



Département des Technologie de
l'information et de la communication (TIC)
Informatique et systèmes de communication
Informatique logicielle

Travail de Bachelor

Concevoir une expérience d'apprentissage interactive à la programmation avec PLX

Ou comment permettre aux enseignants de programmation
de concevoir des cours orientés sur la pratique et le feedback.



Étudiant

Enseignant responsable

Année académique

Samuel Roland

Prof. Bertil Chapuis

2024-25

Yverdon-les-Bains, le 20.05.2025

Département des Technologie de l'information et de la communication (TIC)
Informatique et systèmes de communication
Informatique logicielle
Étudiant : Samuel Roland
Enseignant responsable : Prof. Bertil Chapuis

Travail de Bachelor 2024-25

Concevoir une expérience d'apprentissage
interactive à la programmation avec PLX

Résumé publiable

TODO: résumé publiable...

Étudiant :

Samuel Roland

Date et lieu :

.....

Signature :

.....

Enseignant responsable :

Prof. Bertil Chapuis

Date et lieu :

.....

Signature :

.....

Préambule

Ce travail de Bachelor (ci-après TB) est réalisé en fin de cursus d'études, en vue de l'obtention du titre de Bachelor of Science HES-SO en Ingénierie.

En tant que travail académique, son contenu, sans préjuger de sa valeur, n'engage ni la responsabilité de l'auteur, ni celles du jury du travail de Bachelor et de l'Ecole.

Toute utilisation, même partielle, de ce TB doit être faite dans le respect du droit d'auteur.

HEIG-VD

Vincent Peiris
Chef de département TIC

Yverdon-les-Bains, le 20.05.2025

Authentification

Le soussigné, Samuel Roland, atteste par la présente avoir réalisé ce travail et n'avoir utilisé aucune autre source que celles expressément mentionnées

Yverdon-les-Bains, le 20.05.2025

Samuel Roland

Table des matières

Préambule	4
Authentification	5
Introduction	8
Contexte	8
Problème	8
Défis	9
Comment les enseignants peuvent voir les résultats en temps réel ?	9
Comment faciliter la rédaction et la maintenance des exercices ?	10
Solutions existantes	13
Glossaire	14
Planification	15
Déroulement	15
Planification initiale	15
Planification finale	16
Etat de l'art	17
Format de données humainement éditables existants	17
KHI - Le langage de données universel	17
Bitmark - le standard des contenus éducatifs digitaux	18
NestedText – Un meilleur JSON	20
SDLang - Simple Declarative Language	20
KDL - Cuddly Data language	22
Conclusion	22
Librairies existantes de parsing en Rust	23
Les serveurs de langage et librairies Rust existantes	24
Adoption	25
Librairies disponibles	25
Choix final	25
POC sur lsp-server	26
Systèmes de surglignage de code	28
Textmate	28
Tree-Sitter	29
Semantic highlighting	29
Choix final	31
POC de Tree-Sitter	31
Protocoles de synchronisation et formats de sérialisation existants	35
JSON	35
Protocol Buffers - ProtoBuf	35

MessagePack	36
Websocket	36
gRPC	37
tarpc	38
Choix final	38
POC de synchronisation de messages JSON via websockets avec tungstenite	39
Architecture	43
Implémentation	44
Conclusion	45
Bibliographie	46
Outils utilisés	53
Usage de l'IA	53
Outils techniques	53

Introduction

Contexte

Ce travail de Bachelor vise à développer le projet PLX (voir plx.rs, Terminal User Interface (TUI) écrite en Rust, permettant de faciliter la pratique intense sur des exercices de programmation en retirant un maximum de friction. PLX vise également à apporter le plus vite possible un feedback automatique et riche, dans le but d'appliquer les principes de la pratique délibérée à l'informatique. PLX peut à terme aider de nombreux cours à la HEIG-VD (tels que PRG1, PRG2, PCO, SYE, ...) à passer de long moments de théorie en session d'entraînement dynamique et très interactive. En réfinissant l'expérience des étudiants et des enseignants sur les exercices et laboratoires, l'ambition est qu'à terme, cela génère un apprentissage plus profond de modèles mentaux solides chez les étudiants. Cela aidera les étudiants qui ont beaucoup de peine à s'approprier la programmation à avoir moins de difficultés avec ces cours. Et ceux qui sont plus à l'aise pourront développer des compétences encore plus avancées.

Problème

Le projet est inspiré de Rustlings (TUI pour apprendre le Rust), permettant de s'habituer aux erreurs du compilateur Rust et de prendre en main la syntaxe (1). PLX fournit actuellement une expérience locale similaire pour le C et C++. Les étudiants clonent un repos Git et travaillent localement sur des exercices afin de faire passer des checks automatisés. À chaque sauvegarde, le programme est compilé et les checks sont lancés. Cependant, faire passer les checks n'est que la 1ère étape. Faire du code qualitatif, modulaire, lisible et performant demande des retours humains pour pouvoir progresser. De plus, les exercices existants étant stockés dans des PDF ou des fichiers Markdown, cela nécessite de les migrer à PLX.



Fig. 1. – Aperçu de la page d'accueil de PLX dans le terminal (2)

Basic arguments usage

The 2 first program arguments are the firstname and number of legs of a dog. Print a full sentence about the dog. Make sure there is at least 2 arguments, print an error if not.

Check results - 2/3 passed**1. Joe + 5 legs**

Args: ["Joe", "5"]

Diff:

-The dog is Joeand has 5 legs

+The dog is Joe and has 5 legs

2. No arg → error**3. One arg → error**

Fig. 2. – Aperçu d'un exercice dans PLX, avec un check qui échoue et les 2 suivants qui passent (2)

Défis

Comment les enseignants peuvent voir les résultats en temps réel ?

Ce TB aimerait pousser l'expérience en classe plus loin pour permettre aux étudiants de recevoir des feedbacks sur leur réponse en live, sur des sessions hautement interactives. Cela aide aussi les enseignants à mesurer l'état de compréhension et les compétences des étudiants tout au long du semestre, et à adapter leur cours en fonction des incompréhensions et des lacunes.

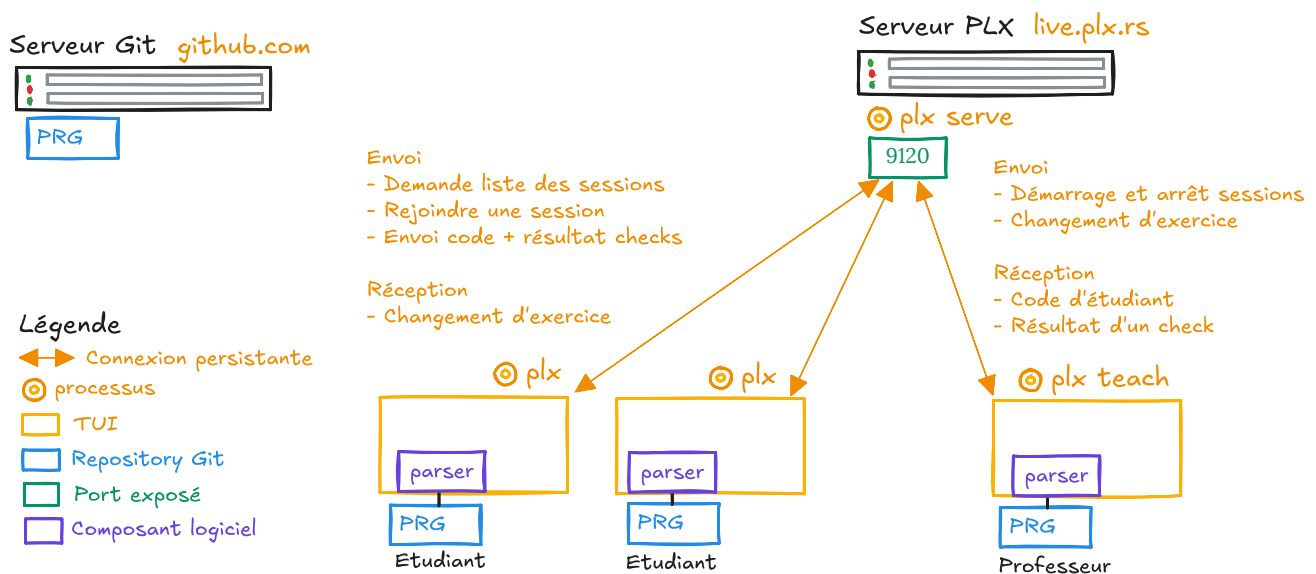


Fig. 3. – Architecture haut niveau décrivant les interactions entre les clients PLX et le serveur de session live

Une fois une session live démarrée par un enseignant et les étudiants ayant rejoint la session, l'enseignant peut choisir de faire un exercice l'un après l'autre en définissant son propre rythme. L'exercice en cours est affichés sur tous les clients PLX. A chaque sauvegarde d'un fichier de code, le code est compilé et les checks sont lancés comme en dehors d'une session live. La différence est que les résultats des checks et le code modifié sera envoyé à l'enseignant de la session, pour qu'il puisse directement dans PLX. L'enseignant pourra ainsi avoir un aperçu global de l'avancement et des checks qui ne passent pas, d'aller inspecter le code de certaines soumissions et au final de faire des feedbacks à la classe en live ou à la fin de l'exercice.

Comment faciliter la rédaction et la maintenance des exercices ?

Pour faciliter la productivité dans la rédaction et maintenance d'exercices ainsi que leur première transcription, on souhaiterait avoir une syntaxe épurée, humainement lisible et éditable, facilement versionnable dans Git. Pour cette raison, nous introduisons une nouvelle syntaxe appelée DY. Elle sera adaptée pour PLX afin de remplacer le format TOML actuel.

Voyons maintenant un exemple concret d'exercice de programmation, très inspiré d'un laboratoire du cours Système d'exploitation (SYE) à la HEIG-VD.

Pipe implementation in our custom shell

A pipe in system programming is a way to forward the standard output of a program to the standard input of another one.

When running this command in our custom shell using this symbol ``|``, we want the output of ``echo`` to be used as the input of ``toupper`` which is just going to print the text in uppercase.

```
```sh
echo hello | toupper
```
```

To test if the output sent through a pipe reaches ``toupper``, you can run the script ``./st``, you can ignore all lines before ``S03: starting the initial process (shell)``. Once you see the prompt ``so3%``, you can type a command.

```
```sh
so3% echo hello | toupper
HELLO
```
```

```
```sh
so3% type ls | toupper
CAT.ELF
ECHO.ELF
LN.ELF
LS.ELF
SH.ELF
```
```

To exit the shell, you have to enter ``Ctrl+x a``, this will exit the Qemu hypervisor.

Snippet 1. – Exemple d'exercice de programmation, rédigé en Markdown, avec une implémentation du pipe dans un shell

Cet exercice en Snippet 1 est adapté à l'affichage et l'export PDF pour être distribué dans un recueil d'exercices ou dans une consigne de laboratoire. Cependant, ce format n'est pas adapté à être parsé par un outil qui aimerait automatiser la vérification du code. En effet, les 2 exemples de commandes à lancer ne pourront être que lancées à la main par l'étudiant, ce qui crée de la friction autour de l'exercice et ralentit l'étudiant dans son apprentissage.

Nous avons besoin d'une syntaxe qui permet de décrire le démarrage du shell, ce que l'étudiant tape à la main dans son terminal, la vérification des outputs à différents endroits et finalement la terminaison du shell.

L'option la plus rapide et facile à ce problème serait de rédiger en format JSON.

```

{
  "exo": "Pipe implementation in our custom shell",
  "instruction": "A pipe in system programming is a way to forward the standard
output\nof a program to the standard input of another one.\n\nWhen running this command
in our custom shell using this symbol `|`,\nwe want the output of `echo` to be used as
the input of `toupper`\nwhich is just going to print the text in uppercase.\n``sh\necho
hello | toupper\n``",
  "checks": [
    {
      "name": "Output sent through a pipe reaches `toupper`",
      "run": "./st",
      "skip": { "mode": "until", "line": "S03: starting the initial process
(shell)" },
      "sequence": [
        { "type": "see", "timeout": "2s", "value": "so3%" },
        { "type": "type", "value": "echo hello | toupper" },
        { "type": "see", "value": "HELLO" },
        { "type": "see", "value": "so3%" },
        { "type": "type", "value": "ls | toupper" },
        { "type": "see",
          "value": "CAT.ELF\nECHO.ELF\nLN.ELF\nLS.ELF\nSH.ELF\n" },
        { "type": "see", "value": "so3%" },
        { "type": "kill", "signal": 9, "value": "qemu-system-arm" }
      ]
    }
  ]
}

```

Snippet 2. – Equivalent JSON de l'exercice défini sur le Snippet 1

Dans cet équivalent JSON, on voit bien que rédiger du contenu Markdown ou l'output sur plusieurs lignes en remplaçant les retours à la ligne `\n` à la main est fastidieux et complique la lisibilité, en plus de tous les guillemets, deux points et accolades nécessaires au-delà du texte.

Voyons maintenant à quoi pourrait ressembler cette nouvelle syntaxe DY beaucoup plus légère pour l'exercice précédent:

```

exo Pipe implementation in our custom shell
A pipe in system programming is a way to forward the standard output
of a program to the standard input of another one.

When running this command in our custom shell using this symbol `|`,
we want the output of `echo` to be used as the input of `toupper`
which is just going to print the text in uppercase.
```sh
echo hello | toupper
```

check Output sent through a pipe reaches `toupper`
run ./st
skip .until S03: starting the initial process (shell)
see .timeout 2s so3%
type echo hello | toupper
see HELLO
see so3%
type ls | toupper
see
CAT.ELF
ECHO.ELF
LN.ELF
LS.ELF
SH.ELF
see so3%
kill .signal 9 qemu-system-arm

```

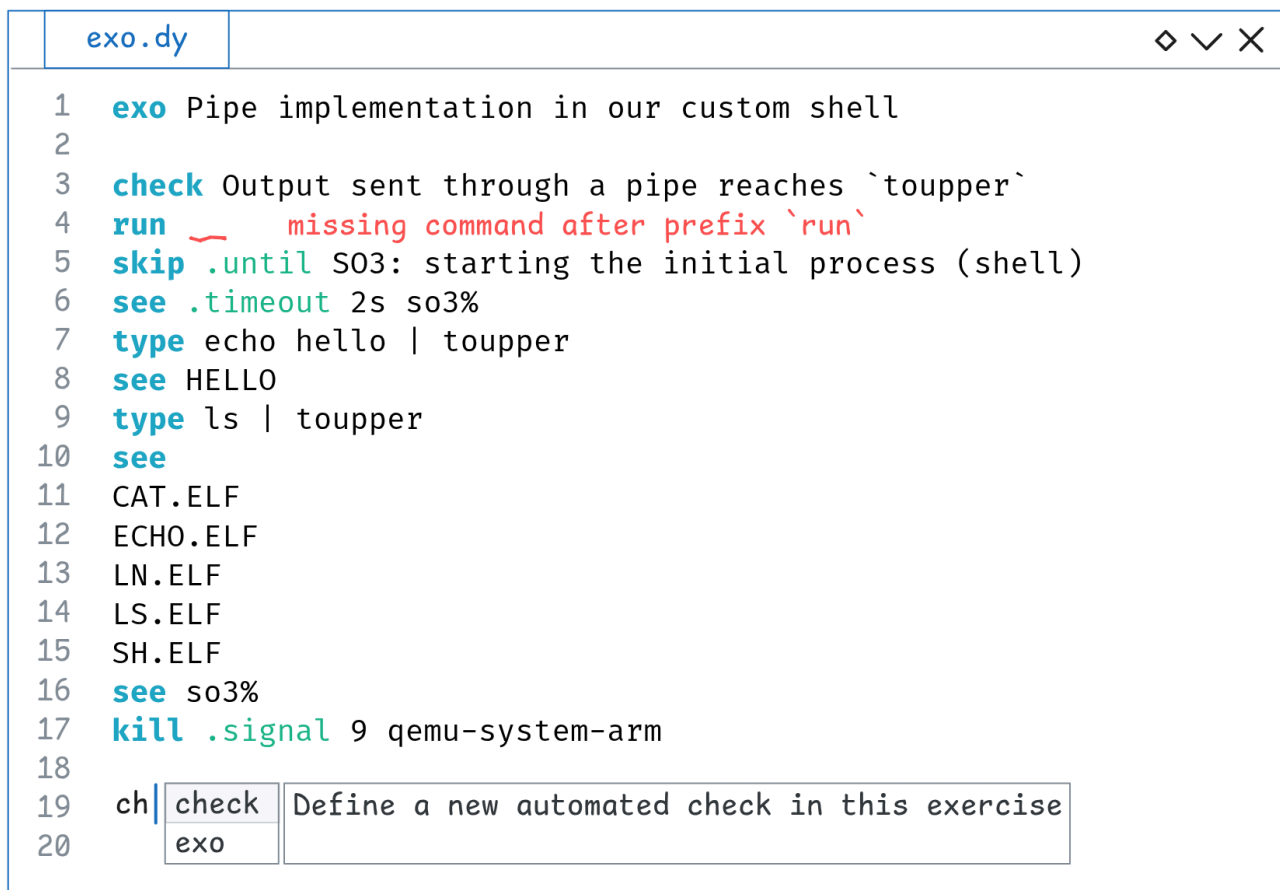
Fig. 4. – Equivalent dans une version préliminaire de la syntaxe DY de l'exercice défini sur le Snippet 1

Ce système de préfixe (en bleu du début des lignes) et de propriétés (après un point) permet de structurer le contenu de l'exercice tout en gardant un style de rédaction proche du Markdown.

Le préfixe `exo` introduit un exercice, avec un titre sur la même ligne et le reste de la consigne en Markdown sur les lignes suivantes. `check` introduit le début d'un check avec un titre, en Markdown également. `run` donne la commande de démarrage du programme. Le préfixe `skip` avec la propriété `.until` permet de cacher toutes les lignes d'output jusqu'à voir la ligne donnée. `see` demande à voir une ou plusieurs ligne en sortie standard. `type` simule une entrée au clavier et finalement `kill` indique comment arrêter le programme, ici en envoyant le `.signal 9` sur le processus `qemu-system-arm` (qui a été lancé par notre script `./st`).

Les fins de ligne définissent la fin du contenu pour les préfixes sur une seule ligne. Le préfixe `exo` supporte plusieurs ligne, son contenu se termine ainsi dès qu'un autre préfixe valide est détecté (ici `check`). La hiérarchie est implicite dans la sémantique, un exercice contient un ou plusieurs checks, sans qu'il y ait besoin d'indentation ou d'accolades pour indiquer les relations de parents et enfants. De même, un check contient une séquence d'action à effectuer (`run`, `see`, `type` et `kill`), ces préfixes n'ont de sens qu'à l'intérieur la définition d'un check (que après une ligne préfixée par `check`).

Cette deuxième partie demande ainsi d'écrire un parseur de cette nouvelle syntaxe et un support des différents IDE. Voici un aperçu de l'expérience imaginée des enseignants pour la rédaction des exercices dans cette syntaxe.



```
1  exo Pipe implementation in our custom shell
2
3  check Output sent through a pipe reaches `toupper`
4  run missing command after prefix `run`
5  skip .until S03: starting the initial process (shell)
6  see .timeout 2s so3%
7  type echo hello | toupper
8  see HELLO
9  type ls | toupper
10 see
11 CAT.ELF
12 ECHO.ELF
13 LN.ELF
14 LS.ELF
15 SH.ELF
16 see so3%
17 kill .signal 9 qemu-system-arm
18
19 ch| check Define a new automated check in this exercise
20    exo
```

Fig. 5. – Aperçu de l'expérience imaginée dans un IDE

On voit dans la Fig. 5 que l'intégration se fait sur 2 points majeures

- le surlignage de code, qui permet de coloriser les préfixes et les propriétés, afin de bien distinguer le contenu des éléments propres à la syntaxe
- intégration avancée de la connaissance et des erreurs du parseur à l'éditeur: comme en ligne 4 avec l'erreur de la commande manquante après le préfixe `run`, et comme en ligne 19 avec une auto-complétion qui propose les préfixes valides à cette position du curseur.

Solutions existantes

Comme mentionné dans l'introduction, PLX est inspiré de Rustlings. Cette TUI propose une centaine d'exercices avec des morceaux de code à faire compiler ou avec des tests à faire passer. L'idée est de faire ces exercices en parallèle de la lecture du « Rust book » (la documentation officielle).

The image shows a Rustlings exercise in a terminal window and the corresponding code in a VS Code editor. The terminal window displays the following output:

```
error: expected type, found ``
  → exercises/02_functions/functions2.rs:2:16
  |
2 | fn call_me(num:) {
  |               ^ expected type

error[E0425]: cannot find value `num` in this scope
  → exercises/02_functions/functions2.rs:3:17
  |
3 |     for i in 0..num {
  |                   ^^^ not found in this scope

For more information about this error, try `rustc --explain E0425`.
error: could not compile `exercises` (bin "functions2") due to 2 previous errors

Progress: [>-----] 0/94
Current exercise: exercises/02\_functions/functions2.rs

h:hint / l:list / c:check all / x:reset / q:quit ?
```

The VS Code editor shows the file `functions2.rs` with the following code:

```
1 // TODO: Add the missing type of the argument `num`
  // after the colon `:`.
2 fn call_me(num: i32) {
3     for i: i32 in 0..num {
4         println!("Ring! Call number {}", i + 1);
5     }
6 }
7
8 fn main() {
9     call_me(num: 3);
10 }
```

Fig. 6. – Un exemple de Rustlings en haut dans le terminal et VSCode en bas, sur un exercice de fonctions

De nombreux autres projets se sont inspirés de ce concept, `clings` pour le C (3), `golings` pour le Go (4), `ziglings` pour Zig (5) et même `haskellings` pour le Haskell (6) ! Ces projets consistent d'une suite d'exercice et d'une TUI pour les exécuter pas à pas, afficher les erreurs de compilation ou les cas de tests qui échouent, pour faciliter la prise en main aux débutants.

Chaque projet se concentre sur un langage et créer des exercices dédiés. PLX prend une approche différente, il n'y a pas d'exercice proposés parce que PLX supporte de multiples langages. Le contenu sera géré indépendamment de l'outil, permettant aux enseignants en école d'intégrer leur propre contenu et compétences enseignées.

Glossaire

- `Cargo.toml`, fichier dans un projet Rust définit les dépendances (les crates) et leur versions minimum à inclure dans le projet, équivalent du `package.json` de NPM
- `crate` : la plus petite unité de compilation avec cargo, concrètement chaque projet contient un ou plusieurs dossiers avec un `Cargo.toml`
- `crates.io` : le registre officiel des crates publiée pour l'écosystème Rust, l'équivalent de `npmjs.com` pour l'écosystème Javascript, ou `mvnrepository.com` pour Java

Planification

Déroulement

Le travail commence le 17 février 2025 et se termine le 24 juillet 2025. Sur les 16 premières semaines, soit du 17 février 2025 au 15 juin 2025, la charge de travail représente 12h par semaine. Les 6 dernières semaines, soit du 16 juin 2025 au 24 juillet 2024, ce travail sera réalisé à plein temps.

Un rendu intermédiaire noté est demandé le 23 mai 2025 avant 17h et le rendu final est prévu pour le 24 juillet 2025 avant 17h.

La défense sera organisée entre le 25 août 2025 et le 12 septembre 2025.

Planification initiale

Note: cette planification est reprise du cahier des charges original

En se basant sur le calendrier des travaux de Bachelor, voici un aperçu du découpage du projet pour les différents rendus.

Rendu 1 - 10 avril 2025 - Cahier des charges

- Rédaction du cahier des charges.
- Analyse de l'état de l'art des parsers, des formats existants de données humainement éditables, du syntax highlighting et des langages servers.
- Analyse de l'état de l'art des protocoles bidirectionnels temps réel (websockets, gRPC, ...) et des formats de sérialisation (JSON, protobuf, ...).
- Prototype avec les bibliothèques disponibles de parsing et de langage servers en Rust, choix du niveau d'abstraction espéré et réutilisation possible.

Rendu 2 - 23 mai 2025 - Rapport intermédiaire

- Rédaction du rapport intermédiaire.
- Définition de la syntaxe DY à parser, des préfixes et flags liés à PLX, et la liste des vérifications et des erreurs associées.
- Définition d'un protocole de synchronisation du code entre les participants d'une session.
- Prototype d'implémentation de cette synchronisation.
- Prototype des tests automatisés sur le serveur PLX.
- Définition du protocole entre les clients PLX et le serveur pour les entraînements live.

Moitié des 6 semaines à temps plein - 4 juillet 2025

- Écriture des tests de validation du protocole et de gestion des erreurs.
- Développement du serveur PLX.
- Rédaction du rapport final par rapport aux développements effectués.

Rendu 3 - 24 juillet 2025 - Rapport final

- Développement d'une bibliothèque `dy`.
- Intégration de cette bibliothèque à PLX.

- Rédaction de l'affiche et du résumé publiable.
- Rédaction du rapport final.

Planification finale

Voici les étapes des jalons majeures atteints durant le travail.

TODO

Etat de l'art

Format de données humainement éditables existants

Ces recherches ignorent les formats de données largement supporté et répandu tel que le XML, JSON, YAML et TOML. Ils sont tout à fait adapter pour des configurations, de la sérialisation et de l'échange de donnée et sont pour la plupart facilement lisible. Cependant la quantité de séparateurs et délimiteurs en plus du contenu qu'ils n'ont pas été optimisé pour la rédaction par des humains.

Le YAML et le TOML, bien que plus léger que le JSON, inclue de nombreux types de données autre que les strings, des tabulations et des guillemets, ce qui rend la rédaction plus fastidieuse qu'en Markdown. Le Markdown a le défaut de ne pas être assez structuré être parsé par une machine. On cherche quelque chose du niveau de simplicité du Markdown en terme de rédaction, mais avec une validation poussée customisable par le projet qui définit le schéma.

Ces recherches se focalisent sur les syntaxes qui ne sont pas spécifique à un domaine ou qui seraient complètement déliée de l'informatique ou de l'éducation. Ainsi, l'auteur ne présente pas Cooklang (7), qui se veut une langage de balise pour les recettes de cuisines, même si l'implémentation du parseur en Rust (8) pourra servir pour d'autres recherches.

On ignore également les projets qui créent une syntaxe très proche du Rust, comme la Rusty Object Notation (RON) (9), de par leur nécessité de connaître un peu la syntaxe du Rust et surtout parce qu'elle ne simplifie pas vraiment l'écriture comparé à du YAML. On ignore également les projets dont la spécification ou l'implémentation est en état de « brouillon » et n'est pas encore utilisable en production.

Contrairement aux langages de programmation qui existent par centaines, les syntaxes de ce genre ne sont pas monnaies courantes. Différentes manières de les nommer existent: langage de balise (markup language), format de donnée, syntaxes, langage de donnée, langage spécifique à un domaine (de l'anglais Domain Specific Language - DSL), ... Les mots-clés utilisés suivants ont été utilisés sur Google, la barre de recherche de Github.com et de crates.io: `data format`, `human friendly`, `human writable`, `human readable`.

KHI - Le langage de données universel

D'abord nommée UDL (Universal Data Language) (10), cette syntaxe a été inventée pour mixer les possibilités du JSON, YAML, TOML, XML, CSV et Latex, afin de supporter toutes les structures de données modernes. Plus concrètement le markup, les structs, les listes, les tuples, les tables/matrices, les enums, les arbres hiérarchiques sont supportés. Les objectifs sont la polyvalence, un format source (fait pour être rédigé à la main), l'esthétisme et la simplicité.

```
{article}:
uuid: 0c5aacfe-d828-43c7-a530-12a802af1df4
type: chemical-element
key: aluminium
title: Aluminium
description: The <@element>:{chemical element} aluminium.
tags: [metal; common]

{chemical-element}:
symbol: Al
number: 13
stp-phase: <Solid>
melting-point: 933.47
boiling-point: 2743
density: 2.7
electron-shells: [2; 8; 3]

{references}:
wikipedia: \https://en.wikipedia.org/wiki/Aluminium
snl: \https://snl.no/aluminium
```

Snippet 3. – Un exemple simplifié de KHI de leur README (11), décrivant un exemple d'article d'encyclopédie.

Une implémentation en Rust est proposée (12). Son dernier commit sur ces 2 repositorys date du 11.11.2024, le projet a l'air de ne pas être fini au vu des nombreux `todo!()` présent dans le code. La large palette de structures supportées implique une charge mentale additionnelle pour se rappeler, ce qui en fait une mauvaise option pour PLX.

Bitmark - le standard des contenus éducatifs digitaux

Bitmark est un standard open-source, qui vise à uniformiser tous les formats de données utilisés pour décrire du contenu éducatif digital sur les nombreuses plateformes existantes (13). Cette diversité de formats rend l'interopérabilité très difficile et freine l'accès à la connaissance et restreint les créateurs de contenus et les éditeurs dans les possibilités de migration entre plateformes. La stratégie est de définir un format basé sur le contenu (Content-first) plus que basé sur son rendu (layout-first) permettant un affichage sur tous type d'appareils incluant les appareils mobiles (13). C'est la Bitmark Association en Suisse à Zurich qui développe ce standard, notamment à travers des Hackatons organisés en 2023 et 2024 (14).

Le standard permet de décrire du contenu statique et interactif, comme des articles ou des quiz de divers formats. 2 formats équivalents sont définis: le bitmark markup language et le bitmark JSON data model (15)

La partie quizzes du standard inclut des textes à trous, des questions à choix multiple, du texte à surligner, des essais, des vrai/faux, des photos à prendre ou audios à enregistrer et de nombreux autres types d'exercices.

```
[.multiple-choice-1]
[!What color is milk?]
[?Cows produce milk.]
[+white]
[-red]
[-blue]
```

Snippet 4. – Un exemple de question à choix multiple tiré de leur documentation (16). L'option correcte `white` est préfixée par `+` et les 2 autres options incorrectes par `-`. Plus haut, `[! ...]` décrit une consigne, `[? ...]` décrit un indice.

```
{
  "markup": "[.multiple-choice-1]\\n[!What color is milk?]\\n[+white]\\n[-red]\\n[-blue]",
  "bit": {
    "type": "multiple-choice-1",
    "format": "text",
    "item": [],
    "instruction": [ { "type": "text", "text": "What color is milk?" } ],
    "body": [],
    "choices": [
      { "choice": "white", "item": [], "isCorrect": true },
      { "choice": "red", "item": [], "isCorrect": false },
      { "choice": "blue", "item": [], "isCorrect": false }
    ],
    "hint": [ { "type": "text", "text": "Cows produce milk." } ],
    "isExample": false,
    "example": []
  }
}
```

Snippet 5. – Equivalent de Snippet 4 dans le Bitmark JSON data model (16)

Open Taskpool, projet qui met à disposition des exercices d'apprentissage de langues (17), fournit une API JSON utilisant le Bitmark JSON data model.

```
curl "https://taskpool.taskbase.com/exercises?translationPair=de->en&word=school&exerciseType=bitmark.cloze"
```

Snippet 6. – Requête HTTP à Open Taskpool pour demander des exercices d'allemand vers anglais autour du mot `school` de format `cloze` (texte à trou)

```
...
"cloze": {
  "type": "cloze",
  "format": "text",
  "instruction": "Gegeben: \"Früher war hier eine Schule.\", schreiben Sie das fehlende Wort",
  "body": [
    { "type": "text", "text": "There used to be a " },
    {
      "type": "gap",
      "solutions": [ "school" ],
      "answer": { "text": "" }
    },
    { "type": "text", "text": " here." }
  ]
},
...
```

Snippet 7. – Extrait simplifié de la réponse JSON, respectant le standard Bitmark (18). La phrase `There used to be a ___ here.` doit être complétée par le mot `school` en s'aidant du texte en allemand.

Un autre exemple d'usage se trouve dans la documentation de Classtime (19), on voit que le système de création d'exercices est basé sur des formulaires. Ces 2 exemples donnent l'impression que la structure JSON est plus utilisée que le markup. Au vu de tous séparateurs et symboles de ponctuations à se rappeler, la syntaxe n'a peut-être pas été imaginée dans le but d'être rédigée à la main directement. Finalement, Bitmark ne spécifie pas de type d'exercices programmation nécessaire à PLX.

NestedText – Un meilleur JSON

NestedText se veut human-friendly, similaire au JSON mais pensé pour être facile à modifier et visualiser par les humains. Le seul type de donnée scalaire supporté est la chaîne de caractères, afin de simplifier la syntaxe et retirer le besoin de mettre des guillemets. La différence avec le YAML, en plus des types de données restreint est la facilité d'intégrer des morceaux de code sans échappements ni guillemets, les caractères de données ne peuvent pas être confondus avec NestedText (20).

```
Margaret Hodge:
  position: vice president
  address:
    > 2586 Marigold Lane
    > Topeka, Kansas 20682
  phone: 1-470-555-0398
  email: margaret.hodge@ku.edu
  additional roles:
    - new membership task force
    - accounting task force
```

Snippet 8. – Exemple tiré de leur README (20)

Ce format a l'air assez léger visuellement et l'idée de faciliter l'intégration de blocs multi-lignes sans contraintes de caractères réservée serait utile à PLX. Cependant, tout comme le JSON la validation du contenu n'est pas géré directement par le parseur mais par des bibliothèques externes qui vérifient le schéma (21). De plus, l'implémentation officielle est en Python et il n'y a pas d'implémentation Rust disponible; il existe une crate réservée mais vide (22).

SDLang - Simple Declarative Language

SDLang se définit comme « une manière simple et concise de représenter des données textuellement. Il a une structure similaire au XML: des tags, des valeurs et des attributs, ce qui en fait un choix polyvalent pour la sérialisation de données, des fichiers de configuration ou des langages déclaratifs. » (Traduction personnelle de leur site web (23)). SDLang définit également différents types de nombres (32bit, 64bit, entier, flottant, ...), 4 valeurs de booléens (`true`, `false`, `on`, `off`) comme en YAML, différents formats de dates et un moyen d'intégrer des données binaires encodées en Base64.

```
// This is a node with a single string value
title "Hello, World"

// Multiple values are supported, too
bookmarks 12 15 188 1234

// Nodes can have attributes
author "Peter Parker" email="peter@example.org" active=true

// Nodes can be arbitrarily nested
contents {
  section "First section" {
    paragraph "This is the first paragraph"
    paragraph "This is the second paragraph"
  }
}

// Anonymous nodes are supported
"This text is the value of an anonymous node!"

// This makes things like matrix definitions very convenient
matrix {
  1 0 0
  0 1 0
  0 0 1
}
```

Snippet 9. – Exemple tiré de leur site web (23)

Ce format s'avère plus intéressant que les précédents de part le faible nombre de caractères réservés et la densité d'information: avec l'auteur décrit par son nom, email et un attribut booléen sur une seule ligne ou la matrice de 9 valeurs définie sur 5 lignes. Il est cependant regrettable de voir de les strings doivent être entourées de guillemets et les textes sur plusieurs lignes doivent être entourés de backticks ```. De même la définition de la hiérarchie d'objets définis nécessite d'utiliser une paire `{ }`, ce qui rend la rédaction un peu plus lente.

KDL - Cuddly Data language

```
package {
  name my-pkg
  version "1.2.3"

  dependencies {
    // Nodes can have standalone values as well as
    // key/value pairs.
    lodash "^3.2.1" optional=#true alias=underscore
  }

  scripts {
    // "Raw" and dedented multi-line strings are supported.
    message ""
      hello
      world
    ""

    build #""
      echo "foo"
      node -c "console.log('hello, world!');"
      echo "foo" > some-file.txt
    ""#
  }
}
```

Snippet 10. – Exemple simplifié tiré de leur site web (24)

Est-ce que cela paraît proche de SDLang vu précédemment ? C'est normal puisque KDL est basé sur SDLang avec quelques améliorations. Celles qui nous intéressent concernent la possibilité d'utiliser des guillemets pour les strings sans espace (`person name=Samuel` au lieu de `person name="Samuel"`). Cette simplification n'inclue malheureusement des strings multilignes, qui demande d'être entourée par `""` . Le problème d'intégration de morceaux de code est également relevé, les strings brutes sont supportées entre `#` sur le mode une ou plusieurs lignes, ainsi pas d'échappements des backslashes à faire par ex.

En plus des autres désavantages restant de hiérarchie avec `{ }` et guillemets, il reste toujours le problème des types de nombres qui posent soucis avec certaines strings si on ne les entoure pas de guillemets. Par exemple ce numéro de version `version "1.2.3"` a besoin de guillemets sinon `1.2.3` est interprété comme une erreur de format de nombre à virgule.

Conclusion

En conclusion, au vu du nombre de tentatives/variantes trouvées, on voit que la verbosité des formats largement répandu du XML, JSON et même du YAML est un problème qui ne touche pas que l'auteur. La diminution de la verbosité des syntaxes décrites en-dessus est réel cible des usages plus avancé de structure de données et types variés. L'auteur pense pouvoir proposer une approche encore plus légère et plus simple, inspirée du Markdown, reprenant les avantages du YAML mais sans les tabulations et uniquement basé sur les strings et les listes.

Librairies existantes de parsing en Rust

Après s'être intéressé aux syntaxes existantes, nous nous intéressons maintenant aux solutions existantes pour simplifier ce parsing de cette nouvelle syntaxe en Rust.

Après quelques recherches avec le tag `parser` sur crates.io (25), j'ai trouvé la liste de librairies suivantes:

- `winnow` (26), fork de `nom`, utilisé notamment par le parseur Rust de KDL (27)
- `nom` (28), utilisé notamment par `cexpr` (29)
- `pest` (30)
- `combine` (31)
- `chumsky` (32)

A noter aussi l'existence de la crate `serde`, un framework de sérialisation et désérialisation très populaire dans l'écosystème Rust (selon lib.rs (33)). Il est notamment utilisé pour les parseurs JSON et TOML. Ce n'est pas une librairie de parsing mais un modèle de donnée basée sur les traits de Rust pour faciliter son travail. Au vu du modèle de données de Serde (34), qui supporte 29 types de données, ce projet paraît à l'auteur apporter plus de complexités qu'autre chose pour trois raisons:

- Seulement les strings, listes et structs sont utiles pour PLX. Par exemple, les 12 types de nombres sont inutiles à différencier et seront propre au besoin de la variante.
- La sérialisation (struct Rust vers syntaxe DY) n'est pas prévue, seulement la désérialisation est utile.
- Le mappage des préfixes et propriétés par rapport aux attributs des structs Rust qui seront générées, n'est pas du 1:1, cela dépendra de la structure définie pour la variante de PLX.

Après ces recherches et quelques essais avec `winnow`, l'auteur a finalement décidé qu'utiliser une librairie était trop compliqué pour le projet et que l'écriture manuelle d'un parseur ferait mieux l'affaire. La syntaxe DY est relativement petite à parser, et sa structure légère et souvent implicite rend compliqué l'usage de librairies pensées pour des langages de programmation très structuré.

Par exemple, une simple expression mathématique `((23+4) * 5)` paraît idéale pour ces outils, les débuts et fin sont claires, une stratégie de combinaisons de parseurs fonctionnerait bien pour les expressions parenthésées, les opérateurs et les nombres. Elles semble bien adapter à exprimer l'ignorance des espaces, extraire les nombres tant qu'il contiennent des chiffres, extraire des opérateurs et les 2 opérandes autour...

Pour DY, l'aspect multilignes et qu'une partie des préfixes optionnel, complique l'approche de définir le début et la fin et d'appeler combiner récursivement des parseurs comme on ne sait pas facilement où est la fin.

```
exo Dog struct
Consigne très longue

en *Markdown*
sur plusieurs lignes

xp 20
checks
...
```

Snippet 11. – Exemple d'un début d'exercice de code, on voit que la consigne se trouve après la ligne `exo` et continue sur plusieurs lignes jusqu'à qu'on trouve un autre préfixe (ici `xp` qui est optionnel ou alors `checks`). `size`

Les serveurs de langage et bibliothèques Rust existantes

Une part importante du support d'un langage dans un éditeur, consiste en l'intégration des erreurs, l'auto-complétion, les propositions de corrections, des informations au survol... et de nombreuses fonctionnalités qui améliorent la compréhension ou l'interaction. L'avantage d'avoir les erreurs de compilation directement soulignées dans l'éditeur, permet de voir et corriger immédiatement les problèmes sans lancer une compilation manuelle dans une interface séparée.

Contrairement au surlignage de code, ces fonctionnalités demandent une compréhension beaucoup plus fine, ils sont implémentés dans des processus séparés de l'éditeur (aucun langage de programmation n'est ainsi imposé). Ces processus séparés sont appelés des serveurs de langage (language server en anglais). Les éditeurs qui intègre Tree-Sitter développe un client LSP qui se charge de lancer ce serveur, de lancer des requêtes et d'intégrer les données des réponses dans leur interface visuelle.

La communication entre l'éditeur et un serveur de langage démarré pour le fichier en cours, se fait via le `Language Server Protocol (LSP)`. Ce protocole inventé par Microsoft pour VSCode, résoud le problème des développeurs de langages qui doivent supporter chaque éditeur de code indépendamment avec des APIs légèrement différentes pour faire la même chose. Le projet a pour but également de simplifier la vie des nouveaux éditeurs pour intégrer rapidement des dizaines de langages via ce protocole commun et standardisé (35).

```

1 fn main() {
2   let name: &str = " John ".trim();
fn(&self) -> &str

```

Returns a string slice with leading and trailing whitespace removed.

'Whitespace' is defined according to the terms of the Unicode Derived Core Property `White_Space`, which includes newlines.

Examples

```

let s = "\n Hello\tworld\t\n";

```

- `trim()`
- `trim_matches(...)`
- `trim_right_matches(...)`
- `trim_ascii()`
- `trim_ascii_start()`
- `trim_start_matches(...)`
- `trim_ascii_end()`
- `trim_end_matches(...)`
- `trim_start()`
- `trim_end()`

Fig. 7. – Exemple d'auto-complétion dans Neovim, générée par le serveur de langage `rust-analyzer` sur l'appel d'une méthode sur les `&str`

Les points clés du protocole à relever sont les suivants:

- **JSON-RPC** (JSON Remote Procedure Call) est utilisé comme format de sérialisation des requêtes. Similaire au HTTP, il possède des entêtes et un corps. Ce standard définit quelques structures de données à respecter. Une requête doit contenir un champ `jsonrpc`, `id`, `method` et optionnellement `params` (36). Il est possible d'envoyer une notification (requête sans attendre de réponse). Par exemple, le champ `method` va indiquer l'action qu'on tente d'appeler, ici une des fonctionnalités du serveur. Voir Snippet 12
- Un serveur de langage n'a pas besoin d'implémenter toutes les fonctionnalités du protocole. Un système « Capabilities » est défini pour annoncer les méthodes implémentées (37).
- Le transport des messages JSON-RPC peut se faire en `stdio` (flux standard entrée et sorties), sockets TCP ou même en HTTP.


```
Content-Length: ... \r\n
\r\n
{
  "jsonrpc": "2.0",
  "id": 1,
  "method": "textDocument/completion",
  "params": {
    ...
  }
}
```

Snippet 12. – Exemple de requête en JSON-RPC envoyé par le client pour demander des propositions d'auto-complétion à une position de curseur données. Tiré de la spécification (38)

Quelques exemples de serveurs de langages implémentés en Rust

- `tinymist`, serveur de langage de Typst (système d'édition de document, utilisé pour la rédaction de ce rapport)
- `rust-analyzer`, serveur de langage officiel du langage Rust
- `asm-lsp` (39), permet d'inclure des erreurs dans du code assembleur

D'autres exemples de serveurs de langages implémentés dans d'autres langages

- `jdtls` le serveur de langage pour Java implémenté en Java (40)
- `tailwindcss-language-server`, le serveur de langage pour le framework CSS TailwindCSS, implémenté en TypeScript (41)
- `typescript-language-server` et pour finir celui pour TypeScript, implémenté en TypeScript également (42)
- et beaucoup d'autres projets existent...

Une crate commune à plusieurs projets est `lsp-types` (43) qui définit les structures de données, comme `Diagnostic`, `Position`, `Range`. Ce projet est utilisé par `lsp-server`, `tower-lsp` et d'autres (44).

Adoption

Selon la liste sur le site de la spécification (45), la liste des IDE qui supportent le LSP est longue: Atom, Eclipse, Emacs, GoLand, IntelliJ IDEA, Helix, Neovim, Visual Studio, VSCode bien sûr et d'autres. La liste des serveurs LSP (46) quand à elle, contient plus de 200 projets, dont 40 implémentés en Rust! Ce large support et ces nombreux exemples va grandement faciliter le développement de ce serveur de langage et son intégrations dans différents IDE.

Librairies disponibles

En cherchant à nouveau sur `crates.io` sur le tag `lsp`, on trouve différent projets dont `async-lsp` (47) utilisée dans `nil` (48) (un serveur de langage pour le système de configuration de NixOS) et de la même auteure.

Le projet `tinymist` a extrait une crate `sync-ls`, mais le README déconseille son usage et conseille `async-lsp` à la place (49). En continuant la recherche on trouve encore un autre `tower-lsp` et un fork `tower-lsp-server` (50)... `rust-analyzer` a également extrait une crate `lsp-server`.

Choix final

L'auteur travaillant dans Neovim, l'intégration se fera en priorité dans Neovim pour ce travail. L'intégration dans VSCode pourra être fait dans le futur et devrait être relativement simple.

Les 2 projets les plus utilisés (en terme de reverse dependencies sur `crates.io`) sont `lsp-server` (51) (56) et `tower-lsp` (85) (52). L'auteur a choisi d'utiliser la crate `lsp-server` étant développé par la communauté Rust, la probabilité d'une maintenance long-terme est plus élevée, et le projet

`tower-lsp` est basée sur des abstractions asynchrones, l'auteur préfère partir sur la version synchrone pour simplifier l'implémentation.

Cette partie est un nice-to-have, l'auteur espère avoir le temps de l'intégrer dans ce travail. Après quelques heures sur le POC suivant, on voit cela semble être assez facile et la possibilité d'ajouter progressivement le support de fonctionnalités est aussi un atout.

POC sur lsp-server

TODO: bien d'avoir cette section séparée pour ce POC ?

L'auteur a modifié et exécuté l'exemple de `goto_def.rs` fourni par la crate `lsp-server` (53). Il a aussi créé un script `demo.fish` permettant de lancer la communication en stdin et attendre entre chaque requête. Cet exemple minimaliste mais clair démontre la communication qui se produit quand on clique sur un `Aller à la définition` dans un IDE. L'IDE va lancer le serveur de langage associé au fichier édité en lançant simplement le processus et en communication via les flux standards. Il y a d'abord une phase d'initialisation et d'annonces des capacités puis l'IDE peut envoyer des requêtes.

```
CLIENT: Content-Length: 85

{"jsonrpc": "2.0", "method": "initialize", "id": 1, "params": {"capabilities": {}}}
SERVER: Content-Length: 78

{"jsonrpc": "2.0", "id": 1, "result": {"capabilities": {"definitionProvider": true}}}
CLIENT: Content-Length: 59

{"jsonrpc": "2.0", "method": "initialized", "params": {}}

CLIENT: Content-Length: 167

{"jsonrpc": "2.0", "method": "textDocument/definition", "id": 2, "params":
{"textDocument": {"uri": "file:///tmp/test.rs"}, "position": {"line": 7, "character": 23}}}
SERVER: Content-Length: 144

{"jsonrpc": "2.0", "id": 2, "result": [{"range": {"end": {"character": 25, "line": 3}, "start": {"character": 12, "line": 3}}, "uri": "file:///tmp/another.rs"]}]
CLIENT: Content-Length: 67

{"jsonrpc": "2.0", "method": "shutdown", "id": 3, "params": null}
SERVER: Content-Length: 38

{"jsonrpc": "2.0", "id": 3, "result": null}
CLIENT: Content-Length: 54

{"jsonrpc": "2.0", "method": "exit", "params": null}
```

Fig. 8. – Exemple de discussion en LSP une demande de `textDocument/definition`, output de `fish demo.fish` dans le dossier `pocs/lsp-server-demo`.

Les lignes après `CLIENT:` sont envoyés en stdin et celles après `SERVER` sont reçues en stdout.

L'initialisation nous montre que le serveur se présente comme supportant uniquement les « aller à la définition » (go to definition) puisque `definitionProvider` est à `true`. Le client envoie ensuite une demande de `textDocument/definition`, en précisant que celle-ci doit être donnée sur le symbole dans fichier `/tmp/test.rs` sur la ligne 7 au caractère 23.

L'auteur a codé en dur une liste de `Location` (positions dans le code pour cette définition), dans `/tmp/another.rs` sur la `Range` de la ligne 3 du caractère 12 à 25. Une fois la réponse envoyée, le client demande au serveur de s'arrêter.

Le code qui gère cette requête du type `GotoDefinition` se présente ainsi.

```
match cast::<GotoDefinition>(req) {
  Ok((id, params)) => {
    let locations = vec![Location::new(
      Uri::from_str("file:///tmp/another.rs")?,
      Range::new(Position::new(3, 12), Position::new(3, 25)),
    )];
    let result = Some(GotoDefinitionResponse::Array(locations));
    let result = serde_json::to_value(&result).unwrap();
    let resp = Response { id, result: Some(result), error: None };
    connection.sender.send(Message::Response(resp))?;
    continue;
  }
  ...
};
```

Snippet 13. – Extrait de `goto_def.rs` modifié pour retourner un `Location` dans la réponse `GotoDefinitionResponse`

Cette communication permet de visualiser les échanges entre l'IDE et un serveur de langage. En pratique après avoir implémenté une logique de résolution des définitions un peu plus réaliste cette communication ne serait pas visible mais bénéficierait à l'intégration dans l'IDE. Si on l'intégrait dans VSCode, la fonctionnalité du clic droit + Aller à la définition fonctionnerait.

Systèmes de surlignage de code

Les IDEs modernes supportent possèdent des systèmes de surlignage de code (syntax highlighting en anglais) permettant de rendre le code plus lisible en colorisant les mots, caractères ou groupe de symboles de même type (séparateur, opérateur, mot clé du langage, variable, fonction, constante, ...). Ces systèmes se distinguent par leur possibilités d'intégration. Les thèmes intégrés aux IDE peuvent définir directement les couleurs pour chaque type de token. Pour un rendu web, une version HTML contenant des classes CSS spécifiques à chaque type de token peut être générée, permettant à des thèmes écrits en CSS de venir appliquer les couleurs. Les possibilités de génération pour le HTML pour le web implique parfois une génération dans le navigateur ou sur le serveur directement.

Un système de surlignage est très différent d'un parseur. Même s'il traite du même langage, dans un cas, on cherche juste à découper le code en tokens et y définir un type de token. Ce qui s'apparente seulement à la premier étape du lexer/tokenizer généralement rencontré dans les parseurs.

Textmate

Textmate est un IDE pour MacOS qui a inventé un système de grammaire Textmate. Elles permettent de décrire comment tokeniser le code basée sur des expressions régulières. Ces expressions régulières viennent de la librairie C Oniguruma (54). VSCode utilise ces grammaires Textmate (55). IntelliJ IDEA l'utilise également pour les langages non supportés par IntelliJ IDEA comme Swift, C++ et Perl (56).

Exemple de grammaire Textmate permettant de décrire un langage nommé `untitled` avec 4 mots clés et des chaînes de caractères entre guillemets, ceci matché avec des expressions régulières.

```
{ scopeName = 'source.untitled';
  fileTypes = ( );
  foldingStartMarker = '\\{\\s*$';
  foldingStopMarker = '^\\s*\\}';
  patterns = (
    { name = 'keyword.control.untitled';
      match = '\\b(if|while|for|return)\\b';
    },
    { name = 'string.quoted.double.untitled';
      begin = '"';
      end = '"';
      patterns = (
        { name = 'constant.character.escape.untitled';
          match = '\\\\.';
        }
      );
    },
  );
}
```

Snippet 14. – Exemple de grammaire Textmate tiré de leur documentation (57).

La documentation précise un choix important de conception: « A noter que ces regex sont matchées contre une seule ligne à la fois. Cela signifie qu'il n'est pas possible d'utiliser une pattern qui matche plusieurs lignes. La raison est technique: être capable de redémarrer le parseur à une ligne arbitraire et devoir reparser seulement un nombre minimal de lignes affectés par un changement. Dans la plupart des situations, il est possible d'utiliser le model `begin / end` pour dépasser cette limite. » (57) (Traduction personnelle, dernier paragraphe section 12.2).

Tree-Sitter

Tree-Sitter (58) se définit comme un « outil de génération de parser et une librairie de parsing incrémentale. Il peut construire un arbre de syntaxe concret (CST) pour depuis un fichier source et efficacement mettre à jour cet arbre quand le fichier source est modifié. » (58) (Traduction personnelle)

Rédiger une grammaire Tree-Sitter consiste en l'écriture d'une grammaire en Javascript dans un fichier `grammar.js`. Le cli `tree-sitter` va ensuite générer un parseur en C qui pourra être utilisé directement via le CLI `tree-sitter` durant le développement et être facilement embarquée comme librairie C sans dépendance dans n'importe quelle type d'application (58, 59).

Tree-Sitter est supporté dans Neovim (60), dans le nouvel éditeur Zed (61), ainsi que d'autres. Tree-Sitter a été inventé par l'équipe derrière Atom (62) et est même utilisé sur GitHub, notamment pour la navigation du code pour trouver les définitions et références et lister tous les symboles (fonctions, classes, structs, etc) (63).

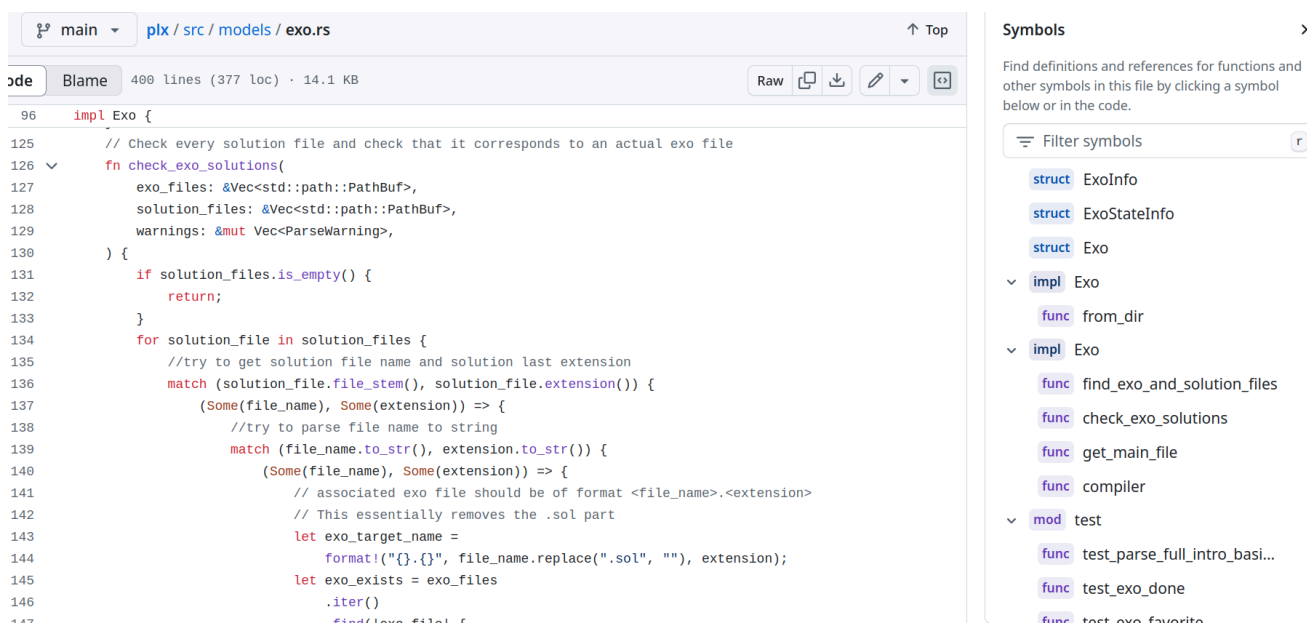


Fig. 9. – Liste de symboles générées par Tree-Sitter, affichés à droite du code sur GitHub pour un exemple de code Rust de PLX

Semantic highlighting

Le surlignage sémantique est une extension du surlignage syntaxique. Les serveurs de langage peuvent ainsi fournir des tokens sémantiques qui apportent une classification plus fine du langage, que les systèmes syntaxiques ne peuvent pas détecter. (64)

Without semantic highlighting:

```

9 | function getFoldingRanges(languageModes: LanguageModes, document: TextDocument): FoldingRange[] {
10 |     let htmlMode = languageModes.getMode('html');
11 |     let range = Range.create(Position.create(0, 0), Position.create(document.lineCount, 0));
12 |     let result: FoldingRange[] = [];
13 |     if (htmlMode && htmlMode.getFoldingRanges) {
14 |         result.push(...htmlMode.getFoldingRanges(document));
15 |     }

```

With semantic highlighting:

```

9 | function getFoldingRanges(languageModes: LanguageModes, document: TextDocument): FoldingRange[] {
10 |     let htmlMode = languageModes.getMode('html');
11 |     let range = Range.create(Position.create(0, 0), Position.create(document.lineCount, 0));
12 |     let result: FoldingRange[] = [];
13 |     if (htmlMode && htmlMode.getFoldingRanges) {
14 |         result.push(...htmlMode.getFoldingRanges(document));
15 |     }

```

Fig. 10. – Exemple tiré de la documentation de VSCode, démontrant quelques améliorations dans le surlignage. Les paramètres `languageModes` et `document` sont colorisés différemment que les variables locales. `Range` et `Position` sont colorisées comme des classes.

`getFoldingRanges` dans la condition est colorisée en tant que fonction ce qui la différencie des autres propriétés. (64)

En voyant la liste des tokens sémantiques possible dans la spécification LSP (65), on comprend mieux l'intérêt et les possibilités de surlignage avancé. Par exemple, on trouve des tokens `macro`, `regexp`, `typeParameter`, `interface`, `enum`, `enumMember`, qui seraient difficile de différencier durant la tokenisation mais qui peuvent être surligné différemment pour mettre en avant leur différence sémantique.

Sur le Snippet 15 surligné ici uniquement grâce à Tree-Sitter (sans surlignage sémantique) on voit que les appels de `HEY` et `hi` dans le `main` ont les mêmes couleurs alors que l'un est une macro, l'autre une fonction. En effet, à l'appel, il n'est pas possible de les différencier, ce n'est que le contexte plus large que seul le serveur de langage possède, qu'on peut déterminer cette différence.

```

#include <stdio.h>

const char *HELLO = "Hey";
#define HEY(name) printf("%s %s\n", HELLO, name)
void hi(char *name) { printf("%s %s\n", HELLO, name); }

int main(int argc, char *argv[]) {
    hi("Samuel");
    HEY("Samuel");
    return 0;
}

```

Snippet 15. – Exemple de code C `hello.c`, avec macro et fonction surligné de la même manière à l'appel

Sur le Snippet 16, on voit que les 2 lignes `hi` et `HEY` sont catégorisés sans surprise comme des fonctions (noeuds `function`, `arguments`, ...).

```
(expression_statement ; [7, 4] - [7, 17]
  (call_expression ; [7, 4] - [7, 16]
    function: (identifieur) ; [7, 4] - [7, 6]
    arguments: (argument_list ; [7, 6] - [7, 16]
      (string_literal ; [7, 7] - [7, 15]
        (string_content)))) ; [7, 8] - [7, 14]
  (expression_statement ; [8, 4] - [8, 18]
    (call_expression ; [8, 4] - [8, 17]
      function: (identifieur) ; [8, 4] - [8, 7]
      arguments: (argument_list ; [8, 7] - [8, 17]
        (string_literal ; [8, 8] - [8, 16]
          (string_content)))) ; [8, 9] - [8, 15]
```

Snippet 16. – Aperçu de l'arbre syntaxique concret généré par Tree-Sitter
récupéré via `tree-sitter parse hello.c`

Si on inspecte l'état de l'éditeur, on peut voir qu'au delà des tokens générés par Tree-Sitter, le serveur de langage (`clangd` ici), a réussi à préciser la notion de macro au-delà du simple appel de fonction.

```
Semantic Tokens
- @lsp.type.macro.c links to PreProc    priority: 125
- @lsp.mod.globalScope.c links to @lsp  priority: 126
- @lsp.type.mod.macro.globalScope.c links to @lsp  priority: 127
```

Snippet 17. – Extrait de la commande `:Inspect` dans Neovim avec le curseur sur le `HEY`

Ainsi dans Neovim une fois `clangd` lancé, l'appel de `HEY` prend ainsi la même couleur que celle attribuée sur sa définition.

Choix final

L'auteur a ignoré l'option du système de SublimeText. pour la simple raison qu'il n'est supporté nativement que dans SublimeText, probablement parce que cet IDE est propriétaire (66). Ce système utilisent des fichiers `.sublime-syntax`, qui ressemble à TextMate (67) mais rédigé en YAML.

Si le temps le permet, une grammaire sera développée avec Tree-Sitter pour supporter du surlignage dans Neovim.

Le choix de ne pas explorer plus les grammaires Textmate, laisse penser que l'auteur du travail délaisse complètement VSCode. Ce qui paraît étonnant comme VSCode est régulièrement utilisé par 73% des 65,437 répondants au sondage de StackOverflow 2024 (68).

Cette décision se justifie notamment par la roadmap de VSCode: entre mars et mai 2025 (69, 70), du travail d'investigation autour de Tree-Sitter a été fait pour explorer les grammaires existantes et l'usage de surlignage de code dans VSCode (71). Des premiers efforts d'exploration avait d'ailleurs déjà eu lieu en septembre 2022 (72).

L'usage du Semantic highlighting n'est pas au programme de ce travail mais pourra être exploré dans le futur si certains éléments sémantiques pourraient en bénéficier.

POC de Tree-Sitter

Ce POC vise à prouver que l'usage de Tree-Sitter fonctionne pour coloriser les préfixes et les propriétés de Snippet 18 pour ne pas avoir cette affichage noir sur blanc qui ne facilite pas la lecture.

```
// Basic MCQ exo
exo Introduction

opt .multiple
- C is an interpreted language
- .ok C is a compiled language
- C is mostly used for web applications
```

Snippet 18. – Un exemple de question choix multiple dans un fichier `mcq.dy`, décrite avec la syntaxe DY. Les préfixes sont `exo` (titre) et `opt` (options). Les propriétés sont `.ok` et `.multiple`.

Une fois la grammaire mise en place avec la commande `tree-sitter init`, il suffit de remplir le fichier `grammar.js`, avec une ensemble de règle construites via des fonctions fournies par Tree-Sitter et des expressions régulières. `seq` indique une liste de tokens qui viendront en séquence, `choice` permet de tester plusieurs options à la même position. On remarque également la liste des préfixes et propriétés insérés dans les tokens de `prefix` et `property`. La documentation **The Grammar DSL** de la documentation explique toutes les options possibles en détails (73).

```
module.exports = grammar({
  name: "dy",
  rules: {
    source_file: ($) => repeat($_line),
    _line: ($) =>
      seq( choice($.commented_line, $.prefixed_line, $.list_line, $.content_line), "\n"),
    prefixed_line: ($) =>
      seq($.prefix, optional(repeat($.property)), optional(seq(" ", $.content))),
    commented_line: (_) => token(seq(/\V\V /, /.+/)),
    list_line: ($) =>
      seq($.dash, repeat($.property), optional(" "), optional($.content)),
    dash: (_) => token(prec(2, /- /)),
    prefix: (_) => token(prec(1, choice("exo", "opt"))),
    property: (_) => token(prec(3, seq(".", choice("multiple", "ok")))),
    content_line: ($) => $.content,
    content: (_) => token(prec(0, /.+/)),
  },
});
```

Snippet 19. – Résultat de la grammaire minimaliste `grammar.js`, définissant un ensemble de règles sous `rules`.

On observe dans le Snippet 19 plusieurs règles:

- `source_file` : décrit le point d'entrée d'un fichier source, défini comme une répétition de ligne.
- `_line` : une ligne est une séquence d'un choix entre 4 types de lignes qui sont chacune décrites en dessous et un retour à la ligne
- `prefixed_line` : une ligne préfixée consiste en séquence de token composé d'un préfix, puis optionnellement d'un ou plusieurs propriétés. Elle se termine optionnellement par un contenu qui commence après un premier espace
- `commented_line` définit les commentaires comme `//` puis un reste
- `list_line`, `dash` et le reste des règles suivent la même logique de définition

Après avoir appelé `tree-sitter generate` pour générer le code du parser C et `tree-sitter build` pour le compiler, on peut demander au CLI de parser un fichier donné et afficher le CST. Dans cet arbre qui démarre avec son noeud racine `source_file`, on y voit les noeuds du même type que les règles définies précédemment, avec le texte extrait dans la plage de caractères associée au noeud. Par exemple, on voit que l'option `C is a compiled language` a bien été extraite à la ligne 5.

entre le byte 6 et 30 (5:6 - 5:30) en tant que `content` . Elle suit un token de `property` avec notre flag `.ok` et le tiret de la règle `dash` .

```
dy> tree-sitter parse -c mcq.dy
0:0 - 7:0      source_file
0:0 - 0:16     commented_line `// Basic MCQ exo`
0:16 - 1:0     "\n"
1:0 - 1:16    prefixed_line
1:0 - 1:3     prefix `exo`
1:3 - 1:4     " "
1:4 - 1:16    content `Introduction`
1:16 - 3:0     "\n"
3:0 - 3:13    prefixed_line
3:0 - 3:3     prefix `opt`
3:3 - 3:13    property `.multiple`
3:13 - 4:0     "\n"
4:0 - 4:30    list_line
4:0 - 4:2     dash `-`
4:2 - 4:30    content `C is an interpreted language`
4:30 - 5:0     "\n"
5:0 - 5:30    list_line
5:0 - 5:2     dash `-`
5:2 - 5:5     property `.ok`
5:5 - 5:6     " "
5:6 - 5:30    content `C is a compiled language`
5:30 - 6:0     "\n"
6:0 - 6:39    list_line
6:0 - 6:2     dash `-`
6:2 - 6:39    content `C is mostly used for web applications`
6:39 - 7:0     "\n"
```

Fig. 11. – Concrete Syntax Tree généré par la grammaire définie sur le fichier `mcq.dy`

La tokenisation fonctionne bien pour cet exemple, chaque élément est correctement découpé et catégorisé. Pour voir ce snippet en couleurs, il nous reste deux choses à définir. La première consiste en un fichier `queries/highlighting.scm` qui décrit des requêtes de surlignage sur l'arbre (highlights query) permettant de sélectionner des noeuds de l'arbre et leur attribuer un nom de surlignage (highlighting name). Ces noms ressemblent à `@variable`, `@constant`, `@function`, `@keyword`, `@string` etc... et des versions plus spécifiques comme `@string.regexp`, `@string.special.path`. Ces noms sont ensuite utilisés par les thèmes pour appliquer un style.

```
(prefix) @keyword
(commented_line) @comment
(content) @string
(property) @property
(dash) @operator
```

Snippet 20. – Aperçu du fichier `queries/highlights.scm`

Le CLI supporte directement la configuration d'un thème via son fichier de configuration, on reprend simplement chaque nom de surlignage en lui donnant une couleur.

```
{
  "parser-directories": [ "/home/sam/code/tree-sitter-grammars" ],
  "theme": {
    "property": "#1bb588",
    "operator": "#20a8c3",
    "string": "#1f2328",
    "keyword": "#20a8c3",
    "comment": "#737a7e"
  }
}
```

Snippet 21. – Contenu du fichier de configuration de Tree-Sitter présent sur Linux au chemin `~/.config/tree-sitter/config.json`

```
// Basic MCQ exo
exo Introduction

opt .multiple
- C is an interpreted language
- .ok C is a compiled language
- C is mostly used for web applications
```

Fig. 12. – Screenshot du résultat de la commande

`tree-sitter highlight mcq.dy` avec notre exercice surligné

L'auteur de ce travail s'est inspiré de l'article **How to write a tree-sitter grammar in an afternoon** (Ben Siraphob) (74) pour ce POC. Le résultat de ce POC est encourageant, même s'il faudra probablement plus que quelques heures pour gérer les détails, comprendre, tester et documenter l'intégration dans Neovim, cette partie n'a pas de chances de pouvoir être réalisée dans ce travail au vu du résultat atteint avec ce POC.

Le semantic highlighting pourrait être utile en attendant l'intégration de Tree-Sitter dans VSCode. L'extension `tree-sitter-vscode` en fait déjà une intégration avec cette approche, qui est beaucoup plus lente qu'une intégration native mais qui fonctionne. A noter que l'extension n'est pas triviale à installer et configurer, qu'on peut considérer son usage encore expérimental. Elle nécessite d'avoir un build WASM de notre parser Tree-Sitter (75).

```
≡ mcq.dy
1  // Basic MCQ exo
2  exo Introduction
3
4  opt .multiple
5  - C is an interpreted language
6  - .ok C is a compiled language
7  - C is mostly used for web applications
8
9
```

Fig. 13. – Screenshot dans VSCode une fois l'extension `tree-sitter-vscode` configuré, le surlignage est fait via notre syntaxe Tree-Sitter via

Protocoles de synchronisation et formats de sérialisation existants

Le serveur de gestion de sessions live a besoin d'un système de communication bidirectionnelle en temps réel, afin de transmettre le code et les résultats des étudiants. Ces messages seront transformées dans un format standard, facile à sérialiser et désérialiser en Rust. Cette section explore les formats textuels et binaires disponibles, ainsi que les protocoles de communication bidirectionnelle.

JSON

Contrairement à toutes les critiques relevées précédemment sur le JSON et d'autres formats, dans leur usage en tant que format source, JSON est une option solide pour la communication entre clients-serveurs. Le format JSON est très populaire pour les APIs REST, les fichiers de configuration, et d'autres usages.

En Rust, avec `serde_json`, il est plutôt facile de parser du JSON dans une `struct`. Une fois la macro `Deserialize` appliquée, on peut directement appeler `serde_json::from_str(json_data)`.

```
use serde::{Deserialize, Serialize};
use serde_json::Result;

#[derive(Serialize, Deserialize)]
struct Person {
    name: String,
    age: u8,
    phones: Vec<String>,
}
// ...
let data = r#" {
    "name": "John Doe",
    "age": 43,
    "phones": [ "+44 1234567", "+44 2345678" ]
} "#;
let p: Person = serde_json::from_str(data).unwrap();
println!("Please call {} at the number {}", p.name, p.phones[0]);
```

Snippet 22. – Exemple simplifié de parsing de JSON, tiré de leur documentation (76).

```
use serde_json::json;

fn main() {
    // The type of `john` is `serde_json::Value`
    let john = json!({
        "name": "John Doe",
        "age": 43,
        "phones": [ "+44 1234567", "+44 2345678" ]
    });
    println!("first phone number: {}", john["phones"][0]);
    println!("{}", john.to_string());
}
```

Snippet 23. – Autre exemple de sérialisation vers JSON d'une structure arbitraire.
Egalement tiré de leur documentation (77).

Protocol Buffers - ProtoBuf

Parmi les formats binaires, on trouve ProtoBuf, un format développé par Google pour sérialiser des données structurées, de manière compacte, rapide et simple. L'idée est de définir un schéma dans

un style non spécifique à un langage de programmation, puis de génération automatiquement du code pour interagir avec ces structures depuis du C++, Java, Go, Ruby, C# et d'autres. (78)

```
edition = "2023";

message Person {
  string name = 1;
  int32 id = 2;
  string email = 3;
}
```

Snippet 24. – Un simple exemple de description d'une personne en ProtoBuf tiré de leur site web (78).

```
Person john = Person.newBuilder()
    .setId(1234)
    .setName("John Doe")
    .setEmail("jdoe@example.com")
    .build();
output = new FileOutputStream(args[0]);
john.writeTo(output);
```

Snippet 25. – Et son usage en Java avec les classes autogénérées à la compilation tiré de leur site web (78).

Le langage Rust n'est pas officiellement supporté mais un projet du nom de PROST! existe (79) et permet de générer du code Rust depuis des fichiers Protobuf.

MessagePack

Le slogan de MessagePack, format binaire de sérialisation: « C'est comme JSON, mais rapide et léger » (Traduction personnelle). Une implémentation en Rust du nom de RPM existe (80).

Websocket

Le protocole Websocket, définie dans la RFC 6455 (81), permet une communication bi-directionnelle entre un client et un serveur. A la place de l'approche de requête-réponses du HTTP, le protocole Websocket définit une manière de garder une connexion TCP ouverte et un moyen d'envoyer des messages dans les 2 sens. On évite ainsi d'ouvrir plusieurs connexions HTTP, une nouvelle à chaque fois qu'un événement se produit ou que le client veut vérifier si le serveur n'a pas d'événements à transmettre. La technologie a été pensée pour être utilisée par des applications dans les navigateurs, mais fonctionne également en dehors (81).

La section **1.5 Design Philosophy** explique que le protocole est conçu pour un « minimal framing » (encadrement minimal autour des données envoyées), juste assez pour permettre de découper le flux TCP en frame (en message d'une durée définie) et de distinguer le texte des données binaires. Le texte doit être encodé en UTF-8. (82)

La section **1.3. Opening Handshake**, nous explique que pour permettre une compatibilité avec les serveurs HTTP et intermédiaires sur le réseau, l'opening handshake (l'initialisation du socket une fois connecté) est compatible avec le format des entêtes HTTP. Cela permet d'utiliser un serveur websocket sur le même port qu'un serveur web, ou d'héberger plusieurs serveurs websocket sur différents routes par exemple `/chat` et `/news`. (83)

Dans l'écosystème Rust, il existe plusieurs crate qui implémente le protocole, parfois côté client, côté serveur ou les deux. Il existe plusieurs approches sync et async, nous nous concentrons ici sur une approche sync avec gestion des threads natifs manuelle pour simplifier l'implémentation et les recherches.

La crate `tungstenite` propose une abstraction du protocole qui permet de facilement interagir avec des `Message`, leur écriture `send()` et leur lecture `read()` de façon très simple (84). Elle passe la Autobahn Test Suite (suite de tests de plus de 500 cas) (85).

```
use std::net::TcpListener;
use std::thread::spawn;
use tungstenite::accept;

/// A WebSocket echo server
fn main () {
    let server = TcpListener::bind("127.0.0.1:9001").unwrap();
    for stream in server.incoming() {
        spawn (move || {
            let mut websocket = accept(stream.unwrap()).unwrap();
            loop {
                let msg = websocket.read().unwrap();

                // We do not want to send back ping/pong messages.
                if msg.is_binary() || msg.is_text() {
                    websocket.send(msg).unwrap();
                }
            }
        });
    }
}
```

Snippet 26. – Exemple de serveur echo en WebSocket avec la crate `tungstenite`. Tiré de leur README (84)

Une version async pour le runtime tokio existe également, elle s'appelle `tokio-tungstenite`, si le besoin de passer à un modèle async avec Tokio se fait sentir, nous devrions pouvoir y migrer (86).

Il existe une crate `websocket` avec une approche sync et async, qui est dépréciée et dont le README (87) conseille l'usage de `tungstenite` ou `tokio-tungstenite` à la place (87).

A noter qu'il existe d'autres crates tel que `fastwebsockets` (88) à disposition, qui ont l'air de permettre de travailler à un plus bas niveau. Pour faciliter l'implémentation nous les ignorons pour ce travail.

gRPC

gRPC est un protocole basé sur ProtoBuf, inventé par Google. Il se veut être un système de Remote Procedure Call (RPC - un système d'appel de fonctions à distance), universelle et performant qui supporte le streaming bi-directionnelle sur HTTP2. La possibilité de travailler avec plusieurs langages reposent sur la génération automatique de code pour les clients et serveurs permettant de gérer la sérialisation en Protobuf et gérant le transport.

En plus des définitions des messages en Protobuf déjà présentée, il est possible de définir des services, avec des méthodes avec un type de message et un type de réponse.

```
// The greeter service definition.
service Greeter {
    // Sends a greeting
    rpc SayHello (HelloRequest) returns (HelloReply) {}
}

// The request message containing the user's name.
message HelloRequest {
    string name = 1;
}

// The response message containing the greetings
message HelloReply {
    string message = 1;
}
```

Snippet 27. – Exemple de fichier .proto définissant 2 messages et un service permettant d'envoyer un nom et de recevoir des salutations en retour. Tiré de leur documentation d'introduction (89)

Comme Protobuf, Rust n'est pas supporté officiellement mais une implémentation du nom de Tonic existe (90), elle utilise Prost! mentionnée précédemment pour l'intégration de Protobuf.

Un article de 2019, intitulé **The state of gRPC in the browser** (91) montre que l'utilisation de gRPC dans les navigateurs web est encore malheureusement mal supportée. En résumé, « il est actuellement impossible d'implémenter la spécification HTTP/2 gRPC dans le navigateur, comme il n'y a simplement pas d'API de navigateur avec un contrôle assez fin sur les requêtes. » (Traduction personnelle). La solution à été trouvée à ce problème est le projet gRPC-Web qui fournit un proxy entre le navigateur et le serveur gRPC, faisant les conversions nécessaires entre gRPC-Web et gRPC.

Il reste malheureusement plusieurs limites: le streaming bi-directionnelle n'est pas possible, le client peut faire des appels unaires (pour un seul message) et peut écouter une server-side streams (flux de messages venant du server). L'autre limite est le nombre maximum de connexions en streaming simultanées dans un navigateur sur HTTP/1.1 fixées à 6 (92), ce qui demande de restructurer ses services gRPC pour ne pas avoir plus de 6 connexions en server-side streaming à la fois.

tarpc

tarpc également développé sur l'organisation GitHub de Google sans être un produit officiel, se définit comme « un framework RPC pour Rust, avec un focus sur la facilité d'utilisation. Définir un service peut être fait avec juste quelques lignes de code et le code boilerplate du serveur est géré pour vous. » (Traduction personnelle) (93)

tarpc est différent de gRPC et Cap'n Proto « en définissant le schéma directement dans le code, plutôt que dans un langage séparé comme Protobuf. Ce qui signifie qu'il n'y a pas de processus de compilation séparée et pas de changement de contexte entre différent langages. » (Traduction personnelle) (93)

Choix final

Par soucis de facilité de debug, d'implémentation et d'intégration, l'auteur a choisi de rester sur un format textuel et d'implémenter la sérialisation en JSON via la crate mentionnée précédemment `serde_json`. L'expérience existante des websockets de l'auteur, sa possibilité de choisir le format de données, et son solide support dans les navigateurs (au cas où PLX avait une version web un jour), font que ce travail utilisera la combinaisons Websockets + JSON.

gRPC aurait pu aussi être une option comme PLX est en dehors du navigateur, il ne serait pas touché par les limites exprimées. Cependant, cela rendrait plus difficile un support d'une version web de PLX si le projet en avait besoin dans le futur.

Quand l'usage de PLX dépassera une dizaines/centaines d'étudiants connectés en même moment et que la latence sera trop forte ou que les coûts d'infrastructures deviendront un soucis, les formats binaires plus légers seront une option à creuser. Au vu des nombreux choix, mesurer la taille des messages, la latence de transport et le temps de sérialisation sera important pour faire un choix. D'autres projets pourraient également être considéré comme Cap'n Proto (94) qui se veut plus rapide que Protobuf, ou encore Apache Thrift (95). Ces dernières options n'ont pas été explorée dans cet état de l'art principalement parce qu'elles proposent un format binaire.

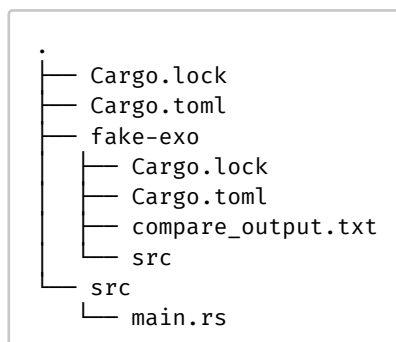
POC de synchronisation de messages JSON via websockets avec tungstenite

Pour vérifier la faisabilité technique d'envoyer des messages en temps réel en Rust via websockets, un petit POC a été développé dans le dossier `pocs/websockets-json`. Le code et les résultats des checks doivent être transmis des étudiants depuis le client PLX des étudiants vers ce lui de l'enseignant, en passant par le serveur de session live.

De part sa nature interactive, il n'est pas évident de retranscrire ce qui s'y passe quand on lance le POC dans 3 shell côte à côte, le mieux serait d'aller compiler et lancer à la main. Nous documentons ici un aperçu du résultat.

Ce petit programme en Rust prend en argument son rôle (`server` , `teacher` ou `student`), tout le code est ainsi dans un seul fichier `main.rs` et un seul binaire.

Ce programme a la structure suivante, le dossier `fake-exo` contient l'exercice à implémenter.



Snippet 28. – Structure de fichiers du POC.

```
// Just print "Hello <name> !" where <name> comes from argument 1
fn main() {
    println!("Hello, world!");
}
```

Snippet 29. – Code Rust de départ de l'exercice fictif à compléter par l'étudiant

Le mini protocole définit pour permettre cette synchronisation est découpé en 2 étapes.

Annnonce des clients

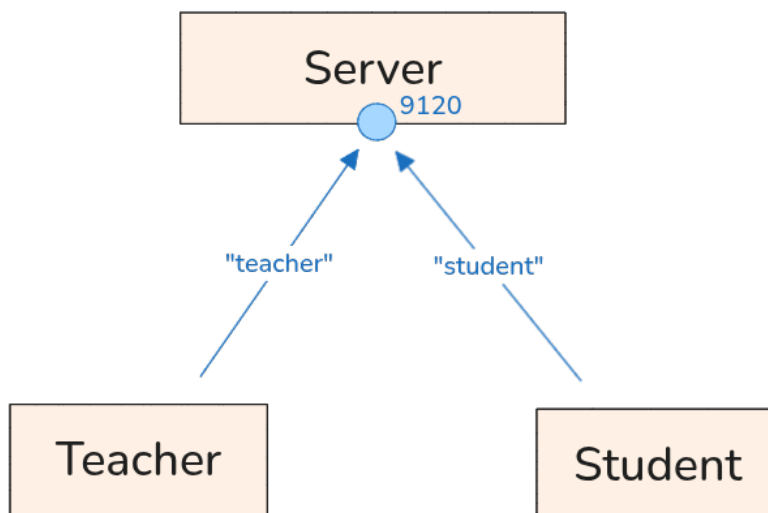


Fig. 14. – La première partie consiste en une mise en place par la connexion et l'annonce des clients de leur rôle, en se connectant puis en envoyant leur rôle en string.

Transfert des résultats des checks

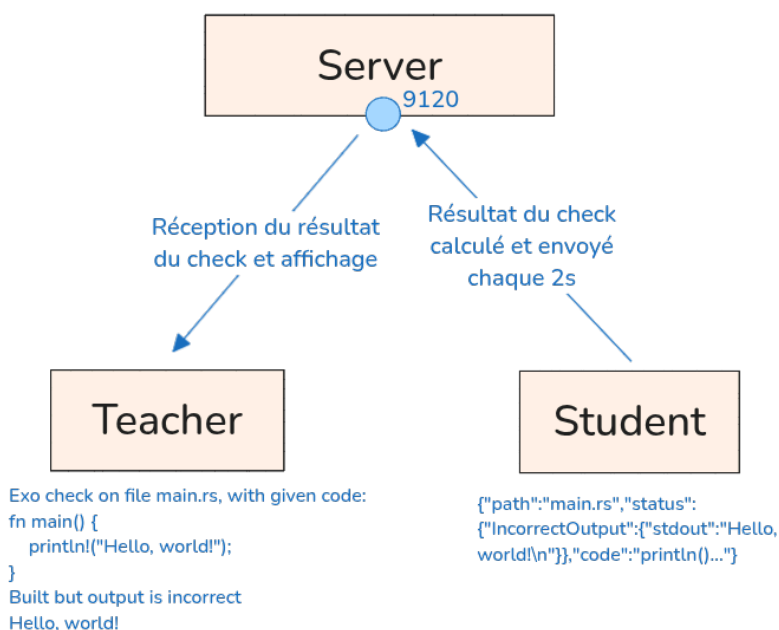


Fig. 15. – La deuxième partie consiste en l'envoi régulier du client du résultat du check vers le serveur, qui ne fait que de transmettre au socket associé au `teacher`.

Dans un premier shell (S1), nous lançons en premier lieu le serveur:

```
websockets-json> cargo run -q server
Starting server process ...
Server started on 127.0.0.1:9120
```

Snippet 30. – Lancement du serveur et attente de connexions sur le port 9120.

Dans un deuxième shell (S2), on lance le `teacher`:


```
websockets-json> cargo run -q teacher
Starting teacher process ...
Sending whoami message
Waiting on student's check results
```

Snippet 31. – Lancement du `teacher`, connexion au serveur et envoi d'un premier message littéral `teacher` pour annoncer son rôle

Dans S1, on voit que le serveur a bien reçu la connexion et a détecté le rôle de `teacher`.

```
...
Teacher connected, saved associated socket.
```

Snippet 32. – `teacher` est bien connecté au serveur

Dans S3, on lance finalement le rôle de l'étudiant:

```
websockets-json> cargo run -q student
Starting student process ...
Sending whoami message
Starting to send check's result every 2000 ms
Sending another check result
{"path":"fake-exo/src/main.rs","status":{"IncorrectOutput":{"stdout":"Hello, world!\n"}},
"code":"// Just print \"Hello <name> !\" where <name> comes from argument 1\nfn main()
{\n    println!(\"Hello, world!\");\n}\n"}\n
```

Snippet 33. – Le processus `student` compile et exécute le check, afin d'envoyer le résultat, ici du type `IncorrectOutput`.

Le Snippet 34 nous montre le détails de ce message.

```
{
  "path": "fake-exo/src/main.rs",
  "status": {
    "IncorrectOutput": {
      "stdout": "Hello, world!\n"
    }
  },
  "code": "// Just print \"Hello <name> !\" where <name> comes from argument 1\nfn main()
{\n    println!(\"Hello, world!\");\n}\n"
}
```

Snippet 34. – Le message envoyé avec un chemin de fichier, le code et le statut. Le statut est une enum définie à « output incorrect », puisque l'exercice n'est pas encore implémenté.

Le serveur sur le S1, on ne voit que le `Forwarded one message to teacher`. Sur le S2, on voit immédiatement ceci:

```
Exo check on file fake-exo/src/main.rs, with given code:
// Just print "Hello <name> !" where <name> comes from argument 1
fn main() {
    println!("Hello, world!");
}
Built but output is incorrect
Hello, world!
```

Snippet 35. – Le `teacher` a bien reçu le message et peut l'afficher, la synchronisation temps réel a fonctionné.

Si l'étudiant introduit une erreur de compilation, un message avec un statut différent est envoyé, voici ce que reçoit le `teacher` :

```
Exo check on file fake-exo/src/main.rs, with given code:
// Just print "Hello <name> !" where <name> comes from argument 1
fn main() {
    println!("Hello, world!", args[3]);
}
failed build with error
  Compiling fake-exo v0.1.0
error: argument never used
  → src/main.rs:3:31
   |
3 |     println!("Hello, world!", args[3]);
   |                                ^^^^^^^ argument never used
   |                                |
   |                                formatting specifier missing
```

Snippet 36. – Le `teacher` a bien reçu le code actuel avec l'erreur et l'output de compilation de Cargo

Le système de synchronisation en temps réel permet ainsi d'envoyer différents messages au serveur qui le retransmet directement au `teacher`. Même si cet exemple est minimale puisqu'il ne vérifie pas la source des messages, et qu'il n'y a qu'un seul étudiant et enseignant impliqué, nous avons démontré que la crate `tungstenite` fonctionne.

Architecture

Implémentation

TODO

Conclusion

TODO

Bibliographie

1. rustlings. Online. 2025. Available from: <https://rustlings.rust-lang.org/>
2. PLX. Project status - PLX docs. Online. 2025. Available from: <https://plx.rs/book/status.html>
3. GitHub - danwritecode/clings: rustlings for C....clings. Online. 2025. Available from: <https://github.com/danwritecode/clings>
4. CONTRIBUTEURS. GitHub - mauricioabreu/golings": rustlings but for go lang this time. Online. 2025. Available from: <https://github.com/mauricioabreu/golings>
5. ziglings/exercises: Learn the ⚡ Zig programming language by fixing tiny broken programs. - Codeberg.org. Online. 2025. Available from: <https://codeberg.org/ziglings/exercises>
6. GitHub - MondayMorningHaskell/haskellings: An automated tutorial to teach you about Haskell!. Online. 2025. Available from: <https://github.com/MondayMorningHaskell/haskellings>
7. DUBOVSKOY, Alexey. Cooklang - Recipe Markup Language. Online. 2025. Available from: <https://cooklang.org/>
8. DUBOVSKOY, Alexey. Canonical Cooklang parser in Rust. Online. 2025. Available from: <https://github.com/cooklang/cooklang-rs>
9. RON, Contributeurs de. Rusty Object Notation. Online. 2025. Available from: <https://github.com/ron-rs/ron>
10. TORM. udl v0.3.1 - Parser for UDL (Universal Data Language). Online. 2023. Available from: <https://crates.io/crates/udl>
11. TORM. The Khi data language. Online. 2024. Available from: <https://github.com/khilang/khi>
12. TORM. Rust Khi parser & library. Online. 2024. Available from: <https://github.com/khilang/khi.rs>
13. ASSOCIATION. bitmark Association website. Online. 2025. Available from: <https://www.bitmark-association.org/>
14. ASSOCIATION. bitmark Hackathon. Online. 2025. Available from: <https://www.bitmark-association.org/bitmarkhackathon>
15. ASSOCIATION. bitmark Documentation. Online. 2025. Available from: <https://docs.bitmark.cloud/>
16. ASSOCIATION. Quizzes - .multiple-choice, .multiple-choice-1. Online. 2025. Available from: <https://docs.bitmark.cloud/quizzes/#multiple-choice-multiple-choice-1>
17. TASKBASE. open-taskpool - 12,000 UK 🇬🇧 → DE 🇩🇪 & DE 🇩🇪 → EN 🇬🇧 learning tasks ready for you to use. Online. 2025. Available from: <https://github.com/taskbase/open-taskpool>
18. ASSOCIATION. Quizzes - .cloze (gap text). Online. 2025. Available from: <https://docs.bitmark.cloud/quizzes/#cloze-gap-text>
19. CLASSTIME. Créer la première question / le premier jeu de questions. Online. 2024. Available from: <https://help.classtime.com/fr/comment-commencer-a-utiliser-classtime/creer-la-premiere-question-le-premier-jeu-de-questions>
20. KUNDERT, Ken. NestedText - A Human Friendly Data Format. Online. 2025. Available from: <https://github.com/KenKundert/nestedtext>

-
21. KEN et KUNDERT, Kale. NestedText documentation - Schemas. Online. 2025. Available from: <https://nestedtext.org/en/latest/schemas.html>
 22. BOB22Z. docs.rs - Crate nestedtext. Online. 2025. Available from: <https://nestedtext/latest/nestedtext/>
 23. LUDWIG, Sönke. SDLang, Simple Declarative Language. Online. 2025. Available from: <https://sdlang.org/>
 24. KAT MARCHÁN (ZKAT), et contributeurs. KDL, a cudlly document language. Online. 2025. Available from: <https://kdl.dev/>
 25. All Crates for keyword 'parser'. Online. 2025. Available from: <https://crates.io/keywords/parser>
 26. (EPAGE), Ed Page. winnow v0.7.8 A byte-oriented, zero-copy, parser combinators library. Online. 2025. Available from: <https://crates.io/crates/winnow>
 27. Dependencies of kdl crate. Online. 2025. Available from: <https://crates.io/crates/kdl/6.3.4/dependencies>
 28. (GEAL), Geoffroy Couprie. nom v8.0.0 A byte-oriented, zero-copy, parser combinators library. Online. 2025. Available from: <https://crates.io/crates/nom>
 29. Reverse dependencies of nom crate. Online. 2025. Available from: https://crates.io/crates/nom/reverse_dependencies
 30. pest v2.8.0 The Elegant Parser. Online. 2025. Available from: <https://crates.io/crates/pest>
 31. (MARWES), Markus Westerlind. combine v4.6.7 Fast parser combinators on arbitrary streams with zero-copy support. Online. 2024. Available from: <https://crates.io/crates/combine>
 32. JOSHUA BARRETTO (ZESTERER), Rune Tynan (CraftSpider), et contributeurs. chumsky v0.10.1 A parser library for humans with powerful error recovery. Online. 2025. Available from: <https://crates.io/crates/chumsky>
 33. Most popular Rust libraries. Online. 2025. Available from: <https://lib.rs/std>
 34. Serde data model. Online. 2025. Available from: <https://serde.rs/data-model.html>
 35. MICROSOFT, et contributeurs. Language Server Protocol. Online. 2025. Available from: <https://microsoft.github.io/language-server-protocol/>
 36. GROUP, JSON-RPC Working. JSON-RPC 2.0 Specification. Online. 2013. Available from: <https://www.jsonrpc.org/specification>
 37. MICROSOFT, et contributeurs. Language Server Protocol Specification - 3.17 - Capabilities. Online. 2025. Available from: <https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/#capabilities>
 38. MICROSOFT, et contributeurs. Language Server Protocol Specification - 3.17 - Content part. Online. 2025. Available from: <https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/#contentPart>
 39. BERGERCOOKIE, et contributeurs. asm-lsp v0.10.0 Language Server for x86/x86_64, ARM, RISCv, and z80 Assembly Code. Online. 2025. Available from: <https://crates.io/crates/asm-lsp>
 40. ORGANISATION, et contributeurs eclipse-jdtls. GitHub - eclipse-jdtls/eclipse.jdt.ls: Java language server. Online. 2025. Available from: <https://github.com/eclipse-jdtls/eclipse.jdt.ls>
 41. TAILWINDLABS, et contributeurs. GitHub - tailwindlabs/tailwindcss-intellisense: Intelligent Tailwind CSS tooling for Visual Studio Code. Online. 2025. Available from: <https://github.com/tailwindlabs/tailwindcss-intellisense>
 42. ORGANISATION, et contributeurs typescript-language-server. GitHub - typescript-language-server/typescript-language-server: TypeScript & JavaScript Language Server. Online. 2025. Available from: <https://github.com/typescript-language-server/typescript-language-server>
 43. LSP-TYPES, Contributeurs de. lsp-types v0.97.0 Types for interaction with a language server, using VSCode's Language Server Protocol. Online. 2024. Available from: <https://crates.io/crates/lsp-types>
-

44. , et contributeurs Organisation gluon-lang. Reverse dependencies of lsp-types crate. Online. 2024. Available from: https://crates.io/crates/lsp-types/reverse_dependencies
45. MICROSOFT, et contributeurs. Implementations - Tools supporting the LSP. Online. 2025. Available from: <https://microsoft.github.io/language-server-protocol/implementors/tools/>
46. MICROSOFT, et contributeurs. Implementations - Language Servers. Online. 2025. Available from: <https://microsoft.github.io/language-server-protocol/implementors/servers/>
47. OXALICA, et contributeurs. async-lsp v0.2.2 Asynchronous Language Server Protocol (LSP) framework based on tower. Online. 2025. Available from: <https://crates.io/crates/async-lsp>
48. OXALICA, et contributeurs. nil/crates/nil/Cargo.toml - Nix Language server, an incremental analysis assistant for writing in Nix. Online. 2025. Available from: <https://github.com/oxalica/nil/blob/577d160da311cc7f5042038456a0713e9863d09e/crates/nil/Cargo.toml#L11>
49. MYRIAD-DREAMIN, et contributeurs. sync-ls - Synchronized language service inspired by async-lsp, primarily for tinymin. Online. 2025. Available from: <https://crates.io/crates/sync-ls>
50. ORGANISATION, et contributeurs tower-lsp-community. tower-lsp-server v0.21.1 Language Server Protocol implementation based on Tower. Online. 2025. Available from: <https://crates.io/crates/tower-lsp-server>
51. ORGANISATION, et contributeurs rust-lang. Reverse dependencies of lsp-server crate. Online. 2024. Available from: https://crates.io/crates/lsp-server/reverse_dependencies
52. EYAL KALDERON, et contributeurs. Reverse dependencies of tower-lsp crate. Online. 2023. Available from: https://crates.io/crates/tower-lsp/reverse_dependencies
53. ORGANISATION, et contributeurs rust-lang. rust-analyzer/lib/lsp-server/examples/goto_def.rs at master · rust-lang/rust-analyzer · GitHub. Online. 2025. Available from: https://github.com/rust-lang/rust-analyzer/blob/master/lib/lsp-server/examples/goto_def.rs
54. Online. 2025. Available from: https://macromates.com/manual/en/regular_expressions
55. Online. 2025. Available from: <https://code.visualstudio.com/api/language-extensions/syntax-highlight-guide>
56. Online. 2025. Available from: <https://www.jetbrains.com/help/idea/textmate.html>
57. LTD, MacroMates. Language Grammars – TextMate 1.x Manual - Example Grammar. Online. Available from: https://macromates.com/manual/en/language_grammars#example_grammar
58. TREE-SITTER, Contributeurs de. Introduction - Tree-sitter. Online. 2025. Available from: <https://tree-sitter.github.io/tree-sitter/>
59. TREE-SITTER, Contributeurs de. Creating Parsers - Getting Started - Tree-sitter. Online. 2025. Available from: <https://tree-sitter.github.io/tree-sitter/creating-parsers/1-getting-started.html>
60. Neovim Documentation - Treesitter. Online. 2025. Available from: <https://neovim.io/doc/user/treesitter.html>
61. Language Extensions - Grammar. Online. 2025. Available from: <https://zed.dev/docs/extensions/languages?#grammar>
62. Creating a Grammar. Online. 2025. Available from: <https://flight-manual.atom-editor.cc/hacking-atom/sections/creating-a-grammar/>
63. GITHUB, et contributeurs. Navigating code on GitHub. Online. 2025. Available from: <https://docs.github.com/en/repositories/working-with-files/using-files/navigating-code-on-github>
64. MICROSOFT, et contributeurs. Semantic Highlight Guide | Visual Studio Code Extension API. Online. 2025. Available from: <https://code.visualstudio.com/api/language-extensions/semantic-highlight-guide>
65. MICROSOFT, et contributeurs. Language Server Protocol Specification - 3.17 - Semantic Tokens. Online. 2025. Available from: https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/#textDocument_semanticTokens

-
66. SUBLIMEHQ. SublimeHQ - End User License Agreement. Online. 2025. Available from: <https://www.sublimehq.com/eula>
 67. SUBLIMEHQ. Syntax Definitions. Online. 2025. Available from: <https://www.sublimetext.com/docs/syntax.html>
 68. STACK EXCHANGE INC. Technology | 2024 Stack Overflow Developer Survey - Integrated development environment. Online. 2025. Available from: <https://survey.stackoverflow.co/2024/technology#1-integrated-development-environment>
 69. MICROSOFT, et contributeurs. Iteration Plan for March 2025 · Issue #243015 · microsoft/vscode · GitHub. Online. 2025. Available from: <https://github.com/microsoft/vscode/issues/243015>
 70. MICROSOFT, et contributeurs. Iteration Plan for May 2025 · Issue #248627 · microsoft/vscode · GitHub. Online. 2025. Available from: <https://github.com/microsoft/vscode/issues/248627>
 71. MICROSOFT, et contributeurs. Explore using tree sitter for syntax highlighting · Issue #210475 · microsoft/vscode · GitHub. Online. 2025. Available from: <https://github.com/microsoft/vscode/issues/210475>
 72. MICROSOFT, et contributeurs. [Exploration] Tree-sitter tokenization exploration (Fixes #161256) by aiday-mar · Pull Request #161479 · microsoft/vscode · GitHub. Online. 2022. Available from: <https://github.com/microsoft/vscode/pull/161479>
 73. TREE-SITTER, Contributeurs de. The Grammar DSL - Tree-sitter. Online. 2025. Available from: <https://tree-sitter.github.io/tree-sitter/creating-parsers/2-the-grammar-dsl.html>
 74. SIRAPHOB, Ben. How to write a tree-sitter grammar in an afternoon | siraben's musings. Online. 2022. Available from: <https://siraben.dev/2022/03/01/tree-sitter.html>
 75. GitHub - AlecGhost/tree-sitter-vscode: Bring the power of Tree-sitter to VSCode. Online. 2025. Available from: <https://github.com/AlecGhost/tree-sitter-vscode>
 76. serde_json - Parsing JSON as strongly typed data structures. Online. 2025. Available from: https://docs.rs/serde_json/latest/serde_json/index.html#parsing-json-as-strongly-typed-data-structures
 77. serde_json - Constructing JSON values. Online. 2025. Available from: https://docs.rs/serde_json/latest/serde_json/index.html#constructing-json-values
 78. GOOGLE, et contributeurs. Protocol Buffers Documentation. Online. 2025. Available from: <https://protobuf.dev/>
 79. CONTRIBUTEURS. GitHub - tokio-rs/prost: PROST! a Protocol Buffers implementation for the Rust Language. Online. 2025. Available from: <https://github.com/tokio-rs/prost>
 80. rmp - The Rust MessagePack Library. Online. 2025. Available from: <https://docs.rs/rmp/latest/rmp/>
 81. IAN FETTE, Alexey Melnikov. RFC 6455: The WebSocket Protocol. Online. 2025. Available from: <https://www.rfc-editor.org/rfc/rfc6455>
 82. IAN FETTE, Alexey Melnikov. RFC 6455: The WebSocket Protocol - 1.5. Design Philosophy. Online. 2025. Available from: <https://www.rfc-editor.org/rfc/rfc6455#section-1.5>
 83. IAN FETTE, Alexey Melnikov. RFC 6455: The WebSocket Protocol - 1.3. Opening Handshake. Online. 2025. Available from: <https://www.rfc-editor.org/rfc/rfc6455#section-1.3>
 84. CONTRIBUTEURS, Snapview GmbH et. Lightweight stream-based WebSocket implementation. Online. 2025. Available from: <https://crates.io/crates/tungstenite>
 85. CONTRIBUTEURS. GitHub - crossbario/autobahn-testsuite: Autobahn WebSocket protocol testsuite. Online. 2025. Available from: <https://github.com/crossbario/autobahn-testsuite>
 86. CONTRIBUTEURS, Snapview GmbH et. tokio-tungstenite. Online. 2025. Available from: <https://crates.io/crates/tokio-tungstenite>
 87. CONTRIBUTEURS. websocket. Online. 2025. Available from: <https://crates.io/crates/websocket>
 88. CONTRIBUTEURS. fastwebsockets. Online. 2025. Available from: <https://crates.io/crates/fastwebsockets>
-

-
89. AUTHORS. Introduction to gRPC | gRPC. Online. 2025. Available from: <https://grpc.io/docs/what-is-grpc/introduction/>
 90. CONTRIBUTEURS. GitHub - hyperium/tonic: A native gRPC client & server implementation with async/await support. Online. 2025. Available from: <https://github.com/hyperium/tonic>
 91. BRANDHORST, Johan. The state of gRPC in the browser | gRPC. Online. 2019. Available from: <https://grpc.io/blog/state-of-grpc-web>
 92. CONTRIBUTORS, MDN. EventSource - Web APIs | MDN. Online. 2025. Available from: <https://developer.mozilla.org/en-US/docs/Web/API/EventSource>
 93. CONTRIBUTEURS. GitHub - google/tarpc: An RPC framework for Rust with a focus on ease of use. Online. 2025. Available from: <https://github.com/google/tarpc>
 94. Cap'n Proto: Introduction. Online. 2025. Available from: <https://capnproto.org/>
 95. Apache Thrift - Home. Online. 2025. Available from: <https://thrift.apache.org/>
 96. JONASLOOS. BibTeX to Hayagriva. Online. 2025. Available from: <https://jonasloos.github.io/bibtex-to-hayagriva-webapp/>

Figures

| | | |
|---------|---|----|
| Fig. 1 | Aperçu de la page d'accueil de PLX dans le terminal (2) | 8 |
| Fig. 2 | Aperçu d'un exercice dans PLX, avec un check qui échoue et les 2 suivants qui passent (2) | 9 |
| Fig. 3 | Architecture haut niveau décrivant les interactions entre les clients PLX et le serveur de session live | 9 |
| Fig. 4 | Equivalent dans une version préliminaire de la syntaxe DY de l'exercice défini sur le Snippet 1 | 12 |
| Fig. 5 | Aperçu de l'expérience imaginée dans un IDE | 13 |
| Fig. 6 | Un exemple de Rustlings en haut dans le terminal et VSCode en bas, sur un exercice de fonctions | 14 |
| Fig. 7 | Exemple d'auto-complétion dans Neovim, générée par le serveur de langage <code>rust-analyzer</code> sur l'appel d'une méthode sur les <code>&str</code> | 24 |
| Fig. 8 | Exemple de discussion en LSP une demande de <code>textDocument/definition</code> , output de <code>fish demo.fish</code> dans le dossier <code>pocs/lsp-server-demo</code> .
Les lignes après <code>CLIENT:</code> sont envoyés en stdin et celles après <code>SERVER</code> sont reçues en stdout. | 26 |
| Fig. 9 | Liste de symboles générées par Tree-Sitter, affichés à droite du code sur GitHub pour un exemple de code Rust de PLX | 29 |
| Fig. 10 | Exemple tiré de la documentation de VSCode, démontrant quelques améliorations dans le surlignage. Les paramètres <code>languageModes</code> et <code>document</code> sont colorisés différemment que les variables locales. <code>Range</code> et <code>Position</code> sont colorisées comme des classes. <code>getFoldingRanges</code> dans la condition est colorisée en tant que fonction ce qui la différencie des autres propriétés. (64) | 30 |
| Fig. 11 | Concrete Syntax Tree généré par la grammaire définie sur le fichier <code>mcq.dy</code> | 33 |
| Fig. 12 | Screenshot du résultat de la commande <code>tree-sitter highlight mcq.dy</code> avec notre exercice surligné | 34 |
| Fig. 13 | Screenshot dans VSCode une fois l'extension <code>tree-sitter-vscode</code> configuré, le surlignage est fait via notre syntaxe Tree-Sitter via | 34 |
| Fig. 14 | La première partie consiste en une mise en place par la connexion et l'annonce des clients de leur rôle, en se connectant puis en envoyant leur rôle en string. | 40 |
| Fig. 15 | La deuxième partie consiste en l'envoi régulier du client du résultat du check vers le serveur, qui ne fait que de transmettre au socket associé au <code>teacher</code> | 40 |

Tables

Outils utilisés

Usage de l'IA

L'auteur de ce travail a utilisé l'IA

- pour chercher des syntaxes humainement éditables, comme certains projets ne sont pas bien référencés sur Google car peu utilisés ou décrit avec d'autres mots-clés
- pour trouver la raison de certaines erreurs dans les POCs fait en Rust

Outils techniques

- Neovim pour l'édition du rapport et l'écriture du code
- Template Typst
- Convertisseur de Bibtex vers Hayagriva (96)