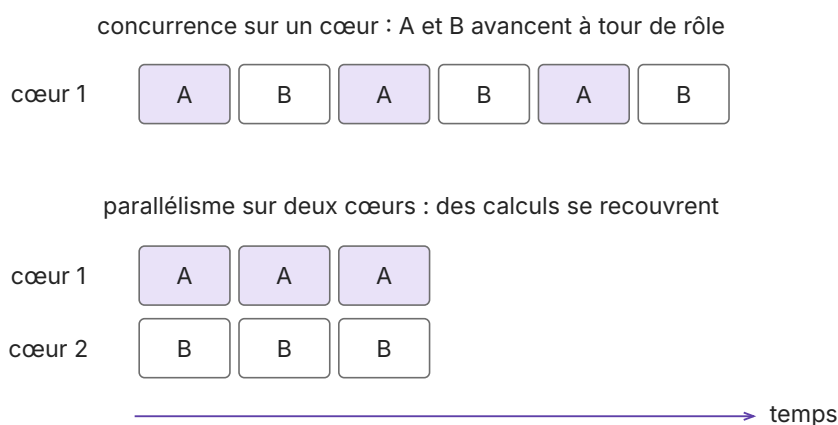
Chaque ligne suit une requête. Entrelacées, les trois requêtes se terminent bien plus tôt, sans que deux calculs aient jamais lieu au même instant. Les longueurs illustrent le principe ; ce ne sont pas des mesures.

Cette organisation s'appelle la *concurrency* : plusieurs tâches sont en cours sur une même période, et leurs étapes s'entrelacent. Elle ne demande pas plusieurs cœurs. Elle demande seulement que le programme soit découpé en tâches capables d'avancer indépendamment, et que quelqu'un choisisse, à chaque instant, laquelle avance.

9.1.2 Plusieurs cœurs qui calculent

Un processeur actuel compte plusieurs *cœurs* : autant d'unités qui suivent chacune le cycle lecture, décodage, exécution du chapitre 6, sur leurs propres instructions. Le *parallélisme* consiste à les faire calculer au même instant.

Il ne profite qu'aux tâches qui calculent. Pour produire deux images, sans aucune attente, l'entrelacement sur un cœur ne fait rien gagner, puisque le cœur est occupé de bout en bout ; seuls deux cœurs travaillant en même temps raccourcissent le total. Dans le schéma suivant, le temps se lit encore de gauche à droite, et les lettres désignent deux tâches qui ne font que calculer.



Sur un cœur, A et B sont concurrentes : elles avancent à tour de rôle, et la durée totale ne change pas. Sur deux cœurs, leurs calculs se recouvrent : c'est le parallélisme. Les segments illustrent des possibilités d'ordonnancement, pas des mesures de durée.

Un programme peut donc être concurrent sans être parallèle, sur un seul cœur. À l'inverse, pour occuper plusieurs cœurs, il faut d'abord disposer de plusieurs tâches prêtes à

avancer, c-à-d. d'un programme concurrent : le parallélisme est une manière d'exécuter un programme concurrent.

Dans ce chapitre, c'est la concurrence qui nous occupe : découper un programme en tâches qui coopèrent, et rester juste quel que soit leur entrelacement. La BEAM répartit ensuite le travail sur les cœurs.

POUR ALLER PLUS LOIN

Quatre cœurs rendent-ils un calcul quatre fois plus rapide ? Rarement. Supposons qu'une fraction p du travail se répartisse entre n cœurs, le reste devant se faire à la suite. La durée passe de 1 à $(1 - p) + p/n$, et le gain vaut

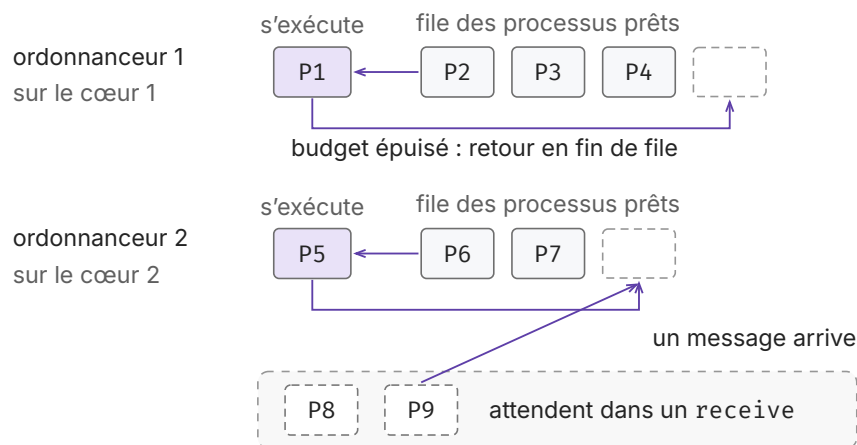
$$S = \frac{1}{(1 - p) + p/n}.$$

Avec $p = 0.9$ et quatre cœurs, $S = 1/(0.1 + 0.225) \approx 3.1$. Même avec une infinité de cœurs, S ne dépasse pas $1/0.1 = 10$: la part séquentielle finit par tout dominer. C'est la *loi d'Amdahl*, énoncée par Gene Amdahl en 1967 ; l'article « *Loi d'Amdahl* »³⁹ de Wikipédia en présente les variantes.

9.1.3 Qui choisit la tâche qui avance ?

Un *ordonnanceur* (*scheduler*) est la partie du système qui répartit le temps des cœurs entre les tâches prêtes. La BEAM lance par défaut un ordonnanceur par cœur, et `System.schedulers_online()` vous en donne le nombre sur votre machine.

Chaque ordonnanceur garde une *file* de processus prêts à avancer. Il prend le premier, le fait exécuter un moment, puis passe au suivant. Un processus quitte son cœur dans deux cas : il attend un message qui n'est pas encore arrivé, ou il a épuisé un petit budget de travail.



Chaque ordonnanceur fait avancer un processus à la fois et garde les autres en file. Un processus qui a épuisé son budget repasse en fin de file ; un processus qui attend un message sort des files et n'y revient qu'à l'arrivée d'un message. Le nombre de processus dessinés est arbitraire.

³⁹ https://fr.wikipedia.org/wiki/Loi_d'Amdahl

Dans le premier cas, le processus sort des files : il ne consomme aucun calcul tant qu'aucun message n'arrive, et c'est l'arrivée d'un message qui le remet dans une file. Dans le second, il repasse simplement en fin de file.

Cette interruption imposée par l'ordonnanceur s'appelle la *préemption* : grâce à elle, un processus qui calcule sans fin n'empêche pas les autres d'avancer, même sur un seul cœur. Les ordonnanceurs s'échangent aussi du travail, et celui dont la file se vide peut prendre des processus dans celle d'un voisin. Vous n'avez donc jamais à choisir le cœur d'un processus : vous décrivez des processus, et la VM les répartit.

POUR ALLER PLUS LOIN

Le budget d'un processus se compte en *réductions* : en gros, une réduction par appel de fonction, et quelques milliers par tour. Or une boucle Elixir est une récursion, donc une suite d'appels : même un calcul sans fin consomme son budget et rend la main. La BEAM peut ainsi interrompre n'importe quel processus, là où la boucle d'événements de JavaScript ou d' `asyncio` attend qu'une tâche rende la main d'elle-même. Le chapitre « *Scheduling* »⁴⁰ de *The BEAM Book*, d'Erik Stenman, détaille files, migrations et comptage.

9.2 Partager la mémoire, ou s'envoyer des messages ?

Des tâches qui avancent en même temps doivent bien, à un moment, échanger des données. Deux grandes façons de faire coexistent : partager une mémoire commune, ou ne rien partager et s'envoyer des messages. Pour les comparer, distinguons d'abord trois sortes d'unités d'exécution.

9.2.1 Processus du système, threads et processus BEAM

Le mot « processus » désigne plusieurs choses voisines, qu'il vaut mieux distinguer dès maintenant. Un *processus du système* est un programme en cours d'exécution, géré par le système d'exploitation : votre navigateur, Livebook, chaque `python` que vous lancez. Le système lui réserve un espace mémoire que les autres ne peuvent pas lire ; le créer est coûteux, et deux processus du système communiquent par des fichiers ou par le réseau.

Un *thread* (*fil d'exécution*) est une exécution à l'intérieur d'un processus du système. Plusieurs threads d'un même programme avancent concurremment et partagent toute sa mémoire. Ils sont plus légers à créer, mais ce partage est la source du souci décrit juste après. En Python, de plus, un verrou global de l'interpréteur, le *GIL*, empêche le plus souvent deux threads d'exécuter du code Python au même instant : ils apportent la concurrence, pas le parallélisme.

Un *processus BEAM* est une troisième sorte d'unité d'exécution, organisée par la machine virtuelle d'Erlang elle-même. Toute la VM tourne dans un seul processus du système, avec notamment un thread par ordonnanceur, et ce sont ces ordonnanceurs qui font avancer les processus BEAM. Ceux-ci empruntent au processus du système son isolation, et au thread sa légèreté, en plus marquée.

⁴⁰ <https://blog.stenmans.org/theBeamBook/#CH-Scheduling>

	processus du système	thread	processus BEAM
organisé par	le système	le système	la VM
mémoire	séparée	partagée	séparée
création	coûteuse	plus légère	très légère
échanges	fichiers, réseau	variables partagées	messages

9.2.2 Deux threads, un compteur

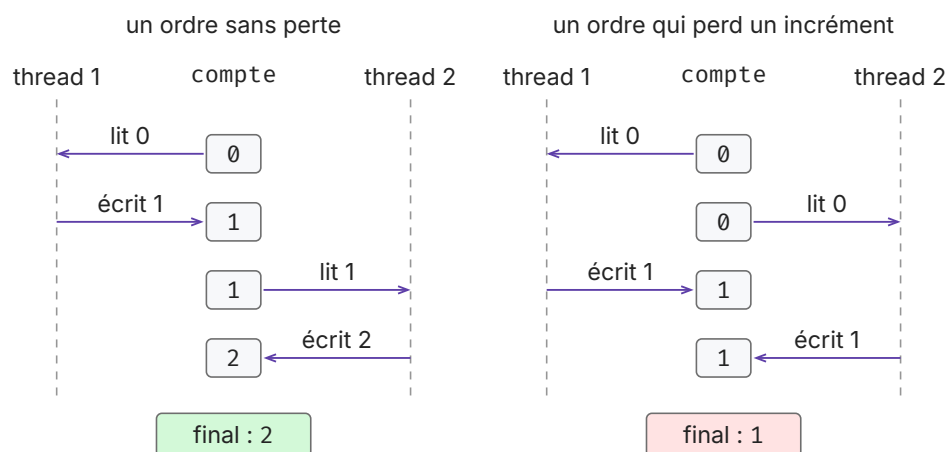
Pourquoi la BEAM a-t-elle choisi d'isoler ses processus ? Regardons ce qui arrive à deux threads qui partagent une variable `compte`, valant 0, et qui exécutent chacun une fois la fonction suivante, écrite ici en Python.

```

1 compte = 0
2
3 def contribuer():
4     global compte
5     compte += 1 # lire compte, puis écrire compte + 1

```

La ligne `compte += 1` paraît indivisible ; elle fait pourtant deux choses : lire la valeur, puis écrire cette valeur plus un. Or l'ordonnanceur peut interrompre un thread entre les deux. Le schéma suivant compare deux ordres possibles des mêmes actions.



Chaque panneau se lit de haut en bas. Une flèche vers un thread est une lecture, une flèche vers `compte` une écriture ; les cases du milieu donnent la valeur de `compte` après chaque action. À droite, le thread 2 lit 0 avant que le thread 1 ait écrit : son écriture recouvre celle du thread 1.

À droite, deux incréments ont eu lieu, mais un seul est compté ! Si le thread 1 avait fait ses deux actions avant que le thread 2 ne lise, comme à gauche, `compte` vaudrait 2. On appelle *condition de course* (*race condition*) une situation où le résultat dépend de l'ordre dans lequel s'entrelacent les actions de plusieurs tâches.

Ce bug est pénible, car il n'apparaît que de temps en temps, selon la charge de la machine : les deux actions d'un thread sont si courtes que l'ordonnanceur les sépare rarement. Un test qui passe cent fois ne prouve donc pas son absence.

Les langages à threads y répondent par des *verrous* : un thread prend le verrou, lit et écrit, puis le rend, et les autres attendent leur tour. Oublier un verrou, ou en prendre deux dans le

mauvais ordre, de sorte que deux threads s’attendent l’un l’autre, reste une source classique d’erreurs.

POUR ALLER PLUS LOIN

Combien d’ordres faudrait-il essayer ? Un ordre des quatre actions se décrit en choisissant, parmi quatre positions, les deux où agit le thread 1 : il y en a $\binom{4}{2} = 6$, dont deux seulement comptent les deux incréments, ceux où un thread fait tout avant l’autre. Avec n actions par thread, on trouve $\binom{2n}{n}$ ordres, soit 184 756 pour $n = 10$, et bien davantage avec trois threads. Un essai n’en explore qu’un, presque toujours le même. Des outils comme [Concuerror](https://concuerror.com)⁴¹, pour Erlang, les énumèrent systématiquement afin de trouver l’entrelacement fautif, ou de prouver qu’il n’existe pas.

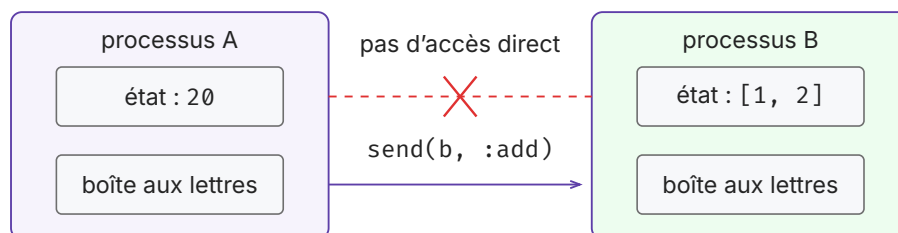
POUR ALLER PLUS LOIN

Rust prend une troisième voie : son compilateur refuse un programme où deux threads pourraient modifier une même donnée sans protection, et la course de ce schéma devient une erreur de compilation. Le chapitre « Fearless Concurrency » du [livre officiel de Rust](https://doc.rust-lang.org/book/ch16-00-concurrency.html)⁴² présente ces règles.

9.2.3 Ne rien partager

La BEAM supprime le partage. Chaque processus BEAM a sa propre mémoire, son propre état et sa propre *boîte aux lettres*, c-à-d. la file des messages qu’il a reçus et pas encore lus. Il est désigné par un *PID* (*process identifier*), qui sert d’adresse pour lui écrire.

Aucun processus ne peut lire ou modifier l’état d’un autre : il ne peut que lui envoyer un message et attendre, éventuellement, une réponse.



Le seul chemin vers l’état d’un autre processus est un message dans sa boîte.

Deux processus, deux états séparés. A ne peut pas toucher la liste de B ; il dépose un message dans sa boîte, et B décidera quoi en faire quand il le lira.

Que devient alors notre compteur ? Il vit dans un seul processus, seul à lire et à écrire son état, et les autres lui envoient des demandes. Or un processus traite ses messages un par un : il ne commence le suivant qu’après avoir complètement fini le précédent.

Lire, ajouter et écrire se font donc dans le traitement d’un même message, que rien ne peut couper de l’extérieur, et l’entrelacement du panneau de droite devient impossible à

⁴¹ <https://concuerror.com>

⁴² <https://doc.rust-lang.org/book/ch16-00-concurrency.html>

l'intérieur de ce processus. Deux questions restent toutefois ouvertes, et nous y reviendrons quand notre compteur existera. L'ordre d'arrivée des messages de deux clients n'est pas fixé ; et si un client découpe son travail en deux demandes, « lire » puis « écrire », un autre client peut s'intercaler entre les deux.

POUR ALLER PLUS LOIN

Ce découpage a un nom en informatique théorique : le *modèle des acteurs*, proposé par Carl Hewitt, Peter Bishop et Richard Steiger en 1973. Dans sa formulation devenue classique, un acteur ne partage rien ; à la réception d'un message, il peut seulement envoyer des messages aux acteurs dont il connaît l'adresse, créer de nouveaux acteurs, et choisir comment il traitera le message suivant. Remplacez « acteur » par « processus » : vous reconnaîtrez `send`, `spawn` et la boucle `loop(next)` de ce chapitre. L'article d'origine, « [A Universal Modular ACTOR Formalism for Artificial Intelligence](#) »⁴³, est en accès libre.

À RETENIR

La concurrence entrelace plusieurs tâches en cours et fait gagner du temps dès que des tâches attendent, même sur un seul cœur ; le parallélisme exécute des calculs au même instant sur plusieurs cœurs. Les processus BEAM ne partagent aucune mémoire et traitent leurs messages un par un : une condition de course « lire, modifier, écrire » ne peut pas couper le traitement d'un message, mais l'ordre d'arrivée des messages de plusieurs clients reste incertain.

9.3 Qui exécute cette fonction ?

Passons à la pratique. Trois primitives suffisent pour commencer : `spawn/1` crée un processus, `send/2` dépose un message dans la boîte d'un PID, et `receive` attend dans sa propre boîte un message qui correspond à l'un de ses motifs.

9.3.1 Créer un processus et lui écrire

`self()` rend le PID du processus qui l'exécute. `spawn/1` lance une fonction dans un nouveau processus et rend aussitôt son PID au créateur ; le nouveau processus se termine lorsque sa fonction se termine.

`receive` s'écrit avec la syntaxe du `case` du chapitre 2 : des motifs, puis une expression à droite de `->`. Le bloc `after` qui le suit borne l'attente : si aucun message ne convient dans le délai indiqué, en millisecondes, c'est sa branche qui s'exécute.

Avant de lire le résultat, prévoyez ce que désignent les deux appels à `self()`. Pourquoi faut-il conserver `parent` avant d'appeler `spawn` ?

⁴³ <https://www.ijcai.org/Proceedings/73/Papers/027B.pdf>

```

1 parent = self()
2 ref = make_ref()
3 child = spawn(fn -> send(parent, {:hello, ref, self()})) end)
4
5 received = receive do
6   {:hello, ^ref, pid} -> pid
7 after
8   1_000 -> raise "Réponse absente"
9 end
10
11 {received == child, received == parent}

```

VALEUR {true, false}

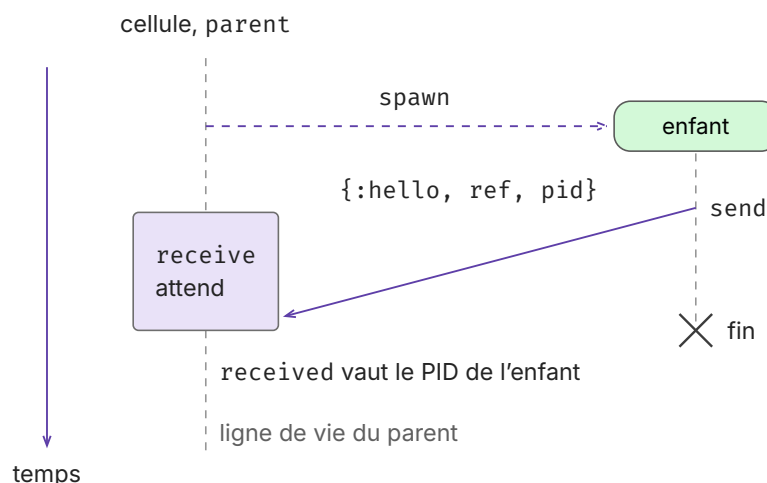
Le `self()` écrit dans la fonction anonyme désigne l'enfant, puisque c'est l'enfant qui exécute cette fonction. La variable `parent`, capturée par la fonction comme les fermetures du chapitre 5, est son adresse de retour. Les numéros imprimés dans un PID changent d'un essai à l'autre : nous vérifions ici des égalités, pas une valeur numérique particulière.

`make_ref()` crée une *référence*, une valeur unique dans toute la VM. Dans le motif, `^ref` exige cette référence déjà connue, comme le pin du chapitre 2 : le message ne se confond donc pas avec celui d'un autre essai. Sans le `^`, `ref` serait dans le motif un nom neuf, relié à la première référence venue, et la clause accepterait n'importe quel message de la forme `{:hello, _, _}`.

Un PID identifie le destinataire ; une référence identifie une demande. Nous verrons bientôt pourquoi la distinction compte.

9.3.2 Lire un diagramme de séquence

Dessignons cet échange. Ce type de schéma, appelé *diagramme de séquence*, reviendra jusqu'à la fin du cours ; prenons donc le temps d'apprendre à le lire.



La cellule crée l'enfant (flèche en pointillé), puis attend dans son `receive`. L'enfant envoie son message et se termine ; le message arrive dans la boîte de la cellule, dont le `receive` le retire.

Un diagramme de séquence obéit à quelques conventions :

- chaque processus possède une colonne, sa *ligne de vie* : un trait vertical en pointillé sous son nom, qui commence à sa création ;

- le temps s'écoule vers le bas : sur une même ligne de vie, un événement dessiné plus bas qu'un autre se produit après lui ;
- une flèche pleine est un message : elle part de l'expéditeur au moment du `send` et aboutit plus bas dans la boîte du destinataire, puisque le message arrive plus tard ;
- une flèche en pointillé marque ici la création d'un processus par `spawn` ;
- une case posée sur une ligne de vie décrit ce que fait le processus pendant ce temps, par ex. attendre dans un `receive` ; une croix marque sa fin.

Lisons celui-ci de haut en bas. La cellule crée l'enfant, puis entre dans son `receive`. L'enfant envoie son message, puis sa fonction se termine. Le message arrive dans la boîte de la cellule, dont le `receive` le retire : `received` vaut alors le PID de l'enfant.

Deux règles évitent les contresens. D'abord, les distances verticales ne mesurent pas des durées ; elles indiquent seulement ce qui vient avant quoi. Ensuite, deux événements situés sur deux lignes de vie différentes, sans flèche qui les relie, n'ont aucun ordre garanti, même si l'un est dessiné plus haut : le schéma montre un déroulement possible parmi d'autres.

Ici, l'enfant a pu envoyer son message avant même que la cellule n'atteigne son `receive` ; le message aurait simplement attendu dans la boîte. Quand un échange vous paraît confus, dessinez-en un au brouillon : c'est le meilleur moyen de savoir qui exécute quoi.

9.3.3 Des processus par milliers

Ces processus sont-ils vraiment légers ? Créons-en dix mille qui attendent chacun un message `:stop`, comptons ceux qui sont vivants en même temps, puis arrêtons-les tous.

```
1 pids = Enum.map(1..10_000, fn _ ->
2   spawn(fn -> receive do :stop -> :ok end end)
3 end)
4 alive = Enum.count(pids, &Process.alive?/1)
5 Enum.each(pids, fn pid -> send(pid, :stop) end)
6 alive
```

VALEUR 10000

`Process.alive?(pid)` rend `true` tant que le processus existe. Les dix mille processus ont existé en même temps, chacun avec sa boîte aux lettres.

D'après le [guide d'efficacité d'Erlang](#)⁴⁴, un processus qui vient de naître occupe 327 mots machine, soit 2,6 Ko environ sur une machine 64 bits ; un thread du système réserve d'ordinaire de l'ordre du mégaoctet pour sa seule pile. Une application Elixir peut donc dédier un processus à chaque connexion, à chaque utilisateur ou à chaque tâche, sans compter.

9.4 Envoyer n'est pas attendre une réponse

`send(pid, message)` envoie une valeur et rend cette même valeur. Il n'attend pas que le destinataire la traite, ni même qu'il existe encore : la valeur rendue par `send` n'est donc pas un accusé de réception. On dit que l'envoi est *asynchrone* : l'expéditeur poursuit aussitôt son travail, comme on poste une lettre sans attendre qu'elle soit lue.

⁴⁴ https://www.erlang.org/doc/system/eff_guide_processes.html

Un appel de fonction ordinaire est au contraire *synchrone*, puisque l'appelant attend son résultat. Pour obtenir une réponse d'un autre processus, il faut donc construire soi-même l'attente : envoyer une demande qui contient sa propre adresse, puis exécuter un `receive`. Cette section en assemble les pièces une à une.

9.4.1 La boîte aux lettres

Chaque message envoyé à un processus s'ajoute au bout de sa boîte aux lettres. `receive` examine la boîte du processus qui l'exécute, jamais celle d'un autre. Il essaie les messages dans l'ordre d'arrivée et, pour chacun, ses clauses dans l'ordre écrit ; il retire le premier message accepté et laisse les autres en place. On parle de *réception sélective*.

Prévoyez le résultat suivant : le premier message envoyé sera-t-il le premier retiré ? Nous écrivons dans notre propre boîte, ce qui nous permet de choisir nous-mêmes l'ordre d'arrivée.

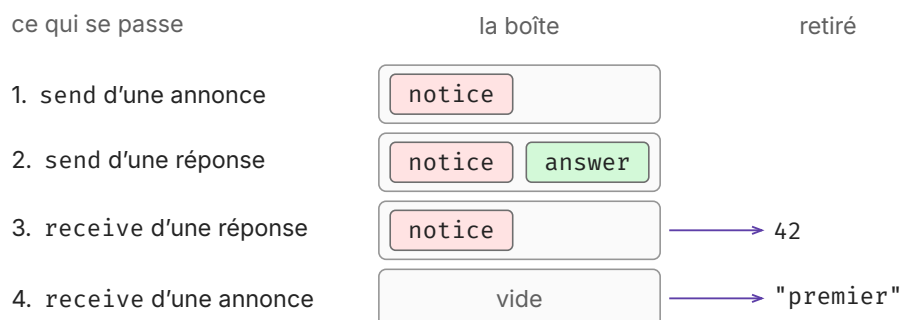
```

1 tag = make_ref()
2 send(self(), {:notice, tag, "premier"})
3 send(self(), {:answer, tag, 42})
4
5 answer = receive do
6   {:answer, ^tag, value} -> value
7 after
8   0 -> :absent
9 end
10
11 notice = receive do
12   {:notice, ^tag, text} -> text
13 after
14   0 -> :absent
15 end
16 {answer, notice}

```

VALEUR {42, "premier"}

Le premier `receive` n'accepte qu'une réponse : il saute l'annonce, retire la réponse et rend 42. L'annonce n'a pas disparu ; le second `receive` la trouve. `after 0` demande ici de ne pas attendre : seuls les messages déjà arrivés sont examinés.

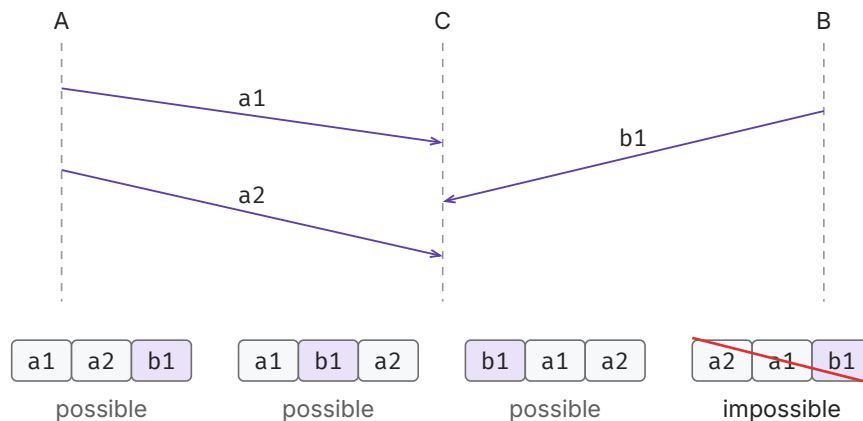


La boîte de la cellule après chaque étape, le message le plus ancien à gauche. La réception de l'étape 3 saute l'annonce et retire la réponse ; l'annonce attend le `receive` suivant.

Avec une clause attrape-tout, `anything -> anything`, le premier `receive` aurait retiré l'annonce. Notez la conséquence, surprenante au premier abord : un `receive` peut attendre devant une boîte non vide, si aucun des messages présents ne correspond à ses clauses.

9.4.2 Dans quel ordre arrivent les messages ?

La BEAM donne une seule garantie d'ordre : deux messages envoyés par un même processus à un même destinataire arrivent dans leur ordre d'envoi. Entre deux expéditeurs différents, en revanche, rien n'est fixé.



A envoie `a1` puis `a2` à C ; B envoie `b1`. Le diagramme montre un déroulement possible. En bas, les ordres d'arrivée dans la boîte de C : `b1` peut se placer n'importe où, mais `a2` n'arrive jamais avant `a1`.

Cette garantie suffit à beaucoup de protocoles. Un client qui envoie une modification, puis une lecture, au même service sait que le service verra la modification avant la lecture. Deux clients distincts, eux, ne savent rien de l'ordre dans lequel leurs demandes seront traitées : un protocole ne doit jamais reposer sur une hypothèse sur la vitesse des clients.

POUR ALLER PLUS LOIN

La garantie vaut pour une paire de processus ; elle ne se transmet pas. Si A envoie `x` à C, puis `y` à B, et que B, en recevant `y`, envoie `z` à C, alors `x` « précède » intuitivement `z` ; pourtant C peut recevoir `z` d'abord, puisque les deux messages viennent d'expéditeurs différents, cf. « [Delivery of Signals](https://www.erlang.org/doc/system/ref_man_processes.html#delivery-of-signals) »⁴⁵ dans le manuel d'Erlang. Cette relation « se produit avant », qui ne relie que les événements d'un même processus ou ceux qu'un message enchaîne, a été formalisée par Leslie Lamport : « [Time, Clocks, and the Ordering of Events in a Distributed System](https://lamport.azurewebsites.net/pubs/time-clocks.pdf) »⁴⁶, *Communications of the ACM*, 1978.

9.4.3 Une demande porte son adresse et sa référence

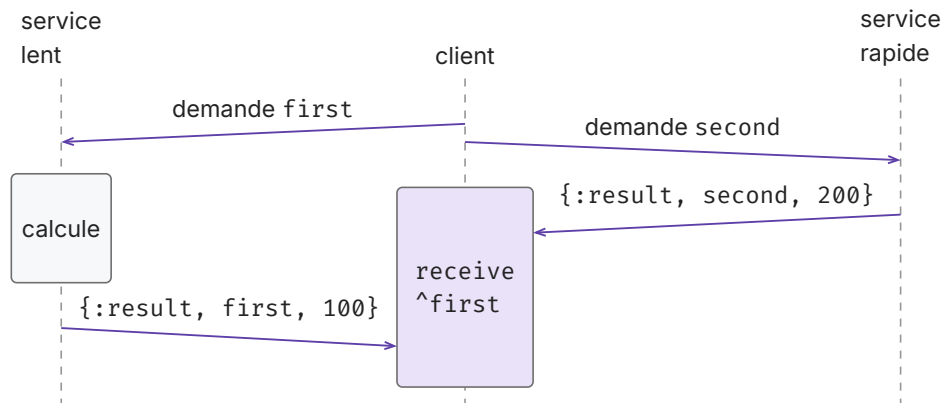
Revenons aux demandes. Un service ne connaît pas ses clients : pour qu'il puisse répondre, la demande doit contenir le PID du client, typiquement `self()` évalué chez le client au moment de l'envoi. Elle contient aussi une référence neuve, que le service recopie dans sa réponse.

Pourquoi cette référence ? Supposons qu'un client ait envoyé deux demandes, l'une à un service lent, l'autre à un service rapide, et que deux réponses `{:result, ..., 100}` et

⁴⁵ https://www.erlang.org/doc/system/ref_man_processes.html#delivery-of-signals

⁴⁶ <https://lamport.azurewebsites.net/pubs/time-clocks.pdf>

`{:result, ..., 200}` arrivent dans un ordre qu'il ne connaît pas. Sans marque, rien ne dirait laquelle répond à quoi !



rend 100 ; la réponse à second reste dans la boîte

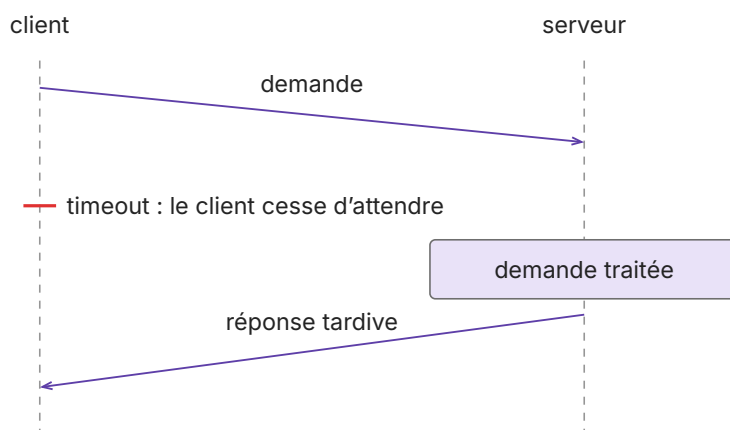
Le client envoie deux demandes, marquées des références `first` et `second`. La réponse du service rapide arrive la première ; le `receive` du client, qui épingle `^first`, la saute et retire celle du service lent.

Grâce à la référence, le client attend précisément la réponse qui l'intéresse ; la réception sélective laisse l'autre dans la boîte, pour plus tard. Sans le `^` devant `first`, au contraire, le motif relierait `first` à la première référence venue et retirerait la réponse à `second` : le client lirait 200 en croyant lire la réponse à sa première demande.

9.4.4 Borner l'attente

`after 1_000` borne l'attente d'un message correspondant à environ une seconde. Le processus ne dort pas volontairement une seconde : si la réponse arrive avant, il la prend aussitôt ; et le délai écoulé, il reprend dès que l'ordonnanceur lui rend la main. `after 0` examine les messages déjà disponibles sans attendre une arrivée future.

Surtout, le délai termine l'attente du client ; il n'annule pas la demande. Le diagramme suivant se lit comme les précédents.



L'attente du client a fini ; le travail du serveur a continué.

Une chronologie possible : la demande réussit après le timeout. Renvoyer la même demande d'ajout risque donc d'ajouter deux fois.

Le serveur a reçu la demande et la traite jusqu'au bout ; sa réponse tardive arrive dans la boîte du client, où elle restera si personne ne l'attend plus. Le danger serait alors de relancer

automatiquement la demande. Pour une lecture, ce n'est pas grave ; un ajout, lui, risque d'être compté deux fois. La référence distingue les réponses ; elle ne rend pas les requêtes répétées inoffensives.

9.4.5 Attendre pour toujours

Un `receive` sans `after` attend aussi longtemps qu'il le faut, c-à-d. éventuellement pour toujours. C'est voulu pour un service qui attend ses demandes, comme la boucle de la section suivante ; c'est un souci pour un client qui attend une réponse qui ne viendra pas.

Plusieurs causes, toutes banales, y conduisent : le service est mort avant de répondre ; sa réponse est partie vers un autre PID, par ex. un `self()` évalué dans le service au lieu de l'adresse du client ; ou elle ne correspond pas au motif attendu, à cause d'une faute de frappe dans un atome. Dans Livebook, la cellule reste alors en cours d'évaluation, sans rien afficher.

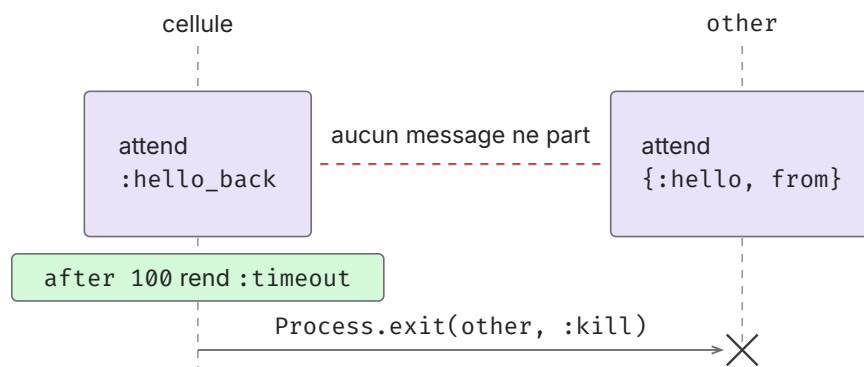
Le cas le plus traître est l'*interblocage* (*deadlock*) : deux processus attendent chacun un message que seul l'autre pourrait envoyer. Dans l'essai ci-dessous, `other` attend qu'on lui dise bonjour avant de répondre, tandis que la cellule attend sa réponse sans lui avoir rien dit. Personne ne fera le premier pas.

```

1 other = spawn(fn ->
2   receive do
3     {:hello, from} -> send(from, :hello_back)
4   end
5 end)
6
7 result = receive do
8   :hello_back -> :ok
9 after
10  100 -> :timeout
11 end
12 Process.exit(other, :kill)
13 result

```

VALEUR :timeout



Les deux cases d'attente se font face, et aucune flèche ne part de l'une vers l'autre. Seul le délai de la cellule met fin à son attente ; la cellule arrête ensuite `other`, qui attendrait sinon pour toujours.

Sans le `after`, la cellule resterait bloquée. Le délai ne résout pas l'interblocage : il le rend visible, en donnant à l'attente une issue que le programme peut traiter. Il ne

fait pas partir le message manquant, et il ne dit pas pourquoi la réponse manque. Le `Process.exit(other, :kill)` final arrête seulement le processus créé pour l'essai.

Voici, pour s'y retrouver, les formes rencontrées jusqu'ici.

forme	ce qu'elle fait	elle rend
<code>self()</code>	désigne le processus qui l'exécute	son PID
<code>spawn(fn -> ... end)</code>	lance la fonction dans un nouveau processus	le PID de l'enfant
<code>send(pid, message)</code>	dépose le message dans la boîte de <code>pid</code> , sans attendre	le message
<code>receive do ... end</code>	retire de sa boîte le premier message qui correspond à une clause ; sinon, attend	la valeur de la clause
<code>after n -> ...</code>	borne l'attente d'un <code>receive</code> à <code>n</code> ms	la valeur de sa branche
<code>make_ref()</code>	crée une référence unique, à épingler avec <code>^ref</code>	la référence
<code>Process.alive?(pid)</code>	demande si le processus existe encore	<code>true</code> ou <code>false</code>
<code>Process.exit(pid, :kill)</code>	termine sans condition le processus	<code>true</code>

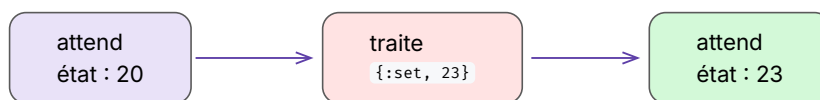
9.5 Comment l'état reste-t-il vivant ?

Un processus n'a pas de variable qui survive d'un message au suivant : nos valeurs sont immuables, et quand sa fonction se termine, il disparaît. Comment un compteur peut-il alors se souvenir de son total ?

9.5.1 Une boucle qui attend des messages

En ne terminant jamais sa fonction : une boucle récursive attend un message, le traite, puis se rappelle elle-même avec le nouvel état en argument, exactement comme les accumulateurs du chapitre 4.

Voici une petite mémoire qui conserve la dernière valeur reçue. Son protocole est explicite : `{:set, value}` remplace l'état, `{:read, client, ref}` demande une réponse, et `:stop` termine le processus. Le schéma suivant en montre le principe, avant le code.



`loop(23)` : le nouvel état devient le prochain argument

Le processus reste le même ; une nouvelle valeur devient l'argument de son prochain tour.

```

1  defmodule Memory do
2    def start(value), do: spawn(fn -> loop(value) end)
3
4    defp loop(value) do
5      receive do
6        {:set, next} -> loop(next)
7        {:read, client, ref} ->
8          send(client, {:value, ref, value})
9          loop(value)
10       :stop -> :ok
11     end
12   end
13 end

```

`start/1` lance la boucle dans un nouveau processus : la valeur initiale, capturée par la fonction anonyme, devient le premier argument de `loop/1`. Celle-ci est privée (`defp`), car elle s'exécute dans le processus de la mémoire et personne d'autre ne doit l'appeler.

Chaque branche décide de la suite. `{:set, next}` rappelle `loop(next)` : le prochain tour connaîtra la nouvelle valeur. `{:read, ...}` répond, puis rappelle `loop(value)` : rien ne change. `:stop` ne rappelle rien : la fonction rend `:ok`, se termine, et le processus disparaît avec son état.

L'appel `loop(next)` est terminal : aucun calcul ne reste à faire après son retour. Les tours n'accumulent donc pas des cadres d'appel, et la boucle peut traiter des messages indéfiniment. Cela ne signifie pas que toute la mémoire est bornée : la valeur gardée ou les messages en attente peuvent grossir.

Le `receive` sans `after` est ici voulu : le service attend des demandes tant que nous ne l'arrêtons pas, et, comme nous l'avons vu, il ne consomme aucun calcul pendant cette attente.

Mettons côte à côte cette boucle et celle de `Calc.produit` au chapitre 4, qui appelait `mult(reste, acc * tete)`. Seule change la provenance de la donnée suivante.

	mult (chapitre 4)	loop de Memory
ce qui est transmis	<code>acc</code> , le produit déjà calculé	<code>value</code> , l'état du processus
la donnée suivante	la tête de la liste	le prochain message de la boîte
l'arrêt	la liste vide <code>[]</code>	le message <code>:stop</code> , ou jamais
l'appel terminal	borne la pile d'appels	idem, même après des milliers de messages

9.5.2 Écrire, puis lire

Essayons la mémoire. La cellule envoie une modification, puis une lecture, et attend la réponse marquée de sa référence.

```

1 memory = Memory.start(20)
2 ref = make_ref()
3 try do
4   send(memory, {:set, 23})
5   send(memory, {:read, self(), ref})
6   receive do
7     {:value, ^ref, value} -> value
8   after
9     1_000 -> :timeout
10  end
11 after
12   Process.exit(memory, :kill)
13 end

```

VALEUR 23

Le même expéditeur envoie `:set`, puis `:read` ; les deux messages arrivent dans cet ordre, et la boucle traite donc la modification avant la lecture.

Le `try ... after` qui entoure l'essai n'a rien à voir avec l'`after` d'un `receive` : il garantit que `Process.exit(memory, :kill)` s'exécute à la fin, même si une ligne a échoué, et arrête seulement le processus créé pour cet essai. Réexécuter une cellule Livebook ne doit pas laisser derrière elle une collection de services oubliés.

Suivons maintenant l'état sur une suite plus longue de messages, tous envoyés par le même client, donc traités dans leur ordre d'envoi.

message traité	état avant	état après
<code>{:set, 23}</code>	20	23
<code>{:read, client, ref}</code>	23	23
<code>{:set, 18}</code>	23	18
<code>:stop</code>	18	processus terminé

Après `:stop`, le processus n'existe plus. Une demande de lecture envoyée ensuite ne reçoit jamais de réponse : `send` réussit, mais personne ne lit plus la boîte, et seul un `after` évite au client d'attendre pour toujours.

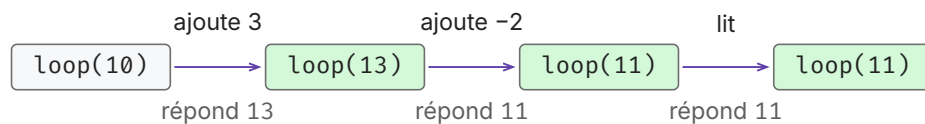
9.5.3 L'état passe d'un tour au suivant

Le même schéma donne un compteur. Imaginons un processus construit comme `Memory`, qui garde un nombre et comprend trois messages ; chaque réponse donne la valeur du compteur après la demande.

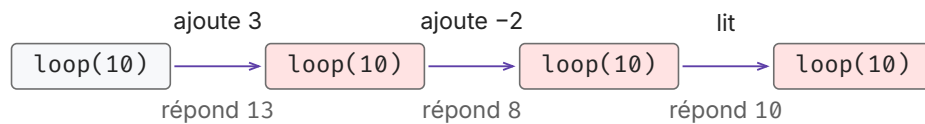
message reçu	réponse	état du tour suivant
<code>{:get, client, ref}</code>	<code>{:result, ref, value}</code>	value
<code>{:add, client, ref, amount}</code>	<code>{:result, ref, value + amount}</code>	value + amount
<code>:stop</code>	aucune	la fonction se termine

Démarrons-le à 10, puis envoyons-lui, depuis un même client, un ajout de 3, un ajout de -2 et une lecture. La première ligne du schéma suivant suit les appels successifs de la boucle.

rappel de `loop(next)`



rappel de `loop(value)` après l'ajout

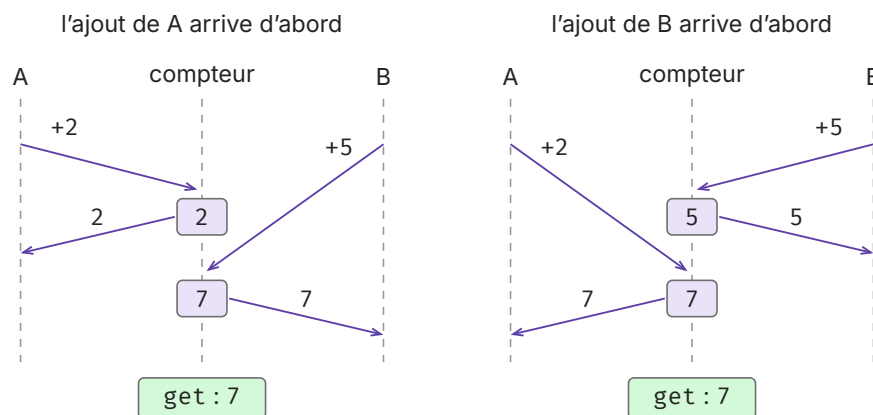


Chaque case est un appel de la boucle, et son argument est l'état du compteur. En haut, chaque ajout devient l'argument du tour suivant : les réponses valent 13, 11 et 11. En bas, la branche d'ajout répond juste, mais rappelle `loop(value)` : chaque tour repart de 10.

La seconde ligne montre une erreur fréquente. La branche d'ajout calcule bien `value + amount` et répond ce nombre, mais rappelle `loop(value)` au lieu de lui passer le nouvel état. La première réponse est juste, 13, et masque l'erreur ; l'ajout de -2 part pourtant de 10 et répond 8, et la lecture rend 10. Une seconde demande suffit donc à détecter l'oubli.

9.5.4 Deux clients, un seul propriétaire

Faisons maintenant écrire deux clients, A et B, au même compteur démarré à 0 : A envoie un ajout de 2, B un ajout de 5, à peu près au même moment. Lequel arrive le premier ? Nous n'en savons rien, et le compteur non plus. Le schéma suivant dessine les deux possibilités.



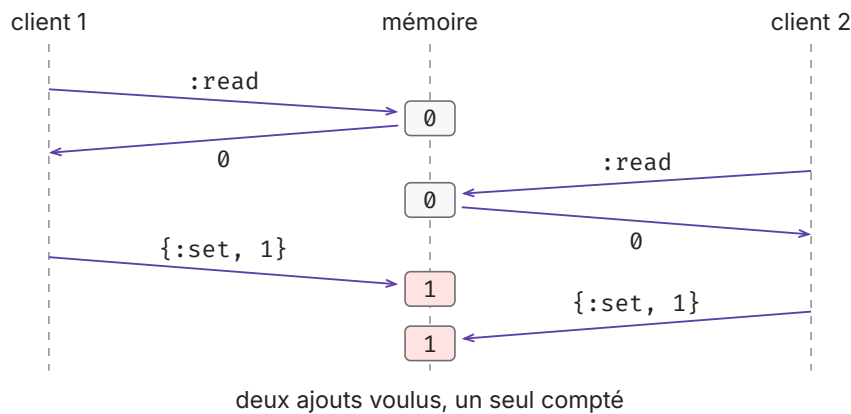
Les deux ajouts partent à peu près en même temps ; le compteur les traite l'un après l'autre, dans leur ordre d'arrivée. Les cases du milieu sont ses états successifs. À gauche, A reçoit 2 et B reçoit 7 ; à droite, B reçoit 5 et A reçoit 7. Une lecture faite ensuite rend 7 dans les deux cas.

Les réponses dépendent de l'ordre d'arrivée : en notant d'abord la réponse à A, on obtient $\{2, 7\}$ ou $\{7, 5\}$. Le total, lui, n'en dépend pas, et aucune contribution n'est perdue. En effet, les clients n'écrivent aucun nombre : ils demandent un ajout, et c'est le compteur qui calcule à partir de son état courant, une demande après l'autre.

Deux compteurs démarrés par deux appels à `start` sont deux processus, donc deux états indépendants, même s'ils exécutent le même module. Le module décrit un comportement ; chaque processus a sa propre valeur.

9.5.5 La course revient entre deux messages

L'isolation interdit-elle pour autant toute condition de course ? Non. Reprenons `Memory`, qui sait seulement remplacer et lire sa valeur, et demandons à deux clients d'y ajouter 1 chacun. Faute de demande d'ajout, chaque client doit lire la valeur, puis renvoyer la valeur lue plus un : deux demandes distinctes, entre lesquelles l'autre client peut agir.



L'entrelacement du début du chapitre, avec des messages : chaque demande est traitée d'un seul tenant, mais le client 2 lit 0 avant que le client 1 ait écrit.

Rejouons cet entrelacement. La cellule tient ici les deux rôles, dans l'ordre du schéma : c'est un déroulement possible parmi d'autres, que nous imposons pour l'observer à coup sûr. La fonction anonyme `read` envoie une demande de lecture et attend la réponse marquée de sa référence.

```

1 memory = Memory.start(0)
2 read = fn ->
3   ref = make_ref()
4   send(memory, {:read, self(), ref})
5   receive do
6     {:value, ^ref, value} -> value
7   after
8     1_000 -> raise "Réponse absente"
9   end
10 end
11
12 try do
13   seen_by_1 = read.()
14   seen_by_2 = read.()
15   send(memory, {:set, seen_by_1 + 1})
16   send(memory, {:set, seen_by_2 + 1})
17   read.()
18 after
19   Process.exit(memory, :kill)
20 end
  
```

VALEUR 1

Le résultat vaut 1, et non 2 : une contribution est perdue, alors que chaque message a été traité d'un seul tenant. La course s'est déplacée d'un cran : elle n'oppose plus deux actions sur une variable, mais deux demandes d'un même client.

La parade consiste à mettre toute la décision dans une seule demande, « ajoute 1 », que le propriétaire calcule à partir de son état courant ; c'est exactement ce que fait le compteur de la section précédente. Le chapitre 10 retrouvera ce piège avec les briques d'OTP.

À RETENIR

Un processus garde son état en le passant à l'appel suivant de sa boucle : il rappelle `loop(next)` après une modification, `loop(value)` quand rien ne change, et ne rappelle rien pour s'arrêter. Cet appel est terminal ; la boucle peut donc traiter des messages indéfiniment sans faire grandir la pile d'appels. Chaque message est traité en entier avant le suivant : deux demandes « ajoute » sont toujours comptées, mais « lire » puis « écrire » en deux demandes peut perdre une contribution.

9.6 Observer une panne

Dans un programme Python, une exception que personne ne rattrape arrête tout le programme. Sur la BEAM, une erreur termine seulement le processus où elle se produit : les autres continuent, puisqu'ils ne partageaient rien avec lui.

Encore faut-il que quelqu'un s'en aperçoive. Le silence d'un service ne suffit pas : il peut ne pas répondre parce qu'il est occupé, bloqué ou déjà terminé. Deux mécanismes y répondent, le monitor et le lien.

9.6.1 Fins normales et fins anormales

Un processus qui se termine le fait avec une *raison de sortie*. Quand sa fonction rend une valeur, il se termine normalement, avec la raison `:normal`. Toutes les autres raisons signalent une fin *anormale*, c-à-d. une panne.

ce qui arrive au processus	raison de sortie	fin
sa fonction rend une valeur	<code>:normal</code>	normale
il exécute <code>exit(:demo)</code>	<code>:demo</code>	anormale
une exception n'est pas rattrapée, par ex. <code>raise "oups"</code>	l'exception et la pile d'appels	anormale
un autre exécute <code>Process.exit(pid, :kill)</code>	<code>:killed</code>	anormale

`exit(raison)` termine le processus qui l'exécute avec cette raison ; c'est la façon la plus simple de provoquer une panne dans un essai. Nous la lancerons toujours dans un processus créé pour l'occasion, jamais dans celui de la cellule.

9.6.2 Un monitor prévient

Un *monitor* observe un processus et informe l'observateur de sa fin par un message ordinaire, déposé dans sa boîte : `{:DOWN, ref, :process, pid, raison}`. `Process.monitor(pid)` installe l'observation et rend la référence `ref` que l'on retrouvera dans ce message.

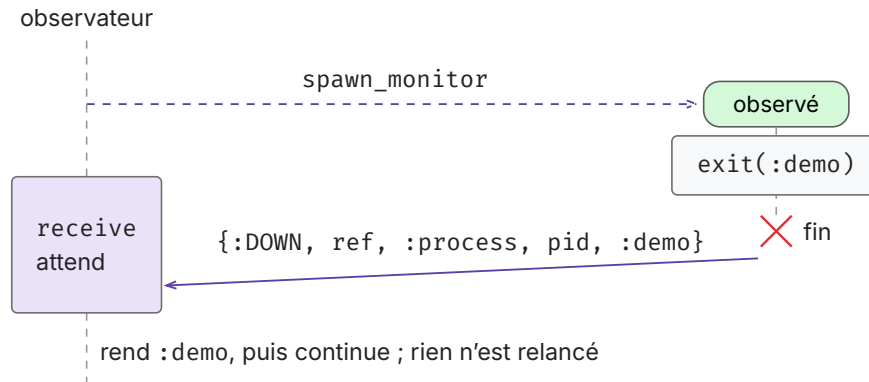
`spawn_monitor` crée le processus et l'observation ensemble, et rend les deux sous la forme `{pid, ref}` : on ne risque pas de commencer à observer un processus déjà terminé.

```

1 {pid, monitor} = spawn_monitor(fn -> exit(:demo) end)
2 receive do
3   {:DOWN, ^monitor, :process, ^pid, reason} -> reason
4 after
5   1_000 -> :timeout
6 end

```

VALEUR :demo



L'observé se termine par `exit(:demo)` ; le monitor transforme cette fin en un message `:DOWN` dans la boîte de l'observateur, dont le `receive` le retire. La cellule continue ; personne ne relance l'observé.

La cellule reste vivante, et la raison transmise est celle que l'observé a donnée à `exit`. Un monitor ne provoque ni redémarrage ni propagation de la panne : il informe, et c'est tout.

Il signale aussi les fins normales. Revenons à `send` : que rend-il quand le destinataire n'existe plus ? Pour le savoir à coup sûr, attendons le `:DOWN` d'un processus qui se termine aussitôt, puis écrivons-lui.

```

1 {pid, monitor} = spawn_monitor(fn -> :ok end)
2 reason = receive do
3   {:DOWN, ^monitor, :process, ^pid, reason} -> reason
4 end
5 {reason, Process.alive?(pid), send(pid, :hello)}

```

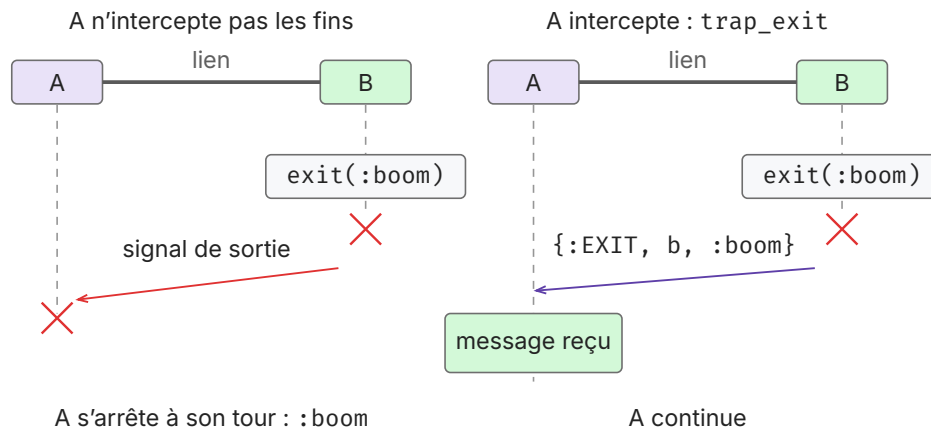
VALEUR {:normal, false, :hello}

Le `:DOWN` porte la raison `:normal`. `Process.alive?` confirme que le PID ne désigne plus personne, et `send` rend pourtant son message, sans erreur : il a déposé une lettre à une adresse vide. Seule une réponse, ou un `:DOWN`, renseigne vraiment sur le destinataire. Un client peut d'ailleurs accepter, dans le même `receive`, la réponse attendue ou le `:DOWN` du service : il n'attendra pas pour toujours un service mort, et c'est ce que fera `GenServer.call` au chapitre 10. Notez enfin qu'un monitor posé sur un PID déjà mort envoie aussitôt un `:DOWN`, avec la raison `:noprocs`.

9.6.3 Un lien relie deux destins

Un *lien* a un autre rôle. Créé par `spawn_link` ou `Process.link`, il relie les destins de deux processus, dans les deux sens. Si l'un se termine anormalement, un *signal de sortie* part vers l'autre, qui se termine à son tour avec la même raison. Une fin normale, elle, ne se propage pas.

Un processus peut toutefois demander à *intercepter* les fins de ses liens, par `Process.flag(:trap_exit, true)`. Le signal de sortie lui arrive alors sous la forme d'un message ordinaire, `{:EXIT, pid, raison}`, qu'il traite dans un `receive` comme les autres.



À gauche, A ne fait rien de particulier : la fin anormale de B le termine à son tour. À droite, A a demandé `trap_exit` : la fin de B lui arrive comme un message, et il continue.

Observons cette propagation de l'extérieur. Un intermédiaire, surveillé par la cellule, crée un enfant lié qui s'arrête aussitôt avec `exit(:boom)`, puis attend un message qui ne viendra jamais.

```
1 {pid, monitor} = spawn_monitor(fn ->
2   spawn_link(fn -> exit(:boom) end)
3   receive do
4     :never -> :ok
5   end
6 end)
7
8 receive do
9   {:DOWN, ^monitor, :process, ^pid, reason} -> reason
10 after
11   1_000 -> :timeout
12 end
```

VALEUR :boom

L'intermédiaire n'a rien fait de mal : il est tombé avec la raison de son enfant. C'est voulu pour des processus qui n'ont pas de sens l'un sans l'autre. Si l'enfant avait simplement terminé sa fonction, l'intermédiaire aurait continué d'attendre.

Le même essai, avec un intermédiaire qui intercepte les fins, donne le panneau de droite :

```

1 parent = self()
2 spawn(fn ->
3   Process.flag(:trap_exit, true)
4   child = spawn_link(fn -> exit(:boom) end)
5   receive do
6     {:EXIT, ^child, reason} -> send(parent, {:trapped, reason})
7   end
8 end)
9
10 receive do
11   {:trapped, reason} -> reason
12 after
13   1_000 -> :timeout
14 end

```

VALEUR :boom

Cette fois, l'intermédiaire survit, reçoit `{:EXIT, child, :boom}` et nous transmet la raison. Seul `Process.exit(pid, :kill)` passe outre : le processus visé s'arrête même s'il intercepte les fins, avec la raison `:killed`.

	monitor	lien
créé par	<code>Process.monitor</code> , <code>spawn_monitor</code>	<code>Process.link</code> , <code>spawn_link</code> , <code>start_link</code>
sens	un observateur, un observé	dans les deux sens
à une fin normale	un message <code>:DOWN</code>	rien
à une fin anormale	un message <code>:DOWN</code>	la fin de l'autre, ou un message <code>:EXIT</code> avec <code>trap_exit</code>

Les fonctions `start_link` employées par OTP créent un lien entre le processus démarré et celui qui le démarre ; c'est par ce lien, converti en messages grâce à `trap_exit`, qu'un superviseur apprend la fin de ses enfants. Pour observer une panne depuis une cellule Livebook ou depuis IEx, on préfère donc un monitor : nous voulons voir la panne, pas faire tomber le processus qui l'observe.

9.7 Laisser planter, puis relancer

Nous savons maintenant apprendre qu'un processus est tombé. Que faire ensuite : réparer sur place, ou repartir d'un état connu ?

9.7.1 Laisser planter

Dans beaucoup de langages, on se protège des erreurs en entourant chaque opération risquée d'un `try`, et en essayant de réparer l'état sur place. La tradition d'Erlang répartit autrement les rôles, sous le slogan « *let it crash* », laisser planter.

Le code d'un processus traite les cas prévus par son contrat, y compris les erreurs attendues, qu'il rend comme des valeurs `{:error, raison}` à la manière du chapitre 4. Face à une situation imprévue, comme un état incohérent ou un message qui n'aurait jamais dû arriver, il ne cherche pas à bricoler une réparation : il s'arrête, et un autre processus décide de la suite.

Pourquoi est-ce raisonnable ? D'abord, l'isolation garantit qu'un état peut-être corrompu disparaît avec son processus, puisque personne d'autre ne le partageait. Ensuite, le remplaçant repart d'un état initial connu et testé, plutôt que d'un état réparé à moitié. Enfin, beaucoup de pannes sont passagères, par ex. un service distant momentanément injoignable, et réessayer depuis un état propre suffit alors.

Le code de chaque processus reste ainsi court, et la politique de reprise est décrite à un seul endroit, dans le processus chargé de relancer les autres : le *superviseur*.

POUR ALLER PLUS LOIN

Joe Armstrong, l'un des créateurs d'Erlang, a exposé cette manière de bâtir des systèmes fiables avec des composants qui peuvent échouer dans sa thèse, « [Making reliable distributed systems in the presence of software errors](https://erlang.org/download/armstrong_thesis_2003.pdf) »⁴⁷ (2003). La documentation d'OTP décrit les arbres de supervision dans ses [principes de conception](https://www.erlang.org/doc/system/design_principles.html)⁴⁸.

9.7.2 Un superviseur relance ses enfants

Un superviseur est un processus dont le métier est de démarrer d'autres processus, ses *enfants*, et de les relancer quand ils tombent. Il leur est lié et intercepte leurs fins : c'est ainsi qu'il apprend chaque panne.

Pour relancer un enfant, il doit savoir le démarrer. Il connaît donc sa recette de démarrage, ou *child spec* : une map qui donne un identifiant à l'enfant et, sous la clé `start`, le module, la fonction et la liste d'arguments à appeler pour le démarrer.

```
1 %{id: :counter, start: {Agent, :start_link, [fn -> 0 end]}}
```

Le superviseur refait cet appel à chaque redémarrage. Avec la stratégie `:one_for_one`, il relance seulement l'enfant qui vient de se terminer. La recette suivante démarre un `Agent`, un petit processus OTP qui garde une valeur et que le chapitre 10 détaille, à zéro ; `{Agent, fn -> 0 end}` est l'abréviation de la map ci-dessus, fournie par `Agent`.

```
1 children = [{Agent, fn -> 0 end}]
2 {ok, supervisor} = Supervisor.start_link(
3   children, strategy: :one_for_one
4 )
5 [{Agent, child, :worker, _}] =
6   Supervisor.which_children(supervisor)
7 Agent.update(child, fn value -> value + 7 end)
8 Agent.get(child, fn value -> value end) # 7
9 Supervisor.stop(supervisor)
```

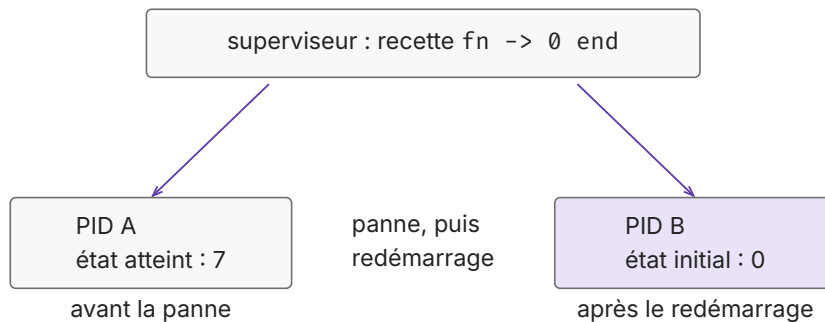
`Supervisor.which_children` liste les enfants avec leur PID, et le motif retient celui de l'`Agent` dans `child`. `Agent.update` remplace sa valeur en lui appliquant une fonction, `Agent.get` la lit : l'`Agent` vaut maintenant 7.

⁴⁷ https://erlang.org/download/armstrong_thesis_2003.pdf

⁴⁸ https://www.erlang.org/doc/system/design_principles.html

Interrompons maintenant l'enfant avec `Process.exit(child, :kill)`, avant d'arrêter le superviseur. Le superviseur, prévenu par son lien, le relance. Le redémarrage est asynchrone : l'ancien PID cesse d'être utilisable, puis un nouveau PID apparaît.

Quel état doit avoir le remplaçant ? La recette ne contient que `fn -> 0 end` ; le nouveau processus part donc de zéro, même si l'ancien avait atteint 7. Le redémarrage crée un nouvel enfant à partir de la recette, et rien d'autre.



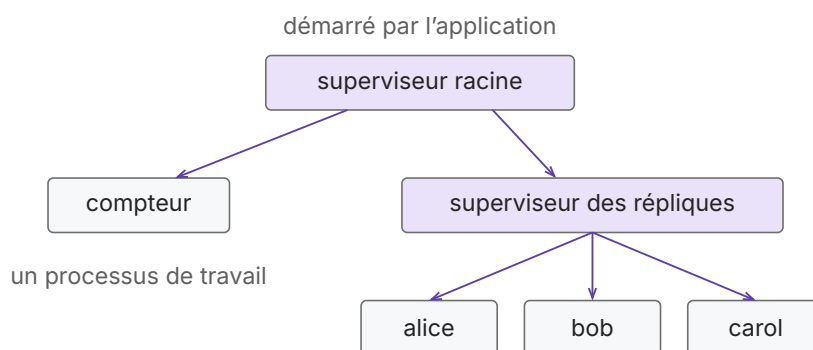
Deux instants : le nouveau PID ne récupère pas l'ancien état.

Les liens partent du même superviseur, à deux instants différents. A et B nomment deux PID distincts. La recette de démarrage fournit zéro ; aucun lien ne transfère la valeur 7 au remplaçant.

Puisque le PID change, comment les clients retrouvent-ils le service ? Par un *nom enregistré* : l'option `name:` de `start_link`, que le chapitre 10 emploiera, associe un nom au processus, et le superviseur l'associe de nouveau au remplaçant. Les clients d'un service supervisé l'appellent donc par son nom plutôt que par un PID conservé dans une variable.

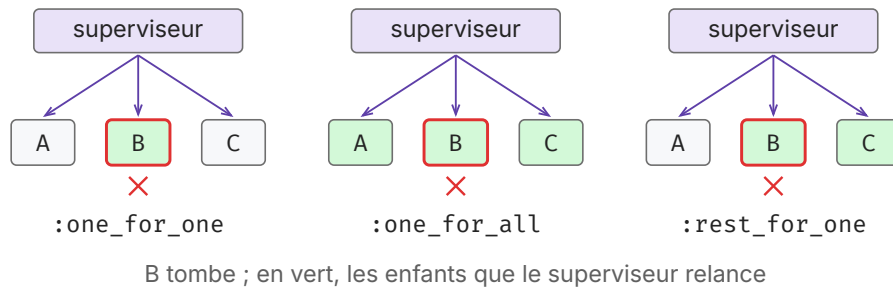
9.7.3 Stratégies et arbre de supervision

Un superviseur est lui-même un processus, et il peut donc être l'enfant d'un autre superviseur. Une application organise ainsi ses processus en *arbre de supervision* : les feuilles travaillent, les nœuds internes surveillent.



Un arbre possible, si l'on réunissait les projets d'AP7 et d'AP8 dans une même application. Les superviseurs, en violet, ne calculent rien : ils démarrent et relancent leurs enfants. Si Bob tombe, seul le superviseur des répliques intervient.

Chaque superviseur suit une *stratégie*, qui dit ce qu'il relance quand l'un de ses enfants tombe. Avec `:one_for_one`, il relance cet enfant seul, ce qui convient à des enfants indépendants, comme nos compteurs. Avec `:one_for_all`, il relance tous ses enfants, qui n'ont de sens qu'ensemble. Avec `:rest_for_one`, il relance l'enfant tombé et ceux démarrés après lui, lorsque chacun dépend des précédents.



Trois superviseurs, trois enfants démarrés dans l'ordre A, B, C. Quand B tombe, chaque stratégie relance un ensemble différent ; un enfant relancé repart de sa recette, avec un nouveau PID.

Nos projets n'utilisent que `:one_for_one`. Chaque enfant précise aussi s'il doit être relancé : toujours (`:permanent`, la valeur par défaut), seulement après une fin anormale (`:transient`), ou jamais (`:temporary`).

Enfin, un superviseur limite la fréquence des redémarrages : par défaut, plus de trois redémarrages en cinq secondes le font abandonner. Il se termine alors à son tour, et c'est son propre superviseur qui décide de la suite. Une panne qui se reproduit à chaque démarrage remonte ainsi l'arbre, au pire jusqu'à arrêter l'application, au lieu de tourner en boucle.

À RETENIR

Un redémarrage reconstruit l'enfant à partir de sa recette : un nouveau PID, l'état initial et une boîte aux lettres vide. Les messages qui attendaient dans l'ancienne boîte sont perdus, et un client qui attendait une réponse ne la recevra pas ; seul un délai, ou un monitor, le lui apprend. La supervision remet donc un service en fonctionnement ; elle ne sauvegarde pas ses données, ne rejoue pas les messages perdus et ne répare pas un programme qui échoue systématiquement. La persistance et la récupération demandent un autre mécanisme.

9.8 Sortir du carnet avec Mix

Une *application OTP* rassemble des modules et un cycle de démarrage et d'arrêt. Elle peut lancer de nombreux processus, mais n'est pas elle-même un processus ; elle n'est pas non plus synonyme de fenêtre graphique ou de serveur web. Avant AP6, la page « [Avant AP6](#) »⁴⁹ fait installer Elixir sur le poste et préparer l'éditeur. En fin de séance, nous créons le projet.

```
1 mix new process_lab --sup
2 cd process_lab
3 mix test
4 iex -S mix
```

`mix.exs` décrit le projet et son application. Les modules vivent dans `lib/`, les tests dans `test/`. L'option `--sup` fournit en plus `lib/process_lab/application.ex`, où la liste `children` décrit les enfants du superviseur, et ajoute dans `mix.exs` la ligne `mod: {ProcessLab.Application, []}`, qui désigne ce module de démarrage.

⁴⁹ <https://ap-2026.noce.fr/avant-ap6/>

Au lancement de l'application, la VM appelle la fonction `start/2` de ce module ; celle-ci démarre le superviseur racine, et avec lui tout l'arbre. La liste `children` générée est vide ; pour donner un premier enfant à l'application, il suffit de la remplacer par `[{Agent, fn -> 0 end}]`, la recette de la section précédente.

`iex -S mix` compile le projet puis démarre son application avant de rendre la main dans IEx. Après avoir modifié un fichier, `recompile()` recharge le code. Cela ne remet pas à zéro les états des processus déjà actifs : pour repartir de l'état initial, on redémarre l'application. Et si l'on quitte IEx puis qu'on le relance, l'Agent repart de 0, pour la même raison que tout à l'heure : rien n'a écrit sa valeur sur le disque.

Les détails des contrats restent consultables dans les documentations [Processes](#)⁵⁰, [Process](#)⁵¹ et [Supervisor](#)⁵².

9.9 Vérifier sa compréhension

Prévoyez chaque situation avant de lire sa réponse, sans faire appel à la vitesse supposée de la machine, et dessinez au besoin le diagramme.

Premier exercice, une boîte et deux réceptions. Que vaut le bloc suivant ? Écrivez le contenu de la boîte après chaque ligne.

```
1 send(self(), :a)
2 send(self(), {:b, 1})
3 send(self(), :c)
4 first = receive do {:b, n} -> n end
5 second = receive do anything -> anything end
6 {first, second}
```

Il vaut `{1, :a}`. Après les trois envois, la boîte contient `:a`, `{:b, 1}` et `:c`, dans cet ordre. Le premier `receive` n'accepte que `{:b, n}` : il saute `:a` et retire `{:b, 1}`. Le second accepte tout et retire le plus ancien message restant, `:a`. La boîte garde `:c`, que personne ne lit.

Deuxième exercice. Une mémoire démarrée à 5 reçoit d'un même client `{:set, 8}`, une lecture, `{:set, 2}`, puis une seconde lecture. Quelles réponses le client reçoit-il ? Et si la branche `{:set, next}` rappelait `loop(value)` ? Les réponses valent 8, puis 2 : l'ordre d'un même expéditeur est conservé, et chaque modification devient l'argument du tour suivant. Avec `loop(value)`, la mémoire répondrait deux fois 5.

Troisième exercice. Deux clients écrivent dans la même mémoire, l'un `{:set, 1}`, l'autre `{:set, 2}` ; le premier envoie ensuite une lecture. Que peut-il lire ? 1 ou 2 : sa lecture suit forcément son propre `:set`, mais le message de l'autre client peut arriver avant, entre les deux, ou après la lecture.

Quatrième exercice. Un processus A est lié à B et surveille C. B et C se terminent normalement, puis D, lui aussi lié à A, avec la raison `:boom`. Qu'arrive-t-il à A ? La fin de B ne lui fait rien, celle de C lui envoie un `:DOWN` avec la raison `:normal`, et celle de D le termine, à moins qu'il n'intercepte les fins et ne reçoive `{:EXIT, d, :boom}`.

⁵⁰ <https://hexdocs.pm/elixir/processes.html>

⁵¹ <https://hexdocs.pm/elixir/Process.html>

⁵² <https://hexdocs.pm/elixir/Supervisor.html>

Cinquième exercice. Un superviseur a trois enfants Agent, démarrés dans cet ordre par la recette `fn -> 0 end`, qui valent 4, 7 et 12. Le deuxième est tué. Que valent-ils ensuite, selon la stratégie ? Avec `:one_for_one`, seul le deuxième est relancé : 4, 0 et 12. Avec `:rest_for_one`, le deuxième et le troisième : 4, 0 et 0. Avec `:one_for_all`, les trois : 0, 0 et 0, sous trois nouveaux PID.

POUR EN PARLER

La concurrence entrelace des tâches, le parallélisme les exécute au même instant. Deux threads qui exécutent `compte += 1` peuvent perdre un incrément, et deux clients qui lisent `Memory`, puis y écrivent la valeur lue plus un, aussi : 1 au lieu de 2.

`send` rend le message sans rien attendre, même vers un PID mort ; `receive` retire le premier message qui correspond à une clause et laisse les autres. Seul l'ordre d'un expéditeur vers un destinataire est garanti.

L'état est l'argument de la boucle : un compteur démarré à 10 répond 13, 11, 11 à « ajoute 3, ajoute -2, lis », et 13, 8, 10 si l'ajout rappelle `loop(value)`.

Un monitor change toute fin en message `:DOWN` ; un lien propage une fin anormale, sauf à qui l'intercepte. Un superviseur relance depuis la recette : l'Agent qui valait 7 repart de 0.

Confier un état à un serveur

SÉANCE AP7 · AGENT, GENSERVER ET SUPERVISION

Imaginons un tableau qui compte les contributions de toute une classe : chaque fois qu'un élève termine un exercice, son poste ajoute un au total, et chacun peut consulter ce total quand il le souhaite. Des dizaines de clients écrivent donc dans le même nombre, parfois au même instant. Où ranger ce nombre pour qu'aucune contribution ne se perde ?

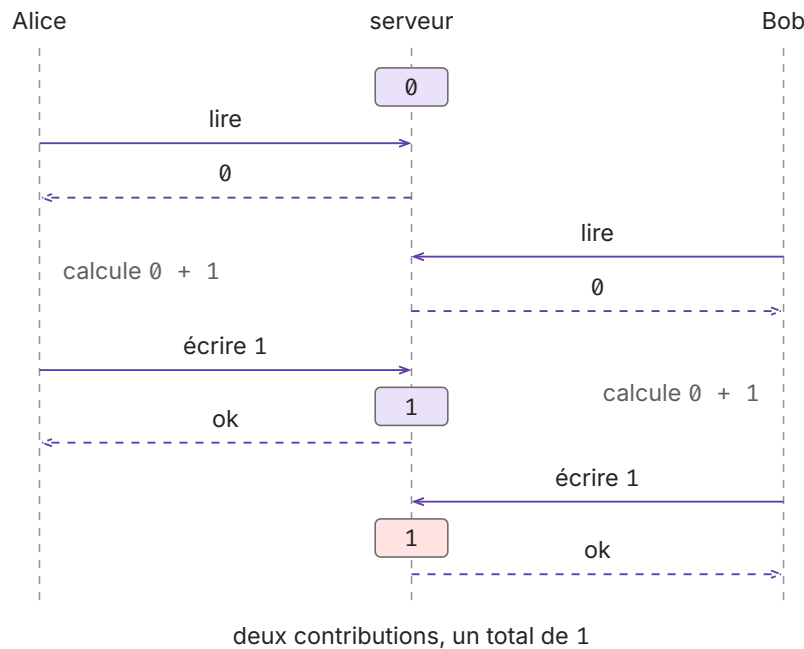
Le chapitre 9 a donné la réponse de principe : confier l'état à un seul processus, qui traite ses messages un par un. Ce chapitre en fait une brique solide. Elixir fournit pour cela, au-dessus d'OTP, `Agent` et `GenServer`, qui écrivent à notre place la boucle de réception ; il permet aussi de nommer ce processus et de le faire relancer par un superviseur. Nous verrons enfin ce que coûte un serveur unique, et comment le tester.

Le projet `compteur-local` d'AP7 construit exactement ce tableau. Nous le suivrons comme exemple, à travers son contrat et son architecture ; les mécanismes, eux, sont expliqués sur de petits modules propres à ce chapitre.

10.1 Un total, plusieurs clients

10.1.1 Lire, puis écrire

Supposons d'abord que le total soit gardé par un serveur très simple, qui ne connaît que deux requêtes : « lis le total » et « écris ce nombre ». Pour ajouter sa contribution, un client lit le total, calcule ce total plus un, puis écrit le résultat. Cela fait deux requêtes. Que se passe-t-il si Alice et Bob procèdent ainsi en même temps, à partir d'un total nul ?



Un déroulement possible, qui se lit de haut en bas. Les flèches pleines portent les requêtes, les flèches en pointillé les réponses ; les cases posées sur la ligne du serveur montrent son total. Alice et Bob lisent tous deux 0 avant d'écrire.

Le serveur a traité quatre requêtes, chacune d'un seul tenant, l'une après l'autre. Pourtant, le total vaut 1 : Bob a écrit $0 + 1$ en se fondant sur une lecture devenue fausse entre-temps. Une contribution est perdue, et aucun message ne s'est égaré !

C'est la condition de course du chapitre 9, qui opposait deux threads autour d'une variable partagée. Elle s'est simplement déplacée d'un cran : des instructions d'un programme vers les requêtes adressées à un serveur.

Le schéma ne montre qu'un déroulement parmi d'autres. Les deux requêtes d'Alice arrivent dans l'ordre où elle les envoie, celles de Bob aussi, mais rien ne fixe la façon dont les deux suites s'entrelacent. Notons L_A la lecture d'Alice, E_A son écriture, et de même L_B et E_B pour Bob. Un entrelacement revient à choisir les deux places d'Alice parmi les quatre arrivées : il y en a $\binom{4}{2} = 6$.

ordre d'arrivée au serveur		total final
L_A, E_A, L_B, E_B	Alice, puis Bob	2
L_B, E_B, L_A, E_A	Bob, puis Alice	2
L_A, L_B, E_A, E_B		1
L_A, L_B, E_B, E_A		1
L_B, L_A, E_A, E_B		1
L_B, L_A, E_B, E_A		1

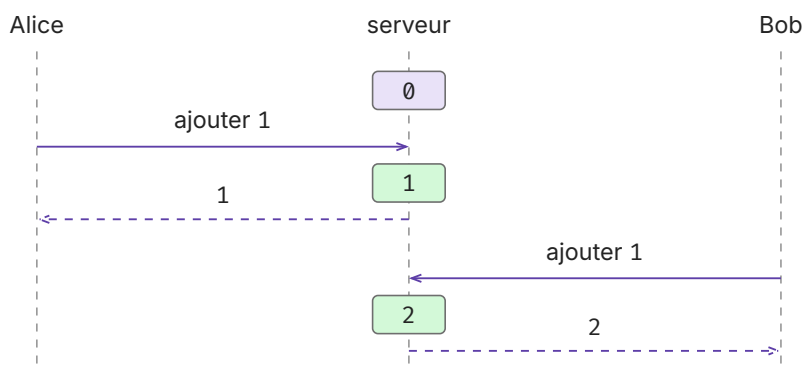
Seuls les deux ordres où l'un des clients a fini avant que l'autre ne lise donnent 2. Le résultat dépend donc de l'ordonnancement, c-à-d. de hasards que le programme ne maîtrise pas : un test peut passer cent fois de suite, puis échouer le jour où la machine est chargée.

POUR ALLER PLUS LOIN

Avec n clients qui lisent puis écrivent, combien d'entrelacements ? Les $2n$ requêtes peuvent arriver dans $(2n)!$ ordres, mais l'ordre interne de chaque client est imposé, ce qui divise par 2 pour chacun : il reste $(2n)!/2^n$ entrelacements, soit 6 pour deux clients et 90 pour trois. Ceux qui ne perdent rien enchaînent, pour chaque client, la lecture et l'écriture sans interruption : ce sont les $n!$ ordres des clients, 2 puis 6. Pour dix clients, à peu près un entrelacement sur 650 millions ne perd rien. Ce décompte ne dit pas la fréquence réelle de chaque ordre, qui dépend de l'ordonnancement ; il montre en revanche qu'on ne peut pas les essayer à la main. L'outil [Concuerror](https://concuerror.com/)⁵³ explore systématiquement les ordonnancements d'un programme Erlang.

10.1.2 Une requête qui porte toute la décision

Le remède ne consiste pas à aller plus vite, ni à espérer un ordre favorable : il consiste à changer de protocole. Au lieu de « écris 1 », le client envoie « ajoute 1 », et c'est le serveur qui calcule le nouveau total à partir de son état courant. La lecture, le calcul et l'écriture se font alors dans le traitement d'une seule requête.



lecture, calcul et écriture dans un seul traitement

Chaque requête « ajouter 1 » lit et modifie le total dans le même traitement. Si Bob passe avant Alice, leurs réponses s'échangent, mais le total final vaut encore 2.

Une telle requête est dite *atomique* : vue de l'extérieur, elle est indivisible. Aucun autre client ne peut observer l'état, ni le modifier, au milieu de son traitement.

Retenez bien que cette garantie porte sur une requête, et non sur une suite de requêtes. Deux appels successifs d'un même client ne forment pas un bloc, et un autre client peut toujours s'intercaler entre eux. La règle vaut pour toute décision qui dépend de l'état : « réserver la dernière place si elle est libre » doit vérifier et réserver dans la même requête, alors que « lire le nombre de places », puis « réserver », laisse deux clients obtenir la même place.

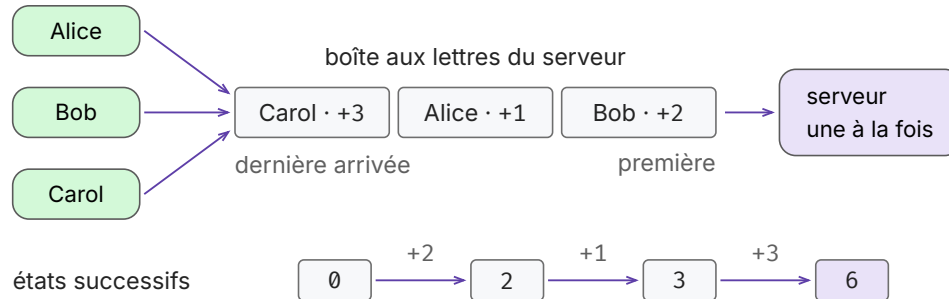
10.1.3 Un propriétaire pour l'état

Cette organisation porte un nom, le motif *client/serveur*. Le *serveur* est le processus propriétaire de l'état : lui seul le lit et le modifie. Les *clients* sont tous les processus qui lui adressent des *requêtes* et attendent, le cas échéant, ses *réponses*. Le *protocole* est la liste des requêtes que le serveur accepte, avec la réponse de chacune.

⁵³ <https://concuerror.com/>

Notez que ces mots ne désignent pas ici des machines : client et serveur sont deux processus de la même VM, et une requête n'est qu'un message.

Pourquoi le serveur ne perd-il aucune contribution ? Parce que sa boîte aux lettres range les requêtes concurrentes en une file, et qu'il les traite une par une, chacune en entier.



Alice, Bob et Carol envoient leurs requêtes au même moment ; elles arrivent dans un ordre que personne ne choisit, ici Bob, Alice, puis Carol. Le serveur prend la plus ancienne, la traite entièrement, puis passe à la suivante.

On dit que le serveur *séréalise* les requêtes : des demandes concurrentes deviennent une suite. L'ordre de cette suite reste incertain, mais il importe peu ici, car des additions peuvent s'effectuer dans n'importe quel ordre : les réponses intermédiaires changent, pas le total final.

Il en irait autrement de requêtes qui ne commutent pas. Avec « écris 5 » et « écris 8 », envoyées par deux clients, l'état final dépend de l'ordre d'arrivée. Le serveur unique empêche les pertes à l'intérieur d'une requête ; il ne fixe pas l'ordre entre les clients.

POUR ALLER PLUS LOIN

Un serveur unique offre une garantie qui porte un nom, la *linéarisabilité* : chaque requête semble prendre effet en un instant précis, situé entre son envoi et sa réponse. Pour notre serveur, cet instant est celui où il traite la requête. En clair, les réponses observées par tous les clients s'expliquent toujours par une suite de requêtes traitées une à une, compatible avec ce que chacun a vu. C'est l'une des garanties les plus fortes que l'on puisse demander à un objet partagé, et l'une des plus coûteuses à obtenir quand l'état est recopié sur plusieurs machines ; le chapitre 11 en choisira une plus faible. Le site de Jepsen en donne une [définition illustrée](https://jepsen.io/consistency/models/linearizable)⁵⁴.

10.2 Agent : une valeur gardée par un processus

10.2.1 Lire et modifier par des fonctions

La brique la plus simple s'appelle `Agent` : un processus qui garde une valeur et l'expose par quelques fonctions. `Agent.start_link(f)` le démarre ; la fonction `f`, exécutée dans le nouveau processus, calcule l'état initial, et l'appel rend `{:ok, pid}`. Trois opérations reçoivent ensuite chacune une fonction :

- `Agent.get(agent, f)` applique `f` à l'état et rend le résultat, sans changer l'état ;

⁵⁴ <https://jepsen.io/consistency/models/linearizable>

- `Agent.update(agent, f)` remplace l'état par le résultat de `f` et rend `:ok` ;
- `Agent.get_and_update(agent, f)` attend de `f` une paire : la réponse, puis l'état suivant.

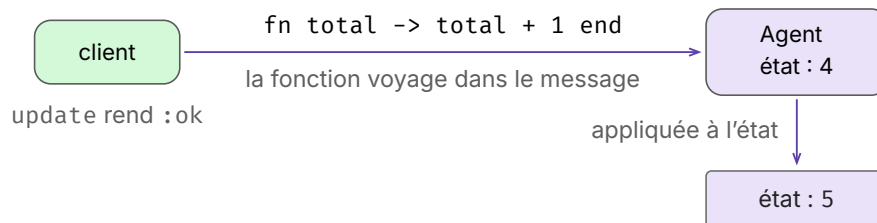
```
1 {:ok, board} = Agent.start_link(fn -> 0 end)

Agent.update(board, fn total -> total + 1 end)  :ok
Agent.update(board, fn total -> total + 2 end)  :ok
Agent.get(board, fn total -> total end)         3
Agent.get(board, fn total -> 10 * total end)    30
Agent.get(board, fn total -> total end)         3
```

`update` rend `:ok`, et non le nouveau total. La fonction passée à `get` décide de la réponse, mais ne touche pas à l'état : l'Agent calcule `30` pour nous, et garde `3`.

10.2.2 La fonction voyage jusqu'à l'Agent

Où s'exécutent ces fonctions ? Pas chez l'appelant : dans le processus de l'Agent. Une fonction est une valeur comme une autre ; elle peut donc voyager dans un message, avec les valeurs qu'elle a capturées, comme les fermetures du chapitre 5. L'Agent la reçoit, l'applique à son état et garde, le cas échéant, le résultat.



Le client n'envoie pas un nombre, mais une fonction. L'Agent l'exécute sur son propre état, `4`, et conserve le résultat, `5` ; le client ne voit jamais cet état.

Puisque l'Agent traite ses messages un par un, une mise à jour qui calcule le nouvel état à partir de l'ancien, comme `fn total -> total + 1 end`, forme exactement la requête atomique « ajoute 1 » de la section précédente. Quand le client veut aussi une réponse calculée sur l'état, `get_and_update` lit et modifie dans la même requête : le premier élément de la paire est la réponse, le second le prochain état.

```
1 {:ok, agent} = Agent.start_link(fn -> 20 end)
2 try do
3   previous = Agent.get_and_update(agent, fn current ->
4     {current, 23}
5   end)
6   {previous, Agent.get(agent, fn current -> current end)}
7 after
8   Agent.stop(agent)
9 end
```

```
VALEUR {20, 23}
```

Le client reçoit `20`, l'ancienne valeur, tandis que l'Agent garde `23`. Comme au chapitre 9, le `try ... after` arrête l'Agent à la fin de l'essai, même si une ligne échoue.

10.2.3 Lire puis écrire, même avec un Agent

L'Agent ne protège pas pour autant d'un mauvais découpage des requêtes. Jouons à la main, dans un seul processus, l'entrelacement de la première figure : Alice lit, Bob lit, puis chacun écrit la valeur qu'il a lue, plus un.

```
1 {:ok, board} = Agent.start_link(fn -> 0 end)

alice = Agent.get(board, fn total -> total end)    0
bob = Agent.get(board, fn total -> total end)      0
Agent.update(board, fn _ -> alice + 1 end)         :ok
Agent.update(board, fn _ -> bob + 1 end)           :ok
Agent.get(board, fn total -> total end)           1
```

Les fonctions `fn _ -> alice + 1 end` ignorent l'état courant : elles écrivent un nombre calculé chez le client, à partir d'une lecture ancienne. Deux processus clients pourraient produire exactement cet ordre ; l'écrire à la main le rend simplement reproductible, ce qui nous resservira pour tester. Avec `fn total -> total + 1 end`, calculée dans l'Agent, le total vaudrait 2 quel que soit l'ordre.

10.2.4 Ce qu'un Agent ne sait pas faire

L'Agent suffit pour garder une valeur partagée par quelques fonctions d'un même module. Il montre pourtant vite ses limites.

- Il n'a pas de protocole. Chaque client lui envoie la fonction de son choix : rien n'empêche d'envoyer `fn _ -> 0 end` par erreur, et les règles du service, par ex. « on ne retire jamais de contribution », sont dispersées chez tous les appelants.
- Une fonction lente bloque tout le monde : elle s'exécute dans l'Agent, et toutes les requêtes suivantes attendent sa fin.
- Il ne réagit qu'à son interface : il ne sait pas traiter un autre message, comme le signal d'un minuteur ou l'avis de fin d'un processus surveillé.

Dès que l'on veut fixer la liste des requêtes permises, tester chacune et décider de son traitement, on écrit un `GenServer`. L'Agent est d'ailleurs lui-même construit sur un `GenServer`, dont le protocole se résume à « applique cette fonction ».

10.3 Une transition et une boucle

Avant d'écrire notre premier `GenServer`, séparons les deux moitiés de tout serveur : ce qu'il fait d'une requête, et la façon dont il enchaîne les requêtes.

10.3.1 La transition, une fonction pure

Prenons un thermomètre qui mémorise la dernière température relevée. Il accepte deux requêtes : `:value` demande la température courante, et `{:record, next}` enregistre une nouvelle valeur. Ce qu'il fait d'une requête tient dans une fonction pure, qui reçoit la requête et l'état courant, puis rend une paire : la réponse, et l'état suivant. On l'appelle une *transition*.

```

1 step = fn
2   :value, value -> {value, value}
3   {:record, next}, _value -> {:ok, next}
4 end
5 requests = [{:record, 23}, :value, {:record, 18}, :value]

step.(:value, 20)           {20, 20}
step.({:record, 23}, 20)    {:ok, 23}
Enum.map_reduce(requests, 20, step) [{:ok, 23, :ok, 18}, 18]

```

Une fonction anonyme peut avoir plusieurs clauses, comme une fonction nommée : la première dont le motif correspond est choisie. Lire `:value` rend l'état sans le changer ; enregistrer 23 répond `:ok` et fait de 23 l'état suivant.

`Enum.map_reduce/3` ressemble à `Enum.reduce/3` du chapitre 4, à ceci près que sa fonction rend une paire : un élément à ranger dans une liste, puis l'accumulateur suivant. Le résultat réunit la liste et l'accumulateur final. Ici, la liste rassemble les réponses, et l'accumulateur, c'est l'état.



Les quatre états successifs du thermomètre. Chaque flèche est une transition : la requête est écrite au-dessus, la réponse en dessous, et l'état d'arrivée devient le point de départ de la requête suivante.

Retenez cette image : un serveur fait un `map_reduce` sur sa boîte aux lettres. Il prend les requêtes dans leur ordre d'arrivée, rend une réponse pour chacune et transmet l'état à la suivante. Tout ce qui relève de la concurrence se ramène alors à une seule question : dans quel ordre la liste des requêtes s'est-elle formée ?

POUR ALLER PLUS LOIN

Un automate qui, à chaque symbole lu, change d'état et produit un symbole en sortie s'appelle une *machine de Mealy*⁵⁵ (1955). Sa définition tient en deux fonctions, l'état suivant $\delta(q, a)$ et la sortie $\lambda(q, a)$, que notre transition rend ensemble, dans une paire. Seule différence : un automate a un nombre fini d'états, alors que notre thermomètre peut garder n'importe quel nombre. Pour les serveurs dont l'état est une phase nommée (porte ouverte, fermée, verrouillée), OTP propose le comportement `:gen_statem`.

10.3.2 La boucle, écrite une fois pour toutes

L'autre moitié est la boucle qui reçoit les requêtes, appelle la transition et renvoie les réponses. Au chapitre 9, la boucle de `Memory` mêlait les deux moitiés : chaque branche du `receive` décidait à la fois de la réponse et de l'état suivant. Séparons-les, en écrivant une boucle qui reçoit la transition en argument.

```

1 defmodule MiniServer do
2   def start(step, state), do: spawn(fn -> loop(step, state) end)

```

⁵⁵ https://fr.wikipedia.org/wiki/Machine_de_Mealy

```

3
4  def call(server, request) do
5      ref = make_ref()
6      send(server, {:call, self(), ref, request})
7      receive do
8          {:reply, ^ref, answer} -> answer
9      after
10         5_000 -> exit(:timeout)
11     end
12 end
13
14 defp loop(step, state) do
15     receive do
16         {:call, client, ref, request} ->
17             {answer, next} = step.(request, state)
18             send(client, {:reply, ref, answer})
19             loop(step, next)
20     end
21 end
22 end

```

`call` s'exécute chez le client : il marque sa requête d'une référence neuve, l'envoie avec sa propre adresse, puis attend la réponse marquée de la même référence. `loop` s'exécute dans le serveur : elle retire une requête, applique la transition, répond, puis se rappelle avec l'état suivant. `exit(:timeout)` termine l'attente par une *sortie*, que nous retrouverons avec `GenServer.call`. Notre thermomètre devient un serveur sans une ligne de plus :

```

1 sensor = MiniServer.start(step, 20)
2 try do
3     :ok = MiniServer.call(sensor, {:record, 23})
4     MiniServer.call(sensor, :value)
5 after
6     Process.exit(sensor, :kill)
7 end

```

VALEUR 23

La boucle ne sait rien des thermomètres, et la transition ne sait rien des messages. C'est exactement le partage des rôles d'OTP. Le module `GenServer`, pour *generic server*, donne accès à la boucle générique d'OTP, éprouvée depuis des décennies ; nous n'écrivons que la partie propre à notre service, sous forme de fonctions qu'il appelle, les *callbacks*. Ce contrat entre une partie générique et des callbacks s'appelle un *comportement* (*behaviour*).

La boucle d'OTP fait aussi bien plus que la nôtre :

notre MiniServer	GenServer
spawn	start_link : lie le serveur à son parent et attend la fin de init
une référence et un <code>receive</code> borné	call : une référence, un monitor du serveur, un délai réglable

notre <code>MiniServer</code>	<code>GenServer</code>
la transition <code>step</code>	les callbacks <code>handle_call</code> , <code>handle_cast</code> et <code>handle_info</code>
les autres messages s'accumulent dans la boîte	<code>handle_info</code> reçoit les messages bruts
un PID seulement	un nom enregistré, au besoin

Le monitor a son importance : si notre serveur meurt avant de répondre, `MiniServer.call` attend ses cinq secondes pour rien. `GenServer.call`, qui surveille le serveur pendant l'attente, échoue dès la réception du message `:DOWN`.

10.4 GenServer : le contrat entre client et serveur

10.4.1 Un premier GenServer

Voici le thermomètre écrit avec `GenServer`. Ses deux clauses `handle_call` reprennent la transition `step`, avec un argument de plus et une réponse emballée dans un tuple.

```

1 defmodule Thermometer do
2   use GenServer
3   def start_link(value) do
4     GenServer.start_link(__MODULE__, value)
5   end
6   def value(pid), do: GenServer.call(pid, :value)
7   def record(pid, value) do
8     GenServer.call(pid, {:record, value})
9   end
10  @impl true
11  def init(value), do: {:ok, value}
12  @impl true
13  def handle_call(:value, _from, value) do
14    {:reply, value, value}
15  end
16  def handle_call({:record, next}, _from, _value) do
17    {:reply, :ok, next}
18  end
19 end

```

Quelques mots sur la forme de ce module. `use GenServer` déclare que le module implémente ce comportement et fournit des callbacks par défaut. La boucle de réception, celle que nous venons d'écrire dans `MiniServer`, est celle d'OTP : lancée par `GenServer.start_link`, c'est elle qui appelle `init/1` au démarrage, puis `handle_call/3` à chaque requête.

`init/1` reçoit l'argument de `start_link` et rend l'état initial, emballé dans `{:ok, état}`. `handle_call/3` reçoit la requête, l'expéditeur et l'état courant. `@impl true` déclare que la fonction qui suit est l'un de ces callbacks, ce qui permet au compilateur de vérifier son nom et son arité ; `__MODULE__` désigne le module courant.

Dans `{:reply, answer, next_state}`, la réponse et l'état sont deux choses différentes : enregistrer 23 répond `:ok`, mais conserve 23. Le paramètre `_from` contient les informations

permettant de répondre ; notre callback n'a pas besoin de les manipuler pour une réponse immédiate.

```

1 { :ok, sensor } = Thermometer.start_link(20)
2 try do
3   :ok = Thermometer.record(sensor, 23)
4   Thermometer.value(sensor)
5 after
6   GenServer.stop(sensor)
7 end

```

VALEUR 23

10.4.2 Démarrer : `start_link` et `init`

`Thermometer.start_link(20)` appelle `GenServer.start_link(__MODULE__, 20)`. OTP crée alors le processus serveur, lié à l'appelant ; dans ce nouveau processus, il appelle `init(20)`, et il attend le résultat de `init` avant de rendre `{:ok, pid}` à l'appelant.

Ainsi, quand `start_link` rend la main, le serveur est prêt : on peut l'appeler aussitôt. En revanche, un `init` lent retarde celui qui démarre le serveur, par ex. toute une application à son lancement ; on n'y met que le nécessaire.

`init` peut aussi refuser de démarrer, en rendant `{:stop, raison}` : `start_link` rend alors `{:error, raison}`, et aucun serveur n'existe. Enfin, le lien créé par `start_link` est celui du chapitre 9 : si l'appelant se termine anormalement, le serveur s'arrête avec lui, et inversement. Pour un essai isolé dans IEx, `GenServer.start` démarre le même serveur sans lien.

10.4.3 La réponse et l'état suivant

Dans `{:reply, réponse, état}`, la réponse part vers le client, et l'état devient l'argument du prochain traitement. Ce sont deux valeurs distinctes, et les confondre est l'erreur la plus fréquente. Le distributeur de tickets suivant, comme à l'entrée d'une préfecture, rend le numéro courant et garde le suivant.

```

1 defmodule Ticket do
2   use GenServer
3   def start_link(first) do
4     GenServer.start_link(__MODULE__, first)
5   end
6   def take(server), do: GenServer.call(server, :take)
7
8   @impl true
9   def init(first), do: {:ok, first}
10
11   @impl true
12   def handle_call(:take, _from, number) do
13     {:reply, number, number + 1}
14   end
15 end

```

```

1 { :ok, tickets } = Ticket.start_link(1)
2 try do
3   { Ticket.take(tickets), Ticket.take(tickets), Ticket.take(tickets) }
4 after
5   GenServer.stop(tickets)
6 end

```

VALEUR {1, 2, 3}

Suivons l'état de requête en requête : chaque ligne part de l'état laissé par la précédente.

requête	état avant	réponse	état après
:take	1	1	2
:take	2	2	3
:take	3	3	4

RÉPONDRE JUSTE, MAIS OUBLIER DE GARDER

Une clause qui rendrait `{:reply, number, number}` répondrait correctement au premier client, puis donnerait le ticket 1 à tout le monde : la réponse est partie, mais l'état n'a pas avancé. Ce défaut ne se voit pas sur une seule requête, il apparaît à la suivante. Un test utile enchaîne donc toujours au moins deux requêtes.

10.4.4 Le contrat du tableau de contributions

Le projet `compteur-local` applique ce motif au tableau du début du chapitre. Son serveur, `CounterLocal.Counter`, garde un total entier. Trois fonctions d'interface envoient chacune une requête par `GenServer.call`, et chacune répond le total après traitement. En notant t le total courant :

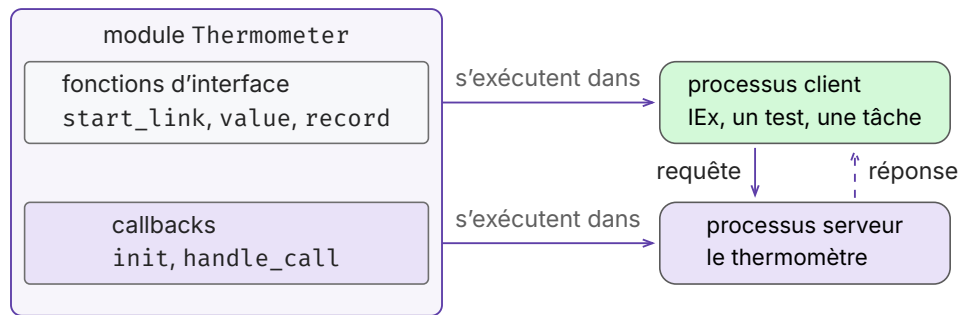
fonction	requête	réponse	total suivant
<code>value(c)</code>	<code>:value</code>	t	t
<code>increment(c, n)</code>	<code>{:increment, n}</code>	$t + n$	$t + n$
<code>reset(c)</code>	<code>:reset</code>	0	0

`c` désigne le compteur, par son PID ou par son nom. Dans la déclaration `def increment(server, amount \\ 1)`, les deux barres obliques inversées donnent à `amount` une valeur par défaut : `increment(c)` ajoute 1. Le contrat ne porte que sur des entiers positifs ou nuls. Ainsi, à partir de zéro, `increment(c)` répond 1, `increment(c, 4)` répond 5, et `value(c)` lit 5.

Ce tableau est un contrat, c-à-d. ce que vérifient les tests du projet ; écrire les clauses qui le réalisent est le travail de la séance. Notez enfin que deux compteurs démarrés à partir du même module sont deux processus, avec deux états indépendants : ajouter au premier ne change pas le second.

10.4.5 Qui exécute quoi ?

Un module `GenServer` réunit dans un même fichier du code qui s'exécute dans deux processus différents. Les fonctions d'interface, `start_link`, `value` et `record`, s'exécutent chez l'appelant ; les callbacks, `init` et `handle_call`, s'exécutent dans le serveur.



Un fichier, deux processus. Le client exécute les fonctions d'interface, qui envoient une requête ; le serveur exécute les callbacks, qui calculent la réponse. Ni l'un ni l'autre n'exécute jamais le code de l'autre moitié.

Vérifions-le avec `self()`, qui désigne toujours le processus qui l'exécute. La fonction d'interface `ask` l'évalue avant d'envoyer sa requête ; le callback l'évalue en la traitant.

```

1 defmodule Who do
2   use GenServer
3   def start_link(arg), do: GenServer.start_link(__MODULE__, arg)
4   def ask(server), do: {self(), GenServer.call(server, :who)}
5
6   @impl true
7   def init(arg), do: {:ok, arg}
8
9   @impl true
10  def handle_call(:who, _from, state) do
11    {:reply, self(), state}
12  end
13 end

```

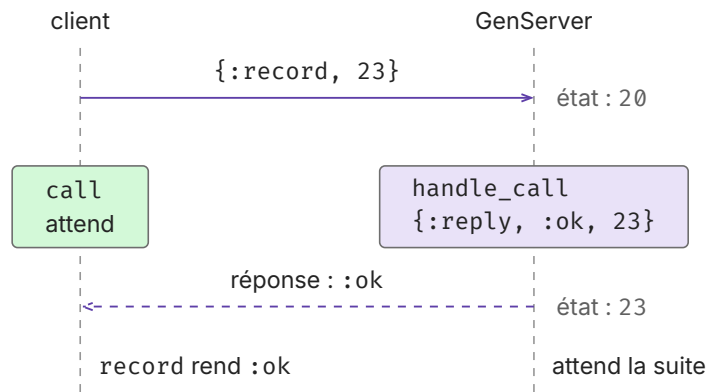
```

1 {:ok, server} = Who.start_link(nil)
2 try do
3   {client_side, server_side} = Who.ask(server)
4   {client_side == self(), server_side == server}
5 after
6   GenServer.stop(server)
7 end

```

VALEUR {true, true}

Le premier `self()` est celui de l'appelant, ici la cellule ou le test qui exécute l'essai ; le second est le PID du serveur. Pour `Thermometer.record(sensor, 23)`, suivons maintenant séparément la réponse et l'état conservé, sur un diagramme de séquence comme au chapitre 9.



Les flèches horizontales transportent la requête puis la réponse. Pendant que le client attend, le callback s'exécute dans le serveur. Le client reçoit `:ok`, tandis que `23` devient l'état du prochain traitement.

Le tableau suivant détaille les mêmes étapes, en indiquant le processus qui exécute chacune. Il montre que `GenServer.call` n'a rien de magique : il reprend le protocole de `MiniServer`, avec un monitor en plus.

	processus	ce qui se passe
1	client	<code>Thermometer.record(sensor, 23)</code> appelle <code>GenServer.call(sensor, {:record, 23})</code> .
2	client	<code>call</code> crée une référence, surveille le serveur par un monitor, lui envoie la requête avec l'adresse du client et la référence, puis attend dans un <code>receive</code> .
3	serveur	La boucle d'OTP, qui attendait dans son propre <code>receive</code> , retire la requête de sa boîte.
4	serveur	Elle appelle notre callback <code>handle_call({:record, 23}, from, 20)</code> .
5	serveur	Le callback rend <code>{:reply, :ok, 23}</code> ; la boucle envoie <code>:ok</code> au client avec la référence, puis se rappelle avec l'état <code>23</code> .
6	client	Le <code>receive</code> de <code>call</code> reconnaît la référence ; <code>record rend :ok</code> .

Cette répartition a des conséquences pratiques. Une vérification écrite dans une fonction d'interface, par ex. une garde `when is_integer(amount) and amount >= 0` sur `increment`, s'exécute chez le client : un appel invalide y échoue avant tout envoi, et le serveur n'en sait rien. Une erreur dans un callback, en revanche, se produit dans le serveur : elle l'arrête, et avec lui l'état que partagent tous les clients. Enfin, dans un callback, `self()` désigne le serveur, jamais le client ; c'est `from` qui identifie l'appelant.

10.5 `call`, `cast` ou `send` ?

Un client dispose de trois manières d'écrire à un `GenServer`. Chacune arrive dans un callback différent, et seule la première attend une réponse.

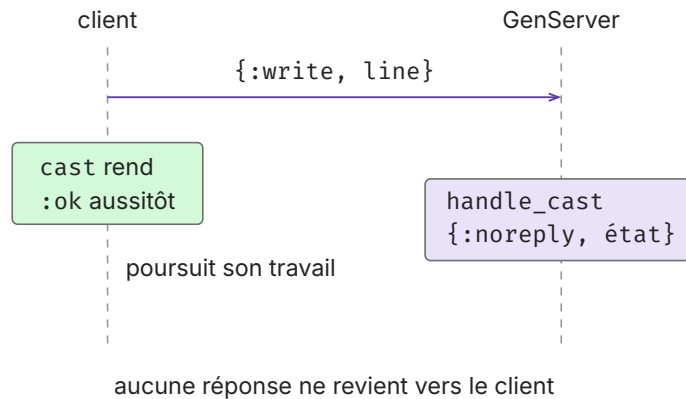
10.5.1 `call` attend la réponse

`GenServer.call(server, requête)` est *synchrone* : le client attend la réponse, et quand `call` rend la main, le serveur a fini de traiter la requête. C'est ce que montrait le diagramme précédent. Cette attente rend les expériences faciles à lire : quand `increment` répond 5,

l'ajout est fait, et le total a bien valu 5. Le projet `compteur-local` utilise donc `call` pour ses trois requêtes, y compris la remise à zéro.

10.5.2 `cast` n'attend rien

`GenServer.cast(server, requête)` dépose la requête dans la boîte du serveur et rend `:ok` aussitôt, sans attendre aucun traitement. Le serveur la traitera plus tard, dans `handle_cast(requête, état)`, qui rend `{:noreply, état}` : il n'y a personne à qui répondre.



Le client dépose sa requête et poursuit son travail. Le serveur la traite quand il arrive à elle dans sa boîte ; son callback ne produit qu'un état suivant, et aucune flèche ne revient.

Ce `:ok` immédiat ne prouve rien : ni que la requête a été traitée, ni même que le serveur existe encore. Un `cast` convient aux requêtes dont le client n'a pas besoin de connaître l'issue, par ex. une ligne de journal. Ce n'est pas une version « plus rapide » de `call`, à employer par défaut.

10.5.3 `send` et `handle_info`

Un `GenServer` reste un processus : on peut aussi lui envoyer un message ordinaire avec `send`. Ces messages bruts, que ni `call` ni `cast` n'ont emballés, arrivent dans `handle_info(message, état)`, qui rend lui aussi `{:noreply, état}`.

C'est par là qu'arrivent, par ex., le signal d'un minuteur que le serveur s'est programmé avec `Process.send_after/3`, les messages `:DOWN` des processus qu'il surveille, ou les messages d'un processus qui ignore tout d'OTP. Sans clause `handle_info`, le comportement par défaut signale le message inattendu dans le journal, puis l'ignore.

Le journal suivant accepte une ligne par chacun des trois chemins.

```

1 defmodule Journal do
2   use GenServer
3   def start_link(lines) do
4     GenServer.start_link(__MODULE__, lines)
5   end
6   def write(log, line), do: GenServer.call(log, {:write, line})
7   def post(log, line), do: GenServer.cast(log, {:write, line})
8   def lines(log), do: GenServer.call(log, :lines)
9
10  @impl true
11  def init(lines), do: {:ok, lines}
12

```

```

13 @impl true
14 def handle_call({:write, line}, _from, lines) do
15   {:reply, :ok, [line | lines]}
16 end
17 def handle_call(:lines, _from, lines) do
18   {:reply, Enum.reverse(lines), lines}
19 end
20
21 @impl true
22 def handle_cast({:write, line}, lines) do
23   {:noreply, [line | lines]}
24 end
25
26 @impl true
27 def handle_info({:note, line}, lines) do
28   {:noreply, [line | lines]}
29 end
30 end

```

```

1 {:ok, log} = Journal.start_link([])
2 try do
3   :ok = Journal.write(log, "par call")
4   :ok = Journal.post(log, "par cast")
5   {:note, _} = send(log, {:note, "par send"})
6   Journal.lines(log)
7 after
8   GenServer.stop(log)
9 end

```

```
VALEUR ["par call", "par cast", "par send"]
```

Les trois messages viennent du même client : ils arrivent dans leur ordre d'envoi, et la lecture, envoyée en dernier, est traitée après eux. C'est pourquoi elle voit la ligne du `cast`, sans que personne ait attendu sa réponse. Le journal ajoute chaque ligne en tête de liste, l'opération la moins coûteuse sur une liste ; `lines` remet la liste dans l'ordre d'arrivée.

10.5.4 Ce que rend le client

Le tableau suivant résume les quatre callbacks d'un `GenServer`, avec le démarrage, qui appelle `init/1`. Les trois premières lignes désignent des fonctions du module `GenServer` ; `M` y est le module du serveur, `__MODULE__` dans `Thermometer`, et `s` le serveur, par son PID ou son nom.

le client écrit	le serveur exécute	le callback rend	le client reçoit
<code>start_link(M, arg)</code>	<code>init(arg)</code>	<code>{:ok, état}</code>	<code>{:ok, pid}</code> , après <code>init</code>
<code>call(s, req)</code>	<code>handle_call(req, from, état)</code>	<code>{:reply, réponse, état}</code>	la réponse, sous 5 s par défaut
<code>cast(s, req)</code>	<code>handle_cast(req, état)</code>	<code>{:noreply, état}</code>	<code>:ok</code> , aussitôt
<code>send(s, msg)</code>	<code>handle_info(msg, état)</code>	<code>{:noreply, état}</code>	<code>msg</code> , aussitôt

Que se passe-t-il si le serveur n'existe plus ? Essayons les trois envois vers un processus déjà terminé.

```

1 dead = spawn(fn -> :ok end)
2 ref = Process.monitor(dead)
3 receive do
4   {:DOWN, ^ref, :process, ^dead, _reason} -> :ok
5 end
6 call =
7   try do
8     GenServer.call(dead, :hello)
9   catch
10    :exit, {reason, _details} -> reason
11  end
12 {GenServer.cast(dead, :hello), send(dead, :hello), call}

```

```
VALEUR {:ok, :hello, :noproc}
```

`cast` et `send` rendent leur valeur habituelle, sans remarquer l'absence du destinataire. `call`, lui, ne peut rendre aucune réponse : il provoque une *sortie* (*exit*) chez le client, avec une raison qui commence par `:noproc`, « pas de processus ».

Une sortie n'est pas une exception levée par `raise`. On l'intercepte, quand on y tient, par une clause `catch :exit, raison` dans un `try`, comme ci-dessus ; le plus souvent, on la laisse terminer le client, qui sera relancé s'il est supervisé.

10.5.5 Le délai d'attente de `call`

Un `call` n'attend pas indéfiniment. Son délai vaut 5 secondes par défaut, et l'on peut en donner un autre, en millisecondes, comme troisième argument de `GenServer.call`. Si la réponse n'est pas arrivée à temps, l'appel provoque une sortie chez le client, avec la raison `:timeout`. Contrairement à la branche `after` d'un `receive`, l'expiration ne rend donc aucune valeur.

Surtout, l'expiration termine l'attente du client ; elle n'annule pas le traitement engagé. Pour le voir, prenons un livre d'or dont chaque signature demande un traitement long, simulé ici par `Process.sleep(200)`.

```

1 defmodule GuestBook do
2   use GenServer
3   def start_link(names) do
4     GenServer.start_link(__MODULE__, names)
5   end
6   def sign(book, name, timeout \\ 5_000) do
7     GenServer.call(book, {:sign, name}, timeout)
8   end
9   def names(book), do: GenServer.call(book, :names)
10
11  @impl true
12  def init(names), do: {:ok, names}
13
14  @impl true
15  def handle_call({:sign, name}, _from, names) do
16    Process.sleep(200)
17    {:reply, :ok, [name | names]}
18  end

```

```

19 def handle_call(:names, _from, names) do
20   {:reply, Enum.reverse(names), names}
21 end
22 end

```

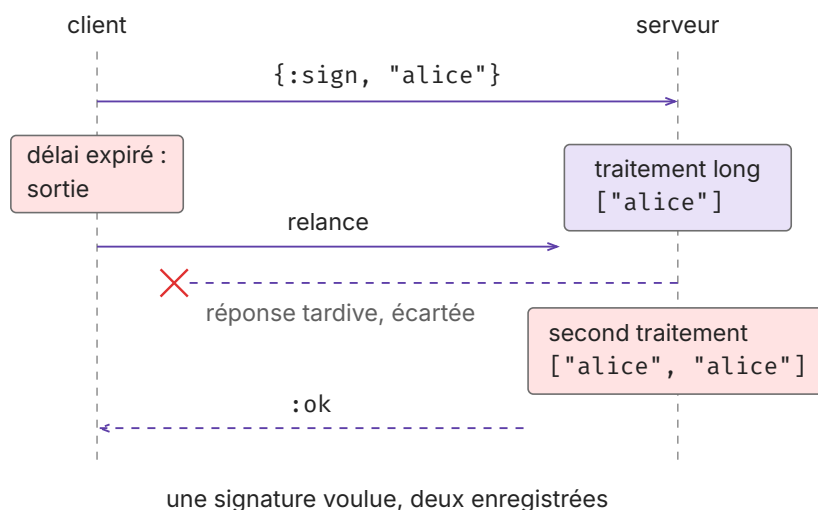
Alice signe avec un délai de 50 millisecondes, trop court ; elle intercepte la sortie, puis relance sa signature avec le délai par défaut.

```

1 {:ok, book} = GuestBook.start_link([])
2 try do
3   first =
4     try do
5       GuestBook.sign(book, "alice", 50)
6     catch
7       :exit, {reason, _details} -> reason
8     end
9   {first, GuestBook.sign(book, "alice"), GuestBook.names(book)}
10 after
11   GenServer.stop(book)
12 end

```

VALEUR {:timeout, :ok, ["alice", "alice"]}



La première requête est traitée malgré l'expiration ; sa réponse arrive trop tard et le client l'écarte. La relance, arrivée dans la boîte pendant ce temps, est traitée à son tour.

Alice a voulu signer une fois, et elle figure deux fois. Sans réponse, elle ne pouvait pas savoir si sa première requête avait été traitée ; la relancer l'a fait compter deux fois. Notez que la réponse tardive ne reste pas dans la boîte du client : depuis OTP 24, une réponse arrivée après l'expiration est écartée. Les délais choisis ici illustrent le mécanisme ; ce ne sont pas des mesures.

POUR ALLER PLUS LOIN

Pourquoi le client ne peut-il pas simplement s'assurer que sa requête a été traitée ? Le [problème des deux généraux](#)⁵⁶ en donne l'intuition. Deux armées doivent attaquer ensemble et ne communiquent que par des messagers, qui peuvent être capturés. Le premier général envoie l'heure de l'attaque et attend un accusé de réception ; mais l'auteur de l'accusé ignore à son tour si celui-ci est arrivé, et ainsi de suite. On montre qu'aucun nombre fini de messages ne donne aux deux la certitude d'être d'accord. Notre client est dans la même position : une réponse manquante peut venir d'un traitement lent, d'un serveur tombé avant ou après le traitement, ou d'une réponse perdue en route.

POUR ALLER PLUS LOIN

Une requête est *idempotente* si la traiter deux fois produit le même effet que la traiter une fois : en notant f son effet sur l'état, $f \circ f = f$. « Écris 23 » l'est ; « ajoute 5 » et « signe » ne le sont pas. Pour rejouer sans risque une requête qui ne l'est pas, le client lui donne un identifiant unique, par ex. `{:sign, "alice", id}` ; le serveur garde les identifiants déjà traités et répond à un doublon sans le traiter de nouveau. C'est le principe des clés d'idempotence des API de paiement, présenté par Stripe dans [« Designing robust and predictable APIs with idempotency »](#)⁵⁷. Le chapitre 11 retrouve cette question avec des états recopiés.

UN CALLBACK QUI APPELLE SON PROPRE SERVEUR

L'interblocage du chapitre 9 a sa version GenServer : un callback ne doit pas faire de `call` vers son propre serveur. Le serveur attendrait une réponse que lui seul peut donner, alors qu'il est occupé à l'attendre ! OTP détecte ce cas simple et fait échouer l'appel par une sortie ; deux serveurs qui s'appellent l'un l'autre dans leurs callbacks, eux, restent bloqués jusqu'à l'expiration d'un délai.

10.6 Nommer le serveur

Pour joindre un serveur, un client a besoin de son adresse. Transmettre le PID à chaque fonction qui en a besoin devient vite pénible ; surtout, nous allons le voir, un PID ne survit pas à un redémarrage.

On donne donc au serveur un *nom enregistré*, par l'option `name:` du démarrage : `GenServer.start_link(module, argument, name: nom)`. Le nom est un atome, souvent celui du module, puisque `CounterLocal.Counter` est lui aussi un atome. Partout où l'on attend un serveur, on peut alors écrire son nom au lieu de son PID.

⁵⁶ https://en.wikipedia.org/wiki/Two_Generals%27_Problem

⁵⁷ <https://stripe.com/blog/idempotency>

```

1 { :ok, pid } = GenServer.start(Thermometer, 20, name: :salon)
2 try do
3   again = GenServer.start(Thermometer, 18, name: :salon)
4   {Thermometer.value(:salon), Process.whereis(:salon) == pid,
5    again == { :error, { :already_started, pid }}}
6 after
7   GenServer.stop(:salon)
8 end

```

```
VALEUR {20, true, true}
```

`Thermometer.value(:salon)` atteint le serveur par son nom. `Process.whereis(nom)` rend le PID qui porte ce nom, ou `nil` si personne ne le porte. Un nom désigne au plus un processus : une seconde demande échoue, avec `{:error, { :already_started, pid }}`, qui indique le titulaire actuel.

`call` cherche le PID associé au nom à chaque appel. Si personne ne porte le nom, il provoque la même sortie `:noproc` que vers un processus mort.

POUR ALLER PLUS LOIN

Un atome n'est jamais libéré par la VM (cf. chapitre 3) : on ne fabrique donc pas un nom par contributeur à partir de données venues du dehors. Pour nommer un serveur par une chaîne, par ex. un compteur par salle, Elixir fournit `Registry` : l'option `name: { :via, Registry, { Rooms, "B204" }}` enregistre le serveur sous la clé "B204" du registre `Rooms`, et les mêmes appels acceptent ce tuple comme nom. La [documentation de Registry](https://hexdocs.pm/elixir/Registry.html)⁵⁸ en donne des exemples.

10.7 Superviser le serveur

10.7.1 La recette de démarrage

Un superviseur démarre ses enfants d'après leur recette, ou *child spec* : une map qui donne un identifiant et, sous la clé `start`, le module, la fonction et les arguments à appeler (cf. chapitre 9). `use GenServer` écrit pour nous la fonction `child_spec/1` qui construit cette recette.

```
1 Thermometer.child_spec(20)
```

```
VALEUR %{id: Thermometer, start: {Thermometer, :start_link, [20]}}
```

Dans une liste d'enfants, la paire `{Thermometer, 20}` abrège exactement cette map : le superviseur appellera `Thermometer.start_link(20)` à chaque démarrage de l'enfant.

10.7.2 Un arbre pour l'application

Le module de démarrage de `compteur-local` est fourni ; il ne fait qu'une chose : lancer le superviseur racine avec son unique enfant.

⁵⁸ <https://hexdocs.pm/elixir/Registry.html>

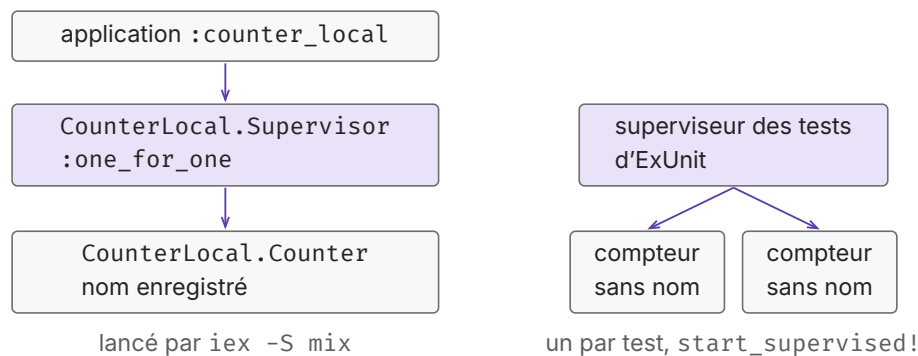
```

1 defmodule CounterLocal.Application do
2   use Application
3
4   @impl true
5   def start(_type, _args) do
6     children = [{CounterLocal.Counter, name: CounterLocal.Counter}]
7     Supervisor.start_link(children,
8       strategy: :one_for_one, name: CounterLocal.Supervisor)
9   end
10 end

```

La liste `children` contient une seule recette abrégée, dont le second élément est cette fois une liste d'options. Le superviseur appellera donc la fonction `start_link` du module `CounterLocal.Counter` avec ces options. Cette fonction, fournie par le projet, y lit la valeur initiale, `initial:`, nulle par défaut, et transmet le nom à `GenServer.start_link`.

Au lancement de l'application, par ex. par `iex -S mix`, la VM appelle `start/2`, que la ligne `mod: {CounterLocal.Application, []}` de `mix.exs` désigne. Le superviseur démarre, puis il démarre le compteur sous son nom : dès le premier prompt, le compteur est joignable par `CounterLocal.Counter`. La stratégie `:one_for_one` relance seulement l'enfant qui s'arrête.



À gauche, l'arbre de l'application, avec son compteur nommé. À droite, les compteurs des tests, démarrés sans nom sous le superviseur d'ExUnit : ils ne touchent jamais au compteur de l'application.

Les tests, en effet, démarrent chacun leur compteur, comme nous le verrons en fin de chapitre. Ils ne se gênent ni entre eux, ni avec l'application.

10.7.3 Tuer le serveur, puis le relire

Que devient un serveur nommé quand il tombe ? Faisons l'expérience avec un thermomètre nommé `:salon`. Sa recette, écrite sous forme de `map`, appelle directement `GenServer.start_link` avec le module, la valeur initiale et le nom. Nous enregistrons 23, tuons le serveur, attendons sa fin, puis relisons le nom.

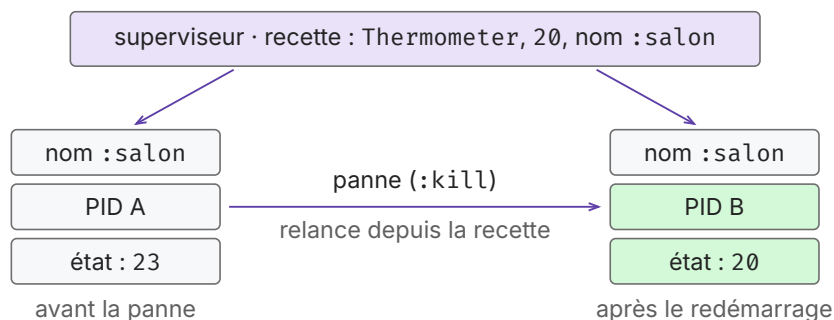
```

1 args = [Thermometer, 20, [name: :salon]]
2 children = [%{id: :salon, start: {GenServer, :start_link, args}}]
3 {:ok, sup} =
4   Supervisor.start_link(children, strategy: :one_for_one)
5 try do
6   :ok = Thermometer.record(:salon, 23)
7   before = Process.whereis(:salon)
8   ref = Process.monitor(before)
9   Process.exit(before, :kill)
10  receive do
11    {:DOWN, ^ref, :process, ^before, :killed} -> :ok
12  end
13  [{:salon, now, :worker, _}] = Supervisor.which_children(sup)
14  {now == Process.whereis(:salon), now != before,
15   Thermometer.value(:salon)}
16 after
17   Supervisor.stop(sup)
18 end

```

VALEUR {true, true, 20}

Le premier élément confirme que le nom désigne bien l'enfant que le superviseur annonce ; le deuxième, que cet enfant est un nouveau venu ; le troisième, qu'il est reparti de la valeur de sa recette, 20, et non de 23.



Le superviseur relance l'enfant depuis la même recette. Le nom `:salon` désigne maintenant le PID B ; l'état atteint par le PID A, `23`, n'a été transmis à personne.

Le nom a suivi le remplaçant ; l'état, lui, n'avait été sauvegardé nulle part. Pourquoi interroger `Supervisor.which_children` avant de relire le nom ? Parce que le redémarrage est asynchrone : juste après la fin de l'ancien serveur, le nom peut ne désigner personne pendant un court instant, et un `call` échouerait avec `:noprocs`. Sur une même VM, l'avis de fin de l'enfant parvient au superviseur avant notre question ; quand il y répond, le remplaçant est en place. Un test préfère une preuve plus directe, que nous verrons en fin de chapitre.

10.7.4 Ce que la supervision ne rend pas

À RETENIR

Un superviseur rétablit un *service* : le même nom répond de nouveau. Il ne rétablit pas les *données* : le remplaçant repart de l'état que donne sa recette, avec une boîte aux lettres vide. Les requêtes qui attendaient dans l'ancienne boîte sont perdues, et un client bloqué dans un `call` reçoit une sortie au lieu de sa réponse.

Cette limite a une conséquence directe pour le tableau de contributions : une seule requête qui fait planter le serveur efface les contributions de tous. Le « laisser planter » du chapitre 9 vise l'imprévu, pas une entrée invalide que l'on pouvait refuser.

On refuse donc une requête hors contrat le plus tôt possible. Une garde dans la fonction d'interface fait échouer le seul client fautif, avant tout envoi ; à défaut, le callback peut répondre `{:error, raison}` sans changer l'état. Le superviseur reste le dernier recours contre ce que personne n'avait prévu.

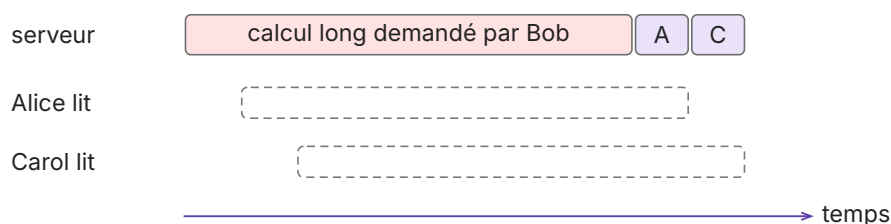
Conserver le total au-delà d'une panne demande un autre mécanisme : l'écrire sur disque à chaque modification, ou en garder des copies dans d'autres processus, voire sur d'autres machines. C'est la question du chapitre 11.

10.8 Un serveur unique, un goulot d'étranglement

10.8.1 Une requête à la fois

La sérialisation qui protège l'état a un prix : le serveur ne traite qu'une requête à la fois, quel que soit le nombre de clients ou de cœurs. Si chaque requête lui demande 10 millisecondes de travail, il en traite au plus cent par seconde. Cent clients qui arrivent ensemble sont servis l'un après l'autre, et le dernier attend environ une seconde.

De plus, une seule requête longue retarde toutes celles qui la suivent dans la boîte, même les plus simples lectures.



Chaque ligne suit un processus. Pendant le long calcul demandé par Bob, les lectures d'Alice et de Carol attendent dans la boîte du serveur (en pointillé), puis sont servies très vite. Les longueurs illustrent le principe ; ce ne sont pas des mesures.

Un tel serveur est un *goulot d'étranglement* (*bottleneck*) : la partie la plus étroite d'un système fixe son débit, comme le col d'une bouteille. On le repère à la longueur de sa boîte aux lettres, que donne `Process.info(pid, :message_queue_len)` : une file qui ne cesse de grandir signale un serveur qui reçoit plus de requêtes qu'il n'en traite.

POUR ALLER PLUS LOIN

La *loi de Little* relie trois moyennes d'un système en régime stable : le nombre L de requêtes présentes, en attente ou en traitement, vaut le débit d'arrivée λ multiplié par le temps W que chacune passe dans le système, $L = \lambda W$. Prenons un serveur qui sait traiter $\mu = 100$ requêtes par seconde. Si les arrivées sont aléatoires, un modèle classique, la file M/M/1, donne $W = 1/(\mu - \lambda)$. À 50 requêtes par seconde, $W = 20$ ms et $L = 1$; à 90 par seconde, $W = 100$ ms et $L = 9$; à 99, W atteint une seconde et $L = 99$. Le traitement n'a pas ralenti, puisque chaque requête demande toujours 10 ms : c'est l'attente qui explose quand on approche de la capacité. Ces chiffres sortent d'un modèle, pas d'une mesure ; cf. la page [Little's law](#)⁵⁹.

10.8.2 Ce que l'on peut y faire

Voici les principaux remèdes, que les projets d'AP9 mettent en œuvre :

- garder les callbacks courts : un calcul coûteux qui ne dépend pas de l'état se fait chez le client, avant l'envoi, et le serveur ne reçoit que la décision à appliquer, « ajoute 3 » plutôt que « écris 15 » ;
- confier les longs calculs à d'autres processus, des *travailleurs*, et ne garder dans le serveur que le suivi de leur avancement ; c'est le sujet du projet [ferme de rendu](#)⁶⁰ ;
- découper l'état, avec un serveur par salle ou par contributeur plutôt qu'un seul pour tout, au prix de lectures qui ne sont plus atomiques d'un serveur à l'autre ;
- servir les lectures sans passer par le serveur, grâce à des tables partagées, les tables ETS, que nous ne détaillerons pas.

Gardez enfin en tête que concurrent ne veut pas dire plus rapide : cent clients d'un même serveur avancent ensemble, mais leurs requêtes sont traitées en série. Le projet [réservations concurrentes](#)⁶¹ reprend, lui, l'idée d'une requête qui vérifie et décide d'un seul tenant.

10.9 Tester un serveur

10.9.1 Un serveur par test

Un test de serveur suit toujours la même forme : démarrer un serveur propre au test, lui envoyer des requêtes, comparer les réponses au contrat. ExUnit, la bibliothèque de tests d'Elixir, fournit pour cela `start_supervised!(recette)` : il démarre le serveur sous un superviseur propre au test, rend son PID, et l'arrête à la fin du test, que celui-ci réussisse ou non.

```
1 defmodule TicketTest do
2   use ExUnit.Case, async: true
3
4   test "chaque client reçoit le numéro suivant" do
5     tickets = start_supervised!({Ticket, 1})
6     assert Ticket.take(tickets) == 1
```

⁵⁹ https://en.wikipedia.org/wiki/Little%27s_law

⁶⁰ <https://ap-2026.noce.fr/seances/9/ferme-rendu/>

⁶¹ <https://ap-2026.noce.fr/seances/9/reservations-concurrentes/>

```

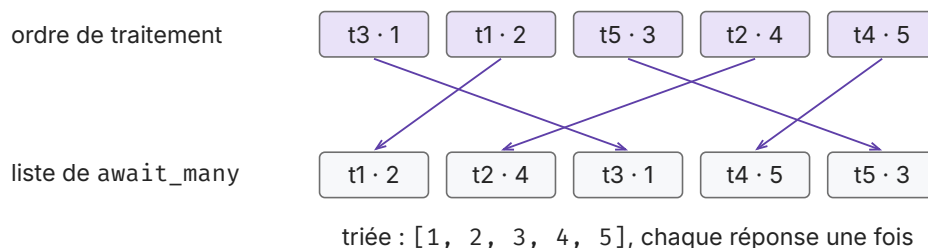
7   assert Ticket.take(tickets) == 2
8   end
9
10  test "cent clients, cent numéros différents" do
11    tickets = start_supervised!({Ticket, 1})
12    clients =
13      for _ <- 1..100 do
14        Task.async(fn -> Ticket.take(tickets) end)
15      end
16    numbers = Task.await_many(clients)
17    assert Enum.sort(numbers) == Enum.to_list(1..100)
18    assert Ticket.take(tickets) == 101
19  end
20 end

```

`use ExUnit.Case, async: true` déclare un module de tests, qui peut s'exécuter en même temps que d'autres : chaque test démarre ses propres serveurs, sans nom, et aucun ne dépend d'un état laissé par un autre. `assert` vérifie une condition ; en cas d'échec, il affiche les deux membres de la comparaison. Le premier test enchaîne deux requêtes, pour détecter un état qui n'avance pas ; le second est le plus parlant.

10.9.2 Cent clients, aucun ticket en double

`Task.async(fn -> ... end)` lance un processus temporaire, une *tâche*, qui calcule un résultat ; il est lié à son lanceur, si bien qu'une erreur dans un client fait échouer le test. `Task.await_many(tasks)` attend les résultats de toutes les tâches et les range dans l'ordre de la liste des tâches, qui n'est pas forcément leur ordre de fin.



Cinq tâches au lieu de cent. En haut, l'ordre dans lequel le serveur a traité leurs requêtes, et le numéro reçu par chacune ; en bas, la liste rendue par `await_many`, rangée dans l'ordre des tâches. Seul le tri retrouve 1 à 5.

Les cent tâches avancent concurremment, mais le distributeur traite leurs demandes une par une : chaque numéro de 1 à 100 sort exactement une fois, dans un ordre imprévisible. Le test trie donc les réponses avant de les comparer, puis vérifie qu'un client de plus reçoit 101.

Le test du projet `compteur-local` fait de même avec cent ajouts : les réponses triées valent 1 à 100, et le total final 100. Il vérifie qu'aucune contribution n'est perdue, pas que cent processus calculent cent fois plus vite !

Remarquez aussi ce qu'un tel test ne prouve pas. Un compteur mal conçu, dont les clients liraient puis écriraient, pourrait le réussir par chance, lors d'un essai où les requêtes ne se sont pas entrelacées. Un test de concurrence qui passe n'établit pas l'absence de condition de course ; il faut aussi des tests où l'entrelacement est fixé.

10.9.3 Des tests déterministes

Un test *déterministe* donne le même résultat à chaque exécution. Quelques règles y conduisent.

- Attendre une réponse, jamais un délai. Un `call` qui répond prouve que sa requête est traitée ; un `Process.sleep` ne prouve rien, sinon que la machine était assez rapide ce jour-là.
- S'appuyer sur l'ordre d'un même expéditeur : après un `cast`, un `call` du même client est traité après lui, comme dans le journal.
- Attendre un événement plutôt qu'un état. Un `monitor` annonce la fin d'un processus par `:DOWN` ; pour un redémarrage, le test fourni de `compteur-local` démarre ses compteurs par une petite fonction qui envoie au test le PID de chaque nouveau compteur, que le test attend avec `assert_receive`.
- Jouer un entrelacement à la main, comme nous l'avons fait avec l'Agent : deux lectures, puis deux écritures, dans cet ordre, à chaque exécution.
- Démarrer ses propres serveurs, et ne jamais tuer celui de l'application.

POUR ALLER PLUS LOIN

La transition pure de ce chapitre fournit un *modèle* du serveur. Un test *par propriétés* engendre au hasard des centaines de listes de requêtes, les envoie au serveur, et compare ses réponses à celles de `Enum.map_reduce(requêtes, état_initial, step)`. Quand une différence apparaît, la bibliothèque réduit la liste fautive au plus petit exemple qui la produit encore. Cette idée vient de QuickCheck, né en Haskell puis porté en Erlang ; la bibliothèque `StreamData`⁶² l'apporte à Elixir.

10.10 Vérifier sa compréhension

Avant de fermer ce chapitre, prévoyez ces situations, puis expliquez votre réponse sans faire appel à la vitesse supposée de la machine.

1. Deux clients appellent `increment` sur le même compteur, qui vaut 0. Le total final vaut 2 si les deux appels réussissent ; les réponses sont 1 et 2, dans un sens ou dans l'autre selon l'ordre d'arrivée.
2. Deux clients lisent un Agent qui vaut 0 avec `get`, puis écrivent chacun `0 + 1` avec `update`. Le total vaut 1 ou 2 selon l'entrelacement ; une seule requête qui ajoute 1 à l'état courant donne 2 dans tous les cas.
3. Le distributeur de tickets rend `{:reply, number, number}`. Trois clients successifs reçoivent tous 1 : la réponse est juste, l'état n'avance pas.
4. Un client envoie un `cast`, puis un `call` de lecture au même serveur. La lecture voit l'effet du `cast`, car les messages d'un même expéditeur arrivent dans l'ordre ; un autre client, lui, peut lire avant.
5. Un `call` expire, et le client le relance. Si le serveur avait traité la première requête, son effet est compté deux fois, sauf si la requête est idempotente ou porte un identifiant que le serveur reconnaît.

⁶² https://hexdocs.pm/stream_data/

6. Un compteur supervisé annonce 12, puis tombe en panne. Son remplaçant répond sous le même nom, avec un autre PID, et repart de sa valeur initiale : un arbre de supervision ne constitue pas une sauvegarde.

Les détails des contrats restent consultables dans la documentation d'Elixir : [GenServer](#)⁶³, [Agent](#)⁶⁴, [Supervisor](#)⁶⁵ et [Registry](#)⁶⁶. Les guides « [Client-server with GenServer](#) »⁶⁷ et « [Supervision trees and applications](#) »⁶⁸ reprennent ces briques sur un autre exemple.

POUR EN PARLER

Un seul processus possède l'état, et les clients lui envoient des requêtes qu'il traite une par une : une requête qui lit et modifie l'état ne peut pas être coupée, mais « lire » puis « écrire » en deux requêtes peut perdre une contribution, même avec un Agent, qui exécute dans son processus la fonction qu'on lui envoie.

Un serveur, c'est une transition, qui calcule la réponse et l'état suivant, et une boucle ; GenServer fournit la boucle, nous écrivons les callbacks, qui s'exécutent dans le serveur, alors que les fonctions d'interface s'exécutent chez le client. Dans `{:reply, :ok, 23}`, le client reçoit `:ok` et le serveur garde `23`.

`call` attend la réponse, 5 s au plus par défaut, et un délai expiré n'annule pas le traitement ; `cast` rend `:ok` aussitôt, même vers un serveur mort ; un message envoyé par `send` arrive dans `handle_info`.

Après un arrêt brutal, le superviseur relance le serveur sous le même nom, avec un nouveau PID et l'état de sa recette : la supervision rétablit le service, pas les données.

Cent clients concurrents ne perdent aucune contribution, parce que le serveur les sérialise ; le même serveur unique devient un goulot d'étranglement dès qu'une requête est longue.

⁶³ <https://hexdocs.pm/elixir/GenServer.html>

⁶⁴ <https://hexdocs.pm/elixir/Agent.html>

⁶⁵ <https://hexdocs.pm/elixir/Supervisor.html>

⁶⁶ <https://hexdocs.pm/elixir/Registry.html>

⁶⁷ <https://hexdocs.pm/elixir/genservers.html>

⁶⁸ <https://hexdocs.pm/elixir/supervisor-and-application.html>

Répliquer un état et retrouver l'accord

SÉANCE AP8 · RÉPLIQUES, CRDT ET CONVERGENCE

Le compteur d'AP7 vivait dans un seul processus, et tout le monde devait lui parler. Ce choix a un grand avantage : le serveur traite les requêtes une par une, et il n'existe jamais deux versions du total. Il a aussi un prix, car il faut pouvoir joindre ce serveur à chaque contribution.

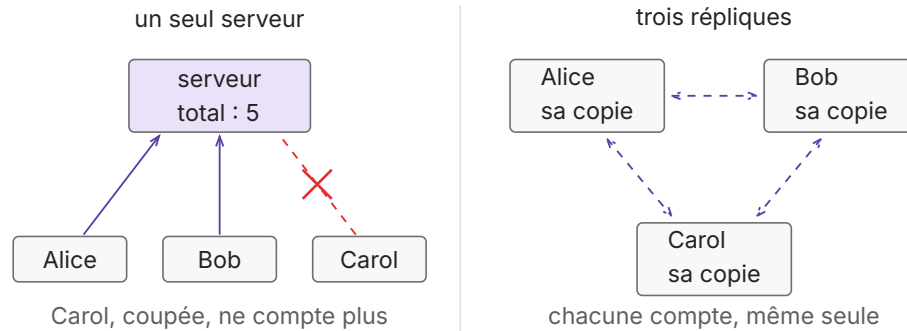
Ce chapitre, qui accompagne AP8, fait le choix inverse : chaque poste garde sa propre copie de l'état, la modifie sans attendre les autres, et les copies se réconcilient plus tard. Nous verrons ce qui peut alors mal tourner, puis une famille de structures de données conçues pour se réunir sans rien perdre ni rien compter deux fois. Le compteur répliqué d'AP8 sert de fil rouge ; l'édition de texte à plusieurs, sujet de deux projets d'AP9, ferme le chapitre.

11.1 Pourquoi garder plusieurs copies ?

Trois raisons reviennent dans les systèmes réels. La première est la *disponibilité*, c-à-d. la capacité à répondre malgré une panne : si l'unique serveur tombe ou devient injoignable, plus personne ne peut compter, alors qu'avec une copie par poste chacun continue.

La deuxième est la *latence*, le délai entre une action et son effet visible. Un serveur lointain impose un aller-retour sur le réseau à chaque modification ; une copie locale répond immédiatement, et l'échange avec les autres peut attendre.

La troisième, ce sont les coupures. Dans le train, en avion ou derrière un réseau saturé, les échanges peuvent s'arrêter pendant des minutes ou des heures. Une application de notes sur téléphone, un éditeur partagé ou un panier d'achat hors ligne doivent pourtant rester utilisables.



À gauche, chaque contribution doit atteindre le serveur : Carol, coupée, ne peut plus compter. À droite, chaque poste modifie sa propre copie ; les flèches en pointillé sont des échanges qui auront lieu quand ils seront possibles.

Tous ces systèmes font le même choix : chaque poste garde sa propre copie de l'état, sa *réplique*, la modifie localement, et les répliques se réconcilient plus tard. On accepte que les copies divergent un temps, à condition de savoir les réunir ensuite.

Notre exemple suivi est un tableau de contributions tenu par trois postes, Alice, Bob et Carol. Chacun ajoute des contributions de son côté, même sans recevoir les messages des autres. Comment réunir ensuite leurs copies sans rien perdre ni rien compter deux fois ?

Le projet `compteur-replique` d'AP8 simule ces trois postes par trois processus d'une même VM. Nous y choisissons à la main quels messages passent, et dans quel ordre, sans aucun réseau à configurer.

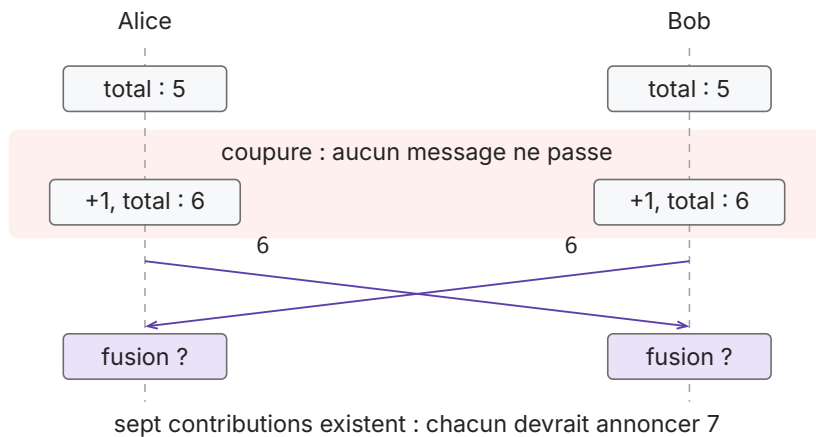
11.2 Quand les copies divergent

Tant que les messages passent vite, les copies restent proches. Les soucis viennent des périodes sans échange, et des messages qui arrivent mal.

11.2.1 Une coupure sépare ce que chacun sait

Une *partition réseau* est une situation où les machines restent en marche, mais se trouvent séparées en groupes qui ne peuvent plus communiquer entre eux : Alice et Bob d'un côté, Carol de l'autre, par ex. Pendant une partition, chaque groupe ne voit que ses propres modifications, et personne ne peut savoir si les autres sont en panne ou seulement injoignables.

Suivons Alice et Bob, qui connaissent tous deux un total de 5 : deux contributions d'Alice et trois de Bob. La connexion tombe ; chacun ajoute une contribution et annonce 6. Aucun des deux n'a tort, et pourtant leurs copies diffèrent.



Le temps descend, avec une colonne par poste. Pendant la coupure, chacun ne voit que son propre ajout. À la reconnexion, chacun reçoit le 6 de l'autre : que faut-il en faire pour obtenir 7 ?

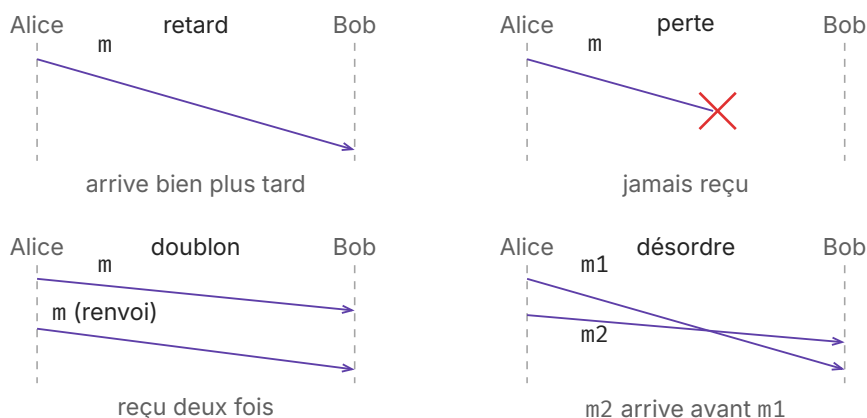
Deux modifications faites ainsi, sans qu'aucun auteur ait vu celle de l'autre, sont dites *concurrentes*. Elles ne sont pas forcément simultanées : Alice a pu ajouter la sienne une heure avant Bob. Ce qui compte, c'est qu'aucune des deux ne tenait compte de l'autre.

11.2.2 Ce que le réseau fait aux messages

À la reconnexion, les copies s'envoient leurs états. Rien ne garantit cependant qu'un message arrive vite, ni qu'il arrive une seule fois, ni dans l'ordre de son envoi, et chacun de ces incidents a une cause ordinaire.

Un message peut attendre dans une file ou derrière une coupure : il arrive en *retard*. Il peut se *perdre* avec une machine qui redémarre. Deux messages qui empruntent des chemins différents peuvent arriver dans le *désordre*.

Le *doublon* mérite qu'on s'y arrête. Un expéditeur qui ne reçoit pas d'accusé de réception renvoie son message, comme le client du chapitre 9 tenta de relancer sa demande après un timeout. Si le premier envoi était seulement lent, le destinataire reçoit deux fois la même chose ! Le schéma suivant résume ces quatre incidents, chacun entre un expéditeur et un destinataire.



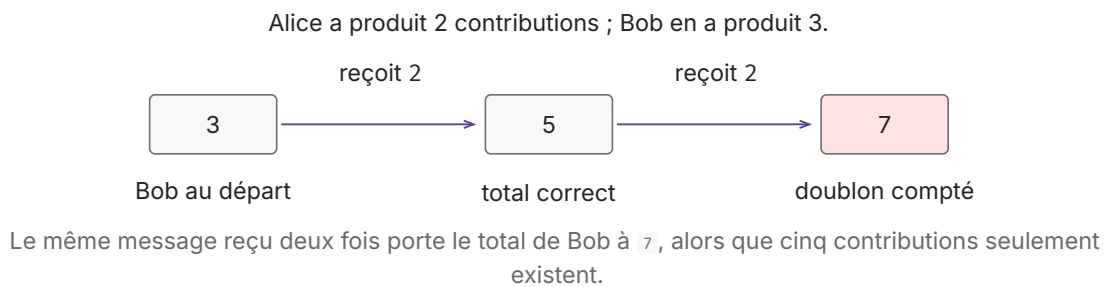
Dans chaque cadre, le temps descend. Un message peut arriver en retard, se perdre, arriver deux fois s'il a été renvoyé, ou être doublé par un message envoyé après lui.

Une perte se rattrape par un envoi ultérieur, à condition qu'il transporte la même information. Les trois autres incidents livrent bien l'information, mais en double, en retard ou en désordre : c'est à la règle de réunion des copies de les supporter.

11.2.3 Deux idées naturelles qui ne suffisent pas

Revenons à Alice, qui a ajouté deux contributions, et Bob, qui en a ajouté trois. Leur résultat commun doit valoir cinq. Si nous gardons seulement un entier par poste, comment fusionner le 2 d'Alice et le 3 de Bob ?

Essayons d'additionner les totaux reçus. Bob, qui annonce 3, reçoit le 2 d'Alice et passe à 5 : c'est juste. Mais si le réseau lui livre ce message une seconde fois, il passe à 7.



Prendre le maximum, $\max(2, 3)$, évite ce doublon, mais perd les deux ajouts propres à Alice.

L'addition a un second défaut, même sans doublon. Reprenons la coupure : Alice et Bob connaissaient tous deux un total de 5, fait de deux contributions d'Alice et de trois de Bob, puis chacun en a ajouté une sans échanger. Les deux copies annoncent 6, alors que sept contributions existent.

La somme des totaux donne 12, car les cinq contributions communes sont comptées deux fois ; le maximum donne 6, car l'une des deux nouvelles disparaît. Un entier par poste ne dit pas ce que les deux copies ont en commun : il manque l'identité de ceux dont les contributions sont déjà comptées.

règle	2 et 3	6 et 6, issus de 5	souci
addition	5, ou 7 avec un doublon	12	recompte les doublons et l'histoire commune
maximum global	3	6	perd des ajouts indépendants
maximum par auteur	$2 + 3 = 5$	$3 + 4 = 7$	demande une composante par auteur

Dans la colonne « 6 et 6 », Alice connaît trois contributions d'elle-même et trois de Bob, tandis que Bob en connaît deux d'Alice et quatre de lui-même. Garder, pour chaque auteur, le plus grand nombre connu donne 3 pour Alice et 4 pour Bob, soit 7. Nous ferons de cette idée une structure de données ; mais demandons-nous d'abord ce que nous voulons promettre.

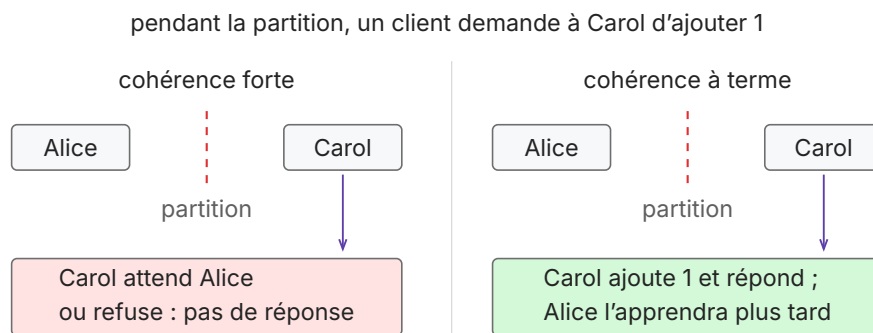
11.3 Cohérence forte ou cohérence à terme ?

Deux grandes familles de garanties s'opposent, selon ce que voit un utilisateur qui lit une copie juste après une écriture faite ailleurs.

La *cohérence forte* promet que tout se passe comme s'il n'existait qu'une seule copie : une lecture voit toujours la dernière écriture, où qu'elle ait été faite. Le compteur d'AP7 la tient, puisqu'un seul processus possède l'état. Avec plusieurs copies, il faut se *coordonner* : avant de confirmer une écriture, une réplique doit s'assurer que les autres en tiennent compte, ou passer par un propriétaire unique.

La *cohérence à terme* (*eventual consistency*) promet moins. Chaque copie répond seule, avec ce qu'elle sait, et les copies peuvent différer pendant un temps ; mais dès que les messages ont pu circuler, elles finissent par porter le même état. On dit alors qu'elles *convergent*.

Que se passe-t-il pendant une partition ? Supposons qu'un client demande à Carol d'ajouter une contribution, alors qu'elle ne peut plus joindre Alice.



La même demande, deux promesses. À gauche, Carol ne répond pas tant qu'elle ne peut pas joindre Alice. À droite, elle répond avec sa copie, qui diffère désormais de celle d'Alice.

Pour garantir la cohérence forte, Carol doit attendre le retour d'Alice, ou refuser la demande : elle n'est plus disponible. Pour rester disponible, elle ajoute la contribution à sa copie et répond ; Alice ne l'apprendra qu'après la partition, et les deux copies auront divergé entre-temps.

En clair, pendant une partition, on ne peut pas à la fois répondre partout et garantir que tout le monde voie le même état. Le bon choix dépend de ce que l'on veut éviter :

application	ce que l'on veut éviter	choix
dernière place d'un train	la vendre deux fois	cohérence forte
solde d'un compte	dépenser deux fois le même argent	cohérence forte
compteur de contributions	bloquer les postes pendant une coupure	cohérence à terme
notes, éditeur partagé	perdre ce qui a été écrit hors ligne	cohérence à terme

Notre compteur choisit la disponibilité : Alice, Bob et Carol comptent même coupés. Le projet *réservations concurrentes*⁶⁹ d'AP9 fait l'autre choix, avec un propriétaire unique qui décide avant de confirmer.

⁶⁹ <https://ap-2026.noce.fr/seances/9/reservations-concurrentes/>

POUR ALLER PLUS LOIN

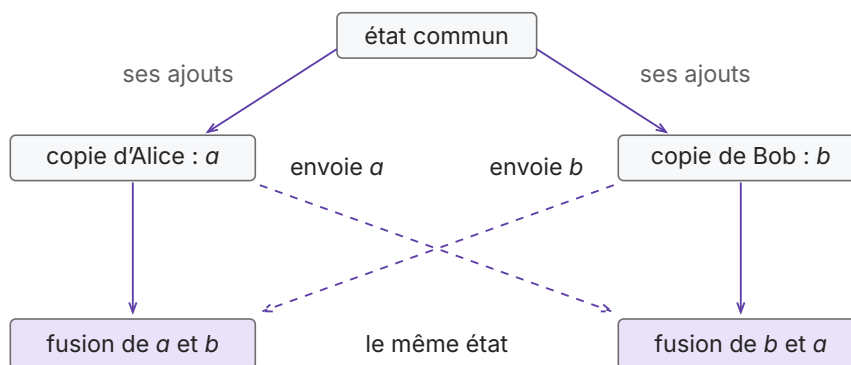
Ce compromis porte le nom de *théorème CAP* : conjecturé par Eric Brewer en 2000, il a été démontré par Seth Gilbert et Nancy Lynch en 2002. Il dit que, si les messages entre deux groupes de machines peuvent être perdus (P, la partition), aucun système ne peut garantir à la fois que toute machine en marche réponde (A, la disponibilité) et que chaque lecture voie la dernière écriture (C, une cohérence très exigeante, dite *linéarisable*). Ce n'est pas un « deux lettres sur trois » à choisir une fois pour toutes : le théorème ne parle que des périodes de partition. Cf. [Gilbert et Lynch \(2002\)](#)⁷⁰, et la section 7.3 des [notes de cours de Martin Kleppmann](#)⁷¹ (Cambridge, en anglais).

11.4 Un état qui se fusionne : le G-Counter

Reprenons la question de la réunion des copies. Plutôt que de chercher une règle qui marche sur des totaux, changeons l'état lui-même pour qu'une bonne règle existe : c'est l'idée des CRDT.

11.4.1 L'idée des CRDT

Un *CRDT* (*Conflict-free Replicated Data Type*, type de données répliqué sans conflit) est une structure de données munie d'une *fusion* : une fonction qui, à partir de deux copies, calcule un état contenant tout ce que chacune savait. Une réplique modifie sa copie localement, sans demander l'avis de personne. De temps en temps, elle envoie son état aux autres, et chaque réplique qui reçoit un état le fusionne avec le sien.



Alice et Bob partent d'un état commun et le modifient chacun de leur côté. Ils s'envoient ensuite leurs états, et chacun fusionne celui qu'il reçoit avec le sien. Une fusion bien conçue donne le même état des deux côtés.

Il n'y a ni vote, ni verrou, ni serveur qui tranche : l'accord vient de la règle de fusion elle-même. Encore faut-il la trouver, et nous avons vu qu'un entier par copie ne le permet pas.

Nous échangeons ici des états entiers : il s'agit de CRDT à *états*. Il existe aussi des CRDT qui diffusent des opérations, comme « ajoute 1 », avec d'autres exigences sur le transport des messages.

⁷⁰ <https://dl.acm.org/doi/10.1145/564585.564601>

⁷¹ <https://www.cl.cam.ac.uk/teaching/2122/ConcDisSys/dist-sys-notes.pdf>

11.4.2 Une case par auteur

L'état `%{alice: 2, bob: 3}` signifie « je connais deux contributions d'Alice et trois de Bob ». Une composante absente vaut zéro. L'état ne décrit pas seulement un total, mais ce que la réplique connaît de chacun des auteurs. Dans les schémas, nous l'écrirons comme une rangée `(2, 3, 0)`, les composantes dans l'ordre Alice, Bob, Carol.

Chaque auteur possède un identifiant distinct et incrémente seulement sa propre composante. Les composantes des autres auteurs ne changent qu'à la réception de leurs états. Deux processus qui incrémenteraient indépendamment la même composante pourraient perdre un ajout : l'identité du propriétaire fait partie du contrat.

Le *G-Counter* est un compteur croissant : ses composantes ne diminuent jamais. Le G vient de *grow-only*. Ce modèle sait ajouter des contributions ; il ne sait pas les annuler ni remettre le compteur à zéro au milieu d'une expérience.

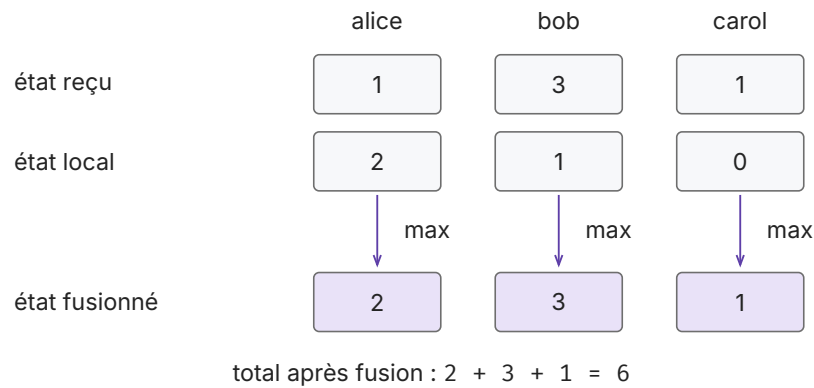
fonction pure	contrat
<code>new()</code>	renvoie la map vide
<code>increment(state, id)</code>	ajoute 1 à la composante de cet auteur
<code>value(state)</code>	additionne toutes les composantes
<code>merge(left, right)</code>	conserve toutes les clés, avec le maximum pour chaque auteur

Voici une fusion sur une seule composante, pour isoler la décision à prendre. Alice a connu deux, puis quatre contributions de Bob ; recevoir plus tard l'ancien état ne doit pas faire diminuer sa connaissance.

```
max(2, 4)  4
max(4, 2)  4
max(4, 4)  4
```

Pour des maps, nous appliquons ce raisonnement séparément à chaque clé. Ainsi, la fusion de `%{alice: 2, bob: 1}` et `%{alice: 1, bob: 3, carol: 1}` vaut `%{alice: 2, bob: 3, carol: 1}`, donc 6. Garder le maximum d'une composante ne choisit pas le poste qui « a raison » sur tout le document ; cela combine leurs connaissances auteur par auteur.

Alignons les auteurs pour voir ce que fait la fusion. Une clé absente est lue comme zéro : c'est ici le cas de Carol dans l'état local.



Chaque colonne garde son maximum, puis le total additionne les colonnes. On ne prend ni la somme des deux totaux, ni le maximum des deux totaux.

11.4.3 Un message est un snapshot

Pour transmettre son état, une réplique envoie un *snapshot* : une copie de l'état connu au moment de sa capture. Dans nos programmes Elixir, c'est une map immuable. Si Alice ajoute ensuite une contribution, le snapshot déjà obtenu ne change pas. Il représente précisément un message ancien que nous pourrions livrer plus tard.

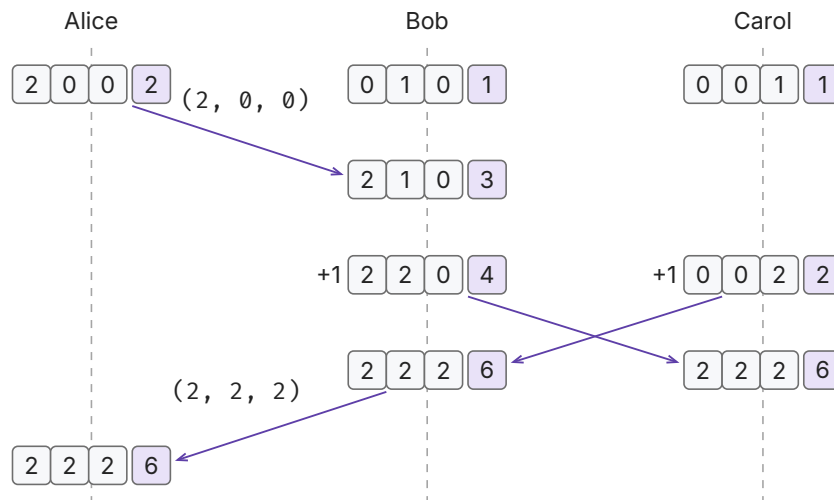
```
1 state = %{alice: 2}
2 snapshot = state
3 state = Map.put(state, :alice, 3)
4 {snapshot, state}
```

VALEUR %{alice: 2}, %{alice: 3}

La variable `state` est reliée à une nouvelle map, comme au chapitre 1. Le snapshot n'est pas une fenêtre donnant accès à l'état vivant d'un processus. C'est cette propriété qui permet de simuler les retards sans utiliser un vrai réseau : garder un snapshot dans une variable, c'est garder un message en transit.

11.4.4 Refaire la fusion à la main sur trois répliques

Suivons maintenant trois répliques qui partent de zéro. Dans le schéma suivant, chaque état est une rangée de cases, la dernière, en violet, donnant le total. Le temps descend ; une flèche est un snapshot envoyé, que la réplique destinataire fusionne aussitôt avec son état.



Cases dans l'ordre Alice, Bob, Carol, puis le total. Bob reçoit l'état d'Alice, puis échange le sien avec Carol ; il relaie enfin vers Alice ce qu'il a appris. Les trois répliques finissent sur (2, 2, 2).

Refaisons chaque étape, case par case.

1. Alice ajoute deux contributions, Bob et Carol une chacun : (2, 0, 0), (0, 1, 0) et (0, 0, 1).
2. Alice envoie son snapshot à Bob, qui calcule $\max(0, 2)$, $\max(1, 0)$ et $\max(0, 0)$, soit (2, 1, 0), de total 3.
3. Bob et Carol ajoutent chacun une contribution, sans rien savoir l'un de l'autre : (2, 2, 0) et (0, 0, 2). Ces deux ajouts sont concurrents.
4. Bob et Carol s'envoient leurs états en même temps, et les flèches se croisent. Bob calcule $\max(2, 0)$, $\max(2, 0)$ et $\max(0, 2)$; Carol calcule les mêmes maxima, arguments échangés. Tous deux obtiennent (2, 2, 2), de total 6.
5. Bob relaie son nouvel état vers Alice, qui obtient (2, 2, 2) à son tour.

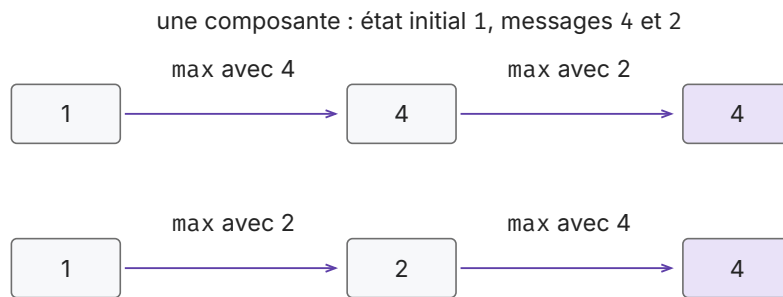
Alice connaît maintenant les deux contributions de Carol, alors qu'elle n'a jamais rien reçu de Carol elle-même : le snapshot de Bob les contenait. Comptons enfin : deux contributions par poste, et les trois répliques annoncent 6, sans doublon ni oubli.

Ajoutons à présent les incidents du réseau. Suivons seulement la copie d'Alice dans une autre expérience ; le tableau indique son état **après** chaque événement. Carol n'a pas besoin d'être déjà connue pour que Bob soit fusionné.

événement	état d'Alice	total
deux ajouts locaux	<code>%{alice: 2}</code>	2
reçoit Bob à 3	<code>%{alice: 2, bob: 3}</code>	5
reçoit encore Bob à 3	<code>%{alice: 2, bob: 3}</code>	5
reçoit Bob ancien à 1	<code>%{alice: 2, bob: 3}</code>	5
reçoit Carol à 1	<code>%{alice: 2, bob: 3, carol: 1}</code>	6

Le doublon et le retard n'ont pas modifié le total. Notez aussi que le message de Carol, même ancien, aurait encore été utile : il apporte une composante qu'Alice n'avait jamais reçue. Ce sont les maxima par auteur qui décident ce que l'on garde, pas la date du message. Ces bonnes propriétés sont-elles une chance ? C'est l'objet de la section suivante.

Avec « garder l'état reçu », deux répliques qui reçoivent les mêmes messages dans des ordres différents finissent en désaccord, et le restent. Avec le maximum, les deux ordres arrivent à 4 :



Les chemins arrivent à 4 ; recevoir encore 4 ou 2 le laisse inchangé.

L'ordre des messages ne change pas le maximum final. Un état ancien ou répété ne fait pas reculer cette composante.

La *commutativité* dit que l'ordre des deux arguments ne compte pas : $a \sqcup b = b \sqcup a$. C'est elle que nous avons observée lors de l'échange croisé entre Bob et Carol. Elle rend le désordre inoffensif.

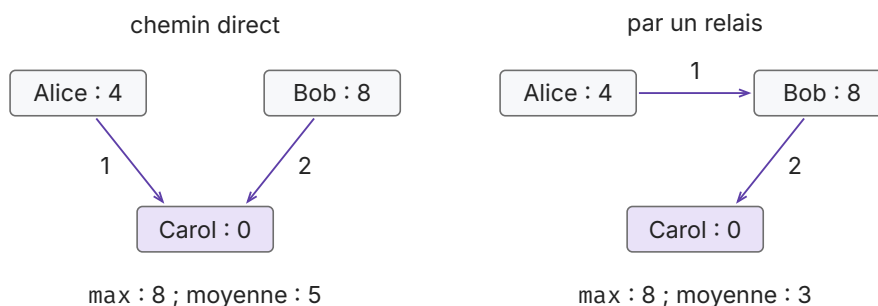
11.5.3 Recevoir par un relais : l'associativité

Un état peut aussi passer par un relais. Si Carol reçoit directement l'état d'Alice, puis celui de Bob, elle fusionne d'abord avec Alice, puis avec Bob. Si Bob a d'abord fusionné l'état d'Alice et transmet le résultat, Carol fusionne une seule fois, avec un état qui contient déjà les deux. Les deux chemins transportent la même information et doivent donner le même résultat.

Quelle règle échouerait ici ? La moyenne, ici entière : la moyenne de 3 et 3 vaut 3, et celle de deux nombres ne dépend pas de leur ordre. Suivons une composante qui vaut 0 chez Carol, 4 chez Alice et 8 chez Bob :

```
1 moyenne = fn a, b -> div(a + b, 2) end
```

moyenne.(moyenne.(0, 4), 8)	5
moyenne.(0, moyenne.(4, 8))	3
max(max(0, 4), 8)	8
max(0, max(4, 8))	8



Les messages sont numérotés dans l'ordre d'envoi. Par le chemin direct, Carol obtiendrait 5 avec la moyenne ; par le relais de Bob, 3. Avec le maximum, les deux chemins donnent 8.

L'*associativité* dit que le regroupement des fusions ne compte pas : $(a \sqcup b) \sqcup c = a \sqcup (b \sqcup c)$. Lisons-la avec les noms de nos postes : par le chemin direct, Carol calcule $(c \sqcup a) \sqcup b$; par le relais, elle calcule $c \sqcup (a \sqcup b)$. L'associativité, avec la commutativité, garantit que ces deux calculs coïncident.

11.5.4 Les trois lois ensemble

Prises ensemble, ces trois lois disent que le résultat d'une suite de fusions ne dépend que de l'*ensemble* des états reçus : ni leur ordre, ni leurs répétitions, ni la manière de les regrouper ne comptent. Le tableau résume ce que protège chaque loi, et une règle de fusion à laquelle elle manque.

loi	égalité	protège des	règle qui ne l'a pas
idempotence	$a \sqcup a = a$	doublons	l'addition
commutativité	$a \sqcup b = b \sqcup a$	messages dans le désordre	garder l'état reçu
associativité	$(a \sqcup b) \sqcup c = a \sqcup (b \sqcup c)$	relais	la moyenne

`max` possède ces trois lois sur les entiers ; les appliquer séparément à chaque auteur les conserve sur les maps, une composante absente valant zéro.

Remarquez que l'addition est commutative et associative : seule l'idempotence lui manque, et c'est précisément la loi qui protège des doublons.

POUR ALLER PLUS LOIN

Pourquoi `max` vérifie-t-il les trois lois ? Le maximum de a et a est a ; $\max(a, b)$ est le plus grand des deux, quel que soit l'ordre où on les donne ; enfin $\max(\max(a, b), c)$ et $\max(a, \max(b, c))$ valent tous deux le plus grand des trois nombres. Pour les maps, deux états sont égaux quand ils ont la même valeur pour chaque auteur, en lisant zéro pour une clé absente. Or la fusion calcule chaque composante par un maximum, indépendamment des autres : chaque égalité se vérifie donc auteur par auteur, avec le raisonnement sur les nombres.

11.5.5 Ce qu'il faut en plus des trois lois

Les trois égalités ne suffisent pas, à elles seules, à rendre le compteur correct. Deux hypothèses du modèle s'y ajoutent. Chaque composante a un seul auteur, le seul à l'incrémenter. De plus, l'état ne fait que croître : un ajout local augmente une composante, et une fusion n'en fait jamais reculer aucune.

Que se passe-t-il si l'on viole la première hypothèse ? Supposons que deux postes utilisent par erreur le même identifiant : `alice`. Chacun ajoute une contribution depuis zéro ; la fusion calcule $\max(1, 1)$ et garde 1. Une contribution a disparu, alors que les trois lois tiennent toujours !

C'est donc la combinaison des lois, des deux hypothèses et d'une diffusion suffisante des états qui rend possible la convergence du compteur. Les lois seules ne justifient pas n'importe quel protocole ni n'importe quelle opération locale.

POUR ALLER PLUS LOIN

Comparons deux états composante par composante : $a \leq b$ quand chaque composante de a est au plus celle de b , c-à-d. quand b sait tout ce que sait a . Cet ordre est *partiel* : $(2, 3)$ et $(1, 4)$ ne sont pas comparables. La fusion $a \sqcup b$ est le plus petit état qui majore a et b , leur *borne supérieure* ; un ensemble ordonné où deux éléments ont toujours une borne supérieure s'appelle un *demi-treillis*. Shapiro, Preguiça, Baquero et Zawirski ont montré en 2011 qu'un objet répliqué dont les états forment un demi-treillis, et dont les modifications locales font croître l'état, converge dès que les états circulent. Cf. « [A comprehensive study of Convergent and Commutative Replicated Data Types](#) »⁷², rapport de recherche Inria, 2011.

EN PRATIQUE

Dans le TP, les tests explorent un petit domaine de maps pour chercher des contre-exemples. Ils ne constituent pas une preuve pour toutes les maps. Reliez chaque test à une loi, puis expliquez pourquoi la propriété de `max` s'étend composante par composante. Un bon cas personnel est « état récent, puis état ancien, puis doublon de l'ancien ».

11.6 Faire vivre trois répliques

Les lois portent sur des fonctions pures. Plaçons maintenant chaque copie dans un processus, puis jouons une coupure et sa réconciliation.

11.6.1 Une réplique par processus

Les fonctions de `GCounter` calculent des états sans lancer de processus. Un `Replica` conserve ensuite `%{id: id, counts: map}` dans son `GenServer`. Son identifiant reste le même ; une requête d'ajout met à jour sa composante et répond par le nouveau total, une requête de fusion fait de même avec `counts`, et une requête de snapshot répond avec la map sans la modifier. Chaque réplique est désignée par un nom, `:alice`, `:bob` ou `:carol`.

appel client	réponse	état conservé
<code>Replica.increment(id)</code>	le nouveau total	<code>counts</code> avec la composante <code>id</code> augmentée de 1
<code>Replica.merge(id, snapshot)</code>	le nouveau total	la fusion de <code>counts</code> et du snapshot
<code>Replica.snapshot(id)</code>	la map <code>counts</code>	inchangé
<code>Replica.value(id)</code>	le total	inchangé

Nous retrouvons la séparation d'AP7 : le calcul pur se teste seul, et le processus garde le résultat entre deux demandes, en distinguant à chaque fois la réponse rendue au client de

⁷² <https://inria.hal.science/inria-00555588>

l'état conservé pour le tour suivant. Le squelette fournit l'API, la supervision et la lecture du snapshot ; vous complétez les clauses qui appellent vos fonctions pures.

Une fois le projet complété, lancez `iex -S mix` et jouez cette trace, qui capture un snapshot d'Alice puis le livre deux fois à Bob :

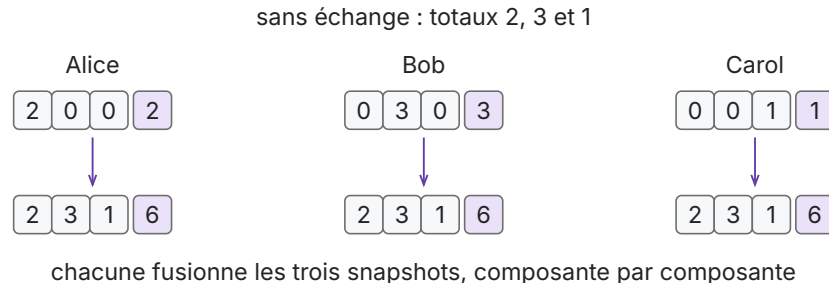
```
1 alias CounterReplicated.Replica
2 Replica.increment(:alice)
3 snapshot = Replica.snapshot(:alice)
4 Replica.merge(:bob, snapshot)
5 Replica.merge(:bob, snapshot)
6 Replica.value(:bob) # 1
```

Avant d'exécuter, prévoyez la valeur de Carol. Elle reste à zéro, car aucun message ne lui a été livré. Cela permet de distinguer « la règle de fusion est correcte » de « tout le monde possède déjà les mêmes informations ».

11.6.2 Couper, puis réconcilier

Dans le TP, nous simulons une coupure en ne livrant aucun snapshot, un doublon en livrant deux fois la même map, et un retard en gardant une ancienne map avant de l'envoyer. Aucun câble n'est débranché : `:alice`, `:bob` et `:carol` sont des processus de la même VM.

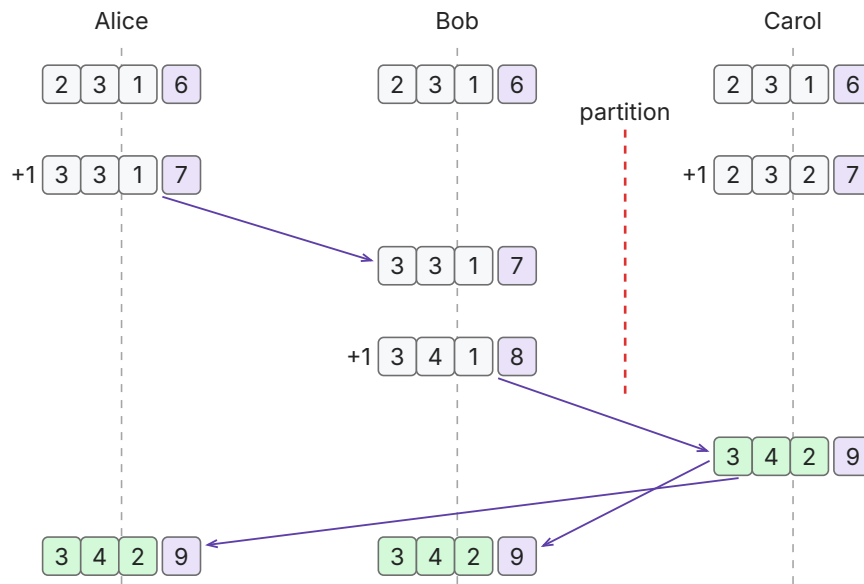
Première expérience : Alice ajoute deux contributions, Bob trois et Carol une, sans aucun échange. Chacune prend ensuite un snapshot, avant la moindre fusion, et les trois snapshots sont livrés à chaque réplique.



Un tour complet livre les trois snapshots à chaque réplique. Ici, aucune contribution n'est ajoutée pendant ce tour : chacune finit sur (2, 3, 1), de total 6.

Si Alice ajoute ensuite une contribution sans nouvel échange, elle annonce 7, tandis que Bob et Carol restent à 6. Ils ne peuvent pas le deviner : leur total restera 6 tant qu'aucun état contenant cette information ne leur parvient, directement ou par une autre réplique.

Rejouons maintenant une vraie partition, en partant de (2, 3, 1) partout. Carol se retrouve séparée d'Alice et de Bob, qui peuvent encore échanger. Alice ajoute une contribution et l'envoie à Bob ; Bob en ajoute une à son tour ; Carol, seule de son côté, en ajoute une aussi.



Le trait rouge sépare Carol des deux autres pendant la partition. À la reconnexion, Bob envoie un snapshot récent à Carol, qui renvoie son nouvel état à Alice et à Bob. Les états finaux, en vert, sont égaux.

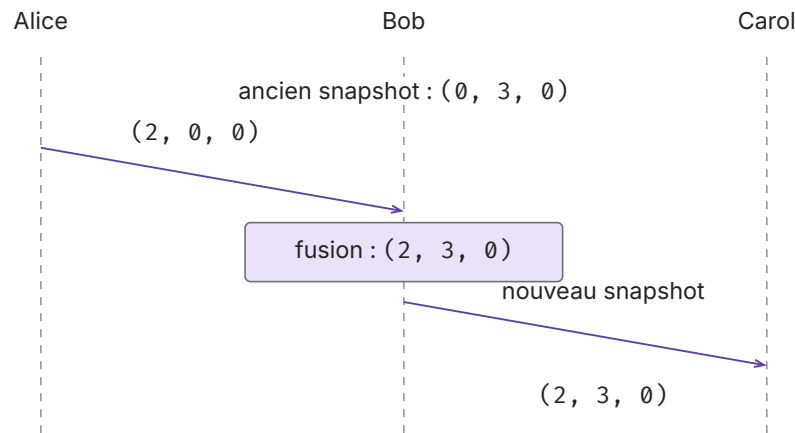
Juste avant la fin de la partition, les copies annoncent 7 chez Alice, 8 chez Bob et 7 chez Carol. Aucune n'est fausse : chacune décrit ce que sa réplique sait. À la reconnexion, Carol fusionne (2, 3, 2) avec le snapshot (3, 4, 1) de Bob et obtient (3, 4, 2), soit 9.

Carol renvoie ensuite ce nouvel état à Alice et à Bob, qui obtiennent la même map. Neuf contributions existent bien : trois d'Alice, quatre de Bob et deux de Carol. Deux échanges ont suffi après la partition, parce que le snapshot de Bob contenait déjà l'ajout d'Alice.

11.6.3 Un relais transmet ce qu'il a appris

Ce dernier point mérite d'être isolé. Alice envoie sa map à Bob, puis Bob prend un *nouveau* snapshot et l'envoie à Carol. Celui-ci contient les contributions propres à Bob et celles qu'il vient d'apprendre d'Alice.

En revanche, un snapshot que Bob avait pris avant de recevoir Alice ne les contient pas : la fusion ne modifie jamais une ancienne valeur conservée dans une variable. Le temps descend dans le schéma suivant, où Bob connaît déjà trois contributions propres.



Le vieux snapshot de Bob contient toujours $(0, 3, 0)$.

L'ordre des composantes est Alice, Bob, Carol. Le nouveau message de Bob relaie ce qu'il vient d'apprendre ; envoyer son ancien snapshot n'aurait transmis que ses trois contributions.

Pour terminer ce trajet, il reste à faire circuler l'état réuni vers les répliques auxquelles une composante manque encore : ici, Alice ignore les contributions de Bob.

POUR ALLER PLUS LOIN

Leslie Lamport a précisé en 1978 la notion de concurrence avec la relation *arrivé avant* : e arrive avant f s'ils ont lieu dans le même processus, e d'abord, si e envoie un message que f reçoit, ou par une chaîne de tels pas ; deux événements qui ne sont reliés dans aucun sens sont concurrents. Les *horloges vectorielles* de Fidge et Mattern (1988) comptent les événements de chaque processus dans un vecteur fusionné par maximum, comme un G-Counter : e arrive avant f quand le vecteur de e est inférieur ou égal à celui de f composante par composante, sans lui être égal. Ainsi $(3, 3, 1)$ et $(2, 3, 2)$, Alice et Carol pendant la partition, sont concurrents. Cf. [l'article de Lamport](https://lamport.azurewebsites.net/pubs/time-clocks.pdf)⁷³, court et lisible, et la section 4.1 des notes de Kleppmann citées plus haut.

11.6.4 Pourquoi les répliques finissent d'accord

Supposons que chacun cesse de contribuer. Les répliques convergent dès que toute information produite finit par être connue de chacune d'elles. Il n'est pas nécessaire de livrer tous les anciens snapshots : un état plus récent peut déjà contenir toute leur information. En revanche, une contribution qui ne quitte jamais sa réplique ne peut pas apparaître ailleurs.

⁷³ <https://lamport.azurewebsites.net/pubs/time-clocks.pdf>

POURQUOI LES RÉPLIQUES FINISSENT D'ACCORD

Supposons que plus personne n'ajoute de contribution, et que l'état de chaque réplique finisse par atteindre toutes les autres, directement ou par relais. Alors toutes les répliques portent la même map. En effet, grâce aux trois lois, l'état d'une réplique vaut, pour chaque auteur, le maximum des valeurs de cette composante qu'elle a produites ou reçues, quels que soient l'ordre, les doublons et les relais. Or la plus grande valeur de la composante d'Alice est celle que possède Alice elle-même, puisqu'elle seule l'incrémente et qu'une fusion ne la fait jamais reculer. Une fois l'état d'Alice parvenu partout, toutes les répliques ont donc la même composante d'Alice ; il en va de même pour chaque auteur.

Reprenons les incidents du réseau à la lumière de cette propriété. Une duplication est supportée par l'idempotence, un désordre par la commutativité, un relais par l'associativité. Une livraison tardive est supportée par la fusion croissante, et une perte peut être rattrapée par un envoi ultérieur qui contient la même information. Une coupure permanente, elle, empêche tout échange : aucun calcul local ne rétablit la communication.

CONVERGER NE VEUT PAS DIRE ÊTRE TOUJOURS D'ACCORD

Après un ajout local, deux répliques peuvent annoncer des totaux différents pendant un temps indéterminé. Si les ajouts continuent, elles peuvent toujours être en train de recevoir des nouveautés. Notre expérience garantit le même résultat une fois les mises à jour diffusées, pas un total instantanément commun ni une durée maximale de rattrapage.

11.7 Converger n'est pas sauvegarder

Nos états vivent seulement en mémoire. Une contribution perdue avant son premier échange ne peut être reconstruite par personne : aucune autre réplique ne la connaît.

Le chapitre 10 distinguait déjà supervision et sauvegarde, et le souci est encore plus visible avec une réplique. Supposons que Bob connaisse `{alice: 2}`, et qu'Alice perde son état puis redémarre à zéro sous le même identifiant. Sa première nouvelle contribution donne `{alice: 1}` ; Bob fusionne 2 et 1, garde 2, et cette contribution disparaît du total !

Reprendre un identifiant ne suffit donc pas à reprendre son histoire. Une application durable doit conserver ses états sur disque, ou donner explicitement un nouvel identifiant à chaque nouvelle incarnation d'un auteur : la fusion, elle, ne devine rien.

Notez enfin ce qu'un G-Counter ne sait pas faire. Avec une dernière place libre, Alice et Bob pourraient chacun accepter une réservation sans échanger, et la fusion compterait deux réservations pour une place. Compter hors connexion et garantir une capacité globale n'imposent pas le même protocole : la seconde garantie demande la cohérence forte décrite plus haut.

11.8 Compter dans les deux sens : le PN-Counter

Un G-Counter ne sait qu'augmenter. Or un stock, un panier ou le nombre de personnes dans une salle montent et descendent.

11.8.1 Décrémenter une case ne marche pas

Pourquoi ne pas simplement décrémenter sa propre composante pour retirer une contribution ? Supposons qu'Alice ait compté trois contributions, et que Bob ait déjà reçu `%{alice: 3}`. Alice retire une contribution comptée par erreur : sa composante redescend à 2. Que se passe-t-il quand Bob reçoit ce nouvel état ? Et quand Alice reçoit plus tard l'état de Bob ?

<code>max(3, 2)</code>	3
<code>max(2, 3)</code>	3

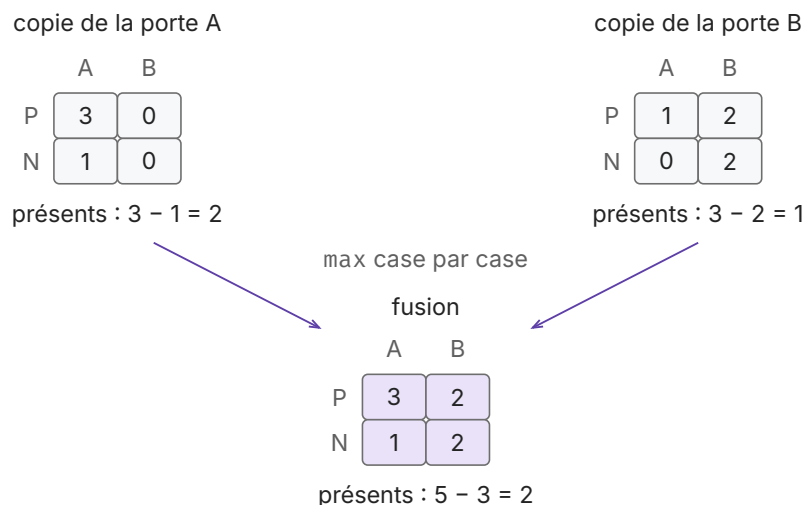
Le retrait est perdu des deux côtés : Bob l'ignore, et Alice elle-même le perd en recevant l'état de Bob. Le maximum suppose qu'une composante ne fait que croître ; une valeur plus petite est lue comme une information ancienne, pas comme un retrait.

11.8.2 Deux compteurs croissants

L'idée du *PN-Counter* est de ne jamais faire reculer une composante, mais de compter séparément les ajouts et les retraits. Son état réunit deux G-Counters : *P*, qui compte les ajouts (*positive*), et *N*, qui compte les retraits (*negative*). La valeur est la somme des composantes de *P* moins celle des composantes de *N*.

Chaque opération ne touche que la case de son auteur. Un ajout augmente sa composante de *P*, un retrait sa composante de *N*. Pour fusionner, on fusionne *P* avec *P* et *N* avec *N*, chacun par maximum case par case.

Prenons une salle à deux portes, A et B, qui tiennent chacune une réplique du nombre de personnes présentes. La porte A a vu trois entrées et une sortie ; la porte B, deux entrées et deux sorties. Avant de recevoir l'état de B, A annonce 2 personnes ; B, qui n'avait reçu qu'un vieil état de A où une seule personne était entrée, annonce 1.



Ligne P : les entrées, ligne N : les sorties ; une colonne par porte. Chaque ligne se fusionne case par case, comme un G-Counter. Après fusion, cinq entrées et trois sorties donnent deux personnes présentes.

Après la fusion, les deux portes annoncent $5 - 3 = 2$ personnes. Comme P et N sont des G-Counters, la fusion d'un PN-Counter hérite des trois lois : elle les applique à chacune des deux moitiés.

Le PN-Counter ne garantit pas pour autant que la valeur reste positive. S'il reste un article en stock et que deux répliques coupées l'une de l'autre le vendent chacune, la fusion annoncera un stock de -1 . Empêcher cela demande, encore une fois, de se coordonner avant de confirmer.

11.9 Des ensembles qui se fusionnent

Les compteurs ne sont qu'un début. Une liste de courses partagée, la liste des membres d'un groupe ou les étiquettes d'un document sont des *ensembles* ; comment les répliquer ?

11.9.1 Le G-Set : ajouter seulement

Le plus simple est le *G-Set*, un ensemble où l'on ne fait qu'ajouter. Sa fusion est l'*union* : un élément est présent dans la fusion s'il est présent dans l'une au moins des deux copies. Avec les `MapSet` d'Elixir, sur une liste de courses où Alice a noté le pain et le lait, et Bob le lait et le sel :

<code>a = MapSet.new([:pain, :lait])</code>	<code>MapSet.new([:lait, :pain])</code>
<code>b = MapSet.new([:lait, :sel])</code>	<code>MapSet.new([:lait, :sel])</code>
<code>MapSet.union(a, b)</code>	<code>MapSet.new([:lait, :pain, :sel])</code>

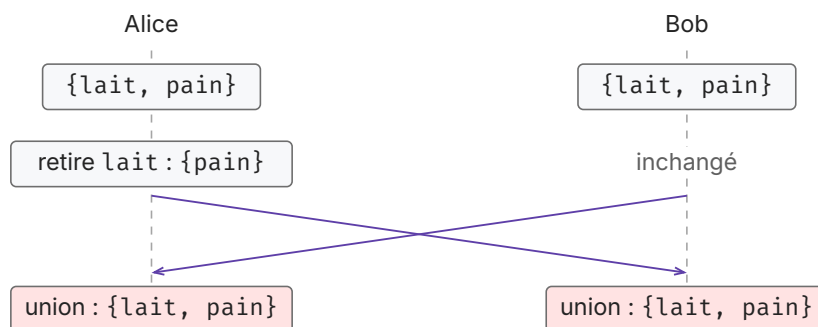
Le lait, noté des deux côtés, n'apparaît qu'une fois. Vérifions deux des lois sur ces copies :

<code>MapSet.union(a, a) == a</code>	<code>true</code>
<code>MapSet.union(a, b) == MapSet.union(b, a)</code>	<code>true</code>

L'union possède les trois lois, pour la même raison que le maximum : « présent dans l'une au moins des copies » ne dépend ni de l'ordre, ni des répétitions, ni des regroupements. Le G-Counter est d'ailleurs une union déguisée : la composante 3 d'Alice résume l'ensemble de ses contributions numéro 1, 2 et 3.

11.9.2 Retirer sans voir revenir l'élément

Que se passe-t-il si l'on retire un élément d'un G-Set ? Alice et Bob partent de `{lait, pain}`. Alice retire le lait ; Bob, qui n'en sait rien, garde sa copie. Puis ils échangent leurs états.



Le temps descend. Après l'échange, l'union contient de nouveau le lait : elle ne distingue pas « jamais ajouté » de « retiré ».

Le lait revient ! Comme pour le compteur, il faut garder la trace du retrait. Le *2P-Set* (*two-phase set*) réunit deux G-Sets : les éléments ajoutés, et les éléments retirés. Un élément retiré reste dans le second ensemble, comme une *pierre tombale* (*tombstone*) qui rappelle sa suppression. Un élément est présent s'il a été ajouté et n'a pas été retiré.

<code>ajoutes = MapSet.new([:lait, :pain])</code>	<code>MapSet.new([:lait, :pain])</code>
<code>retires = MapSet.new([:lait])</code>	<code>MapSet.new([:lait])</code>
<code>MapSet.difference(ajoutes, retires)</code>	<code>MapSet.new([:pain])</code>

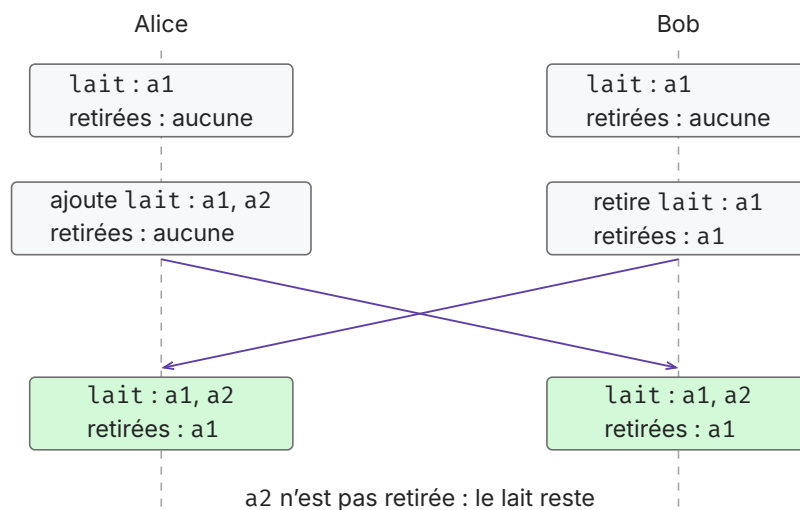
Pour fusionner, on fait l'union des ajouts d'un côté, celle des retraits de l'autre : c'est deux fois la fusion d'un G-Set. Après l'échange, Alice et Bob ont les mêmes ajouts, {lait, pain}, et les mêmes retraits, {lait} : il leur reste le pain.

Le prix à payer est qu'un élément retiré ne peut plus jamais revenir, puisque sa pierre tombale reste à jamais dans l'ensemble des retraits. Pour une liste de courses, où l'on rachète du lait chaque semaine, c'est gênant.

11.9.3 L'OR-Set : une étiquette par ajout

L'*OR-Set* (*observed-remove set*) lève cette limite par une idée simple : chaque ajout reçoit une *étiquette* unique, formée par ex. du nom de son auteur et d'un numéro, a1, a2... Un retrait ne vise que les étiquettes que son auteur a observées. Un élément est présent s'il lui reste au moins une étiquette non retirée.

Rejouons le lait. Alice l'a ajouté avec l'étiquette a1, et Bob l'a appris. Puis, sans échanger, Bob retire le lait, c-à-d. l'étiquette a1 qu'il connaît, tandis qu'Alice, qui veut du lait frais, l'ajoute de nouveau avec une étiquette neuve, a2.



Chaque copie garde les étiquettes du lait et les étiquettes retirées ; la fusion fait l'union des deux. Bob n'a retiré que a1, qu'il avait vue : a2 survit, et le lait reste présent.

Après l'échange, le lait garde l'étiquette a2, que personne n'a retirée : il reste dans la liste. Quand un ajout et un retrait sont concurrents, l'ajout l'emporte, ce qui est souvent le comportement attendu : Bob a retiré le lait qu'il voyait, pas celui qu'Alice venait de rajouter. L'état garde en contrepartie toutes les étiquettes retirées, et grossit avec le temps.

Récapitulons les CRDT rencontrés. Tous fusionnent par maximum ou par union, éventuellement sur deux moitiés : c'est ce qui leur donne les trois lois.

CRDT	état	fusion	limite
G-Counter	une composante par auteur	maximum par composante	ne décroît pas
PN-Counter	deux G-Counters, P et N	deux fusions de G-Counter	aucune borne garantie
G-Set	un ensemble	union	aucun retrait
2P-Set	ajoutés et retirés	deux unions	pas de nouvel ajout après un retrait
OR-Set	étiquettes et étiquettes retirées	deux unions	l'état grossit

11.10 Éditer un texte à plusieurs

Un compteur ne possède ni position, ni caractère supprimé ; un texte, si. Voyons pourquoi le maximum ne suffit plus, puis les deux familles de solutions étudiées par les projets éditeurs d'AP9.

11.10.1 Une position dépend de l'histoire

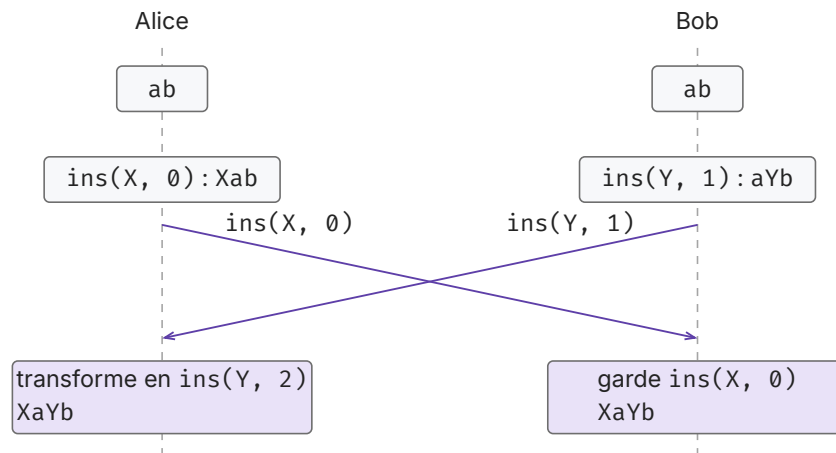
Dans un texte, « insérer X à la position 1 » n'a pas le même sens avant et après la suppression du caractère 0. Partons de ab : Alice insère x à la position 1 et produit axb , Bob supprime le caractère 0 et produit b .

Si Bob applique l'insertion à sa position d'origine, il obtient bx ; si Alice applique la suppression du caractère 0, elle obtient xb . Les deux opérations étaient correctes sur leur document de départ : la position doit être interprétée dans son contexte.

Deux insertions posent le même souci. Alice insère x à la position 0, Bob insère y à la position 1. Appliquées avec leurs positions d'origine, elles donnent $xyab$ si l'on applique Alice puis Bob, et $xa yb$ dans l'autre ordre. Les deux copies divergent, alors que chaque opération était correcte !

11.10.2 Transformer les opérations

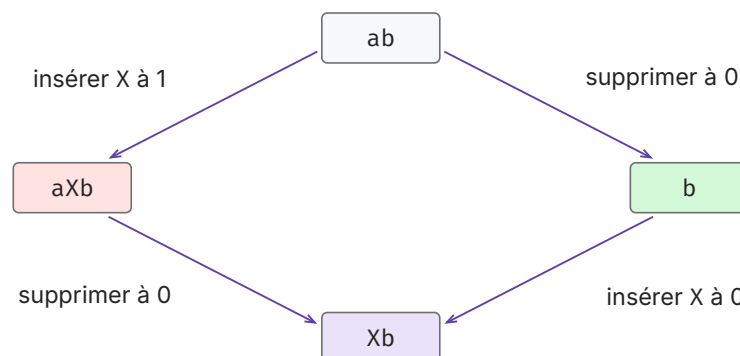
La *transformation opérationnelle* (OT, *operational transformation*) corrige la position d'une opération reçue en tenant compte des opérations déjà appliquées localement. Notons $ins(x, 0)$ l'opération « insérer x à la position 0 ».



Chaque éditeur applique d'abord sa propre opération, puis reçoit celle de l'autre. Alice décale l'insertion de Bob, puisque son x a été inséré avant ; Bob garde celle d'Alice, située avant son y . Les deux copies affichent $xaYb$.

Chez Alice, le texte est déjà xab quand arrive $ins(Y, 1)$. Le x a été inséré avant la position 1 : tous les caractères suivants ont avancé d'un cran, et l'insertion devient $ins(Y, 2)$, ce qui donne $xaYb$. Chez Bob, l'insertion d'Alice, à la position 0, a lieu avant son y : elle ne change pas, et donne aussi $xaYb$.

Pour une suppression, le raisonnement est symétrique : un caractère supprimé avant la position visée la fait reculer d'un cran. C'est ce qui répare notre premier exemple.



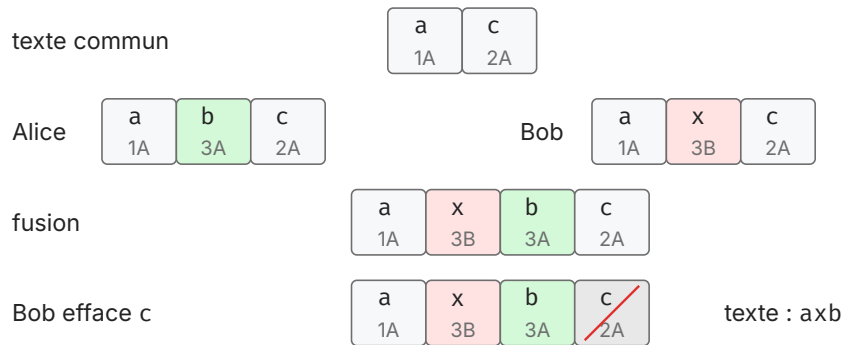
L'OT transforme l'insertion à la position 1 en une insertion à la position 0 après la suppression. Les deux chemins donnent x_b .

Les cas délicats arrivent vite. Deux insertions à la même position demandent une règle fixe pour départager, par ex. « l'opération acceptée la première passe devant ». Deux suppressions du même caractère ne doivent pas en supprimer deux. À l'usage, un serveur central fixe l'ordre des opérations et transforme chaque nouvelle opération contre celles déjà acceptées : c'est l'organisation des éditeurs en ligne comme Google Docs.

11.10.3 Nommer les caractères : les CRDT de séquence

Le *CRDT de séquence* évite au contraire les positions. Chaque caractère reçoit à sa création une étiquette unique et stable, formée par ex. d'un compteur et du nom de son auteur. Une insertion dit alors « b après le caractère a » plutôt que « b à la position 1 », ce qui garde le même sens quoi qu'il se soit passé ailleurs dans le texte.

Dans le schéma suivant, l'étiquette 3A désigne le compteur 3 et l'autrice Alice. Le texte commun est `ac`. Alice insère `b` après `a`, Bob insère `x` après `a` : deux insertions concurrentes au même endroit.



Sous chaque caractère, son étiquette. Après fusion, les deux insertions suivent `a` ; une règle fixe, ici « la plus grande étiquette d'abord », place `x` (3B) avant `b` (3A), sur toutes les copies. L'effacement de `c` laisse une pierre tombale.

Quand deux auteurs insèrent au même endroit, une règle fixe, comme la comparaison de leurs étiquettes, choisit leur ordre, le même sur toutes les copies : ici `axbc`. Enfin, une suppression ne retire pas le caractère : elle le marque comme effacé, pour que les insertions qui y font référence restent interprétables. Nous retrouvons la pierre tombale des ensembles.

La fusion de deux copies devient alors une union : celle des caractères étiquetés, et celle des marques d'effacement. Les trois lois viennent, une fois de plus, des propriétés de l'union.

11.10.4 Deux familles de solutions

	OT	CRDT de séquence
ce qui circule	des opérations à positions	des états ou des opérations étiquetées
qui ordonne	en général un serveur central	une règle fixe, sur chaque copie
ce que l'on garde	l'historique des opérations récentes	les étiquettes et les pierres tombales
projet d'AP9	éditeur OT ⁷⁴	éditeur concurrent ⁷⁵

Dans les deux projets, l'interface est fournie et votre travail porte sur l'algorithme Elixir et sur les raisons de ses résultats. Leurs garanties reposent sur les mêmes idées que notre compteur : une réunion qui ne dépend pas de l'ordre de livraison, et des copies qui convergent une fois les échanges terminés.

⁷⁴ <https://ap-2026.noce.fr/seances/9/editeur-ot/>

⁷⁵ <https://ap-2026.noce.fr/seances/9/editeur-concurrent/>

POUR ALLER PLUS LOIN

La transformation opérationnelle remonte à l'éditeur GROVE d'Ellis et Gibbs, en 1989 ; garantir qu'elle converge avec plus de deux éditeurs et sans serveur s'est révélé difficile, d'où le serveur central des systèmes actuels. Les CRDT de séquence sont plus récents, avec WOOT (2006) puis RGA (2011), et les bibliothèques Automerge et Yjs les utilisent aujourd'hui. La section 8.1 des [notes de Kleppmann](#)⁷⁶ compare les deux approches, et le site [crdt.tech](#)⁷⁷ rassemble articles et implémentations.

11.11 Vérifier sa compréhension

Refaites sur papier ces situations, en prédisant chaque réponse avant de lire celle qui suit.

Premier exercice, une fusion à la main. Alice connaît $\%{\text{alice: 4, bob: 1}}$ et Carol $\%{\text{bob: 2, carol: 3}}$. Que vaut leur fusion, et quel total annonce-t-elle ? Que donnerait la somme des deux totaux ? La fusion vaut $\%{\text{alice: 4, bob: 2, carol: 3}}$, de total 9. La somme des totaux donnerait $5 + 5 = 10$: la contribution de Bob connue des deux côtés serait comptée deux fois.

Deuxième exercice, trois questions sur le modèle.

1. Alice reçoit deux fois un snapshot de Bob, puis un snapshot plus ancien. Son état peut-il reculer ? Non : chaque composante garde son maximum.
2. Alice ajoute une contribution pendant une coupure. Bob peut-il afficher immédiatement le nouveau total ? Non : les lois de fusion ne lui donnent pas une information qu'il n'a jamais reçue.
3. Deux auteurs utilisent accidentellement le même identifiant. Chacun ajoute une contribution depuis zéro : leur fusion donne-t-elle 2 ? Non : le maximum de 1 et 1 vaut 1. Nous avons violé le contrat d'un propriétaire unique par composante.

Troisième exercice, un PN-Counter. La porte A connaît $P = \%{\text{a: 4, b: 1}}$ et $N = \%{\text{a: 2}}$, la porte B $P = \%{\text{a: 3, b: 2}}$ et $N = \%{\text{b: 1}}$. Combien de personnes chacune annonce-t-elle, puis la fusion ? A annonce $5 - 2 = 3$ et B $5 - 1 = 4$. La fusion donne $P = \%{\text{a: 4, b: 2}}$ et $N = \%{\text{a: 2, b: 1}}$, soit $6 - 3 = 3$ personnes.

Quatrième exercice, des ensembles. Dans un 2P-Set, Alice retire le lait, puis Bob l'ajoute de nouveau après avoir reçu l'état d'Alice. Le lait est-il présent après fusion ? Non : sa pierre tombale reste dans les retraits. Dans un OR-Set, le nouvel ajout de Bob porterait une étiquette neuve, que le retrait d'Alice n'a pas vue : le lait serait présent.

Enfin, une transformation. Sur `abc`, Alice supprime le caractère `0` et Bob insère `x` à la position 2. Que devient l'opération de Bob chez Alice, et quel texte obtiennent-ils ? Chez Alice, le texte est `bc` : un caractère `a` disparu avant la position 2, qui devient 1. Les deux copies affichent `bxc`.

⁷⁶ <https://www.cl.cam.ac.uk/teaching/2122/ConcDisSys/dist-sys-notes.pdf>

⁷⁷ <https://crdt.tech/>

Pour revenir au travail de séance, reprenez les tests et les traces du guide AP8⁷⁸, puis comparez les deux sujets éditeurs dans le catalogue d'AP9⁷⁹. La présentation de référence des familles de CRDT est celle de Preguiça, Baquero et Shapiro (2018)⁸⁰.

POUR EN PARLER

Additionner des totaux recompte les doublons et l'histoire commune : deux copies à 6 issues d'un même 5 donnent 12 au lieu de 7, et le maximum des totaux donne 6.

Pendant une partition, il faut choisir entre répondre partout et garantir que tout le monde voie le même état : notre compteur reste disponible et converge à terme, une réservation de dernière place demande la cohérence forte.

Un G-Counter garde une composante par auteur, que seul cet auteur incrémente ; la fusion garde le maximum de chaque composante, et le total les additionne : Alice 2 contre 1, Bob 1 contre 3 et Carol absente contre 1 donnent $2 + 3 + 1 = 6$.

L'idempotence protège des doublons, et l'addition ne l'a pas ; la commutativité protège du désordre, et « garder l'état reçu » ne l'a pas ; l'associativité protège des relais, et la moyenne ne l'a pas : 5 par un chemin, 3 par l'autre.

Un snapshot est une valeur immuable : pris avant une fusion, il ne relaie pas ce que sa réplique apprend ensuite.

Un PN-Counter compte ajouts et retraits dans deux G-Counters ; un 2P-Set garde ses retraits comme pierres tombales, un OR-Set étiquette chaque ajout pour qu'un retrait ne vise que ce qu'il a vu.

Dans un texte, une position dépend de l'histoire : l'OT transforme les opérations reçues, un CRDT de séquence étiquette chaque caractère et garde les effacés.

⁷⁸ <https://ap-2026.noce.fr/seances/8/>

⁷⁹ <https://ap-2026.noce.fr/seances/9/>

⁸⁰ <https://arxiv.org/abs/1805.06358>

Relier ce que nous avons construit

Au début du cours, une fonction transformait une valeur. Nous avons ensuite transformé un texte en arbre, un arbre en instructions, des observations en paramètres, puis des messages en nouveaux états. Ces objets sont différents, mais quatre questions permettent de les comprendre tous.

Quelle est la représentation ? Une liste chaînée favorise le traitement par la tête, un arbre rend explicites les priorités d'un calcul, une matrice décrit un lot d'exemples, et les composantes d'un G-Counter gardent une information qu'un total seul ferait perdre. **Qu'est-ce qui reste vrai ?** L'accumulateur résume la partie déjà parcourue, la pile d'une machine contient les résultats intermédiaires attendus, une réplique n'incrémente que sa propre composante : ces invariants servent à écrire le code et à trouver le premier endroit où une trace s'écarte du prévu. **Qu'a-t-on effectivement vérifié ?** Un programme typé peut encore diviser par zéro, une récursion terminale peut ne jamais terminer, une faible erreur d'apprentissage ne garantit rien sur un autre lot, et un superviseur redémarre un processus sans retrouver ses données. **Comment le vérifier autrement ?** En comparant l'interpréteur et la machine sur la même expression, en changeant l'ordre des messages, en livrant deux fois un snapshot.

À RETENIR

Pour votre projet d'AP9, choisissez un petit comportement complet que vous savez expliquer. Donnez ses entrées, ses sorties, une trace et un cas limite, puis ajoutez une difficulté. Un résultat mesuré et ses limites apprennent davantage qu'une longue liste de fonctionnalités inachevées.

Les sujets, leurs squelettes et leurs tests restent les références pour le travail attendu. Pour toute question, vous pouvez me contacter par email : aurelien.noce@minesparis.psl.eu.

Aide-mémoire

Cette annexe rassemble ce qu'il faut avoir sous les yeux pour relire un programme Elixir ou en écrire un, longtemps après le cours. Rien n'y est nouveau : chaque ligne renvoie au chapitre qui l'explique, avec ses exemples et ses pièges. Le glossaire qui la suit définit en une phrase les notions de langage, de machine, d'apprentissage et de concurrence que le poly a introduites.

Les valeurs

forme	exemples	à savoir	chap.
entier	<code>42</code> , <code>1_000_000</code>	précision arbitraire, comme en Python	1
flottant	<code>3.14</code> , <code>1.0e10</code>	64 bits, donc approché ; <code>/</code> en rend toujours un	1
chaîne	<code>"Ada"</code> , <code>"#{n} ans"</code>	guillemets doubles ; <code><></code> concatène, <code>#{...}</code> interpole	1
atome	<code>:ok</code> , <code>true</code> , <code>nil</code>	constante qui se nomme elle-même ; seuls <code>false</code> et <code>nil</code> sont faux	1
tuple	<code>{:ok, 42}</code>	quelques cases contiguës de rôles connus ; <code>elem/2</code> , <code>tuple_size/1</code>	1, 3
liste	<code>[1, 2]</code> , <code>[t q]</code>	chaînée : ajout en tête immédiat, accès au <i>i</i> -ème en <i>i</i> pas	3
map	<code>%{"a" => 1}</code> , <code>%{nom: "Ada"}</code>	clé → valeur, sans ordre ; clés de tout type	3
keyword list	<code>[taille: 42]</code>	liste de couples <code>{atome, valeur}</code> , pour les options	3
intervalle	<code>1..10</code> , <code>1..10//3</code>	deux bornes et un pas, sans liste construite	3
fonction	<code>fn x -> x * 2 end</code>	une valeur comme une autre ; s'appelle avec <code>f.(x)</code>	4

Les opérateurs

opérateurs	sens	chap.
<code>+</code> , <code>-</code> , <code>*</code> , <code>**</code>	arithmétique ; <code>2 ** 100</code> reste exact	1
<code>/</code> , <code>div</code> , <code>rem</code>	<code>10 / 2</code> vaut <code>5.0</code> ; <code>div(10, 3)</code> vaut <code>3</code> , <code>rem(10, 3)</code> vaut <code>1</code>	1
<code>==</code> , <code>!=</code> , <code><</code> , <code><=</code> , <code>></code> , <code>>=</code>	comparaisons ; <code>1 == 1.0</code> est vrai	1

opérateurs	sens	chap.
<code>==</code> , <code>!=</code>	égalité stricte, type compris : <code>1 == 1.0</code> est faux	1
<code>and</code> , <code>or</code> , <code>not</code>	logique sur des booléens seulement	1
<code>&&</code> , <code> </code> , <code>!</code>	logique sur toute valeur ; <code>x défaut</code> fournit un repli	1
<code><></code>	concatène deux chaînes ; dans un motif, reconnaît un préfixe	1, 2
<code>++</code> , <code>--</code>	concatène deux listes ; retire la première occurrence de chaque élément	3
<code>[x liste]</code> , <code>in</code>	ajoute une tête ; teste l'appartenance	3
<code> ></code>	<code>a > f(b)</code> signifie <code>f(a, b)</code>	3
<code>=</code> , <code>^</code>	confronte un motif à une valeur ; le pin compare au lieu de lier	2

Les motifs

Un motif décrit la forme attendue d'une valeur. Il apparaît à gauche de `=`, dans les clauses de `case`, de `receive` et des fonctions, et à gauche de `<-` dans `with`.

motif	ce qu'il exige de la valeur	ce qu'il lie
<code>x</code>	rien : une variable accepte tout	<code>x</code>
<code>42</code> , <code>:ok</code> , <code>"ok"</code>	exactement cette valeur	rien
<code>_</code>	rien, et ne lie aucun nom	rien
<code>{a, b}</code>	un tuple d'exactly deux cases	<code>a</code> , <code>b</code>
<code>{:ok, v}</code>	un tuple de deux cases commençant par <code>:ok</code>	<code>v</code>
<code>[t q]</code>	une liste non vide	<code>t</code> , <code>q</code>
<code>[]</code>	la liste vide	rien
<code>%{cle: v}</code>	une map ayant <i>au moins</i> la clé <code>:cle</code>	<code>v</code>
<code>%{^k => v}</code>	une map ayant la clé contenue dans <code>k</code>	<code>v</code>
<code>"+" <> r</code>	une chaîne qui commence par <code>+</code>	<code>r</code>
<code>^x</code>	la valeur actuelle de <code>x</code>	rien
<code>m = motif</code>	ce qu'exige <code>motif</code> ; <code>m</code> reçoit la valeur entière	<code>m</code> et ceux du motif

Les fonctions, dans l'ordre du chapitre 4

Une fonction nommée vit dans un module ; une fonction anonyme est une valeur. Voici les formes de déclaration, dans l'ordre où le chapitre 4 les présente, d'abord dans un module :

```

1 # un module range des fonctions nommées
2 defmodule Notes do
3   # publique : Notes.moyenne/1, corps en bloc
4   def moyenne(liste) do
5     somme(liste) / length(liste)
6   end
7
8   # forme courte do:, une clause par cas, une garde
9   def mention(n) when n >= 16, do: "TB"
10  def mention(n) when n >= 10, do: "admis"
11  def mention(_n), do: "ajourné"
12
13  # valeur par défaut : bonus/1 et bonus/2
14  def bonus(n, points \\ 1), do: n + points
15
16  # privée : appellable dans le module seulement
17  defp somme(liste), do: Enum.sum(liste)
18 end

```

puis hors d'un module, où les fonctions anonymes sont des valeurs que l'on range sous un nom :

```

1 # anonyme, appelée avec un point : double.(5)
2 double = fn x -> x * 2 end
3
4 # anonyme à plusieurs clauses
5 lire = fn
6   {:ok, v} -> v
7   {:error, _} -> 0
8 end
9
10 # capture abrégée : &1 est le premier argument
11 triple = &(61 * 3)
12
13 # capture d'une fonction existante, nom/arité
14 taille = &String.length/1

```

appel	quand l'écrire
Notes.moyenne([12, 15])	une fonction publique, depuis l'extérieur du module
somme(liste)	une fonction du même module, publique ou privée
double.(5)	une fonction anonyme rangée dans une variable : le point est obligatoire
Enum.map(l, &Notes.mention/1)	passer une fonction nommée en argument
Enum.map(l, fn x -> x + 1 end)	passer un calcul écrit sur place

nom/arité identifie une fonction : `bonus/1` et `bonus/2` sont deux fonctions distinctes. Une garde (`when`) n'accepte que des opérations simples, comparaisons, arithmétique, tests de type comme `is_list/1`, `in`, `length/1`, `map_size/1`, jamais une fonction écrite par vous.

Choisir et enchaîner

forme	quand l'employer	chap.
<code>case v do ... end</code>	plusieurs formes possibles d'une valeur, une clause motif <code>-> ...</code> par forme	2
<code>cond do ... end</code>	des conditions essayées dans l'ordre, par ex. des seuils	2
<code>if c, do: a, else: b</code>	deux branches selon une condition ; la valeur du <code>if</code> est celle de la branche	1, 2
clauses de fonction	une clause par forme d'argument, avec ou sans garde	4
<code>with ... <- ... do ... end</code>	enchaîner des étapes qui réussissent ; rendre la première non-correspondance	4
réursion	traiter la tête, rappeler sur le reste ; un cas de base sur <code>[]</code>	4
<code>Enum.reduce/3</code>	parcourir en transportant un accumulateur	4

Lire une erreur

erreur	ce qu'elle signifie	chap.
<code>MatchError</code>	le motif d'un <code>=</code> ne correspond pas à la valeur, affichée après le message	2
<code>CaseClauseError</code>	aucune clause du <code>case</code> n'accepte la valeur	2
<code>FunctionClauseError</code>	aucune clause de la fonction n'accepte les arguments, listés sous le message	4
<code>UndefinedFunctionError</code>	fonction inexistante, mauvaise arité, ou privée appelée de l'extérieur	4
<code>KeyError</code>	<code>map.cle</code> ou <code>%{map cle: v}</code> sur une clé absente	3
<code>ArithmeticError</code>	calcul sur autre chose qu'un nombre, ou division par zéro	1
<code>BadBooleanError</code>	<code>and</code> , <code>or</code> ou <code>not</code> sur une valeur qui n'est pas un booléen	1
<code>Protocol.UndefinedError</code>	<code>Enum</code> appliqué à une valeur non énumérable, un tuple par ex.	3
<code>Enum.EmptyError</code>	<code>Enum.max/1</code> ou <code>Enum.min/1</code> sur une collection vide	4
<code>ExUnit.AssertionError</code>	un <code>assert</code> faux : <code>left</code> est calculé, <code>right</code> attendu	1
<code>undefined variable</code>	erreur de compilation : nom jamais lié ici, par ex. lié dans une branche et lu après	2

Les fonctions de bibliothèque à connaître

fonction	ce qu'elle fait
<code>Enum.map(c, f)</code> , <code>Enum.filter(c, f)</code>	transforme chaque élément ; garde ceux pour lesquels <code>f</code> répond vrai
<code>Enum.reduce(c, acc, f)</code>	replie la collection ; <code>f</code> reçoit l'élément puis l'accumulateur
<code>Enum.sum</code> , <code>Enum.count</code> , <code>Enum.min</code> , <code>Enum.max</code>	des réductions déjà écrites

fonction	ce qu'elle fait
<code>Enum.all?</code> , <code>Enum.any?</code>	<code>f</code> répond-elle vrai pour tous, pour au moins un ?
<code>Enum.sort</code> , <code>Enum.sort_by</code> , <code>Enum.reverse</code>	trier, trier selon une clé, retourner
<code>Enum.at</code> , <code>Enum.take</code> , <code>Enum.uniq</code> , <code>Enum.chunk_every</code>	<i>i</i> -ème, premiers, sans doublons, par paquets
<code>Enum.join(l, sep)</code> , <code>Enum.each(c, f)</code>	assembler en chaîne ; appeler <code>f</code> pour son effet, rend <code>:ok</code>
<code>String.split/1</code> , <code>String.split/2</code>	découper sur les blancs ; sur un séparateur, en gardant les vides
<code>String.length</code> , <code>String.upcase</code> , <code>String.trim</code>	longueur, majuscules, blancs des extrémités retirés
<code>String.to_integer</code> , <code>Integer.parse</code>	convertir ; <code>Integer.parse</code> rend <code>{n, reste}</code> ou <code>:error</code>
<code>Map.get(m, k, défaut)</code> , <code>Map.put(m, k, v)</code>	lire avec repli ; rendre une map avec une clé de plus ou changée
<code>Map.keys</code> , <code>Map.values</code> , <code>Map.has_key?</code>	les clés, les valeurs, la présence d'une clé
<code>I0.puts(x)</code> , <code>I0.inspect(x, label: "...")</code>	afficher et rendre <code>:ok</code> ; afficher et rendre <code>x</code> inchangé

Processus et OTP

forme	ce qu'elle fait	chap.
<code>self()</code>	le PID du processus qui exécute ce code	9
<code>spawn(fn -> ... end)</code>	lance la fonction dans un nouveau processus et rend son PID	9
<code>send(pid, message)</code>	dépose le message dans la boîte de <code>pid</code> , sans attendre ; rend le message	9
<code>receive do ... after n -> ... end</code>	retire le premier message qui correspond à une clause ; <code>after</code> borne l'attente à <code>n</code> millisecondes	9
<code>make_ref()</code>	une référence unique, recopiée dans la réponse pour l'apparier à sa demande	9
<code>Process.monitor(pid)</code> , <code>spawn_monitor</code>	demande un message <code>{:DOWN, ref, :process, pid, raison}</code> à la fin du processus	9
<code>Process.link(pid)</code> , <code>spawn_link</code>	lie deux destins : une fin anormale se propage, ou arrive en <code>{:EXIT, pid, raison}</code>	9
<code>Agent.start_link(fn -> état end)</code>	un processus qui garde un état ; <code>get</code> , <code>update</code> et <code>get_and_update</code> lui confient une fonction	10
<code>GenServer.start_link(M, arg)</code>	démarre un serveur ; OTP appelle <code>M.init(arg)</code> , qui rend <code>{:ok, état}</code>	10

forme	ce qu'elle fait	chap.
<code>GenServer.call(s, requête)</code>	envoie et attend ; <code>handle_call(requête, from, état)</code> rend <code>{:reply, réponse, état}</code>	10
<code>GenServer.cast(s, requête)</code>	envoie sans attendre ; <code>handle_cast(requête, état)</code> rend <code>{:noreply, état}</code>	10
<code>handle_info(message, état)</code>	traite un message ordinaire, envoyé par <code>send</code> ou par un monitor	10
<code>Supervisor.start_link(enfants, strategy: ...)</code>	démarre les enfants décrits par leurs recettes et relance ceux qui tombent, selon la stratégie, par ex. <code>:one_for_one</code>	9
<code>Task.async</code> , <code>Task.await_many</code>	lancer des calculs dans des processus, puis attendre leurs résultats	10
<code>mix new nom --sup</code> , <code>mix test</code> , <code>iex -S mix</code>	créer un projet supervisé, lancer ses tests, l'ouvrir dans IEx	9
<code>Map.merge(a, b, fun)</code>	fusion de deux maps ; avec <code>fn _k, x, y -> max(x, y) end</code> , celle de deux G-Counters, composante par composante	11

Nx et Axon

forme	ce qu'elle fait (chapitres 7 et 8)
<code>Nx.tensor([[1, 2], [3, 4]])</code>	un tenseur, ici de forme <code>{2, 2}</code>
<code>Nx.shape(t)</code> , <code>Nx.type(t)</code>	la forme, tuple des tailles des axes ; le type des éléments, par ex. <code>{:f, 32}</code>
<code>Nx.add</code> , <code>Nx.subtract</code> , <code>Nx.multiply</code> , <code>Nx.pow</code>	opérations case par case, avec broadcasting quand les formes s'y prêtent
<code>Nx.sum(t, axes: [0])</code> , <code>Nx.mean</code>	réduit l'axe 0, qui disparaît de la forme ; <code>keep_axes: true</code> le garde en taille 1
<code>Nx.dot(a, b)</code>	produit scalaire, matrice-vecteur ou matriciel, selon les formes
<code>Nx.reshape(t, forme)</code> , <code>Nx.transpose(t)</code>	mêmes valeurs dans un autre rangement ; axes échangés
<code>Nx.to_number(t)</code> , <code>Nx.to_list(t)</code>	revenir à un nombre ou à une liste Elixir
<code>defn f(x) do ... end</code>	fonction numérique, après <code>import Nx.Defn</code> : Nx en construit le graphe
<code>grad(x, fn x -> ... end)</code>	dans un <code>defn</code> , la dérivée ou le gradient au point <code>x</code> ; <code>Nx.Defn.grad</code> hors d'un <code>defn</code>
<code>Axon.input("points", shape: {nil, 2})</code>	l'entrée d'un réseau ; <code>nil</code> laisse libre le nombre d'exemples
<code>Axon.dense(couche, k,</code> <code>activation: :relu)</code>	une couche dense de <code>k</code> unités, puis une activation
<code>Axon.Loop.trainer(model, perte,</code> <code>optimiseur)</code>	assemble la boucle d'entraînement

forme	ce qu'elle fait (chapitres 7 et 8)
<code>Axon.Loop.run(boucle, lots, état, epochs: e)</code>	exécute <code>e</code> époques et rend les paramètres appris
<code>Axon.build(model)</code>	rend deux fonctions : initialiser les paramètres, prédire
<code>Polaris.Optimizers.adam(...)</code>	l'optimiseur Adam ; <code>learning_rate</code> : fixe le taux η

Glossaire

Chaque notion est définie en une phrase, avec le chapitre qui l'explique et l'illustre. Les termes sont rangés par partie, dans l'ordre où le poly les rencontre.

Écrire en Elixir

expression : un bout de code qui a une valeur ; en Elixir, tout en est une, `if` compris. (ch. 1)

valeur et console : ce qu'une expression rend, que le programme réutilise ; ce qu'elle affiche, que seul le lecteur voit. (ch. 1)

atome : une constante dont le nom est la valeur, comme `:ok` ; `true`, `false` et `nil` en sont. (ch. 1)

immutabilité : une valeur ne change jamais ; on en fabrique une neuve à partir de l'ancienne. (ch. 1)

liaison : l'association d'un nom à une valeur ; réutiliser le nom le relie à une valeur neuve sans modifier l'ancienne. (ch. 1)

motif : la forme attendue d'une valeur, avec des trous à remplir, confrontée à une valeur par `=`, `case` ou une clause. (ch. 2)

déstructurer : extraire les morceaux d'une valeur en la confrontant à un motif de même forme. (ch. 2)

pin : l'opérateur `^`, qui fait comparer une variable à sa valeur actuelle au lieu de la relier. (ch. 2)

clause : une ligne `motif -> expression` d'un `case`, ou une définition d'une fonction pour une forme d'arguments. (ch. 2, 4)

garde : une condition `when` ajoutée à une clause, limitée à des opérations simples. (ch. 4)

tête et queue : le premier élément d'une liste non vide ; la liste de tous les suivants. (ch. 2, 3)

liste chaînée : une suite de maillons, chacun portant une valeur et un lien vers le suivant. (ch. 3)

map : un ensemble de paires clé-valeur, sans ordre. (ch. 3)

keyword list : une liste de couples `{atome, valeur}`, employée pour les options. (ch. 3)

pipe : l'opérateur `|>`, qui passe la valeur de gauche en premier argument de la fonction de droite. (ch. 3)

module : un regroupement de fonctions nommées, déclaré par `defmodule`. (ch. 4)

arité : le nombre d'arguments d'une fonction ; `nom/arité` l'identifie. (ch. 4)

fonction anonyme : une fonction sans nom, que l'on range ou que l'on passe comme une valeur, appelée avec un point. (ch. 3, 4)

capture : l'opérateur `&`, qui abrège une fonction anonyme (`&(&1 * 2)`) ou désigne une fonction existante (`&String.upcase/1`). (ch. 4)

réursion : une fonction qui s'appelle sur une entrée plus petite, jusqu'à un cas de base.

(ch. 4)

cas de base : la clause qui répond sans rappel, sur `[]` pour une liste. (ch. 4)

accumulateur : un argument qui transporte le résultat partiel d'un parcours. (ch. 4)

appel terminal : un appel dont le résultat est rendu tel quel, sans calcul en attente ; la pile d'appels reste bornée. (ch. 4)

énumérable : une collection que les fonctions d' `Enum` savent parcourir : liste, map ou intervalle, mais pas tuple. (ch. 4)

Langages et machines

langage cible, langage d'implémentation : le langage que notre outil lit et exécute ; celui dans lequel l'outil est écrit. (ch. 5)

token : un « mot » du langage, comme `:plus` ou `{:number, 12.0}`. (ch. 5)

lexer : la brique qui découpe le texte en tokens, de gauche à droite. (ch. 5)

parser : la brique qui organise les tokens en arbre selon la grammaire. (ch. 5)

arbre de syntaxe abstraite (AST) : l'arbre qui dit quelle opération s'applique à quoi, sans la ponctuation. (ch. 5)

postordre : un parcours qui traite chaque nœud après ses enfants. (ch. 5, 6)

grammaire : un ensemble de règles qui décrit tous les textes admis par un langage. (ch. 5)

BNF : la notation des grammaires de Backus et Naur : `:=`, terminaux entre guillemets, `|`, `*`, `?`. (ch. 5)

dérivation : la vérification pas à pas, règle après règle, qu'un texte est admis. (ch. 5)

grammaire ambiguë : une grammaire qui admet deux arbres pour le même texte. (ch. 5)

descente récursive : un parser écrit avec une fonction de lecture par règle, qui rend l'arbre et le reste. (ch. 5)

syntaxe et sémantique : la forme d'un texte ; le sens de ce qu'il calcule. (ch. 5)

environnement : les liaisons connues à un endroit du programme, la plus récente d'abord. (ch. 5)

portée : la partie du programme où une liaison est visible ; lexicale si elle se lit dans le texte, dynamique si elle dépend de l'appelant. (ch. 5)

masquer : cacher une liaison par une autre du même nom, sans modifier la première. (ch. 5)

variable libre : un nom du corps d'une fonction qui n'est ni un paramètre ni une liaison locale. (ch. 5)

fermeture : la valeur d'une fonction : ses paramètres, son corps et l'environnement de sa définition. (ch. 5)

évaluation stricte : les arguments sont calculés avant d'entrer dans le corps de la fonction. (ch. 5)

forme spéciale : une construction qui décide elle-même quels sous-arbres évaluer, comme `case`. (ch. 5)

compilateur : un programme qui traduit, une fois pour toutes, un arbre en instructions pour une machine. (ch. 6)

bit et octet : un chiffre valant 0 ou 1 ; un groupe de huit bits, soit un nombre de 0 à 255. (ch. 6)

adresse : le numéro d'une case de la mémoire, à distinguer de son contenu. (ch. 6)

processeur (CPU) : le circuit qui exécute les instructions d'un programme, l'une après l'autre. (ch. 6)

registre : l'un des quelques emplacements nommés du processeur qui gardent des valeurs de travail. (ch. 6)

compteur ordinal (PC) : le registre qui contient l'adresse de la prochaine instruction. (ch. 6)

cycle lire-décoder-exécuter : la boucle du processeur : lire les octets au PC, reconnaître l'instruction, l'appliquer, choisir le PC suivant. (ch. 6)

cache : une petite mémoire rapide, gérée par le matériel, qui garde des copies de la mémoire principale. (ch. 6)

hexadécimal : l'écriture en base seize, de 0 à F ; un chiffre vaut quatre bits. (ch. 6)

big endian, little endian : l'octet de poids fort d'abord, ou l'octet de poids faible d'abord. (ch. 6)

format binaire : la convention qui fixe position, taille, ordre et sens de chaque champ ; un contrat entre qui écrit et qui lit. (ch. 6)

machine virtuelle (VM) : un programme qui joue le rôle d'un processeur en lisant et appliquant des instructions. (ch. 6)

machine à pile : une machine dont les instructions prennent leurs opérandes au sommet d'une pile et y déposent leur résultat. (ch. 6)

opcode, opérande : le premier octet d'une instruction, qui dit l'opération ; la donnée qui le suit. (ch. 6)

bytecode : un programme codé en octets pour une machine virtuelle. (ch. 6)

saut : une instruction qui donne au PC une autre valeur que l'adresse suivante ; conditionnel s'il dépend d'un booléen. (ch. 6)

étiquette et assembleur : un nom qui tient la place d'une adresse ; l'outil qui remplace les étiquettes par des adresses, en deux passes. (ch. 6)

contrôle statique, dynamique : une vérification faite avant l'exécution, sur le texte ; ou pendant, sur les valeurs. (ch. 6)

typeur : un outil qui associe un type à chaque nœud de l'arbre sans exécuter le programme. (ch. 6)

Calculer et apprendre

apprentissage supervisé : ajuster les paramètres d'une formule à partir d'exemples dont la réponse est connue. (ch. 7)

modèle et paramètres : la formule à paramètres ; les nombres que l'entraînement modifie, poids et biais. (ch. 7)

exemple, étiquette, caractéristiques : un cas connu ; sa réponse attendue ; les nombres qui le décrivent. (ch. 7)

tenseur : un tableau de nombres de même type rangés selon des axes, l'équivalent d'un tableau NumPy. (ch. 7)

forme et axe : le tuple des tailles, par ex. {16, 2} ; une direction de rangement, numérotée à partir de 0. (ch. 7)

réduction : remplacer tout un axe par une seule valeur, somme ou moyenne ; l'axe disparaît de la forme. (ch. 7)

broadcasting : étendre un tenseur en répétant ses valeurs pour l'accorder à la forme d'un autre. (ch. 7)

somme pondérée : la somme des entrées multipliées chacune par son poids ; le produit scalaire de deux vecteurs. (ch. 7)

produit matrice-vecteur : le produit scalaire de chaque ligne d'une matrice avec un vecteur. (ch. 7)

graphe de calcul : les opérations d'une fonction numérique reliées par leurs dépendances. (ch. 7)

perte : un nombre qui dit à quel point les prédictions sont fausses. (ch. 8)

erreur quadratique moyenne (MSE) : la moyenne des carrés des écarts entre prédictions et réponses. (ch. 8)

dérivée : la pente de la tangente ; de combien le résultat bouge quand l'entrée bouge un peu. (ch. 8)

dérivée partielle : la dérivée par rapport à un seul paramètre, les autres étant gelés. (ch. 8)

gradient : le vecteur des dérivées partielles, une par paramètre, noté ∇L . (ch. 8)

règle de la chaîne : le long d'une composition, les dérivées locales se multiplient. (ch. 8)

descente de gradient : retrancher η fois le gradient aux paramètres, puis recommencer. (ch. 8)

taux d'apprentissage : le nombre positif η qui règle la longueur du pas. (ch. 8)

lot et époque : un paquet d'exemples traités ensemble ; un passage sur tous les exemples d'apprentissage. (ch. 8)

frontière de décision : les points où le score atteint le seuil ; une droite pour un score affine. (ch. 8)

apprentissage, validation, test : les exemples qui ajustent les paramètres ; ceux qui comparent des recettes ; ceux qui mesurent une seule fois la recette retenue. (ch. 8)

surapprentissage : ajuster les particularités des exemples d'apprentissage sans bénéfice sur les autres. (ch. 8)

neurone et activation : une somme pondérée, un biais, puis une fonction non linéaire appliquée au résultat. (ch. 8)

couche dense : k neurones sur les mêmes entrées, soit un produit par une matrice de poids. (ch. 8)

sigmoïde : $\sigma(z) = 1/(1 + e^{-z})$, qui écrase un score entre 0 et 1. (ch. 8)

entropie croisée binaire : $-\ln$ de la probabilité donnée à la bonne classe ; elle punit l'assurance dans l'erreur. (ch. 8)

optimiseur : la règle de mise à jour des paramètres, de la descente simple à Adam. (ch. 8)

hyperparamètres : les choix faits avant l'entraînement : largeur, activation, taux, époques, graine. (ch. 8)

exactitude : la proportion de décisions correctes. (ch. 8)

Faire coopérer des processus

concurrence : plusieurs tâches en cours sur une même période, dont les étapes s'entrelacent. (ch. 9)

parallélisme : plusieurs cœurs qui calculent au même instant. (ch. 9)

ordonnanceur : la partie du système qui répartit le temps des cœurs entre les tâches prêtes. (ch. 9)

thread : une exécution à l'intérieur d'un processus du système, qui partage toute sa mémoire avec les autres threads. (ch. 9)

processus BEAM : une unité d'exécution de la VM, isolée et légère, identifiée par son PID. (ch. 9)

boîte aux lettres : la file des messages qu'un processus a reçus et pas encore lus. (ch. 9)

condition de course : une situation où le résultat dépend de l'ordre dans lequel s'entrelacent plusieurs tâches. (ch. 9)

verrou : une protection prise avant d'agir sur une donnée partagée, puis rendue. (ch. 9)

asynchrone, synchrone : l'expéditeur continue sans attendre ; l'appelant attend le résultat. (ch. 9)

réception sélective : `receive` retire le premier message qui correspond à une clause et laisse les autres. (ch. 9)

diagramme de séquence : un schéma où chaque processus a une ligne de vie verticale, le temps descend et les flèches sont des messages. (ch. 9)

interblocage : deux processus qui attendent chacun un message que seul l'autre pourrait envoyer. (ch. 9)

lien et monitor : relier les destins de deux processus ; observer la fin d'un processus par un message `:DOWN`. (ch. 9)

laisser planter : s'arrêter face à l'imprévu plutôt que réparer son état, et laisser un superviseur repartir d'un état connu. (ch. 9)

superviseur : un processus qui démarre ses enfants et les relance selon une stratégie. (ch. 9)

arbre de supervision : l'organisation d'une application, où des superviseurs surveillent des travailleurs ou d'autres superviseurs. (ch. 9)

motif client/serveur : un processus serveur possède l'état ; les clients lui adressent des requêtes. (ch. 10)

callback : une fonction du serveur qu'OTP appelle, comme `init`, `handle_call` ou `handle_cast`. (ch. 10)

disponibilité et latence : pouvoir travailler même si un serveur tombe ; le délai entre une action et son effet visible. (ch. 11)

partition réseau : des machines en marche mais séparées en groupes qui ne communiquent plus. (ch. 11)

réplique : la copie de l'état que garde chaque poste, modifiée localement puis réconciliée. (ch. 11)

G-Counter : un compteur croissant à une composante par auteur, fusionné par maximum composante par composante. (ch. 11)

snapshot : une copie immuable de l'état à un instant, qui ne suit pas les modifications futures. (ch. 11)

fusion : l'opération $a \sqcup b$ qui calcule un état contenant tout ce que chacun savait. (ch. 11)

idempotence, commutativité, associativité : un doublon, l'ordre de deux messages, le passage par un relais ne changent rien à la fusion. (ch. 11)

convergence à terme : les répliques finissent par porter le même état dès que les mises à jour ont circulé. (ch. 11)

CRDT : une structure de données répliquée conçue pour converger sans coordination centrale. (ch. 11)

transformation opérationnelle (OT) : transformer une opération d'édition selon celles déjà intégrées, avec un serveur qui fixe l'ordre. (ch. 11)

Lectures pour continuer

Ces lectures prolongent chaque partie ; aucune n'est nécessaire pour suivre le cours. Les chapitres signalent en outre, dans leurs encarts « Pour aller plus loin », des articles et des documentations plus ciblés.

Écrire en Elixir. La [leçon d'Elixir School](https://elixirschool.com/fr)⁸¹ en français, puis le [guide officiel](https://hexdocs.pm/elixir/introduction.html)⁸², qui reprend le langage pas à pas. Pour un livre complet, *Elixir in Action* de Saša Jurić (Manning), qui va jusqu'à OTP.

Langages et machines. *Crafting Interpreters* de Robert Nystrom, [lisible en ligne](https://craftinginterpreters.com)⁸³, construit un interpréteur puis une machine à bytecode, exactement notre chemin d'AP2 et d'AP3. Le projet [nand2tetris](https://www.nand2tetris.org)⁸⁴ part des portes logiques pour arriver à un compilateur. *Structure and Interpretation of Computer Programs*⁸⁵ d'Abelson et Sussman reste le classique sur les environnements et les interpréteurs.

Calculer et apprendre. *Machine Learning in Elixir* de Sean Moriarity (Pragmatic Bookshelf) reprend Nx et Axon en détail. La série « [Neural networks](https://www.3blue1brown.com/topics/neural-networks) »⁸⁶ de 3Blue1Brown donne une intuition visuelle du gradient et de la rétropropagation, que [micrograd](https://github.com/karpathy/micrograd)⁸⁷ d'Andrej Karpathy écrit en une centaine de lignes. Pour aller beaucoup plus loin, *Deep Learning* de Goodfellow, Bengio et Courville est [en ligne](https://www.deeplearningbook.org)⁸⁸.

Faire coopérer des processus. *Learn You Some Erlang for Great Good!*⁸⁹ de Fred Hébert explique processus, liens et supervision avec humour. La [thèse de Joe Armstrong](https://erlang.org/download/armstrong_thesis_2003.pdf)⁹⁰ raconte pourquoi Erlang laisse planter. Sur la réplication, *Designing Data-Intensive Applications* de Martin Kleppmann (O'Reilly) et [l'article fondateur sur les CRDT](https://hal.inria.fr/inria-00555588)⁹¹ de Shapiro, Preguiça, Baquero et Zawirski.

⁸¹ <https://elixirschool.com/fr>

⁸² <https://hexdocs.pm/elixir/introduction.html>

⁸³ <https://craftinginterpreters.com>

⁸⁴ <https://www.nand2tetris.org>

⁸⁵ https://mitp-content-server.mit.edu/books/content/sectbyfn/books_pres_0/6515/sicp.zip/index.html

⁸⁶ <https://www.3blue1brown.com/topics/neural-networks>

⁸⁷ <https://github.com/karpathy/micrograd>

⁸⁸ <https://www.deeplearningbook.org>

⁸⁹ <https://learnyousomeerlang.com>

⁹⁰ https://erlang.org/download/armstrong_thesis_2003.pdf

⁹¹ <https://hal.inria.fr/inria-00555588>