NOHAR

Zeste de savoir

L'Advent of Code 2020 en Go, jours 16 à 20

20 décembre 2020

Table des matières

Ho, ho, ho!

Me revoilà pour un quatrième épisode de l'Advent Of Code 2020. Pour commencer, voici les liens vers les épisodes précédents:

— Jours 1 à 5 , où l'on plante le décor;
— Jours 6 à 10 , où la difficulté augmente d'un cran,
— Jours 11 à 15 , où bourriner ne marche plus,

Comme vous allez le voir, les exos n'ont plus rien de facile, maintenant!

1. Jour 16 : Faire preuve de déduction

L'exercice du 16e jour consistait à deviner le format d'un document par déduction, à partir d'un ensemble d'exemples.

1.1. Le format d'entrée

L'entrée que l'on nous donnait était composée de trois éléments. Le premier était un ensemble de 20 descriptions de champs, dans le format que voici:

```
departure location: 36-363 or 377-962
```

Il faut ici comprendre que le champ `departure location` est un entier compris entre 36 et 363, ou bien entre 377 et 962. **Tous** les champs sont caractérisés par deux intervalles comme celui-ci.

Le second élément était notre propre billet de train, sous la forme d'une vingtaine d'entiers:

```
89,179,173,167,157,127,163,113,137,109,151,131,97,149,107,83,79,139,59,53
```

Enfin, nous avions une liste de 245 autres exemples de billets de train. Le but ultime de l'exercice est d'arriver à comprendre où se trouvent tous les champs "utiles" de notre billet de train, c'est-à-dire tous les champs dont le nom commence par `departure`.

1.2. Modéliser les données

La modélisation des données est plutôt directe:

```
89,179,173,167,157,127,163,113,137,109,151,131,97,149,107,83,79,139,59,53
```

Un ticket est donc une séquence d'entiers, et un `Field` une structure avec un nom, deux intervalles et… *un masque*, puisque nous allons avoir besoin de réaliser des opérations ensemblistes sur l'ensemble des champs possibles (soit un ensemble de 20 éléments).

Je vous ferai grâce du parsing des entrées dans le billet (le code est là). D'ailleurs, à moins que l'on ne tombe d'ici Noël sur un format qui nécessite une grammaire (ce qui m'étonnerait quand même beaucoup), je pense ne pas trop risquer de me tromper en vous disant que je ferai toujours l'impasse dessus dorénavant.

Ajoutons plutôt une paire de méthodes qui servent à vérifier qu'un nombre est une valeur correcte pour un champ donné:

```
89,179,173,167,157,127,163,113,137,109,151,131,97,149,107,83,79,139,59,53
```

Le but est simplement de rendre le code plus facile à lire par la suite. Si j'avais le temps, je vous montrerais comment vérifier que le compilateur Go va *inliner* ces méthodes à la compilation, mais… on verra ça une autre fois. 🍑

1.3. Partie 1 : Filtrer les tickets mal formés

La première question nous fait rentrer doucement dans le bain: le but est de trouver, dans les tickets d'exemple, toutes les valeurs qui ne peuvent correspondre à aucun champ, et donc de virer ces tickets de notre liste pour la suite.

La fonction suivante calcule le "taux d'erreur" (c'est comme ça que l'appelle l'énoncé) d'un billet unique en additionnant tous les champs erronés:

```
89,179,173,167,157,127,163,113,137,109,151,131,97,149,107,83,79,139,59,53
```

Rien de *très* particulier à noter, si ce n'est que j'ai encore une fois utilisé la syntaxe `continue` sur un label. Cela dit, remarquez que cette fonction retourne quand même un booléen (`ok`) en plus du taux d'erreur. En effet, si l'on reposait sur le fait que l'erreur soit non-nulle pour savoir si un billet est valide, nous nous exposerions à un bug assez vicieux en tombant sur une valeur nulle (acceptée par aucun champ).

Passons maintenant à une fonction qui filtre les tickets en supprimant tous ceux qui sont invalides:

```
89,179,173,167,157,127,163,113,137,109,151,131,97,149,107,83,79,139,59,53
```

Cette fonction n'est pas non plus très compliquée, mais elle comprend quand même un petit *hack* qui vaut le coup d'être vu au moins une fois! Regardez la slice `valid`: plutôt que de l'allouer dynamiquement, je l'initialise comme étant la slice vide qui pointe sur les données de `tickets`. C'est une astuce assez commode pour réutiliser la mémoire déjà allouée: la slice sur laquelle nous écrivons et celle que nous lisons partagent le même espace mémoire sous-jacent. Ainsi, nos appels à `append()` n'auront jamais besoin de provoquer une réallocation des données.

Du moment que nous savons que nous n'essayerons jamais d'écrire plus loin que l'élément que nous sommes en train de lire, nous pouvons tranquillement écraser les données du tableau en même temps que l'on itère dessus. Au pire, si tous les tickets sont valides, cette fonction va simplement copier la mémoire du tableau sur elle-même.

1.4. Partie 2 : Identifier les champs qui nous intéressent par déduction

C'est maintenant qu'on rigole!

Maintenant que nous avons uniquement des tickets bien formés, il va falloir que nous trouvions la position des champs qui nous intéressent, sachant que chaque champ se trouvera systématiquement au même indice dans tous les tickets (et heureusement, sinon on ne pourrait pas résoudre le problème…).

Pour ce faire, il faut nous y prendre par éliminations successives. Commençons par réaliser une première passe sur nos tickets pour déterminer un premier ensemble de "candidats" par indice: nous voulons associer, à chaque indice des tickets, l'ensemble des champs qui peuvent y correspondre. Pour ce faire, c'est assez simple: on prend un champ, et on essaye d'appliquer sa

contrainte au champ d'indice `i` de tous les tickets. Si la contrainte du champ est systématiquement validé sur tous les tickets à la position `i`, alors c'est un candidat possible.

Pour représenter les ensembles de candidats maintenant, souvenez-vous que chaque champ est associé à un "`Mask`": un bit unique dans un entier non-signé. L'ensemble des candidats est donc à son tour un masque binaire. C'est exactement la même logique que celle que j'avais appliquée le 6e jour .

```
89,179,173,167,157,127,163,113,137,109,151,131,97,149,107,83,79,139,59,53
```

Nous avons donc maintenant un tableau `candidates` qui associe à chaque indice l'ensemble des `Field` potentiels. Ensuite, générons un masque `wanted`, dont les bits à 1 indiquent les champs qui nous intéressent (les champs dont le nom commence par `departure`):

```
89,179,173,167,157,127,163,113,137,109,151,131,97,149,107,83,79,139,59,53
```

Tout est en place, nous n'avons plus qu'à jouer au Sudoku! 🍊

Concrètement, on va maintenir un ensemble de champs *connus*, dans lequel on va ajouter les bits des champs dont nous avons pu déterminer la position avec certitude. Puis nous allons itérer sur tous nos candidats tant que tous les champs `wanted` ne sont pas encore connus.

À chaque itération, nous allons raffiner nos ensembles de candidats en leur retirant tous les champs dont la position est déjà connue: s'il ne reste plus qu'un candidat, c'est que nous avons trouvé la position d'un nouveau champ, on l'ajoute à l'ensemble des champs connus.

Voici comment je l'ai implémenté:

```
89,179,173,167,157,127,163,113,137,109,151,131,97,149,107,83,79,139,59,53
```

Remarquez que je vérifie, à chaque itération, que de nouveaux champs ont bien été identifiés en me servant d'une variable `found` que je synchronise avec `known` en fin d'itération. En effet, si l'entrée du problème ne permettait pas de déduire toutes les règles sans ambiguïté, cette fonction serait susceptible de partir en boucle infinie.

Cela dit, si on en arrivait là, ce serait plutôt signe que l'une des fonctions précédentes est buggée, car si nous partions sciemment avec des données ambiguës, l'exercice appartiendrait alors à une toute autre classe de problèmes!

2. Jour 17 : Et si on testait les (futurs) generics de Go ?

L'énoncé du Jour 17 consiste à implémenter non pas une, mais *deux* chimères: il s'agit de généraliser le Jeu de la vie de Conway en **trois**, puis en **quatre** dimensions! Je crois que le

record de mon exo préféré du mois vient d'être battu. Vous allez bientôt comprendre pourquoi. 🍊

2.1. Une implémentation "éparse" du jeu de la vie

Avant de nous jeter tête baissée dans le code, remarquons qu'il s'agit du second automate cellulaire que nous devons implémenter ce mois-ci. Histoire de changer par rapport à celui du jour 11 , j'ai décidé d'implémenter un automate qui soit beaucoup plus facile à généraliser à un espace:

— virtuellement infini,
— de dimension arbitraire.

Pour cela, je vais contredire mes conclusions du Jour 15 , en modélisant le monde dans lequel évoluent nos cellules par *une* `map` plutôt qu'une *slice*. Mais il y a deux bonnes raisons à cela.

Dans le jeu de la vie, on peut engendrer certains *patterns* qui vont partir comme des projectiles et avancer indéfiniment dans la même direction. Pour cette raison, il est préférable de considérer que le monde de la simulation s'étend à l'infini dans toutes les directions. Et bien sûr, il arrive un moment où ça ne tient plus dans la mémoire de l'ordinateur, alors même que 99,999…999% des cases sont **vides**. Dans ces conditions, le meilleur moyen de ne pas saturer inutilement la mémoire est de ne garder en mémoire que les 0.000…001% de cellules vivantes.

? Tu as dit qu'il y avait deux raisons. C'est quoi la deuxième?

Parce que ça m'amuse! 🍊

2.1.1. Modélisation

Avant de me lancer dans les implémentations 3D et 4D du JDLV, commençons par une version beaucoup plus classique sur une grille en 2D.

```
89,179,173,167,157,127,163,113,137,109,151,131,97,149,107,83,79,139,59,53
```

Comme vous le voyez, une cellule est une simple paire de coordonnées, et je l'ai muni d'une méthode `IterNeighborhood` qui sert à itérer sur **la totalité** de son voisinage 8-connexe, y compris elle-même. Dans cette implémentation, je ne prends même pas la peine de vérifier les *ranges* des coordonnées. Si on arrive aux limites représentables par un `int`, le voisinage va simplement *wrapper* pour se retrouver téléporté à l'autre bout de l'univers, ce qui est parfaitement acceptable. Cela dit avant d'en arriver là sur des entiers à 64 bits, et même si nous pouvions calculer une génération *toutes les nanosecondes*, nous ne verrions pas ce genre de choses se produire avant 7000 ans (à moins de le faire exprès en initialisant la simulation aux bornes de l'univers)…

Bref, passons à la "grille", que je vais appeler `World2D` ici.

```
89,179,173,167,157,127,163,113,137,109,151,131,97,149,107,83,79,139,59,53
```

Pour mettre à jour le monde, on va itérer sur toutes les cellules vivantes pour incrémenter des compteurs sur tout leur voisinage. Ensuite, les compteurs vont nous servir à changer l'état des cellules:

— Si une cellule "morte" a précisément 3 voisins actifs, elle s'active.
— Si une cellule active a 2 ou 3 voisins actifs, son compteur arrivera à 3 ou 4, et elle pourra rester active.
— Autrement, si une cellule a moins de 3 cellules actives ou plus de 4 (toujours en se comptant elle-même), alors cette cellule meurt.

Plutôt simple, non?

2.2. Généraliser ce code en 3D et en 4D

Vous remarquerez certainement que la seule méthode que nous avons réellement besoin d'implémenter pour généraliser ce monde en 3D, c'est `IterNeighborhood`. En effet, c'est le seul code qui dépend réellement de la dimension du modèle. ***Ce serait quand même bien si on avait des generics en Go*** pour généraliser ce code sans avoir à tout recopier…

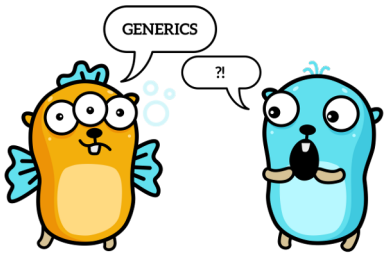

FIGURE 2.1. – Illustration par Maria Letta distribuée sous la Free Gophers Licence v1

Et ça tombe bien, car si vous vous souvenez de ce que je disais le 6e jour , il existe déjà une implémentation des generics avec laquelle on peut jouer en Go, et qui est (pour l'instant) prévue pour Go 1.17 en août prochain. C'est donc sur le go2go playground que ça va se passer.

Ce que nous voulons, c'est créer une implémentation de `World` qui accepte n'importe quel type de cellules. Nous n'aurions plus qu'à instancier un `World[Cell2D]` pour exécuter un JDLV en 2D, ou un `World[Cell3D]`, ou un `World[Cell4D]`…

Essayons déjà ceci pour commencer, et laissons-nous engueuler par le compilateur pour découvrir ce qu'il attend de nous:

```
89,179,173,167,157,127,163,113,137,109,151,131,97,149,107,83,79,139,59,53
```

Comme prévu, ça ne compile pas, et `go2go` va nous cracher les erreurs suivantes:

```
prog.go2:17:5: c.IterNeighborhood undefined (type bound for T has
    no method IterNeighborhood)
prog.go2:3:23: invalid map key type T (missing comparable
    constraint)
prog.go2:15:22: invalid map key type T (missing comparable
    constraint)
```

Le problème qui se produit à la ligne 3 et à la ligne 15, c'est que l'on ne peut pas utiliser n'importe quel type (`any`) comme clé d'un `map`. Le compilateur nous dit d'utiliser plutôt la contrainte `comparable` sur notre paramètre `T`. L'autre erreur (celle qui apparaît en premier) nous indique que *rien* ne nous garantit que le type `T` passé en paramètre disposera d'une méthode `IterNeighborhood` avec la bonne signature.

Qu'à cela ne tienne, il nous suffit de créer une *interface générique* `Cell[T]` pour décrire le contrat que nos cellules doivent respecter, et l'utiliser pour contraindre le paramètre de notre type générique `World`:


```
prog.go2:17:5: c.IterNeighborhood undefined (type bound for T has
    no method IterNeighborhood)
prog.go2:3:23: invalid map key type T (missing comparable
    constraint)
prog.go2:15:22: invalid map key type T (missing comparable
    constraint)
```


Et voilà, le compilo est satisfait! Il ne nous reste plus qu'à implémenter les cellules 3D et 4D, ce qui est trivial. 🍊

Vous pourrez voir ma solution générique complète à cet exo sur le go2go playground à cette adresse . Sinon, vous trouverez une implémentation sans generics sur mon dépôt gitlab .

2.3. Conclusion de cette expérience

Depuis environ un an que j'ai abandonné Python pour faire du Go quotidiennement et professionnellement, cet exercice a été la toute première occasion que j'aie croisée où les *generics* étaient vraiment justifiés. Je dois dire que leur prise en main m'a agréablement surpris: on est sur un comportement assez proche de ce qui se fait en Rust (comprendre par là que la généricité est contrainte par des *traits*, sauf que nous on appelle ça banalement des interfaces) tout en restant fidèle à la philosophie de Go qui veut que l'on n'a pas besoin de déclarer explicitement quand un type implémente une interface donnée.

Ce qui est *explicite*, par contre, ce sont les erreurs que m'a craché `go2go`: bien sûr, on ne peut pas généraliser sur une si petite expérience, mais je remarque qu'en laissant le compilateur me dire ce qu'il voulait, j'ai pu régler mon problème en deux coups de cuiller à pot, sans avoir besoin **ni de déchiffrer ces erreurs** (pensez à `gcc` quand vous faites une erreur en instanciant un template de templates de templates en C++), **ni de les googler** pour comprendre leur signification.

Simple, idiomatique, efficace.

À première vue, on dirait bien que ce mécanisme de *generics* a tout ce que l'on peut attendre de lui. 🍊

3. Jour 18 : Précédence d'opérateurs arithmétiques

Ce que j'aime bien avec l'Advent Of Code, c'est que chaque jour, je contredis ce que je racontais l'avant-veille. Hier, je détruisais ma conclusion d'il y a trois jours à propos de l'utilisation des `map` ou des *slices*… voyons voir ce que j'ai pu vous raconter avant-hier et que je vais devoir contredire aujourd'hui.

Ah, voilà, j'ai trouvé!

> Je vous ferai grâce du parsing des entrées dans le billet. D'ailleurs, à moins que l'on ne tombe d'ici Noël sur un format qui nécessite une grammaire (ce qui m'étonnerait quand même beaucoup), je pense ne pas trop risquer de me tromper en vous disant que je ferai toujours l'impasse dessus dorénavant.
>
> *Moi-même, avant-hier*

Et aujourd'hui, on implémente une grammaire , comme par hasard! 🍊

Nos entrées vont donc se présenter sous la forme d'expressions arithmétiques, comme celle-ci:

```
3 * 8 * 2 + 9 * ((6 + 5 * 3) * 7 * 9 * 7 * 7) * (7 * 4 + 5 + 8 * 8)
```

Le but est bien sûr de les évaluer, mais:

— Partie 1: en considérant que les opérateurs `+` et `*` ont **la même précédence**.
— Partie 2: en considérant que l'opérateur `+` a la priorité sur `*`.

C'est somme toute un exercice "assez banal" pour comprendre l'impact de la précédence des opérateurs sur la grammaire d'un langage.

3.1. Partie 1 : aucune règle de précédence

Pour commencer, on considère que les opérateurs ont tous la même précédence.

3.1.1. Ignorons les parenthèses

Si l'on fait le choix de commencer en ignorant les parenthèses, on peut implémenter une toute première fonction qui va évaluer nos expressions d'une seule traite:

```
3 * 8 * 2 + 9 * ((6 + 5 * 3) * 7 * 9 * 7 * 7) * (7 * 4 + 5 + 8 * 8)
```

Voilà. Ce n'est rien de vraiment sorcier, mais ça fait déjà une bonne base sur laquelle on peut itérer.

3.1.2. Évaluer des sous-expressions

Maintenant que cette évaluation sans parenthèses est en place, nous allons pouvoir réfléchir à comment gérer des "sous-expressions".

Pour ce faire, le plus simple est encore d'utiliser une approche récursive: dès que l'on tombe sur une parenthèse ouvrante, on appelle récursivement la fonction pour évaluer la sous-expression, puis on l'accumule dans le résultat.

La subtilité vient du fait que l'on va devoir garder un contrôle explicite sur la position du caractère que l'on est en train d'évaluer dans l'expression.

```
3 * 8 * 2 + 9 * ((6 + 5 * 3) * 7 * 9 * 7 * 7) * (7 * 4 + 5 + 8 * 8)
```

Ceci suffit à répondre à la partie 1. Ce qu'il faut retenir, c'est que **pour évaluer une sous-expression, on réalise un appel récursif**.

3.2. Partie 2 : une précédence à l'inverse des règles classiques

Dans la partie 2, on introduit à nouveau la précédence des opérateurs, mais en inversant les priorités par rapport à d'habitude: l'addition est prioritaire sur la multiplication. C'est là que le problème se corse, surtout si vous n'avez jamais implémenté de grammaire auparavant.

D'habitude, la façon dont on peut approcher ce problème est de considérer qu'une expression arithmétique sera toujours *une somme de produits*, sachant qu'un produit peut avoir dans ses termes une expression complète entre parenthèses (donc un terme peut être à son tour une somme de produits). Ici, avec les priorités inversées, on peut appliquer la même logique, et considérer qu'une expression sera toujours *un produit de sommes*.

Nous allons donc définir deux fonctions:

```
3 * 8 * 2 + 9 * ((6 + 5 * 3) * 7 * 9 * 7 * 7) * (7 * 4 + 5 + 8 * 8)
```

Prenons le temps de dérouler un exemple pour expliquer comment ça va marcher:

```
2 * (6 + 9 * 8 + 6) + 6
```

- Nous allons commencer notre évaluation par un appel à `evalProduct` (puisqu'une expression est un produit).
- `evalProduct` tombe sur `2` et délégue l'évaluation du premier terme à `evalSum` :
  - `evalSum` parse le `2` et l'accumule dans son résultat (`0 + 2 = 2`).
  - `evalSum` tombe sur le `*` (priorité plus faible que `+`), donc retourne `2` ainsi que la position **juste avant** le `*`.
- `evalProduct` accumule la valeur de retour dans son résultat (`1 * 2 = 2`).
- `evalProduct` tombe sur `*` (sa priorité à lui) et l'ignore.
- `evalProduct` tombe sur `(` et délègue son évaluation à `evalSum` :
  - `evalSum` tombe sur `(`, le consomme, et délègue l'évaluation de la sous-expression à `evalProduct` :
    - `evalProduct` tombe sur `6` et délègue l'évaluation du terme à `evalSum` :
      - `evalSum` parse le `6` et l'accumule dans son résultat (`0 + 6 = 6`)
      - `evalSum` tombe sur `+` (sa priorité à lui) et l'ignore
      - `evalSum` parse le `9` et l'accumule dans son résultat (`6 + 9 = 15`)
      - `evalSum` tombe sur `*` (priorité plus faible), et retourne son résultat (et la position d'avant)
    - `evalProduct` accumule la valeur de retour (`1 * 15 = 15`)
    - `evalProduct` tombe sur `*` (sa priorité à lui) et l'ignore
    - `evalProduct` tombe sur `8` et délègue l'évaluation du terme à `evalSum` :
      - `evalSum` parse le `8` et l'accumule dans son résultat (`0 + 8 = 8`)
      - `evalSum` tombe sur `+` (sa priorité à lui) et l'ignore
      - `evalSum` parse le `6` et l'accumule (`8 + 6 = 14`)
      - `evalSum` tombe sur `)` et retourne son résultat **ainsi que la position juste avant**‘
    - `evalProduct` accumule la valeur de retour (`15 * 14 = 210`)
    - `evalProduct` tombe sur `)` et retourne son résultat
  - `evalSum` accumule la valeur de retour (`0 + 210 = 210`)
  - `evalSum` tombe sur le `+` (sa priorité à lui) et l'ignore
  - `evalSum` parse le `6` et l'accumule (`210 + 6 = 216`)
  - fin de l'entrée, `evalSum` retourne son résultat
- `evalProduct` accumule le résultat (`2 * 216 = 432`)
- fin de l'entrée, `evalProduct` retourne le résultat.

Cet exemple a le mérite de contenir *tous les cas de figure*. On peut donc s'en servir pour implémenter ces deux fonctions (mais n'oubliez pas d'écrire des tests avant, c'est important). 🍊

```
2 * (6 + 9 * 8 + 6) + 6
```

Et voilà le travail.

J'aimerais pouvoir dire que ce n'était "pas difficile". La vérité, c'est que ce genre de problème **est compliqué** quand on y touche pour la première fois, et cet exercice m'a rappelé les nombreuses heures que j'avais passées à implémenter des parseurs avec le *Dragon Book* sur les genoux pour comprendre comment ça marche il y a plusieurs années. Bref, non, ce n'était pas un exercice facile et ce n'est pas non plus un domaine facile.

De plus, ici, *je pars du principe que les expressions sont bien formées*, et notamment que les parenthèses sont bien équilibrées:

— Si une parenthèse ouvrante n'est jamais fermée, on agit comme si la fermeture était impliquée à la fin.
— Si on croise une parenthèse fermante non équilibrée avant la fin de l'expression, on court-circuite l'évaluation du reste.

Pire encore: dans cette implémentation, le symbole `+` est optionnel, et `8 9` sera automatiquement évalué comme `8 + 9` puisque `+` est l'opérateur de priorité maximale…

Mais ces considérations dépassent le cadre de cet exercice: si je me lançais dans un tutoriel pour écrire une grammaire robuste qui valide ses entrées, il me faudrait bien plus de place que celle que je m'autorise dans ces billets déjà bien denses…

4. Jour 19 : Et encore une grammaire !

Décidément, moi qui pensais qu'on ne toucherait pas à des grammaires ce mois-ci, j'étais totalement à côté de la plaque! 🍊

Aujourd'hui , le but du jeu était d'implémenter un algorithme de *matching* pour un langage **non régulier**, autrement dit, pour un langage qui *ne peut pas s'exprimer* comme une expression régulière.

L'entrée nous fournit une *grammaire* comme celle-ci:

```
0: 1 2
1: "a"
2: 1 3 | 3 1
3: "b"
```

Posons un peu de vocabulaire:

— Cette grammaire est composée de quatre **règles** (*rules*).
— Les règles 1 et 3 matchent un caractère précis, on dit qu'elles sont **terminales**.
— Les règles 0 et 2 sont dites **non-terminales**.
— L'expression `1 2` (dans la règle 0) signifie "matche 1 suivi de 2".
— La règle 2 est composée de deux **séquences** alternatives, il faut la lire comme: "matche 1 puis 3, ou bien 3 puis 1".

Ce qui va différencier les parties 1 et 2, c'est que dans la première partie, la grammaire ne sera pas *récursive* (il n'y a pas de cycle dans le graphe formé par les règles), alors que dans la seconde partie, nous avons deux règles explicitement récursives. Si je dis plus haut que ce langage est **non régulier**, c'est parce que l'une des règles récursives de la seconde partie s'exprime comme ceci:

```
11: 42 31 | 42 11 31
```

Autrement dit: ”La règle 42 répétée un certain nombre (N) de fois, suivi de 31 répété précisément N fois”. Aucun opérateur des expressions régulières ne permet de faire une telle chose (sinon, ça voudrait dire que l'on pourrait vérifier l'équilibrage des parenthèses au moyen d'une regexp). Je vous ferai grâce de la théorie qui vient derrière, mais on peut en conclure que *le langage ainsi formé ne peut pas être représenté par un automate fini* à cause de ce détail.

Cependant, ce ”détail” ne m'a pas empêché d'appliquer un algorithme bien connu d'évaluation d'automates non-déterministes, que j'ai mentionné le deuxième jour , je veux bien sûr parler de *l'algorithme de Thompson*. Bien que cette solution soit assez bourrine dans le cadre de cet exo (il existe des façons plus simples de le résoudre), cet algorithme a le mérite d'être vraiment robuste.

4.1. Modélisation du problème

Pour représenter la grammaire, j'ai opté pour la modélisation suivante:

```
11: 42 31 | 42 11 31
```

Autrement dit, la grammaire suivante:

```
0: 1 2
1: "a"
2: 1 3 | 3 1
3: "b"
```

Sera modélisée comme ceci:

```
0: 1 2
1: "a"
2: 1 3 | 3 1
3: "b"
```

C'est une modélisation plutôt directe.

4.2. Principe de l'algorithme de Thompson

L'idée de l'algorithme de Thompson, c'est de ne parcourir la chaîne d'entrée *qu'une fois et une seule*: si plusieurs chemins sont possibles, alors on maintient autant de ”*threads*” simultanés qu'il existe de chemins valables. À chaque fois que l'on consomme un caractère de l'entrée, on fait avancer tous les threads d'un cran, on supprime ceux qui échouent et on en crée de nouveaux si certains d'entre eux arrivent à une nouvelle bifurcation. Quand on a fini de parcourir la chaîne d'entrée, si un des threads vient de se terminer sur un *match*, alors la chaîne est validée.

Un exemple vaudra mieux qu'un long discours. Imaginons que nous voulions évaluer la chaîne `aba` avec notre grammaire d'exemple.

La première chose à faire, ça va être d'initialiser nos threads:

```
Entrée: .a b a
Threads:
           T0
[Rule: 0, Seq: 0, Sub: 0]
```

Ensuite, nous allons "étendre" (ou "dérouler) nos threads, c'est-à-dire que l'on va référencer les règles qu'ils pointent jusqu'à savoir quel est le prochain caractère qu'ils doivent matcher.

Typiquement, la sous-règle `0` de la séquence `0` de la règle `0` est "1", et cette règle est un terminal (`"a"`). Notre thread va donc être déroulé comme ceci:

```
           T0
[Rule: 1 (terminal: 'a')]
[Rule: 0, Seq: 0, Sub: 0]
```

Maintenant, il est temps de consommer le premier caractère de l'entrée (`a`). Nous voyons bien que l'actuel thread 0 coïncide avec cette entrée, donc nous allons le faire avancer d'un cran, c'est-à-dire dépiler la règle terminale et fair avancer la *frame* d'en-dessous, vers la sous-règle suivante:

```
Entrée: a.b a
Threads:
           T0
[Rule: 0, Seq: 0, Sub: 1]
```

Avant de continuer, déroulons à nouveau nos threads : la sous-règle `1` de la règle `0` est la règle `2`. Celle-ci est composée de deux séquences alternatives. **C'est là que tout se joue**.

Pour gérer ces deux séquences alternatives, nous allons *forker* (cloner) le thread `T0`, de manière à avoir un thread par séquence de la règle 2 :

```
           T0              |              T1
[Rule: 2, Seq: 0, Sub: 0] | [Rule: 2, Seq: 1, Sub: 0]
[Rule: 0, Seq: 0, Sub: 1] | [Rule: 0, Seq: 0, Sub: 1]
```

Continuons l'extension: la première séquence de la règle `2` est `1 3`, alors que la seconde séquence est `3 1`.

```
           T0             |              T1
[Rule: 1 (terminal: 'a')] | [Rule: 3 (terminal: 'b')]
[Rule: 2, Seq: 0, Sub: 0] | [Rule: 2, Seq: 1, Sub: 0]
[Rule: 0, Seq: 0, Sub: 1] | [Rule: 0, Seq: 0, Sub: 1]
```

Il est temps de consommer le second caractère de l'entrée: le `b`. De toute évidence, le thread `T0` ne va pas matcher, alors que `T1`, oui. Donc nous pouvons détruire le thread `T0` et faire avancer `T1` d'un cran :

```
Entrée: a b.a
Threads:
            T1
[Rule: 2, Seq: 1, Sub: 1]
[Rule: 0, Seq: 0, Sub: 1]
```

Allez, plus qu'une itération. Commençons par étendre nos threads, la deuxième sous-règle de la deuxième séquence de la règle `2` pointe sur la règle `1` (le terminal `a`) :

```
            T1
[Rule: 1 (terminal: 'a')]
[Rule: 2, Seq: 1, Sub: 1]
[Rule: 0, Seq: 0, Sub: 1]
```

Consommons le caractère d'entrée (`a`, comme par hasard !). Ça matche! Cette fois, quand nous allons faire avancer T1 d'un cran :

— On va dépiler la règle 1 qui vient de matcher.
— On va vouloir faire avancer la règle 2 sous-jacente, sauf que cette séquence n'a plus de sous-règles, donc elle a fini de matcher également, donc on la dépile.
— On va vouloir faire avancer la règle 0 sous-jacente, sauf que cette séquence non plus n'a plus de sous-règle, donc elle a fini de matcher.

Nous nous retrouvons avec une pile vide dans `T1`, caractéristique de **la fin** de son exécution.

```
            T1
         <terminé>
```

Nous n'avons plus de caractère à consommer dans l'entrée, et nous avons justement sous la main un thread qui vient de se terminer avec succès: on en conclut que la chaîne d'entrée respecte bien notre grammaire.

4.3. Implémentation

4.3.1. Modélisation des threads

Voici comment sont modélisés mes `Thread` :

```
            T1
          <terminé>
```

Je ne vous détaillerai pas toutes les méthodes du type `Thread`, vous pourrez les lire ici . Je vais plutôt me contenter d'en résumer l'interface publique:

- `NewThread()` initialise un thread à l'état `0`, `0`, `0`.
- `thread.Clone()` retourne une copie du `Thread`.
- `thread.Frame()` retourne l'état courant du thread (la *frame* en haut de la pile).
- `thread.Set(frame)` empile la *frame* sur le thread et retourne la *frame* empilée.
- `thread.Done()` retourne `true` si le thread a fini de s'exécuter (i.e. si sa pile est vide).
- `thread.Match(char, grammar)` vérifie si le thread pointe sur un terminal égal à `c` et avance d'un pas si c'est le cas. Cette méthode retourne `true` en cas de succès.

4.3.2. L'algorithme général de matching

Implémentons maintnant l'algorithme tel que nous venons de le décrire plus haut: à chaque itération, on étend les threads, puis on les `Match()` sur le caractère d'entrée (ce qui les fait avancer d'un pas). Si on tombe à court de threads, alors le match a échoué. Lorsque l'on arrive à la fin de la chaîne d'entrée, on retourne `true` si au moins un des threads a terminé son exécution.

```
            T1
          <terminé>
```

Comme vous le voyez, j'ai délégué l'extension et l'avancement d'une case à deux fonctions séparées. Les voici:

```
            T1
          <terminé>
```

Ce code fait précisément ce que je décrivais plus haut, et suffit à répondre aux deux parties de l'exercice.

Finalement, cette implémentation était assez verbeuse, mais je pense qu'elle décrit on ne peut plus explicitement le principe de cet algorithme. Dans le cadre de cet exercice, cela revient à allouer dynamiquement beaucoup d'objets contrairement à ce qui se passe dans un automate fini (où l'état d'un thread peut être totalement modélisé par un unique entier), donc cette solution

revient un peu à sortir le lance-roquette. Mais bon, ça valait le coup d'essayer pour constater que ça fonctionne.

5. Jour 20 : Reconstruire un puzzle

L'exo d'aujourd'hui était particulièrement long!

On nous donnait un ensemble de 144 "pièces" de 10x10 comme celle-ci:

```
Tile 1321:
.###.#.#..
#####.....
.....#..#.
#..##..#.#
...#..#..#
##........
.#.#....#.
#...##....
#.#.#..#..
#.#..#....
```

On nous explique ensuite que ces *tiles* sont les pièces d'un puzzle (de 12x12) à reconstruire, avec les contraintes suivantes:

- Les *tiles* peuvent avoir été tournées,
- Les *tiles* peuvent avoir été renversés horizontalement ou verticalement.
- Les bords de chaque *tile* sont faits de telle manière à ce leur reconstruction ne soit pas ambiguë: les pièces du bord de l'image ont des bords *uniques* (même renversés), et deux pièces ont un bord en commun (modulo renversement) si et seulement si ce sont deux pièces adjacentes.

Dans ce billet, je vais passer rapidement sur beaucoup d'étapes intermédiaires (plus laborieuses que difficiles à coder) et ne vais détailler que les étapes importantes de l'exercice.

5.1. Partie 1 : Commencer par les coins !

La première question nous demande de retrouver les *tiles* qui constituent les 4 coins de l'image, exactement comme quand on reconstruit un puzzle dans la réalité. Pour ce faire, commençons déjà par modéliser notre problème:

```
Tile 1321:
.###.#.#..
#####.....
.....#..#.
#..##..#.#
```

```
...#..#..#
##........
.#.#....#.
#...##....
#.#.#..#..
#.#..#....
```

Comme vous le voyez, j'ai distingué le type `Image` du type `Tile`, parce que j'anticipe le fait que nous allons avoir besoin d'implémenter des méthodes de transformation d'images qui n'ont pas grand chose à voir avec la manipulation des pièces du puzzle.

En revanche, dans le type `Tile`, en plus de l'identifiant unique de la pièce et de l'image associée, j'associe un tableau de 8 `Border`. C'est-à-dire un tableau qui décrit le motif formé par chacun des 4 bords de l'image, ainsi que leur inversion. Pour modéliser efficacement ces "bords", j'ai simplement décidé que ceux-ci seraient des nombres entiers non-signés sur 10 bits, chaque bit représentant un pixel de la bordure (ce qui fait 1024 possibilités en tout).

Enfin, vous remarquerez que ce tableau `Borders` est rempli en suivant un ordre précis, de manière à identifier plus facilement ces bords par la suite.

5.1.1. Répondre à la question

Pour répondre à la première question, j'ai suivi le raisonnement suivant:

— Toutes les pièces se trouvant au bord du puzzle ont forcément au moins une bordure unique (2 en comptant son renversement).
— Les pièces aux quatre coin possèdent deux bordures uniques (4 en comptant leurs renversements).

Pour commencer, je vais donc stocker tous mes `Tile` dans un `map` indexé par le `TileID`:

```
Tile 1321:
.###.#.#..
#####.....
.....#..#.
#..##..#.#
...#..#..#
##........
.#.#....#.
#...##....
#.#.#..#..
#.#..#....
```

C'est cette structure que je crée en premier, en parsant mes entrées. À partir de là, je vais créer un tableau pour indexer mes pièces selon leurs bords.

```
Tile 1321:
.###.#.#..
#####.....
.....#..#.
#..##..#.#
...#..#..#
##........
.#.#....#.
#...##....
#.#.#..#..
#.#..#....
```

Ainsi, il sera facile, plus tard, de retrouver les pièces du puzzle dont la bordure coïncide avec une pièce que nous venons de poser. Pour le moment, nous allons nous servir de cette structure pour retrouver les coins du puzzle (les pièces dont deux bordures sont uniques):

```
Tile 1321:
.###.#.#..
#####.....
.....#..#.
#..##..#.#
...#..#..#
##........
.#.#....#.
#...##....
#.#.#..#..
#.#..#....
```

Et ceci suffit à répondre à la première question: il n'y a qu'à multiplier les quatre `TileID` correspondants entre eux.

5.2. Partie 2 : Reconstruire le puzzle et rechercher des monstres marins

5.2.1. Transformations d'images

Avant de procéder à la reconstruction elle-même, nous allons commencer par implémenter les transformations d'images dont nous allons avoir besoin. J'ai codées celles-ci dans le fichier `image.go` . Je vais simplement les résumer ici.

```
FlipH (renversement horizontal):
  1 2 3      3 2 1
  4 5 6  ->  6 5 4
  7 8 9      9 8 7
```

```
5
6  FlipV (renversement vertical):
7    1 2 3      7 8 9
8    4 5 6  ->  4 5 6
9    7 8 9      1 2 3
10
11 FlipD (renversement diagonal):
12   1 2 3      1 4 7
13   4 5 6  ->  2 5 8
14   7 8 9      3 6 9
15
16 RotateRight (rotation à droite = FlipD + FlipH):
17   1 2 3      7 4 1
18   4 5 6  ->  8 5 2
19   7 8 9      9 6 3
20
21 RotateLeft (rotation à gauche = FlipD + FlipV):
22   1 2 3      3 6 9
23   4 5 6  ->  2 5 8
24   7 8 9      1 4 7
```

5.2.2. Remettre une pièce dans le bon sens

Pour reconstruire notre image, nous allons procéder de façon simple:

- On va d'abord prendre un coin que l'on va placer en haut à gauche.
- On va ensuite compléter la première ligne, en trouvant à chaque fois la pièce dont un bord coïncide avec le bord droit de la dernière pièce posée, et en la transformant pour que ce bord soit à sa gauche.
- Pour commencer la ligne suivante, on va chercher une pièce dont un bord coïncide avec le bord du bas de la pièce au-dessus, puis la transformer de façon à ce que ce bord se retrouve en haut.

Nous avons donc besoin de 3 méthodes:

- Une pour placer le coin "unique" d'une pièce "du coin" en haut à gauche (pour démarrer le puzzle),
- Une pour replacer un bord arbitraire d'une pièce à sa gauche,
- Une pour replace un bord arbitraire d'une pièce en haut.

Je vous montre juste la méthode `ToLeftBorder` ici:

```
1  FlipH (renversement horizontal):
2    1 2 3      3 2 1
3    4 5 6  ->  6 5 4
4    7 8 9      9 8 7
5
```

```
FlipV (renversement vertical):
  1 2 3      7 8 9
  4 5 6  ->  4 5 6
  7 8 9      1 2 3

FlipD (renversement diagonal):
  1 2 3      1 4 7
  4 5 6  ->  2 5 8
  7 8 9      3 6 9

RotateRight (rotation à droite = FlipD + FlipH):
  1 2 3      7 4 1
  4 5 6  ->  8 5 2
  7 8 9      9 6 3

RotateLeft (rotation à gauche = FlipD + FlipV):
  1 2 3      3 6 9
  4 5 6  ->  2 5 8
  7 8 9      1 4 7
```

Autrement dit, on trouve dans quel sens notre pièce est tournée (où se trouve le bord que nous voulons remettre à gauche), puis en fonction de cette information, on transforme l'image et on recalcule les bords de la pièce de manière à pouvoir retrouver la suivante.

Les deux autres méthodes sont du même acabit, je vous laisse consulter le fichier `tile.go` si cela vous intéresse.

5.2.3. Reconstruire l'image

Pour reconstruire l'image, nous allons nous y prendre en deux temps:

— D'abord, il faut réordonner et transformer les pièces pour qu'elles se présentent toutes dans le bon sens.
— Ensuite, on recopie leur contenu *en supprimant les bords de chaque pièce* dans une nouvelle image carrée.

La fonction `reorder()` réalise exactement l'opération que je vous ai décrite plus haut:

```
FlipH (renversement horizontal):
  1 2 3      3 2 1
  4 5 6  ->  6 5 4
  7 8 9      9 8 7

FlipV (renversement vertical):
  1 2 3      7 8 9
  4 5 6  ->  4 5 6
  7 8 9      1 2 3

```

```
FlipD (renversement diagonal):
  1 2 3      1 4 7
  4 5 6  ->  2 5 8
  7 8 9      3 6 9

RotateRight (rotation à droite = FlipD + FlipH):
  1 2 3      7 4 1
  4 5 6  ->  8 5 2
  7 8 9      9 6 3

RotateLeft (rotation à gauche = FlipD + FlipV):
  1 2 3      3 6 9
  4 5 6  ->  2 5 8
  7 8 9      1 4 7
```

La création de l'image elle-même revient juste à recopier des données, vous la trouverez implémentée dans la fonction `assemble()` .

5.2.4. Trouver les monstres marins

Le but ultime de l'exercice est de compter le nombre de monstres marins que l'on peut trouver dans l'image. Un monstre correspond au *pattern* suivant:

```
                  #
#    ##    ##    ###
 #  #  #  #  #  #
```

L'énoncé nous précise que tous les monstres de l'image sont tournés dans le même sens, mais bien sûr, l'image que nous avons reconstruite *peut très bien* ne pas être dans le bon sens. Autrement dit, il faut essayer de trouver ce *pattern* dans l'image en la retournant dans tous les sens, jusqu'à ce que l'on trouve plus de 0 monstre dedans.

J'ai implémenté ma recherche comme ceci:

```
                  #
#    ##    ##    ###
 #  #  #  #  #  #
```

Pour trouver qu'il y avait finalement 15 monstres dans mes données d'entrée. 🍊

Finalement, cet exercice était plutôt rigolo, mais heureusement qu'on est dimanche, parce qu'il m'aura pris un paquet de temps!

Pour résumer ces 5 jours, je dirais qu'on est largement sortis du cadre des gentils exos d'algorithmique.

Retrouvait face à des situations non triviales où l'on devait écrire du code assez spécialisé (du *parsing*, un moteur de *matching*, du traitement d'image…). Au final, chacun de ces exercices contribuait à améliorer notre culture informatique générale, même si ça commence à devenir franchement laborieux à implémenter proprement. 🍊

Allez, encore 5 jours et ça sera fini. Restons motivés!