Université de Liège
Faculté des Sciences appliquées

Projet : Solveur d'automates

INFO0054 : Programmation Fonctionnelle

Auteurs :
Samuel Charlier s203252
Manon Gerard s201354
Christopher Ligotti s221009

Professeur :
C. Debruyne

Année académique 2024 - 2025

1 Introduction

Dans le cadre du cours de programmation fonctionnelle à l'université de Liège (INFO0054), il nous a été demandé de développer un solveur générique pour les automates finis déterministes (AFD). Pour cela, nous avons décidé de répartir la résolution du problème en plusieurs fichiers :

- `AFD.scala`
- `Somme4.scala`
- `ChainesBinairesImpaires.scala`
- `LoupMoutonChou.scala`
- `Taquin.scala`

Le premier fichier sert à la création d'AFD ainsi qu'à la manipulation de ceux-ci. Tandis que les autres résolvent les différentes situations mises en avant dans l'énoncé.
Pour ce qui est de la compréhension et de l'exécution rapide du code, nous avons respectivement :

- `README.md`
- `Makefile`

Le fichier `README.md` fournit des explications complémentaires sur l'utilisation et le but des différents fichiers exécutables.

2 Choix d'implémentation et d'ADT

Dans cette section, nous nous concentrerons sur nos choix d'implémentations des fonctions. Pour savoir ce que font les fonctions, il faut se référer aux spécifications des fonctions écrites dans le code. Ces spécifications ont été faites en suivant le guide de style Scaladoc.

2.1 La fonction accept

Dans le fichier `AFD.scala` qui définit la classe AFD, nous avons la méthode `solve` qui prend en entrée un mot de type générique B et renvoie un booléen. Un mot est un synonyme de type pour une liste de type B.
Son implémentation fonctionne de la manière suivante, on parcourt le mot de gauche à droite via la fonction `foldLeft` en partant d'un `Option` de l'état initial `Some(s)`. Ensuite, on réalise pour chaque symbole dans le mot via la paire état et symbole, la détermination du prochain état via la fonction de transition par un `flatmap` car chaque état est contenu dans un `Option`. Finalement, via la fonction `contains`, on regarde si l'état final obtenu est contenu dans "les états accepteurs de l'AFD" (F). Ceci nous renverra `false` si l'état final ne fait pas partie de F ou si le mot mène à un état inexistant sinon on aura `true`.
Via cette implémentation, on ne parcourt le mot qu'une seule fois.

2.2 La fonction solve

Dans le fichier `AFD.scala`, nous avons implémenté la méthode `solve`, qui est une fonction générale permettant de trouver tous les mots sans cycles menant à des états accepteurs. Cette méthode accepte une heuristique comme paramètre optionnel. Si aucune heuristique n'est fournie, une heuristique constante (`_ => 0`) est utilisée par défaut, ce qui garantit que tous les états sont explorés sans priorisation particulière.

Définition et fonctionnement L'implémentation de `solve` repose sur une fonction interne récursive, `recherche`, qui explore les chemins possibles à l'aide d'une structure de type `File`. Dans le code, `File` est un synonyme de type représentant une liste triée contenant les informations suivantes :

- **L'état actuel** (`etatActuel`).
- **Le chemin actuel** (`chemin`), représenté dans l'ordre inversé pour permettre des ajouts rapides.
- **Le coût heuristique** (`heuristique(etatActuel)`).
- **Les états visités** (`visites`), utilisés pour éviter les cycles.

La méthode suit les étapes suivantes :

- **Initialisation** : Lors de l'appel de `solve`, une file est initialisée avec l'état initial (`s`), un chemin vide, un coût heuristique nul, et un ensemble vide d'états visités.
- **Exploration des états** : À chaque itération, nous extrayons le premier élément de la file et calculons ses voisins atteignables via la méthode `adjacence`. Ces voisins sont ajoutés à la file si l'état n'a pas encore été visité.

  Rôle de la méthode adjacence Dans notre implémentation, la méthode `adjacence` est utilisée pour calculer tous les voisins atteignables depuis un état donné. Elle parcourt l'ensemble de l'alphabet (`sigma`) et utilise la fonction de transition (`delta`) pour déterminer les états accessibles avec chaque symbole. Le résultat est un ensemble de paires `(symbole, état)`, représentant toutes les transitions possibles à partir de l'état actuel. Cette méthode nous permet de séparer la logique de calcul des voisins, rendant le code plus lisible et modulaire. Elle est essentielle pour explorer efficacement les chemins dans l'automate
- **Ajout aux solutions** : Si l'état courant est un état accepteur (`F.contains(etatActuel)`), le chemin correspondant est ajouté à la liste des solutions après avoir été inversé (`chemin.reverse`).
- **Mise à jour de la file** : La file est triée après chaque mise à jour pour maintenir l'ordre basé sur le coût heuristique.
- **Terminaison** : Lorsque la file est vide, cela signifie que tous les chemins possibles ont été explorés, et nous retournons la liste des solutions.

Utilisation de foldLeft Pour gérer les voisins de l'état courant, nous utilisons `foldLeft` sur la liste des voisins. Cette fonction permet de parcourir chaque voisin et de mettre à jour la file accumulée à chaque étape. Par exemple, pour chaque voisin, nous vérifions s'il n'a pas encore été visité. Si ce n'est pas le cas, nous ajoutons cet état à la file avec le chemin mis à jour et son coût heuristique. Le code correspondant est le suivant :

```
val nouvelleFile = adjacence(etatActuel).foldLeft(reste) {
    case (acc, (symbole, etatAdjacent))
        if !visites.contains(etatAdjacent) =>
        (etatAdjacent, symbole :: chemin,
         heuristique(etatAdjacent),
         visites + etatActuel) :: acc
    case (acc, _) => acc
}.sortBy(_._3)
```

Cette approche permet de construire la nouvelle file tout en maintenant une structure immuable et en respectant les principes de la programmation fonctionnelle.

Justification de l'implémentation Dans notre implémentation, nous avons choisi de ne pas utiliser directement la fonction `accept` pour vérifier si un chemin mène à un état accepteur. En effet, nous avons déjà accès à l'état final du chemin à chaque itération, ce qui nous permet de vérifier directement si cet état est dans l'ensemble `F`. Si nous avions utilisé `accept`, cela aurait été moins efficace, car il aurait fallu reconstruire le chemin dans l'ordre correct pour le parcourir. Si nécessaire, nous aurions pu appeler `accept(chemin.reverse)` pour vérifier cette condition, mais cela n'est pas nécessaire ici.

Exemple d'utilisation Nous avons testé cette fonction avec un automate simple qui accepte les mots contenant un nombre impair de `1`. Voici comment cet automate est défini :

- **Alphabet** : $\Sigma = \{0, 1\}$.
- **États** : $\{pair, impair\}$, où `pair` représente un nombre pair de `1` et `impair`, un nombre impair.
- **Transitions** :
  — $\delta(pair, 0) = pair$, $\delta(pair, 1) = impair$.
  — $\delta(impair, 0) = impair$, $\delta(impair, 1) = pair$.
- **État initial** : `pair`.
- **États accepteurs** : $\{impair\}$.

Lors de l'exécution de `solve` sur cet automate, la fonction retourne les solutions suivantes : `[[1], [0, 1, 1]]`. Ces mots mènent tous à l'état accepteur `impair`. Cette implémentation garantit une exploration efficace grâce à l'utilisation de structures immuables (`Set` pour les états visités et `::` pour construire les chemins) et à la gestion ordonnée de la file.

Efficacité et modularité L'implémentation de `solve` est une fonction récursive terminale, ce qui signifie que Scala est capable de transformer cette récursion en une boucle itérative sous-jacente. Cela garantit une utilisation optimisée de la mémoire et prévient les dépassements de pile lors de l'exploration de grands automates. En outre, l'utilisation de paramètres par défaut pour l'heuristique rend cette méthode flexible et facilement adaptable à des besoins spécifiques, comme l'exploration avec des heuristiques différentes.

2.3 La fonction lazysolve - évaluation non stricte

Dans le fichier `AFD.scala`, nous avons implémenté la méthode `lazysolve`, qui est une version paresseuse de la fonction `solve`. Cette méthode partage les mêmes principes fondamentaux que `solve`, décrits dans la section précédente, mais elle retourne une `LazyList`, permettant une génération paresseuse des solutions.

Lien avec solve L'implémentation de `lazysolve` repose sur la même structure et suit les mêmes étapes générales que `solve`, notamment l'utilisation de la méthode `adjacence`, d'une file (`File`) pour gérer les états à explorer, et d'un ensemble des états visités pour éviter les cycles. La principale différence réside dans le type de résultat produit :

- `solve` retourne une liste complète contenant tous les mots acceptés par l'automate, générés immédiatement.
- `lazysolve` retourne une `LazyList`, où les mots ne sont calculés qu'au moment où ils sont explicitement demandés.

Évaluation paresseuse Pour garantir l'évaluation non stricte, nous avons remplacé la construction classique des listes (avec `::`) par l'utilisation de l'opérateur `#::`. Cela permet d'ajouter un élément à la `LazyList` sans évaluer immédiatement les éléments suivants. De plus, cette approche impose que la fonction interne `recherche` ne soit pas récursive terminale, car cela forcerait l'évaluation complète avant de retourner la `LazyList`, ce qui contredirait le principe d'évaluation paresseuse.

Pourquoi utiliser lazysolve ? Nous avons choisi une évaluation paresseuse pour répondre à des besoins spécifiques :

- **Exploration partielle** : Avec `lazysolve`, seuls les mots nécessaires sont calculés. Cela est particulièrement utile lorsque seule une partie des solutions est requise.
- **Optimisation mémoire** : Contrairement à `solve`, qui génère et stocke tous les mots, `lazysolve` économise de la mémoire en ne stockant que les éléments déjà consommés.
- **Automates complexes ou infinis** : Dans des cas où l'espace des solutions est très large ou infini, une génération immédiate serait inefficace ou impossible. L'évaluation paresseuse permet de gérer ces situations de manière progressive.

Exemple d'utilisation L'exemple d'automate décrit dans la section `solve` (acceptant les mots contenant un nombre pair de `1`) peut également être utilisé pour illustrer `lazysolve`. La différence principale réside dans la façon dont les solutions sont générées :

```
val solutions = afd.lazysolve()
println(solutions.take(3).toList) // Génère uniquement les 3 premiers mots
```

Dans cet exemple, seuls les mots `[[0], [0, 0], [1, 1]]` sont calculés, et les suivants ne seront générés que si nécessaire.

Conclusion La méthode `lazysolve` est une alternative efficace et économe en mémoire à `solve`. Elle s'appuie sur une logique similaire, tout en offrant une flexibilité accrue grâce à l'évaluation paresseuse. Cela la rend particulièrement adaptée pour des explorations partielles ou pour des automates complexes où la génération immédiate des solutions serait coûteuse.

2.4 Chaînes binaires impaires

Comme stipulé dans l'énoncé, la représentation de la chaîne prise dans notre AFD sera soit paire ou impaire pour cela, on réalise un `sealed trait` que l'on nomme `EtatBinaire`. Celui-ci est étendu en deux cas : `Pair` et `Impair`.
Suite à cela, on aura besoin de définir les différentes caractéristiques de notre AFD :

- une condition initiale qui est l'état paire
- la représentation de l'alphabet, vu qu'on aura une chaine binaire par soit un 0 ou soit un 1
- la fonction de transition δ est défini par la fonction `deltaBinaire` qui prend en entrée un tuple composé d'un état et d'une valeur binaire de notre chaîne à analyser pour renvoyer un `Option` du nouvel état calculé. Ceci se fait de la manière suivante : si on croise un '0', l'état reste inchangé, si on croise un '1', l'état change vers l'autre état et si tout autre cas est rencontré, on renvoie un `None`
- pour ce qui est de l'état accepteur, on met un ensemble contenant simplement notre état impair

Pour ce qui est des résultats demandés dans l'énoncé suite à l'exécution de notre code on obtient ceci :

```
ex.solve() -> List(List(1))
ex.lazysolve().take(4).toList -> List(List(1))
ex.accept(List(1, 0, 1, 0, 1, 0, 1, 1)) -> true
ex.accept(List(1, 0, 1, 0, 1, 0, 1, 0)) -> false
```

Quant à la question : "L'utilisation de la fonction `solve` doit renvoyer tous les mots sans cycles. Combien de mots de ce type doivent être renvoyés ?"
Cela dépend de la définition de notre condition initiale. Pour un problème tel que notre solveur, on aura toujours une liste contenant une liste avec un 1, car cela fera directement passer notre état invalide (Pair) vers un état accepteur (Impair). Cependant, si nous décidions que notre condition de départ serait impaire pour une extension de notre problème, i.e. si on veut savoir si, suite à une liste impaire, on y ajoute une autre liste X aurait-on état accepteur ? Le `solve` nous donnerait alors une liste contenant une liste vide car l'état serait un état accepteur est donc il n'aurait rien à rajouter.

2.5 Problème du loup, du mouton et du chou

Pour la création d'un AFD résolvant ce puzzle, on va poser comme représentation des états un tuple de 2 ensembles pour mettre en évidence les 2 côtés de la rive. Avec chacune des rives possédant un ensemble de `sealed trait` que l'on nomme `Intervenant` qui s'étend en ces différents objets : "P" pour le passeur, "L" pour loup, "C" pour le chou et "M" pour le mouton.

Lors de la création de ce dit AFD, on passe les arguments suivant afin d'être capable de résoudre le problème :

- un état initial plaçant l'entièreté des intervenants dans le premier ensemble (laissant le deuxième ensemble vide) afin de représenter le début du puzzle où tous les intervenants sont sur la rive gauche
- l'alphabet qui correspond aux caractères "p", "l", "m" et "c" représentant l'éventuel intervenant avec qui le passeur traverse
- la fonction de transition δ correspond à notre fonction `deltaLoupMoutonChou`. Cette fonction en dépend d'une autre qui est `estInterdit` qui indique si il s'agit d'un état où un intervenant en mange un autre. Pour en revenir à la fonction δ, on regarde que le passeur et l'éventuel intervenant qui viendrait avec lui sont contenu du même côté. Si c'est le cas et que nous n'étions pas dans un état interdit, les intervenants changent de rives. Sinon, `None` est retourné.
- l'état accepteur qui représente l'état final du problème où tous les intervenants se situent dans le deuxième ensemble (laissant le premier ensemble vide) représentant la rive droite

Suite à notre implémentation, on exécute le code et nous obtenons les résultats suivants pour ce qui est demandé dans l'énoncé

```
ex.solve() -> List(List(m, p, c, m, l, p, m), List(m, p, l, m, c, p, m))
ex.lazysolve().take(2).toList -> List(List(m, p, l, m, c, p, m), List(m, p, c, m, l, p, m))
ex.accept(List(m, p, c, m, l, c, m, l, c, m, l, p, m)) -> true
ex.accept(List(p, p, p)) -> false
```

2.6 Le taquin

Pour représenter les cases du taquin, nous avons défini un ADT `Piece` qui contient deux constructeurs `Trou` et `Nombre` qui prend comme valeur un nombre naturel.

Pour implémenter le taquin, nous avons créé une `case class Taquin` qui prend en entrée une liste de liste de Piece qui représente la grille du taquin. Nous avons opter pour une case class puisque leurs instances sont comparées par structure et non par référence. Cela est utile pour la détection de cycle dans les fonctions solve, puisqu'il suffira que la grille soit la même pour être considéré comme le même état. De plus, nous avons généralisé le problème à des taquins rectangulaires de taille $N \times M$.

La grille peut être formatée à partir de notre fonction `parseTaquin`. Cette fonction n'était demandée dans le projet initial mais nous avons trouvé cela utile afin de pouvoir fournir une grille du taquin dans le format de l'énoncé, i.e. [[_ 2][1 3]]. La fonction vérifie que l'entrée est bien formatée et en s'appuyant sur des `foldRight`, `flatMap` et `map` permet d'obtenir la grille dans le bon format. Nous avions discuté avec M. Debruyne a une version intermédiaire du projet où les erreurs n'étaient pas gérés correctement. Cela est maintenant fait, mais par la suite nous devons utiliser un `getOrElse` afin d'obtenir notre grille.

Pour pouvoir créer un AFD pour résoudre notre taquin, nous avons :

- crée l'état initial en instanciant un objet Taquin avec la valeur retournée par `parseTaquin`
- représenté l'alphabet `sigma` qui correspond à un ensemble qui contient "u", "d", "l" et "r". Cela a été fait dans la case classe `Taquin`.
- crée la fonction de transition `delta` qui appelle la fonction `move` d'un taquin qui consiste en un swap du Trou et d'une case voisine grâce à la méthode `updated` définie pour les Listes.
  La case voisine est définie grâce à un filtrage par motif sur l'action à effectuer et intègre des gardes pour s'assurer de ne pas sortir de la gille.
  La position du Trou est obtenu grâce à la fonction `findTrou` qui parcourt la grille avec `zipWithIndex` pour associer un indice à chaque ligne, puis utilise `flatMap` pour extraire les coordonnées des trous de chaque ligne. Enfin, `headOption` permet de récupérer la première position trouvée ou de retourner None si aucun trou n'est présent.
- Créé l'ensemble des états accepteurs, composé uniquement de l'état final du taquin. Cet état final consiste en un objet Taquin généré avec comme grille les nombres de 1 à $N \times M - 1$, et le trou à la fin. Dans un premier temps, une liste est créée en utilisant `foldRight` pour accumuler les valeurs à partir de la fin vers le début. Cela permet d'utiliser l'opérateur : :. Ensuite, la liste est découpée en lignes de taille M grâce à `grouped(M).toList`.
  Cet état final est évalué de manière paresseuse puisqu'il crée un nouvel objet Taquin, donc nous ne souhaite pas que pou ce nouvel objet l'état final soit évalué.

Afin d'améliorer l'affichage des grilles du Taquin, nous avons modifié la fonction `toString` pour qu'elle affiche la grille tel que montré dans l'énoncé. Cela est fait grâce à un `map` de la grille où l'on transforme chaque case des lignes en un nombre ou _. Ces lignes sont entourées de crochets et le résultat final aussi.

Voici ce que l'on obtiens pour ce qui est demandé dans l'énoncé ainsi que des tests additionnels avec le taquin 3×3 et un taquin 2×3

```
Tests pour le taquin : [[_ 2][1 3]]
ex1.solve() -> List(List(d, r), List(r, d, l, u, r, d, l, u, r, d))
ex1.accept(List(d, u, d, r)) -> true
Tests pour le taquin : [[2 1][3 _]]
ex2.solve() -> List()
Tests pour le taquin : [[2 3 6][1 _ 5][7 8 4]]
ex3.lazysolve(taquin => taquin.H1).take(1).toList -> List(List(r, u, l, d, l, u, r, d, l, u, r, r, d, l,
u, r, d, l, u, r, d, l, l, d, r, u, l, d, r, u, l, d, r, r, u, l, d, r, u, l, d, l, u, r, r, d, l, l, u,
r, r, d, l, l, u, r, r, d, l, l, u, r, d, l, u, r, r, d))
ex3.accept(List(r, u, l, d, l, u, r, d, l, u, r, r, d, l, u, r, d, l, u, r, d, d, l, u, r, d, l, u, r, d,
 l, l, u, r, d, l, u, r, d, r, u, l, l, d, r, r, u, l, l, d, r, r, u, l, d, r, u, l, d, l, u, r, r, d, l,
 u, r, d, l, u, l, d, r, u, r, d, l, u, r, d, l, l, u, r, r, d, l, l, u, r, d, l, u, r, d, r)) -> true
Tests pour le taquin rectangulaire : [[1 2 3][4 _ 5]]
ex4.lazysolve(taquin => taquin.H2).take(1).toList -> List(List(r))
```

2.7 Heuristiques (bonus)

Nous avons déjà mentionné que les fonctions `solve` et `lazy` acceptaient une heuristique dans les parties ci-dessus, donc nous ne reviendrons pas sur leur implémentation.

Dans la case class `Taquin`, nous avons dû définir les fonctions H1 et H2.
Pour avoir H1, nous avons aplati (`flatten`) la grille afin d'avoir une liste à laquelle nous associons les indices des cases obtenu avec `zipWithIndex`. Grâce à cela, nous pouvons compter le nombre de pièces qui ne sont pas associés au bon indice, c'est-à-dire le nombre de pièces mal placées.
Pour obtenir H2, nous avons aussi aplati la grille et associé les indices des cases. Ensuite, nous avons appliqué un `map` afin d'avoir la distance de Manhattan pour chacune des cases. Enfin, nous faisons la somme de ces distances grâce à `sum`.

```
Temps de résolution avec H1 pour [[2 3 6][1 _ 5][7 8 4]] afin d'avoir 1 solutions : 56 ms
Temps de résolution avec H2 pour [[2 3 6][1 _ 5][7 8 4]] afin d'avoir 1 solutions : 8 ms
Temps de résolution avec H1 pour [[2 3 6][1 _ 5][7 8 4]] afin d'avoir 5 solutions : 309 ms
Temps de résolution avec H2 pour [[2 3 6][1 _ 5][7 8 4]] afin d'avoir 5 solutions : 18 ms
Temps de résolution avec H1 pour [[2 3 6][1 _ 5][7 8 4]] afin d'avoir 10 solutions : 1002 ms
Temps de résolution avec H2 pour [[2 3 6][1 _ 5][7 8 4]] afin d'avoir 10 solutions : 69 ms
```

Nous pouvons voir que H2 semble être une meilleure heuristique puisqu'elle permet d'avoir des solutions beaucoup plus rapidement que H1. Intuitivement, il semble normal que H2 se dirige plus vite vers des solutions que H1 puisque H2 donne une meilleure estimation du cout restant. En effet, lorsque seulement une case est mal placée, H1 retourne 1 alors que H2 va donner une indication du nombre de coup minimum pour que la case atteigne son état final. H2 permet donc de savoir si une action est plus promettante qu'une autre.

2.8 Entrée utilisateur

Par soucis de maniabilité, nous avons mis à disposition de l'utilisateur, une possibilité de tester la fonction `accept` en lui fournissant une entrée personnalisable directement via le terminal. Ainsi, en plus de rendre le code interactif, on peut aussi le tester facilement sous différents situations.
Pour ce qui est de son implémentation, nous importons depuis la librairie standard de Scala par `scala.io.StdIn` la fonction `readLine()` qui permet de lire la commande de l'utilisateur. Ensuite pour pouvoir traiter sa requête et l'appliquer à `accept`, nous avons besoin d'une autre fonction. Ainsi, nous avons créé dans la classe AFD, une méthode `parseAccept` ayant pour argument un `String` et une fonction transformant un `String` en un mot. Cette méthode renvoie ensuite un booléen en accordance avec le fait que la méthode `accept`

de l'ADF a pu être appliquer par un `map` sur l'entièreté du mot obtenu par l'utilisation de la fonction pris en argument sur l'entrée de l'utilisateur. Si son exécution se fait sans rencontré d'erreur alors on renvoie le résultat de `accept` sur le mot sinon si l'entrée est erroné est qu'on a donc reçu une erreur par le `try`, alors on renvoie directement `false` par un `getOrElse`.

3 Conclusion

Ce projet nous a permis d'explorer les concepts fondamentaux de la programmation fonctionnelle tout en appliquant ces principes à la résolution d'un problème concret : le développement d'un solveur générique pour les automates finis déterministes (AFD). Tout au long de ce travail, nous avons cherché à optimiser l'efficacité, la modularité et la lisibilité de notre code, en mettant en pratique des structures immuables, des fonctions récursives terminales, ainsi que l'évaluation paresseuse lorsque cela était nécessaire.

La conception de la méthode `solve` et sa déclinaison paresseuse `lazysolve` ont représenté des défis intéressants. Ces implémentations ont nécessité une réflexion approfondie sur les structures de données appropriées (`File` et `LazyList`), les algorithmes de parcours et la gestion efficace des états visités pour éviter les cycles. Nous avons également fait usage des fonctions d'ordre supérieur comme `foldLeft` pour simplifier et clarifier certaines opérations.

En abordant des exemples spécifiques tels que les chaînes binaires impaires, le problème du loup, du mouton et du chou, ainsi que le taquin, nous avons démontré la flexibilité et la robustesse de notre solveur. Chaque problématique nous a permis de concevoir des ADT adaptés et de vérifier que nos implémentations respectaient les spécifications et les propriétés des automates déterministes.

Pour améliorer la performance du solveur, nous avons exploré l'utilisation d'heuristiques. Nous avons constaté que l'heuristique H2, basée sur la distance de Manhattan, fournissait des résultats plus rapides et plus pertinents que H1, renforçant ainsi l'importance d'une bonne évaluation heuristique dans les problèmes d'exploration.

Avec le recul, certaines améliorations pourraient encore être envisagées, comme par exemple le fait de renforcer la gestion des exceptions pour rendre le solveur plus robuste face à des entrées malformées ou à des automates incomplets.

En conclusion, ce projet a été une opportunité précieuse pour approfondir notre compréhension de la programmation fonctionnelle et de ses avantages dans le traitement de problèmes complexes. Grâce à l'utilisation de concepts comme les structures immuables, les fonctions d'ordre supérieur et l'évaluation paresseuse, nous avons développé un solveur générique qui allie efficacité et simplicité d'utilisation, tout en respectant les principes de modularité et de réutilisabilité.