



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

Bertil Chapuis

2024-25

Yverdon-les-Bains, le 19.06.2025

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

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 19.06.2025

Samuel Roland

Table des matières

Préambule	2
Authentification	3
Introduction	6
Contexte	6
Problèmes de l'expérience originale	7
L'approche de PLX	9
Nouveaux défis	10
Défi 1: Comment les enseignants peuvent voir le code et les résultats en temps réel ? ...	10
Défi 2: Comment faciliter la rédaction et la maintenance des exercices ?	11
Solutions existantes	17
Glossaire	18
Planification	19
Déroulement	19
Planification initiale	19
Planification finale	20
État de l'art	21
Format de données humainement éditables existants	21
KHI - Le langage de données universel	21
Bitmark - le standard des contenus éducatifs digitaux	22
NestedText – Un meilleur JSON	24
SDLang - Simple Declarative Language	25
KDL - Cuddly Data language	26
Conclusion	26
Librairies existantes de parsing en Rust	27
Les serveurs de langage et librairies Rust existantes	28
Adoption	29
Librairies disponibles	29
Choix final	29
POC de serveur de langage avec lsp-server	30
Systèmes de surlignage de code	32
Textmate	32
Tree-Sitter	33
Surlignage sémantique	34
Choix final	35
POC de surlignage de notre syntaxe avec Tree-Sitter	35
Protocoles de synchronisation et formats de sérialisation existants	39
JSON	39

Protocol Buffers - Protobuf	39
MessagePack	40
Websocket	40
gRPC	41
tarpc	42
Choix final	42
POC de synchronisation de messages JSON via Websocket avec tungstenite	43
Architecture	47
Protocole de synchronisation	47
Syntaxe DY	48
Définition semi-formelle de la syntaxe DY en abstrait	48
Usage de la syntaxe dans PLX	48
Exemple d'usage dans PLX	49
Implémentation	50
Implémentation des POC	50
Implémentation du serveur PLX	50
Implémentation de la librairie <code>dy</code>	50
Intégration de <code>dy</code> dans PLX	50
Implémentation de la syntaxe Tree-Sitter	50
Implémentation du serveur de langage	50
Conclusion	51
Bibliographie	52
Annexes	59
Outils utilisés	59
Usage de l'intelligence artificielle	59
Outils techniques	59
Logo	59
Cahier des charges original	60
Concevoir une expérience d'apprentissage interactive à la programmation avec PLX	60

Introduction

Contexte

L'informatique et particulièrement la programmation, sont des domaines abstraits et complexes à apprendre. Dans la majorité des universités, l'informatique est enseignée sur des cours composés d'une partie théorique, dispensée par un·e professeur·e, et d'une partie pratique, sous forme de laboratoires, encadrée par des assistant·es. Les sessions théoriques sont souvent données sous forme magistrale: des slides durant 2 périodes pour présenter différents concepts, morceaux de code et études de cas. Les étudiant·es ont rarement la possibilité d'être actif·ves, ce qui limite fortement la concentration et la rétention de l'information. Une grande partie de l'auditoire décroche et préfère travailler sur des laboratoires ou réviser d'autres cours.

Lors des rares sessions d'exercice en classe et durant la révision en dehors des cours, un temps important est perdu à mettre en place les exercices et les vérifications manuelles. Ce processus fastidieux se fait au détriment de la pratique délibérée, concept popularisé par le psychologue Anders Ericsson dans ses recherches en expertise (1). Cette méthode consiste à travailler de manière concentrée sur des sous-compétences spécifiques. Afin de constamment corriger et affiner son modèle mental, elle demande de recevoir un feedback rapide et régulier. La solidité du modèle mental construit par l'expérience, permet d'attendre un haut niveau d'expertise.

Ce travail de Bachelor s'inscrit dans ce contexte et vise à redéfinir l'expérience d'apprentissage et d'enseignement de la programmation en s'inspirant de cette méthode.

Problèmes de l'expérience originale

Pour mieux comprendre à quel point le processus actuel d'entraînement est fastidieux, regardons un exercice concret de C pour débutant. Une enseignante qui suit une classe de 40 étudiant·es, fournit la consigne suivante sur un serveur, comme premier exercice de la session.

Salue-moi

Un petit programme qui te salue avec ton nom complet.

Assure toi d'avoir la même sortie que ce scénario, en répondant `John` et `Doe` manuellement.

```
> ./main
Quel est ton prénom ? John
Salut John, qu'est est ton nom de famille ? Doe
Passe une belle journée John Doe !
>
```

Démarre avec ce bout de code.

```
int main(int argc, char *argv[]) {
    // ???
}
```

Vérifie que ton programme ait terminé avec le code de fin 0, en lançant cette commande.

```
> echo $?
0
```

Un titre, une consigne et un scénario pour tester le bon fonctionnement sont fournis. L'enseignante annonce un temps alloué de dix minutes. Une fois la consigne récupérée et lue par un des étudiant·es, il prend le code de départ et de crée un nouveau fichier dans ses fichiers personnels. L'étudiant ouvre ensuite son IDE favori dans le dossier de l'exercice et configure la compilation avec CMake. Après trois de mise en place, il peut enfin commencer à coder.

Une première solution est développée et est prête à être testée après deux minutes. Il lance un terminal, compile le code, rentre `John` et `Doe` et s'assure du résultat. Après relecture de la sortie générée, il se rend compte d'une erreur sur `Passe une belle journée Doe !` : seul le nom de famille s'affiche, le prénom a été oublié. Deux minutes pour tester son code se sont écoulées. Après une minute de correction, l'étudiant retourne dans son terminal et recommence le processus de validation. L'exercice est terminé juste à la fin du temps alloué et l'étudiant peut suivre la correction. S'il avait eu une erreur de plus, il aurait eu besoin de quelques minutes de plus. Certain·es étudiant·es à ses côtés n'ont pas eu le temps de finir et doivent s'arrêter.

En résumé, sur les dix minutes seulement trois ont été utilisées pour de l'écriture de code. Tout le reste a été perdu sur des tâches « administratives » autour de l'exercice.

Durant la correction, l'enseignante va présenter sa solution et demander s'il y a des questions. Certain·es étudiant·es les plus avancé·es poseront peut-être des questions sur l'approche ou une fonction spécifique. Il est cependant rare d'entendre une question du type « Je suis complètement paumé·e, vous pouvez réexpliquer ce que fait cette fonction ? » ou encore « Je ne n'ai pas ce qui est flou mais je n'ai vraiment pas compris votre solution ». D'autres qui n'ont pas pu terminer l'exercice ne savent pas si leur début partait dans la bonne direction, même si la solution était bien expliquée.

Faire fonctionner le programme n'est que la première étape. Faire du code robuste, modulaire, lisible et performant demande des retours humains pour pouvoir progresser. Les étudiant·es moins expérimenté·es ne savent pas immédiatement si la compétence est acquise, comme le feedback n'arrive que dans les corrections des évaluations notées, plusieurs semaines plus tard.

Côté de l'enseignante, en comptant uniquement sur les questions des étudiant·es, savoir si le message de l'exercice est passé reste un challenge. Il est difficile aussi de savoir quand l'exercice doit se terminer. Peut-être qu'il aurait fallu 5 minutes de plus pour qu'une majorité ait le temps de finir ? Pour avoir accès aux réponses, elles doivent être manuellement rendues sur un serveur. Ce rendu prend à nouveau du temps pour chaque étudiant·e. Pour l'enseignante, les récupérer, ouvrir et fermer 40 fichiers, pour évaluer l'état d'avancement à l'oeil et comprendre les différentes approches, prendrait trop de temps en classe.

Une autre approche serait de coder dans un fichier Google Docs partagé à toute la classe. L'enseignante a maintenant un moyen de relire au fur et à mesure, détecter les incompréhensions, mais les étudiant·es ont perdu toute l'expérience du développement en local. Dans Google Docs, il n'y a pas de couleur sur le code, pas d'auto-complétion et pas d'erreur de compilation visible dans le code. Tous les raccourcis, le formatage automatique et les informations au survol manquent terriblement. Pour tester leur programme, les étudiant·es doivent constamment copier leur code dans un fichier local.

En conclusion, le problème c'est que l'entraînement est fastidieux pour les étudiants, ce qui implique moins d'exercices effectués, moins de motivation à avancer et freine l'apprentissage en profondeur. Le manque de retour ralentit également la progression des compétences autour de la qualité du code produit. Les enseignant·es n'ont pas accès aux réponses des étudiant·es, ce qui empêche d'avoir une vision précise des incompréhensions et de donner de feedbacks.

L'approche de PLX

Ce travail de Bachelor vise à poursuivre le développement du projet PLX (2), Terminal User Interface (TUI) écrite en Rust. Cette application permet aux étudiant·es de se concentrer pleinement sur l'écriture du code. PLX 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 (3). PLX fournit actuellement une expérience locale similaire pour le C et C++.

Pour commencer à s'entraîner, les étudiant·es clonent un repository Git et travaillent localement dans leur IDE favori qui s'exécute en parallèle de PLX. Les scénarios de vérifications, exécutés auparavant manuellement, sont lancés automatiquement à chaque sauvegarde de fichier. Ces suites de tests automatisées, appelées « checks », permettent d'apporter un feedback automatisé rapide, continu et riche. Au lieu de perdre sept minutes sur dix sur des tâches « administratives », PLX en automatise la majorité et permet à l'étudiant·e de réduire ce temps à une minute.



Fig. 1. – Aperçu de la page d'accueil de PLX dans le terminal (4)

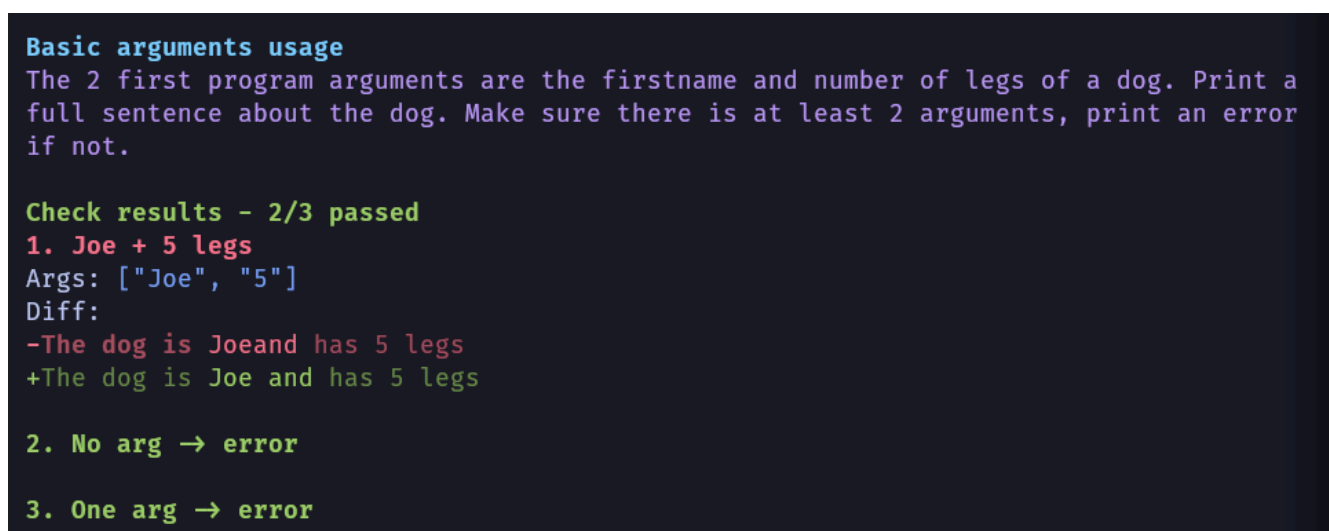


Fig. 2. – Aperçu d'un exercice de C dans PLX, avec un check qui échoue et 2 autres qui passent (4)

Nouveaux défis

Le besoin de feedback humain pour les étudiant·es en plus du feedback automatisé, et celui de permettre aux enseignant·es d'accéder aux réponses, ne sont pas encore résolus par PLX. Ces nouveaux défis sont le point de départ des deux extensions majeures qui seront développées dans le cadre de ce travail.

Défi 1: Comment les enseignants peuvent voir le code et les résultats en temps réel ?

Comme mentionné précédemment, le rendu manuel d'exercices prend un peu de temps et ne sera pas fait fréquemment durant un entraînement. De plus, avoir accès à une archive de fichiers de code, demanderait encore de les lancer localement avant de pouvoir construire des statistiques de l'état des checks.

Comme l'application fonctionne localement et s'exécute à chaque sauvegarde, le code et les résultats des checks sont déjà connus par PLX. Il suffirait d'avoir un serveur central, qui héberge les sessions d'entraînement en classe (appelées « sessions live »), vers lequel PLX envoie le code et l'état des checks en continu. Ces informations peuvent ensuite être récupérées du serveur et affichées sur l'interface PLX de l'enseignant·e.

Des interfaces de visualisation sur une échelle globale ou plus zoomée, permettront aux enseignant·es de rapidement comprendre les lacunes des étudiant·es, en relisant les différentes réponses affichées. Grâce à l'état des checks, il sera facile de voir si la classe a terminé l'exercice ou de filtrer les réponses pour concentrer sa relecture. Il sera possible de sélectionner certaines réponses particulières pour les afficher au beamer, pouvoir les commenter ou demander à la classe de critiquer constructivement le code choisi.

Sur la Fig. 3 suivante, on voit qu'avant de commencer, les étudiants ont dû cloner le repository Git du cours sur leur machine pour accéder aux exercices. Une fois une session live démarrée par un·e enseignant·e et les étudiant·es ayant rejoint la session, l'enseignant·e peut choisir de faire un exercice l'un après l'autre en choisissant le rythme.

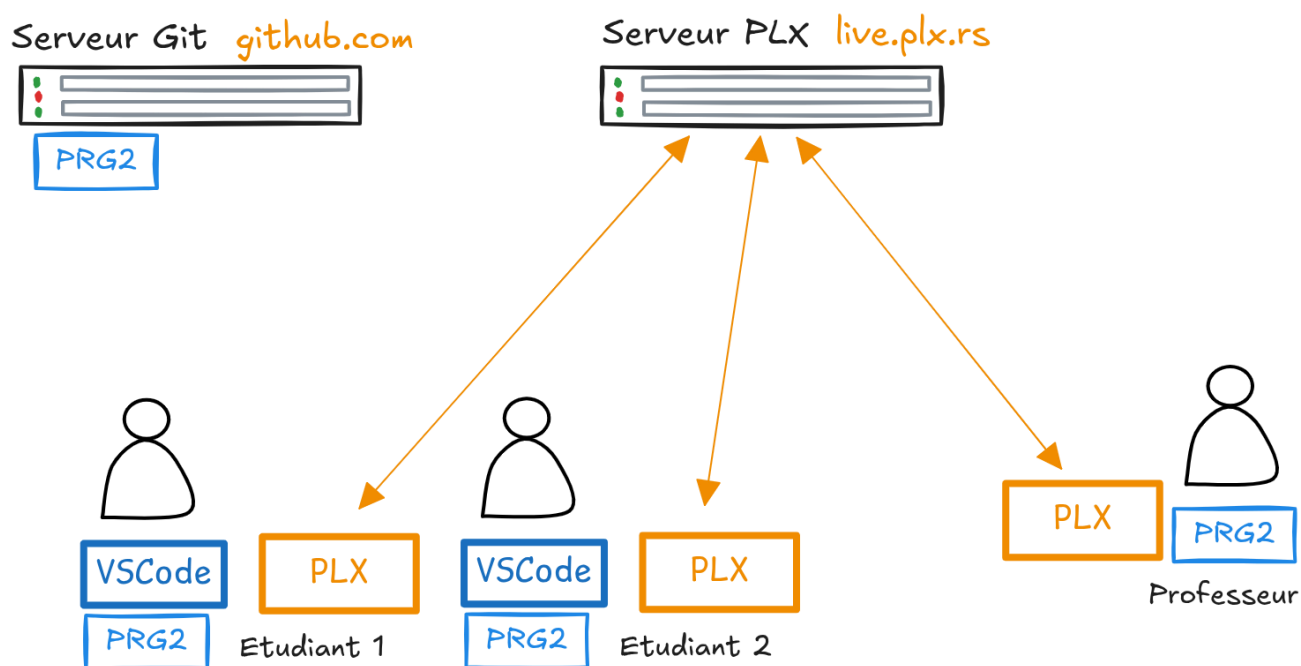


Fig. 3. – Interactions entre les clients PLX chez l'enseignant·e et les étudiant·es, le code est synchronisé via un serveur central, le cours « PRG2 » a un repository Git publique

L'exercice en cours est affiché sur tous les clients PLX. À chaque sauvegarde d'un fichier de code, le code est compilé et les checks sont lancés. Les résultats des checks et le code modifié seront envoyés à l'enseignant de la session.

Ce premier défi nécessite le développement d'un serveur central et d'un protocole de synchronisation. Elle implique aussi l'utilisation d'un protocole de communication bidirectionnel pour permettre cette expérience en direct en classe.

Défi 2: Comment faciliter la rédaction et la maintenance des exercices ?

La rédaction de contenu sous forme de fichier textes, au lieu de l'approche classique de formulaires, semble particulièrement plaire en informatique. En effet, de nombreux enseignant·es à la HEIG-VD rédigent tout ou partie de leur contenu (exercices, slides, consignes de laboratoires) dans divers formats textuels.

Un exemple d'usage du Markdown est le recueil d'exercices du cours de PRG1 (cours de C++) (5). On note également l'usage de balises HTML `<details>` et `<summary>`, pour rendre disponible la solution tout en la cachant par défaut. Pour combler le manque de mise en page du Markdown, d'autres enseignant·es utilisent Latex ou Typst (6).

Pour faciliter l'adoption de PLX, nous avons besoin d'un format de données simple à prendre en main, pour décrire les différents types d'exercices supportés. Si on reprend l'exercice présenté plus tôt, qu'on le rédige en Markdown, en y ajoutant la solution comme sur le recueil de PRG1, cela donne Snippet 1.

```
# Salut-moi
Un petit programme qui te salue avec ton nom complet.

Assure toi d'avoir la même sortie que ce scénario, en répondant `John` et `Doe`
manuellement.
```
> ./main
Quel est ton prénom ? John
Salut John, qu'est est ton nom de famille ? Doe
Passe une belle journée John Doe !
>
...

Démontre avec ce bout de code.
```c
int main(int argc, char *argv[]) {
    // ???
}
...

Vérifie que ton programme ait terminé avec le code de fin 0, en lançant cette commande.
```sh
> echo $?
0
...

<details>
<summary>Solution</summary>

```c
#include <stdio.h>

#define NAME_MAX_SIZE 100
int main(int argc, char *argv[]) {
    char firstname[NAME_MAX_SIZE];
    char lastname[NAME_MAX_SIZE];

    printf("Quel est ton prénom ? ");
    fflush(stdout);
    scanf("%s", firstname);

    printf("Salut %s, qu'est est ton nom de famille ? ", firstname);
    fflush(stdout);
    scanf("%s", lastname);

    printf("Passe une belle journée %s %s !\n", firstname, lastname);
    return 0;
}
...
</details>
```

Snippet 1. – Exemple d'exercice de programmation, rédigé en Markdown

Ce format en Snippet 1 est pensé pour un document lisible par des humains. Cependant, si on voulait pouvoir automatiser l'exécution du code et des étapes manuelles de rentrer le prénom, le nom et de vérifier l'output, nous aurions besoin d'extraire chaque information sans ambiguïté.

Hors cette structure, bien que reproductible sur d'autres exercices, n'est pas assez standardisée pour une extraction automatique.

En effet, sans comprendre le langage naturel, comment savoir que `John` et `Doe` doivent être rentrés à la main et ne font pas partie de l'output ? Comment le parseur peut détecter qu'on parle du code d'exit du programme et que ce code doit valoir zéro ? Et si on avait différents scénarios comment pourrait-on les décrire et différencier ? Comment distinguer la consigne utile du reste des instructions générique, comme *en répondant* `John` et `Doe` *manuellement*, qui ne devrait pas apparaître si le scénario a pu être automatisé ? Ce qui peut être clair sur le langage naturel pour les humains du découpage mentale des informations, devient une tâche impossible pour un parseur qui doit être prédictible et rapide (on exclut l'usage de l'intelligence artificielle pour ces raisons).

De plus, ce format possède plusieurs parties qui demandent plus de travail à la rédaction. Le code de la solution est développé dans un fichier `main.c` séparé et doit être copié manuellement. Une partie du texte comme *Assure toi d'avoir la même sortie que ce scénario* est générique et doit pourtant être constamment répétée à chaque exercice pour introduire le snippet. L'output est à maintenir à jour avec le code de la solution, si celle-ci évolue, on risque d'oublier de mettre à jour la consigne de l'exercice.

Dans le contexte du développement de PLX, la manière la plus rapide et facile à mettre en place serait simplement de définir un schéma JSON à respecter. On aurait d'abord un champ pour le titre (sous la clé `exo` pour raccourcir le mot `exercice`) et la consigne.

Ensuite une liste de checks serait fournie. Chaque check serait défini par un titre et une séquence d'opérations à effectuer. Chaque opération serait de type `see` (ce que l'on s'attend à « voir » dans l'output), `type` (ce qu'on tape dans le terminal) et finalement `exit` (pour définir le code d'exit attendu). Il serait pratique de définir cette séquence dans un objet, avec en clé `see`, `type` ou `exit` et en valeur, un paramètre. Comme les clés des objets en JSON n'ont pas d'ordre et doivent être uniques (7), nous ne pourrions pas répéter plusieurs étapes `see`. Nous devons décrire la séquence comme un tableau d'objets. Voici un exemple d'usage de ce schéma sur le Snippet 2.

```
{
  "exo": "Salut-moi",
  "instruction": "Un petit programme qui te salue avec ton nom complet.",
  "checks": [
    {
      "name": "Il est possible d'être salué avec son nom complet",
      "sequence": [
        { "kind": "see", "value": "Quel est ton prénom ?" },
        { "kind": "type", "value": "John" },
        { "kind": "see", "value": "Salut John, qu'est est ton nom de famille ?" },
        { "kind": "type", "value": "Doe" },
        { "kind": "see", "value": "Passe une belle journée John Doe !" },
        { "kind": "exit", "code": 0 }
      ]
    }
  ]
}
```

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

Cet exemple d'exercice en Snippet 2 est minimal, mais montre clairement que rédiger dans ce format serait fastidieux. Si la consigne s'étalait sur plusieurs lignes, nous aurions du remplacer manuellement les retours à la ligne par des `\n`. Au-delà du texte brut, tous les guillemets, deux points, crochets et accolades nécessaires demande un effort de rédaction important.

Un autre format plus léger à rédiger est le YAML, regardons ce que cela donne:

```
exo: Salue-moi
instruction: Un petit programme qui te salue avec ton nom complet.
checks:
  - name: Il est possible d'être salué avec son nom complet
    sequence:
      - kind: see
        value: Quel est ton prénom ?
      - kind: type
        value: John
      - kind: see
        value: Salut John, qu'est est ton nom de famille ?
      - kind: type
        value: Doe
      - kind: see
        value: Passe une belle journée John Doe !
      - type: exit
        value: 0
```

Snippet 3. – Equivalent YAML de l'exercice défini sur le Snippet 1

Le YAML nous a permis ici de retirer tous les guillemets, les accolades et crochets. Cependant, malgré sa légèreté, il contient encore plusieurs points de friction:

- Les tirets sont nécessaires pour chaque élément de liste et les deux points pour chaque clé
- Pour avoir plus d'une information par ligne, il faut ajouter une paire d'accolades autour des clés
(- { kind: see, value: Passe une belle journée John Doe ! })
- Les tabulations sont difficiles à gérer dès qu'on dépasse 3-4 niveaux, elles sont aussi nécessaires sur du contenu multiligne
- Certaines situations nécessitent encore des guillemets autours des chaînes de caractères

L'intérêt clair du YAML, tout comme le JSON est la possibilité de définir des paires de clés/valeurs, ce qui n'est pas possible en Markdown. On pourrait définir une convention par dessus Markdown: définir qu'un titre de niveau 1 est le titre de l'exercice, qu'un bloc de code sans langage défini est l'output ou encore que le texte entre le titre et l'output est la consigne. Quand on arrive sur des champs plus spécifiques aux exercices de programmation, cela se corse un peu: comment définir le code d'exit attendu, définir la commande pour stopper un programme, ou encore définir les parties de l'output qui sont des entrées utilisateurs ?

Pour résoudre ces problèmes, nous proposons une nouvelle syntaxe, nommée DY, à mi-chemin entre le Markdown et le YAML, concise et compacte. Pour les clés, on définit un système de préfixe qui permet de structurer le contenu de l'exercice. On reprend les idées de `see`, `type`, et `exit`. Voici un exemple en Fig. 4.

```

exo Salue-moi
Un petit programme qui te salue avec ton nom complet.

check Il est possible d'être salué avec son nom complet
see Quel est ton prénom ?
type John
see Salut John, qu'est est ton nom de famille ?
type Doe
see Passe une belle journée John Doe !
exit 0

```

Fig. 4. – Equivalent de l'exercice du Snippet 1, dans une version préliminaire de la syntaxe DY

Dans cette syntaxe, plus de tabulation, de deux points, juste des informations séparées ligne par ligne, avec un préfixe avant chaque nouveau champ. La consigne est définie dans la suite du titre et peut s'étendre sur plusieurs lignes. Le Markdown est toujours supportés dans le titre et la consigne.

Ce deuxième défi demande d'écrire un parseur de cette nouvelle syntaxe. Ce n'est que la première étape, car lire du texte structuré blanc sur fond noir sans aucune couleur, sans feedback sur la validité du contenu, mène à une expérience un peu froide. En plus du parseur, il est indispensable d'avoir un support solide dans les IDE modernes pour proposer une expérience d'édition productive.

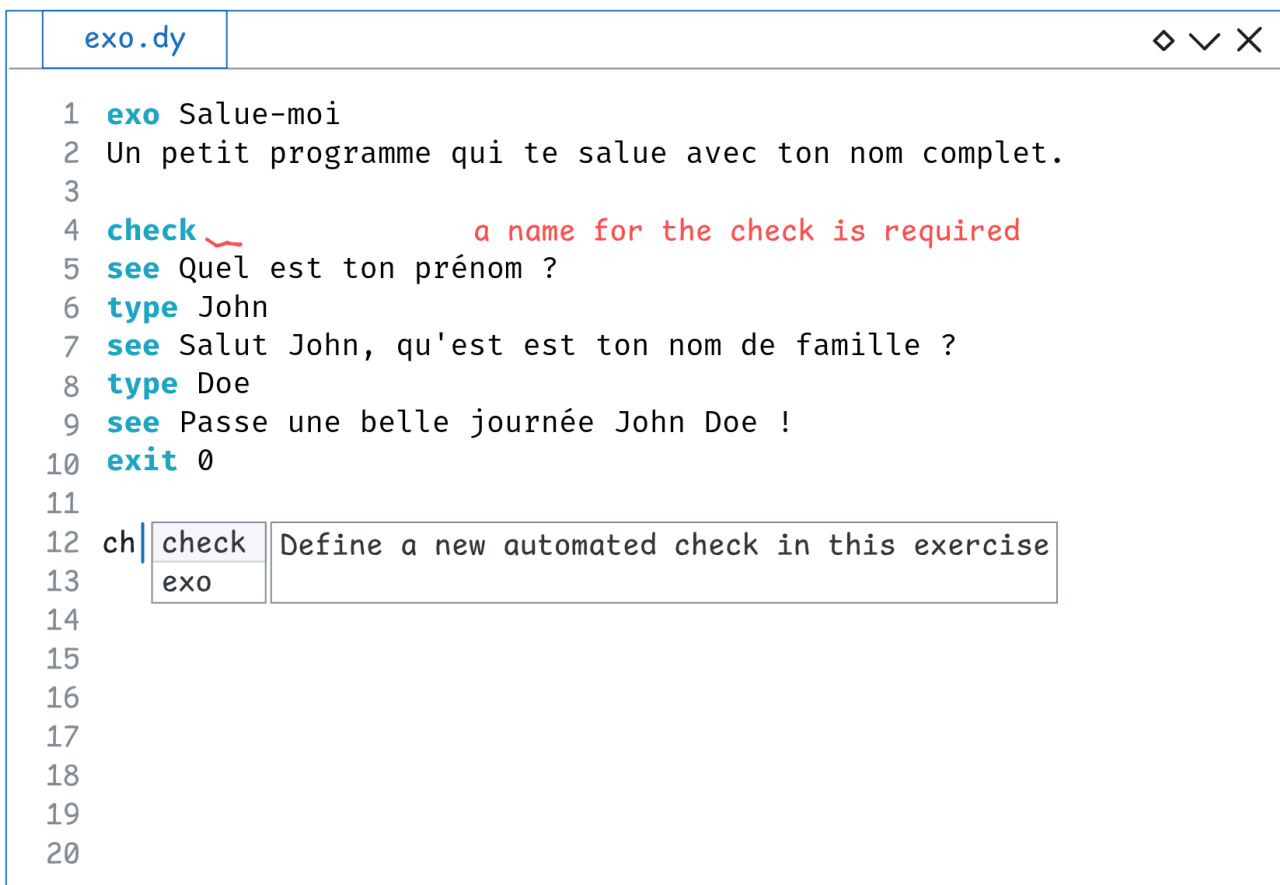


Fig. 5. – Aperçu de l'expérience imaginée de rédaction imaginée dans un IDE

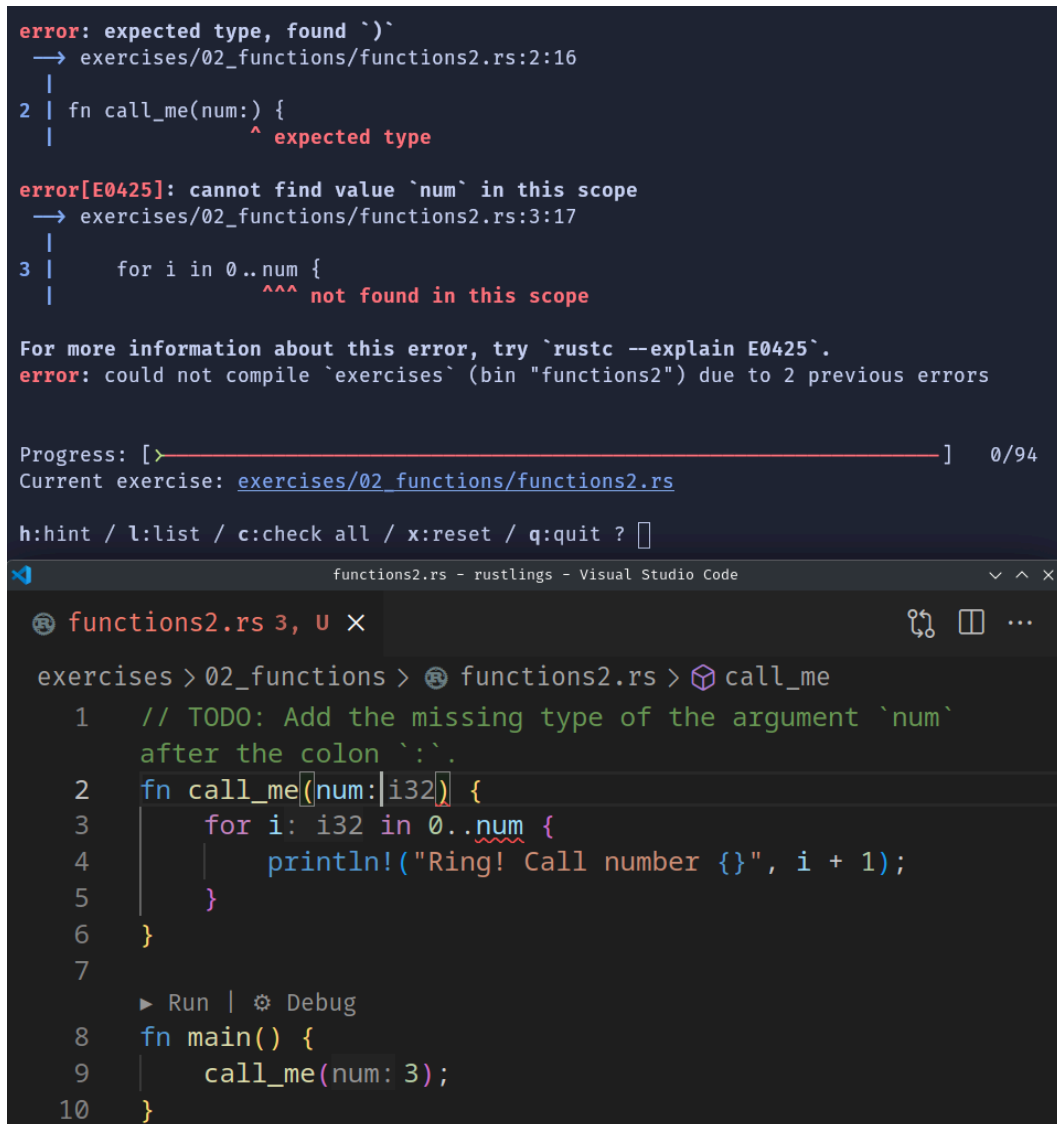
On voit dans la Fig. 5 que l'intégration se fait sur 2 points majeurs:

1. le surlignage de code, qui permet de coloriser les préfixes et les propriétés, afin de bien distinguer les préfixes et contenu
2. intégration avancée des erreurs du parseur et de la documentation à l'éditeur. On le voit en ligne 4 avec l'erreur du nom de check manquant après le préfixe `check`, en ligne 19 avec une auto-complétion qui propose les préfixes valides à cette position du curseur.

Cette nouvelle syntaxe, son parseur et support d'IDE permettront de remplacer le format TOML actuellement utilisé dans PLX.

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 terminal window at the top and a VS Code editor window at the bottom. The terminal window displays Rust compilation errors for a file named `functions2.rs`. The errors are:

```
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
```

Below the errors, the terminal shows progress information: `Progress: [>] 0/94` and the current exercise: `exercises/02_functions/functions2.rs`. It also displays a hint command: `h:hint / l:list / c:check all / x:reset / q:quit ?`.

The VS Code editor window 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 (8), `golings` pour le Go (9), `ziglings` pour Zig (10) et même `haskellings` pour Haskell (11) ! Ces projets incluent une suite d'exercice et 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 de programmation et crée des exercices dédiés. PLX prend une approche différente, il n'y a pas d'exercice proposé 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

L'auteur de ce travail se permet un certain nombre d'anglicismes quand un équivalent français n'existe pas ou n'est pas couramment utilisé. Certaines constructions de programmations bien connues comme les `strings` au lieu d'écrire `chaînes de caractères` sont également utilisées. Certaines sont spécifiques à certains langages et sont décrites ci-dessous pour aider à la lecture.

- `POC` : *Proof Of Concept*, preuve qu'un concept fonctionne en pratique. Consiste ici en un petit morceau de code développé juste pour démontrer que le concept est fonctionnel, sans soin particulier apporté à la qualité de l'implémentation. Ce code n'est pas réutilisé par la suite, il sert seulement d'inspiration pour l'implémentation réelle.
- `exo` : abréviation familière de `exercice`. Elle est utilisée dans la syntaxe DY pour rendre plus concis la rédaction.
- `check` : nom choisi pour décrire un ou plusieurs tests unitaires ou vérifications automatisées du code
- `Cargo` : le gestionnaire de dépendances, de compilation et de test des projets Rust
- `crate` : la plus petite unité de compilation avec Cargo, concrètement chaque projet contient un ou plusieurs dossiers avec un `Cargo.toml`, ce sont des crates locales. Les dépendances sont également des crates qui ont été publiées sur le registre officiel.
- `Cargo.toml`, configuration de Cargo dans un projet Rust définit les dépendances (les crates) et leurs versions minimum à inclure dans le projet, équivalent du `package.json` de NPM
- `crates.io` : le registre officiel des crates publiées pour l'écosystème Rust, l'équivalent de `npmjs.com` pour l'écosystème JavaScript ou `mvnrepository.com` pour Java
- `parsing` ou `désérialisation` : processus d'un parseur, visant à extraire de l'information brute vers une forme structurée facilement manipulable
- `sérialisation` : inverse du processus du parseur, qui vise à transformer une structure de données quelconque en une forme brute (une string par exemple) afin de la stocker sur le disque ou l'envoyer à travers le réseau
- `struct` : structure de données regroupant plusieurs champs, disponible en C, en Rust et d'autres langages inspirés
- `backtick` : caractère accent grave utilisé sans lettre, délimiteur fréquent de mention de variable ou fonction dans un rapport en Markdown
- `README` ou `README.md` : Point d'entrée de la documentation d'un repository Git, généralement écrit en Markdown, affiché directement sur la plupart des hébergeurs de repository Git
- `regex` : raccourci pour les expressions régulières
- `snippet` : court morceau de code ou de données

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 annexe, avec quelques corrections mineures.

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 serveurs de langages.
- Analyse de l'état de l'art des protocoles bidirectionnels temps réel (Websocket, gRPC...) et des formats de sérialisation (JSON, Protobuf, ...).
- Prototype avec les bibliothèques disponibles de parsing et de serveurs de langages 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 propriétés 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 librairie `dy`.

- Intégration de cette librairie 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

État de l'art

Format de données humainement éditables existants

Nous avons introduit plus tôt une nouvelle syntaxe DY, mais avant de commencer le développement, il est nécessaire de chercher les syntaxes existantes qui visent les mêmes objectifs et voir si elles peuvent être supportée en Rust, afin de justifier le choix de cette invention.

On ignore le XML et JSON qui sont parfaitement adaptés 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 optimisés pour la rédaction par des humains.

Le YAML et le TOML, bien que plus léger que le JSON, inclue de nombreux types de données autres 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 pour être parsé par une machine. En termes de rédaction, on cherche quelque chose du niveau de simplicité du Markdown, mais avec une validation poussée et spécifique au projet qui définit le schéma et les règles de validation.

Ces recherches se focalisent sur les syntaxes qui ne sont pas spécifiques à un domaine ou qui seraient complètement déliées de l'informatique ou de l'éducation. Ainsi, l'auteur ne présente pas Cooklang (12), qui se veut un langage de balise pour les recettes de cuisines, même si l'implémentation du parseur en Rust (13) 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) (14), à cause de la contrainte de connaître un peu la syntaxe du Rust et surtout parce qu'elle ne simplifie pas vraiment l'écriture comparée à du YAML. On ignore aussi les projets dont la spécification ou l'implémentation est en état de « brouillon » et n'est pas encore utilisable en production.

Différentes manières de les nommer existent : langage de balise (*markup language*), format de donnée, syntaxes, langage de donnée, langage spécifique à un domaine (*Domain Specific Language* - DSL), ... Pour trouver les projets suivants, la recherche a principalement été faite avec les mots-clés suivants 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*) (15), 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, les balises, 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 4. – Un exemple simplifié de KHI de leur README (16), décrivant un exemple d'article d'encyclopédie.

Une implémentation en Rust est proposée (17). Son dernier commit sur ces 2 repository Git 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 (18). 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 tout type d'appareils incluant les appareils mobiles (18). C'est la Bitmark Association en Suisse à Zurich qui développe ce standard, notamment à travers des Hackatons organisés en 2023 et 2024 (19).

Le standard permet de décrire du contenu statique et interactif, comme des articles ou des quiz de divers formats. Deux équivalents sont définis : le *bitmark markup language* et le *bitmark JSON data model* (20)

La partie quiz 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 5. – Un exemple de question à choix multiple tiré de leur documentation (21). 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 6. – Equivalent de Snippet 5 dans le Bitmark JSON data model (21)

Open Taskpool, projet qui met à disposition des exercices d'apprentissage de langues (22), 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 7. – 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 8. – Extrait simplifié de la réponse JSON, respectant le standard Bitmark (23). 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 (24), 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 restreints est la facilité d'intégrer des morceaux de code sans échappements ni guillemets, les caractères de données ne peuvent pas être confondus avec NestedText (25).

```
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 9. – Exemple tiré de leur README (25)

Ce format a l'air assez léger visuellement et l'idée de faciliter l'intégration de blocs multilignes 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 (26). De plus, l'implémentation officielle est en Python et il n'y a pas d'implémentation Rust disponible, il existe une crate réservée qui est restée vide (27).

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 (28)). SDLang définit également différents types de nombres (32 bits, 64 bits, entiers, flottants...), 4 valeurs de booléens (`true`, `false`, `on`, `off`), 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 10. – Exemple tiré de leur site web (28)

Ce format s'avère plus intéressant que les précédents par 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 neuf valeurs définie sur cinq lignes. Il est cependant regrettable que 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 11. – Exemple simplifié tiré de leur site web (29)

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'inclut malheureusement des strings multilignes, qui demande d'être entourée par `""`. Le problème d'intégration de morceaux de code est également relevé, les strings bruts 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 répandus 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 cible des usages plus avancés 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 (30), j'ai trouvé la liste de librairies suivantes :

- `winnow` (31), fork de `nom`, utilisé notamment par le parseur Rust de KDL (32)
- `nom` (33), utilisé notamment par `cexpr` (34)
- `pest` (35)
- `combine` (36)
- `chumsky` (37)

À noter aussi l'existence de la crate `serde`, un framework de sérialisation et désérialisation très populaire dans l'écosystème Rust (selon lib.rs (38)). 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 (39), qui supporte 29 types de données, ce projet paraît à l'auteur apporter plus de complexités qu'autre chose pour trois raisons :

1. 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.
2. La sérialisation (struct Rust vers syntaxe DY) n'est pas prévue, seul la désérialisation est utile.
3. 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 semblent bien adaptées à exprimer l'ignorance des espaces, extraire les nombres tant qu'ils contiennent des chiffres, extraire des opérateurs et les deux opérandes autour...

Pour DY, l'aspect multiligne et le fait que une partie des préfixes est optionnelle, complique l'approche de définir le début et la fin et de 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

check la struct a la bonne taille
see sizeof(Dog) = 12
exit 0
...
```

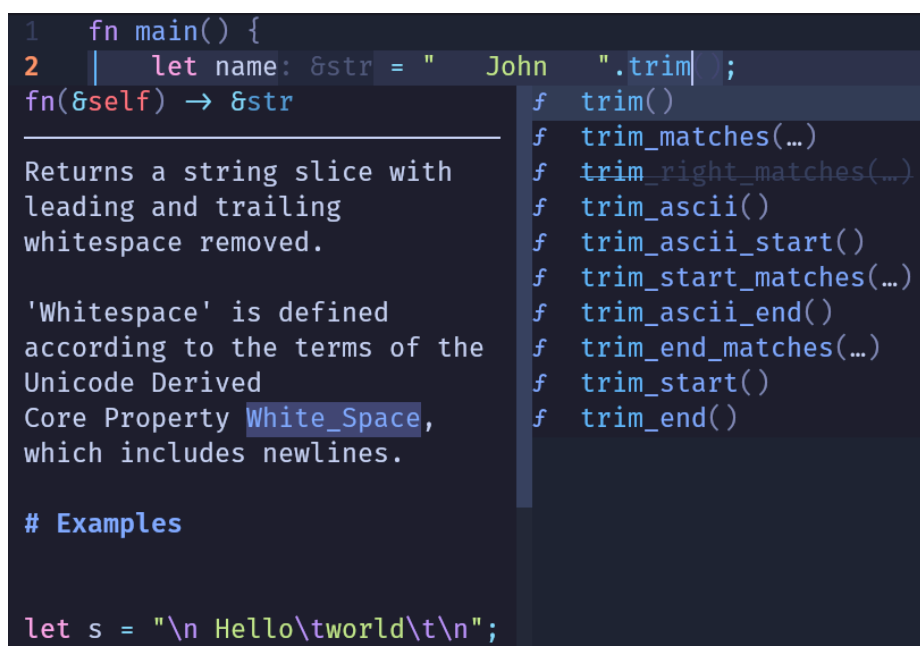
Snippet 12. – Exemple d'un début d'exercice de code, on voit que la consigne se trouve après la ligne `exo` et continue sur plusieurs lignes jusqu'à qu'on trouve un autre préfixe (ici `checks`). Même problème sur pour la fin du contenu de `see`, jusqu'au prochain `see` ou `exit` ou `check`

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*). Les éditeurs qui intègrent Tree-Sitter développent 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, résout le problème des développeurs de langages qui doivent supporter chaque éditeur de code indépendamment avec des API 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 (40).



```

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` (41). 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 13
- Un serveur de langage n'a pas besoin d'implémenter toutes les fonctionnalités du protocole. Un système de capacités (*Capabilities*) est défini pour annoncer les méthodes implémentées (42).
- Le transport des messages JSON-RPC peut se faire en `stdio` (flux standards d'entrée/sortie), sockets TCP ou même en HTTP.

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

Snippet 13. – 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 (43)

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` (44), 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 (45)
- `tailwindcss-language-server`, le serveur de langage pour le framework CSS TailwindCSS, implémenté en TypeScript (46)
- `typescript-language-server` et pour finir celui pour TypeScript, implémenté en TypeScript également (47)
- et beaucoup d'autres projets existent...

Une crate commune à plusieurs projets est `lsp-types` (48) qui définit les structures de données, comme `Diagnostic`, `Position`, `Range`. Ce projet est utilisé par `lsp-server`, `tower-lsp` et d'autres (49).

Adoption

Selon la liste sur le site de la spécification (50), 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 (51) quant à elle, contient plus de 200 projets, dont 40 implémentés en Rust ! Ce large support et ces nombreux exemples faciliteront le développement de ce serveur de langage et son intégration dans différents IDE.

Librairies disponibles

En cherchant à nouveau sur `crates.io` sur le tag `lsp`, on trouve différents projets dont `async-lsp` (52) utilisée dans `nil` (53) (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 (54). En continuant la recherche, on trouve encore un autre `tower-lsp` et un fork `tower-lsp-server` (55)... `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 termes de *reverse dependencies* sur `crates.io`) sont `lsp-server` (56) et `tower-lsp` (85) (57). 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 de serveur de langage avec `lsp-server`

L'auteur a modifié et exécuté l'exemple de `goto_def.rs` fourni par la crate `lsp-server` (58). Il a aussi créé un script `demo.fish` permettant de lancer la communication en stdin et attendre entre chaque requête. Cet exemple 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 envoyées 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 un 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 plage (*Range*) sur ligne 3 entre les caractères 12 et 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 14. – 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 profitera 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 IDE modernes supportent possèdent des systèmes de surlignage de code (ou surlignage syntaxique - *syntax highlighting*) 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 leurs 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 impliquent 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 processus est très similaire à la première 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 (59). VSCode utilise ces grammaires Textmate (60). IntelliJ IDEA l'utilise également pour les langages non supportés par IntelliJ IDEA comme Swift, C++ et Perl (61).

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 15. – Exemple de grammaire Textmate tiré de leur documentation (62).

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. » (62) (Traduction personnelle, dernier paragraphe section 12.2).

Tree-Sitter

Tree-Sitter (63) se définit comme un « outil de génération de parser et une librairie de parsing incrémentale. Il peut construire un arbre de syntaxe concret (CST) depuis un fichier source et efficacement mettre à jour cet arbre quand le fichier source est modifié. » (63) (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 quel type d'application (63, 64).

Tree-Sitter est supporté dans Neovim (65), dans le nouvel éditeur Zed (66), ainsi que d'autres. Tree-Sitter a été inventé par l'équipe derrière Atom (67) 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) (68).

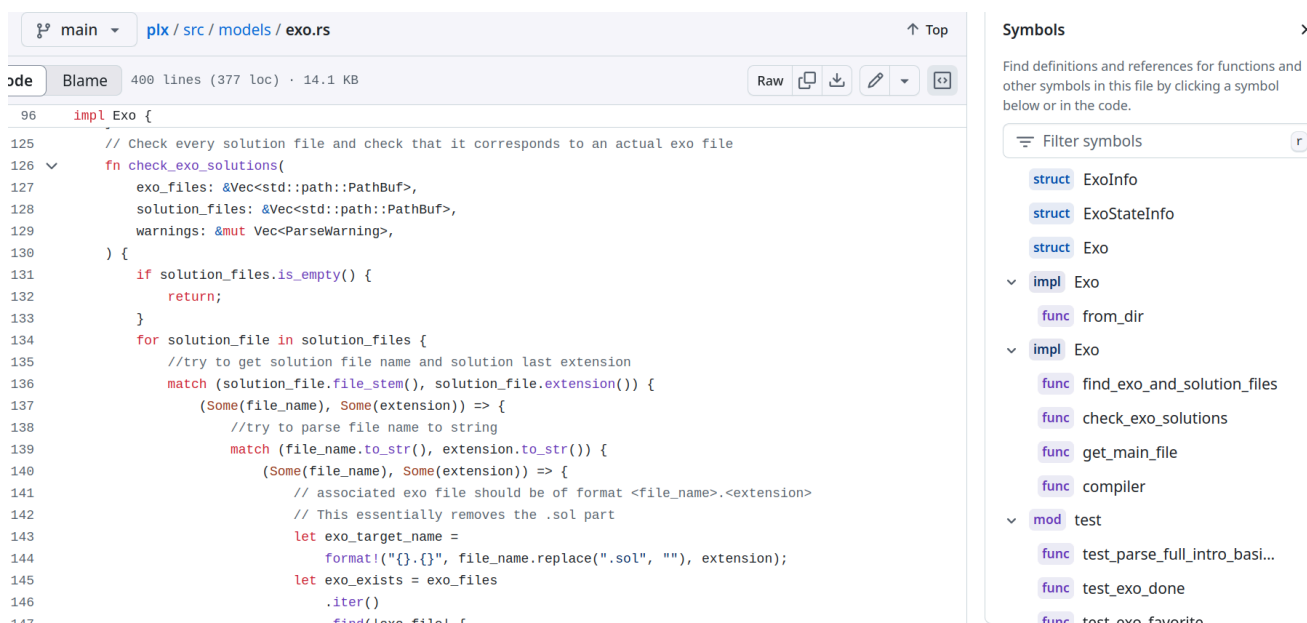


Fig. 9. – Liste de symboles générées par Tree-Sitter, affichés à droite du code sur GitHub pour un exemple de code Rust de PLX

Surlignage sémantique

Le surlignage sémantique (*Semantic highlighting*) 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. (69)

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. (69)

En voyant la liste des tokens sémantiques possible dans la spécification LSP (70), on comprend mieux l'intérêt et les possibilités de surlignage avancé. Par exemple, on trouve des tokens `macro`, `regex`, `typeParameter`, `interface`, `enum`, `enumMember`, qui seraient difficiles de différencier durant la tokenisation, mais qui peuvent être surligné différemment pour mettre en avant leur différence sémantique.

Sur le Snippet 16 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 16. – Exemple de code C `hello.c`, avec macro et fonction surligné de la même manière à l'appel

Sur le Snippet 17, 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 17. – 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.typemod.macro.globalScope.c links to @lsp  priority: 127
```

Snippet 18. – 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 (71). Ce système utilisent des fichiers `.sublime-syntax`, qui ressemble à TextMate (72), 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 (73).

Cette décision se justifie notamment par la roadmap de VSCode: entre mars et mai 2025 (74, 75), 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 (76). Des premiers efforts d'exploration avait d'ailleurs déjà eu lieu en septembre 2022 (77).

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

POC de surlignage de notre syntaxe avec Tree-Sitter

Ce POC vise à prouver que l'usage de Tree-Sitter fonctionne pour coloriser les préfixes et les propriétés de Snippet 19 pour ne pas avoir cet 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 19. – 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 règles 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 (78).

```
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(" ", $_content))),
    commented_line: (_) => token(seq("//", /./, /.+/)),
    list_line: ($) =>
      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 20. – Résultat de la grammaire minimaliste `grammar.js`, définissant un ensemble de règles sous `rules`.

On observe dans le Snippet 20 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 préfixe, 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 propriété `.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`... et des versions plus spécifiques comme `@string.regex`, `@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 21. – 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 22. – 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** (79) 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 nice to have a des chances de pouvoir être réalisée dans ce travail au vu du résultat atteint avec ce POC.

Le surlignage sémantique 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. À 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 parseur Tree-Sitter (80).

```
≡ 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 transformés 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 client-serveurs. Le format JSON est très populaire pour les API REST, les fichiers de configuration, et d'autres usages.

```
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 23. – Exemple simplifié de parsing de JSON, tiré de leur documentation (81).

```
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 24. – Autre exemple de sérialisation vers JSON d'une structure arbitraire.
Egalement tiré de leur documentation (82).

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

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. (83)

```
edition = "2023";

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

Snippet 25. – Un simple exemple de description d'une personne en ProtoBuf tiré de leur site web (83).

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

Snippet 26. – Et son usage en Java avec les classes autogénérées à la compilation tiré de leur site web (83).

Le langage Rust n'est pas officiellement supporté, mais un projet du nom de PROST! existe (84) 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 (85).

Websocket

Le protocole Websocket, définie dans la RFC 6455 (86), permet une communication bidirectionnelle 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 (86).

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 variable définie) et de distinguer le texte des données binaires. Le texte doit être encodé en UTF-8. (87)

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 différentes routes par exemple `/chat` et `/news`. (88)

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 (synchrone) et async (asynchrone), 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 (89). Elle passe la *Autobahn Test Suite* (suite de tests de plus de 500 cas pour vérifier une implémentation WebSocket) (90).

```
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 27. – Exemple de serveur echo en WebSocket avec la crate `tungstenite`. Tiré de leur README (89)

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 (91).

Il existe une crate `websocket` avec une approche sync et async, qui est dépréciée et dont le README (92) conseille l'usage de `tungstenite` ou `tokio-tungstenite` à la place (92).

Pour conclure cette section, il est intéressant de relever qu'il existe d'autres crates tel que `fastwebsockets` (93) 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 bidirectionnel 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 présentés, 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 28. – 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 (94)

Comme Protobuf, Rust n'est pas supporté officiellement, mais une implémentation du nom de Tonic existe (95), elle utilise PROST! mentionnée précédemment pour l'intégration de Protobuf.

Un article de 2019, intitulé **The state of gRPC in the browser** (96) 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 bidirectionnel 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 serveur). L'autre limite est le nombre maximum de connexions en streaming simultanées dans un navigateur sur HTTP/1.1 fixées à 6 (97), ce qui demande de restructurer ses services gRPC pour ne pas avoir plus de six 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) (98)

tarpc est différent de gRPC et Cap'n Proto « en définissant le schéma directement dans le code, au lieu d'utiliser 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 différents langages. » (Traduction personnelle) (98)

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 websocket 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 combinaison de Websocket et 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 des dizaines/centaines d'étudiants connectés en même moment et que la latence sera trop forte ou que les coûts d'infrastructures deviendront un souci, 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 considérés comme Cap'n Proto (99) qui se veut plus rapide que Protobuf, ou encore Apache Thrift (100). Ces dernières options n'ont pas été explorés dans cet état de l'art principalement parce qu'elles proposent un format binaire.

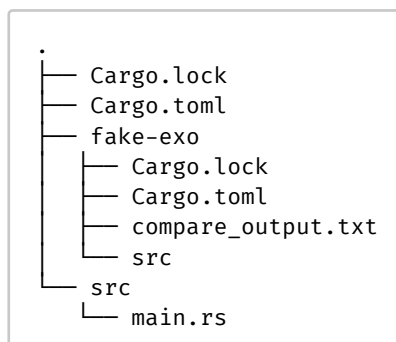
POC de synchronisation de messages JSON via Websocket avec tungstenite

Pour vérifier la faisabilité technique d'envoyer des messages en temps réel en Rust via websocket, 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.

À cause de sa nature interactive, il n'est pas évident de retranscrire ce qui s'y passe quand on lance le POC dans trois shells 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 29. – Structure de fichiers du POC.

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

Snippet 30. – Code Rust de départ de l'exercice fictif à compléter par l'étudiant

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

Annnonce des clients

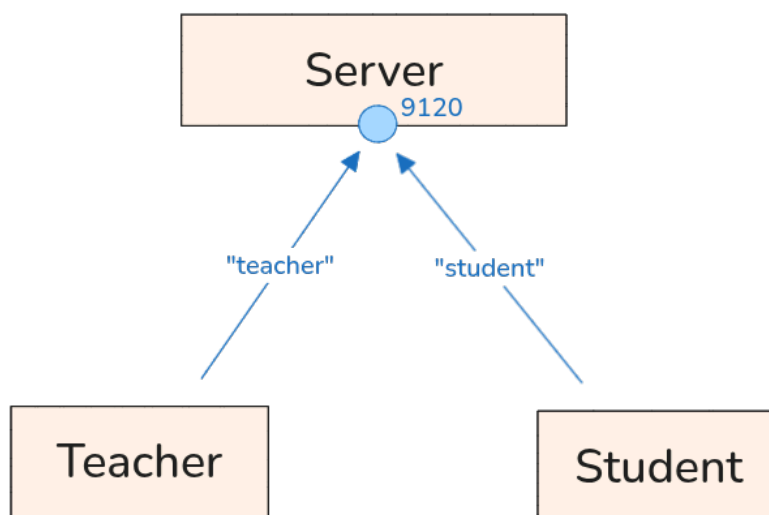


Fig. 14. – La première partie consiste en une mise en place par la connexion et l'annonce des clients de leur rôle, en se connectant puis en envoyant leur rôle en string.

Transfert des résultats des checks

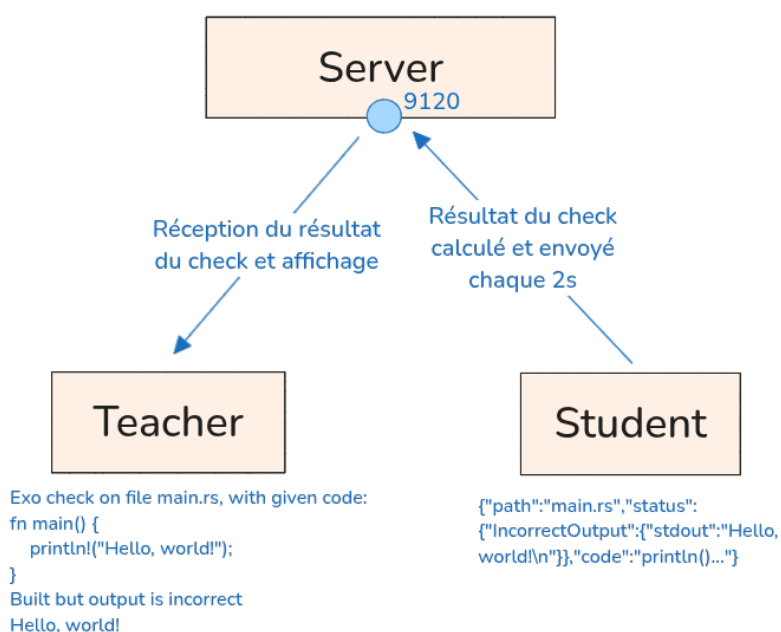


Fig. 15. – La deuxième partie consiste en l'envoi régulier du client du résultat du check vers le serveur, qui ne fait que de transmettre au socket associé au `teacher`.

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 31. – 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 32. – 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 33. – `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 34. – Le processus `student` compile et exécute le check, afin d'envoyer le résultat, ici du type `IncorrectOutput`.

Le Snippet 35 nous montre le détail 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 35. – 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 36. – 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 37. – 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 minimal 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

Protocole de synchronisation

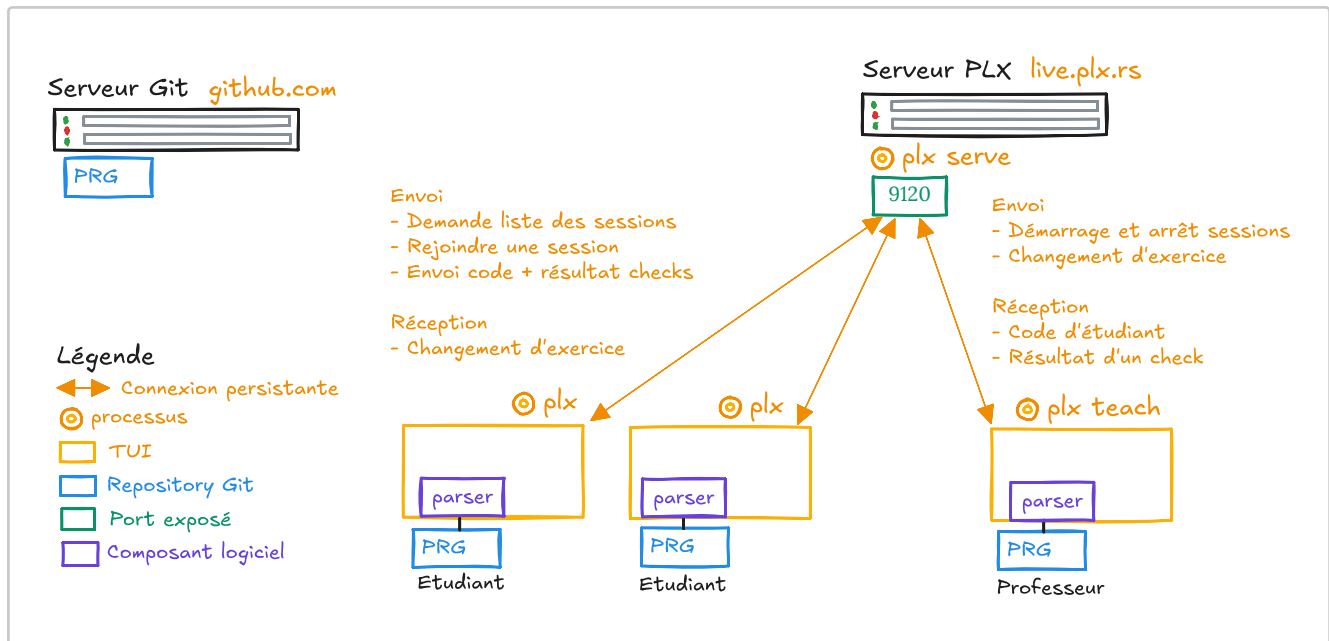


Fig. 16. – Architecture haut niveau décrivant les interactions entre les clients PLX et le serveur de session live

Avertissement: Le protocole détaillé de synchronisation n'a pas encore été défini.

Syntaxe DY

Cette partie décrit d'une manière semi-formelle la syntaxe DY et son usage dans PLX.

Avertissement: ceci est une brouillon, il sera continué les semaines suivantes.

Définition semi-formelle de la syntaxe DY en abstrait

Les préfixes

TODO

Les types de préfixes

TODO

Les propriétés

TODO

Longueurs et types de contenu

TODO

Hiérarchie implicite

Les fins de ligne définissent la fin du contenu pour les préfixes sur une seule ligne. Le préfixe `exo` supporte plusieurs lignes, 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 (uniquement après une ligne préfixée par `check`).

TODO

Détection d'erreurs générales

Usage de la syntaxe dans PLX

TODO

Exemple d'usage dans PLX

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. 17. – Aperçu des possibilités de DY sur un exercice plus complexe

Le Fig. 17 nous montre qu'il existe plusieurs préfixes

- 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.
- **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 lignes 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**).

Toutes les propriétés sont optionnelles, soit elles ont une valeur par défaut, soit la configuration est implicite.

Détection d'erreurs spécifiques à PLX

TODO

Implémentation

Implémentation des POC

Les POC ont été implémenté dans le dossier `pocs` du repository Git du projet. Ce dossier est accessible depuis un navigateur sur <https://github.com/samuelroland/tb-docs/tree/main/pocs>

Implémentation du serveur PLX

TODO

Implémentation de la librairie `dy`

TODO

Intégration de `dy` dans PLX

TODO

Implémentation de la syntaxe Tree-Sitter

TODO

Implémentation du serveur de langage

TODO

Conclusion

Ce rendu intermédiaire termine ainsi les recherches et la rédaction sur l'état de l'art, de nombreuses technologies ont été parcourue. Pour le choix des différentes librairies, une approche basée sur la réduction de la complexité a été privilégiée. Des POC ont été développés pour mieux comprendre et prouver le fonctionnement de Tree-Sitter, la crate `lsp-server` et l'usage de Websocket en Rust via la crate `tungstenite` ainsi que l'envoi des messages en JSON.

TODO

Bibliographie

Avertissement: le format de cette bibliographie n'est pas encore tout à fait correct, notamment sur la gestion des auteurs et des contributeurs. Il manque certains nom d'auteurs ou dates de consultation. Cela sera corrigé par la suite avant la rendu final.

1. ERICSSON, Anders et POOL, Robert. International Edition. Penguin Canada, 2017.
2. PLX. PLX website. En ligne. 2025. Disponible à l'adresse: <https://plx.rs/>
3. rustlings. En ligne. 2025. Disponible à l'adresse: <https://rustlings.rust-lang.org/>
4. PLX. Project status - PLX docs. En ligne. 2025. Disponible à l'adresse: <https://plx.rs/book/status.html>
5. HEIG-VD. GitHub - PRG1-HEIGVD/PRG1_Recueil_Exercices at 6b04c4559d924c394878993c7606e36ae82237ce. En ligne. 30 janvier 2025. Disponible à l'adresse: https://github.com/PRG1-HEIGVD/PRG1_Recueil_Exercices/tree/6b04c4559d924c394878993c7606e36ae82237ce
6. Typst: Compose papers faster. En ligne. 2025. Disponible à l'adresse: <https://typst.app/>
7. CROCKFORD, Douglas et BRAY, Tim. RFC 7159 - The JavaScript Object Notation (JSON) Data Interchange Format - Section 1 Introduction. En ligne. 2025. Disponible à l'adresse: <https://www.rfc-editor.org/rfc/rfc7159.html#section-1>
8. GitHub - danwritecode/clings: rustlings for C....clings. En ligne. 2025. Disponible à l'adresse: <https://github.com/danwritecode/clings>
9. CONTRIBUTEURS. GitHub - mauricioabreu/golings:" rustlings but for golang this time. En ligne. 2025. Disponible à l'adresse: <https://github.com/mauricioabreu/golings>
10. ziglings/exercises: Learn the ⚡ Zig programming language by fixing tiny broken programs. - Codeberg.org. En ligne. 2025. Disponible à l'adresse: <https://codeberg.org/ziglings/exercises>
11. GitHub - MondayMorningHaskell/haskellings: An automated tutorial to teach you about Haskell!. En ligne. 2025. Disponible à l'adresse: <https://github.com/MondayMorningHaskell/haskellings>
12. DUBOVSKOY, Alexey. Cooklang - Recipe Markup Language. En ligne. 2025. Disponible à l'adresse: <https://cooklang.org/>
13. DUBOVSKOY, Alexey. Canonical Cooklang parser in Rust. En ligne. 2025. Disponible à l'adresse: <https://github.com/cooklang/cooklang-rs>
14. RON, Contributeurs de. Rusty Object Notation. En ligne. 2025. Disponible à l'adresse: <https://github.com/ron-rs/ron>
15. TORM. udl v0.3.1 - Parser for UDL (Universal Data Language). En ligne. 2023. Disponible à l'adresse: <https://crates.io/crates/udl>
16. TORM. The Khi data language. En ligne. 2024. Disponible à l'adresse: <https://github.com/khilang/khi>
17. TORM. Rust Khi parser & library. En ligne. 2024. Disponible à l'adresse: <https://github.com/khilang/khi.rs>
18. bitmark Association website. En ligne. 2025. Disponible à l'adresse: <https://www.bitmark-association.org/>
19. BITMARK ASSOCIATION. bitmark Hackathon. En ligne. 2025. Disponible à l'adresse: <https://www.bitmark-association.org/bitmarkhackathon>
20. ASSOCIATION. bitmark Documentation. En ligne. 2025. Disponible à l'adresse: <https://docs.bitmark.cloud/>
21. BITMARK ASSOCIATION. Quizzes - .multiple-choice, .multiple-choice-1. En ligne. 2025. Disponible à l'adresse: <https://docs.bitmark.cloud/quizzes/#multiple-choice-multiple-choice-1>
22. TASKBASE. open-taskpool - 12,000 UK 🇬🇧 → DE 🇩🇪 & DE 🇩🇪 → EN 🇬🇧 learning tasks ready for you to use. En ligne. 2025. Disponible à l'adresse: <https://github.com/taskbase/open-taskpool>
23. BITMARK ASSOCIATION. Quizzes - .cloze (gap text). En ligne. 2025. Disponible à l'adresse: <https://docs.bitmark.cloud/quizzes/#cloze-gap-text>

24. CLASSTIME. Créer la première question / le premier jeu de questions. En ligne. 2024. Disponible à l'adresse: <https://help.classtime.com/fr/comment-commencer-a-utiliser-classtime/creer-la-premiere-question-le-premier-jeu-de-questions>
25. KUNDERT, Ken et KUNDERT, Kale. NestedText – A Human Friendly Data Format. En ligne. 2025. Disponible à l'adresse: <https://github.com/KenKundert/nestedtext>
26. KUNDERT, Ken et KUNDERT, Kale. NestedText documentation - Schemas. En ligne. 2025. Disponible à l'adresse: <https://nestedtext.org/en/latest/schemas.html>
27. BOB22Z. docs.rs - Crate nestedtext. En ligne. 2025. Disponible à l'adresse: <https://nestedtext/latest/nestedtext/>
28. LUDWIG, Sönke. SDLang, Simple Declarative Language. En ligne. 2025. Disponible à l'adresse: <https://sdlang.org/>
29. KAT MARCHÁN (ZKAT), et contributeurs. KDL, a cuddly document language. En ligne. 2025. Disponible à l'adresse: <https://kdl.dev/>
30. All Crates for keyword 'parser'. En ligne. 2025. Disponible à l'adresse: <https://crates.io/keywords/parser>
31. PAGE, Ed (epage). winnow v0.7.8 A byte-oriented, zero-copy, parser combinators library. En ligne. 2025. Disponible à l'adresse: <https://crates.io/crates/winnow>
32. Dependencies of kdl crate. En ligne. 2025. Disponible à l'adresse: <https://crates.io/crates/kdl/6.3.4/dependencies>
33. COUPRIE, Geoffroy (Geal). nom v8.0.0 A byte-oriented, zero-copy, parser combinators library. En ligne. 2025. Disponible à l'adresse: <https://crates.io/crates/nom>
34. Reverse dependencies of nom crate. En ligne. 2025. Disponible à l'adresse: https://crates.io/crates/nom/reverse_dependencies
35. pest v2.8.0 The Elegant Parser. En ligne. 2025. Disponible à l'adresse: <https://crates.io/crates/pest>
36. (MARWES), Markus Westerlind. combine v4.6.7 Fast parser combinators on arbitrary streams with zero-copy support. En ligne. 2024. Disponible à l'adresse: <https://crates.io/crates/combine>
37. JOSHUA BARRETTO (ZESTERER), Rune Tynan (CraftSpider), et contributeurs. chumsky v0.10.1 A parser library for humans with powerful error recovery. En ligne. 2025. Disponible à l'adresse: <https://crates.io/crates/chumsky>
38. Most popular Rust libraries. En ligne. 2025. Disponible à l'adresse: <https://lib.rs/std>
39. Serde data model. En ligne. 2025. Disponible à l'adresse: <https://serde.rs/data-model.html>
40. MICROSOFT, et contributeurs. Language Server Protocol. En ligne. 2025. Disponible à l'adresse: <https://microsoft.github.io/language-server-protocol/>
41. JSON-RPC WORKING GROUP. JSON-RPC 2.0 Specification. En ligne. 2013. Disponible à l'adresse: <https://www.jsonrpc.org/specification>
42. MICROSOFT, et contributeurs. Language Server Protocol Specification - 3.17 - Capabilities. En ligne. 2025. Disponible à l'adresse: <https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/#capabilities>
43. MICROSOFT, et contributeurs. Language Server Protocol Specification - 3.17 - Content part. En ligne. 2025. Disponible à l'adresse: <https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/#contentPart>
44. BERGERCOOKIE, et contributeurs. asm-lsp v0.10.0 Language Server for x86/x86_64, ARM, RISC-V, and z80 Assembly Code. En ligne. 2025. Disponible à l'adresse: <https://crates.io/crates/asm-lsp>
45. ORGANISATION, et contributeurs eclipse-jdtls. GitHub - eclipse-jdtls/eclipse.jdt.ls: Java language server. En ligne. 2025. Disponible à l'adresse: <https://github.com/eclipse-jdtls/eclipse.jdt.ls>
46. TAILWINDLABS, et contributeurs. GitHub - tailwindlabs/tailwindcss-intellisense: Intelligent Tailwind CSS tooling for Visual Studio Code. En ligne. 2025. Disponible à l'adresse: <https://github.com/tailwindlabs/tailwindcss-intellisense>
47. ORGANISATION, et contributeurs typescript-language-server. GitHub - typescript-language-server/typescript-language-server: TypeScript & JavaScript Language Server. En ligne. 2025. Disponible à l'adresse: <https://github.com/typescript-language-server/typescript-language-server>
48. LSP-TYPES, Contributeurs de. lsp-types v0.97.0 Types for interaction with a language server, using VSCode's Language Server Protocol. En ligne. 2024. Disponible à l'adresse: <https://crates.io/crates/lsp-types>
49. ORGANISATION GLUON-LANG, et contributeurs. Reverse dependencies of lsp-types crate. En ligne. 2024. Disponible à l'adresse: https://crates.io/crates/lsp-types/reverse_dependencies
50. MICROSOFT, et contributeurs. Implementations - Tools supporting the LSP. En ligne. 2025. Disponible à l'adresse: <https://microsoft.github.io/language-server-protocol/implementors/tools/>

51. MICROSOFT, et contributeurs. Implementations - Language Servers. En ligne. 2025. Disponible à l'adresse: <https://microsoft.github.io/language-server-protocol/implementors/servers/>
52. OXALICA, et contributeurs. async-lsp v0.2.2 Asynchronous Language Server Protocol (LSP) framework based on tower. En ligne. 2025. Disponible à l'adresse: <https://crates.io/crates/async-lsp>
53. OXALICA, et contributeurs. nil/crates/nil/Cargo.toml - Nix Language server, an incremental analysis assistant for writing in Nix. En ligne. 2025. Disponible à l'adresse: <https://github.com/oxalica/nil/blob/577d160da311cc7f5042038456a0713e9863d09e/crates/nil/Cargo.toml#L11>
54. MYRIAD-DREAMIN, et contributeurs. sync-ls - Synchronized language service inspired by async-lsp, primarily for tinyml. En ligne. 2025. Disponible à l'adresse: <https://crates.io/crates/sync-ls>
55. ORGANISATION, et contributeurs tower-lsp-community. tower-lsp-server v0.21.1 Language Server Protocol implementation based on Tower. En ligne. 2025. Disponible à l'adresse: <https://crates.io/crates/tower-lsp-server>
56. ORGANISATION, et contributeurs rust-lang. Reverse dependencies of lsp-server crate. En ligne. 2024. Disponible à l'adresse: https://crates.io/crates/lsp-server/reverse_dependencies
57. EYAL KALDERON, et contributeurs. Reverse dependencies of tower-lsp crate. En ligne. 2023. Disponible à l'adresse: https://crates.io/crates/tower-lsp/reverse_dependencies
58. ORGANISATION RUST-LANG, ET CONTRIBUTEURS. rust-analyzer/lib/lsp-server/examples/goto_def.rs at master · rust-lang/rust-analyzer · GitHub. En ligne. 2025. Disponible à l'adresse: https://github.com/rust-lang/rust-analyzer/blob/master/lib/lsp-server/examples/goto_def.rs
59. En ligne. 2025. Disponible à l'adresse: https://macromates.com/manual/en/regular_expressions
60. En ligne. 2025. Disponible à l'adresse: <https://code.visualstudio.com/api/language-extensions/syntax-highlight-guide>
61. En ligne. 2025. Disponible à l'adresse: <https://www.jetbrains.com/help/idea/textmate.html>
62. LTD, MacroMates. Language Grammars – TextMate 1.x Manual - Example Grammar. En ligne. Disponible à l'adresse: https://macromates.com/manual/en/language_grammars#example_grammar
63. CONTRIBUTEURS DE TREE-SITTER. Introduction - Tree-sitter. En ligne. 2025. Disponible à l'adresse: <https://tree-sitter.github.io/tree-sitter/>
64. CONTRIBUTEURS DE TREE-SITTER. Creating Parsers - Getting Started - Tree-sitter. En ligne. 2025. Disponible à l'adresse: <https://tree-sitter.github.io/tree-sitter/creating-parsers/1-getting-started.html>
65. Neovim Documentation - Treesitter. En ligne. 2025. Disponible à l'adresse: <https://neovim.io/doc/user/treesitter.html>
66. Language Extensions - Grammar. En ligne. 2025. Disponible à l'adresse: <https://zed.dev/docs/extensions/languages?#grammar>
67. Creating a Grammar. En ligne. 2025. Disponible à l'adresse: <https://flight-manual.atom-editor.cc/hacking-atom/sections/creating-a-grammar/>
68. GITHUB, et contributeurs. Navigating code on GitHub. En ligne. 2025. Disponible à l'adresse: <https://docs.github.com/en/repositories/working-with-files/using-files/navigating-code-on-github>
69. MICROSOFT, et contributeurs. Semantic Highlight Guide | Visual Studio Code Extension API. En ligne. 2025. Disponible à l'adresse: <https://code.visualstudio.com/api/language-extensions/semantic-highlight-guide>
70. MICROSOFT, et contributeurs. Language Server Protocol Specification - 3.17 - Semantic Tokens. En ligne. 2025. Disponible à l'adresse: https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/#textDocument_semanticTokens
71. SUBLIMEHQ. SublimeHQ - End User License Agreement. En ligne. 2025. Disponible à l'adresse: <https://www.sublimehq.com/eula>
72. SUBLIMEHQ. Syntax Definitions. En ligne. 2025. Disponible à l'adresse: <https://www.sublimetext.com/docs/syntax.html>
73. STACK EXCHANGE INC. Technology | 2024 Stack Overflow Developer Survey - Integrated development environment. En ligne. 2025. Disponible à l'adresse: <https://survey.stackoverflow.co/2024/technology#1-integrated-development-environment>
74. MICROSOFT, et contributeurs. Iteration Plan for March 2025 · Issue #243015 · microsoft/vscode · GitHub. En ligne. 2025. Disponible à l'adresse: <https://github.com/microsoft/vscode/issues/243015>
75. MICROSOFT, et contributeurs. Iteration Plan for May 2025 · Issue #248627 · microsoft/vscode · GitHub. En ligne. 2025. Disponible à l'adresse: <https://github.com/microsoft/vscode/issues/248627>

-
76. MICROSOFT, et contributeurs. Explore using tree sitter for syntax highlighting · Issue #210475 · microsoft/vscode · GitHub. En ligne. 2025. Disponible à l'adresse: <https://github.com/microsoft/vscode/issues/210475>
 77. MICROSOFT, et contributeurs. [Exploration] Tree-sitter tokenization exploration (Fixes #161256) by aiday-mar · Pull Request #161479 · microsoft/vscode · GitHub. En ligne. 2022. Disponible à l'adresse: <https://github.com/microsoft/vscode/pull/161479>
 78. CONTRIBUTEURS DE TREE-SITTER. The Grammar DSL - Tree-sitter. En ligne. 2025. Disponible à l'adresse: <https://tree-sitter.github.io/tree-sitter/creating-parsers/2-the-grammar-dsl.html>
 79. SIRAPHOB, Ben. How to write a tree-sitter grammar in an afternoon | siraben's musings. En ligne. 2022. Disponible à l'adresse: <https://siraben.dev/2022/03/01/tree-sitter.html>
 80. GitHub - AlecGhost/tree-sitter-vscode: Bring the power of Tree-sitter to VSCode. En ligne. 2025. Disponible à l'adresse: <https://github.com/AlecGhost/tree-sitter-vscode>
 81. serde_json - Parsing JSON as strongly typed data structures. En ligne. 2025. Disponible à l'adresse: https://docs.rs/serde_json/latest/serde_json/index.html#parsing-json-as-strongly-typed-data-structures
 82. serde_json - Constructing JSON values. En ligne. 2025. Disponible à l'adresse: https://docs.rs/serde_json/latest/serde_json/index.html#constructing-json-values
 83. GOOGLE, et contributeurs. Protocol Buffers Documentation. En ligne. 2025. Disponible à l'adresse: <https://protobuf.dev/>
 84. CONTRIBUTEURS. GitHub - tokio-rs/prost: PROST! a Protocol Buffers implementation for the Rust Language. En ligne. 2025. Disponible à l'adresse: <https://github.com/tokio-rs/prost>
 85. rmp - The Rust MessagePack Library. En ligne. 2025. Disponible à l'adresse: <https://docs.rs/rmp/latest/rmp/>
 86. IAN FETTE, Alexey Melnikov. RFC 6455: The WebSocket Protocol. En ligne. 2025. Disponible à l'adresse: <https://www.rfc-editor.org/rfc/rfc6455>
 87. IAN FETTE, Alexey Melnikov. RFC 6455: The WebSocket Protocol - 1.5. Design Philosophy. En ligne. 2025. Disponible à l'adresse: <https://www.rfc-editor.org/rfc/rfc6455#section-1.5>
 88. IAN FETTE, Alexey Melnikov. RFC 6455: The WebSocket Protocol - 1.3. Opening Handshake. En ligne. 2025. Disponible à l'adresse: <https://www.rfc-editor.org/rfc/rfc6455#section-1.3>
 89. CONTRIBUTEURS, Snapview GmbH et. Lightweight stream-based WebSocket implementation. En ligne. 2025. Disponible à l'adresse: <https://crates.io/crates/tungstenite>
 90. CONTRIBUTEURS. GitHub - crossbario/autobahn-testsuite: Autobahn WebSocket protocol testsuite. En ligne. 2025. Disponible à l'adresse: <https://github.com/crossbario/autobahn-testsuite>
 91. CONTRIBUTEURS, Snapview GmbH et. tokio-tungstenite. En ligne. 2025. Disponible à l'adresse: <https://crates.io/crates/tokio-tungstenite>
 92. CONTRIBUTEURS. websocket. En ligne. 2025. Disponible à l'adresse: <https://crates.io/crates/websocket>
 93. CONTRIBUTEURS. fastwebsockets. En ligne. 2025. Disponible à l'adresse: <https://crates.io/crates/fastwebsockets>
 94. AUTHORS. Introduction to gRPC | gRPC. En ligne. 2025. Disponible à l'adresse: <https://grpc.io/docs/what-is-grpc/introduction/>
 95. CONTRIBUTEURS. GitHub - hyperium/tonic: A native gRPC client & server implementation with async/await support. En ligne. 2025. Disponible à l'adresse: <https://github.com/hyperium/tonic>
 96. BRANDHORST, Johan. The state of gRPC in the browser | gRPC. En ligne. 2019. Disponible à l'adresse: <https://grpc.io/blog/state-of-grpc-web>
 97. CONTRIBUTEURS, MDN. EventSource - Web APIs | MDN. En ligne. 2025. Disponible à l'adresse: <https://developer.mozilla.org/en-US/docs/Web/API/EventSource>
 98. CONTRIBUTEURS. GitHub - google/tarpc: An RPC framework for Rust with a focus on ease of use. En ligne. 2025. Disponible à l'adresse: <https://github.com/google/tarpc>
 99. Cap'n Proto: Introduction. En ligne. 2025. Disponible à l'adresse: <https://capnproto.org/>
 100. Apache Thrift - Home. En ligne. 2025. Disponible à l'adresse: <https://thrift.apache.org/>
 101. LEARNEAO. Free AI Grammar Checker - LanguageTool. En ligne. 2025. Disponible à l'adresse: <https://languagetool.org/>
 102. GitHub - DACC4/HEIG-VD-typst-template-for-TB: This template is a typst version of a LaTeX template for the travail de bachelor (TB) used at the HEIG-VD. En ligne. 2025. Disponible à l'adresse: <https://github.com/DACC4/HEIG-VD-typst-template-for-TB>
-

-
103. JONASLOOS. BibTeX to Hayagriva. En ligne. 2025. Disponible à l'adresse: <https://jonasloos.github.io/bibtex-to-hayagriva-webapp/>
 104. PLX. Development - PLX docs - Logo design. En ligne. 2025. Disponible à l'adresse: <https://plx.rs/book/dev.html#logo-design>

Figures

| | | |
|---------|---|----|
| Fig. 1 | Aperçu de la page d'accueil de PLX dans le terminal (4) | 9 |
| Fig. 2 | Aperçu d'un exercice de C dans PLX, avec un check qui échoue et 2 autres qui passent (4) | 9 |
| Fig. 3 | Interactions entre les clients PLX chez l'enseignant·e et les étudiant·es, le code est synchronisé via un serveur central, le cours « PRG2 » a un repository Git publique | 10 |
| Fig. 4 | Equivalent de l'exercice du Snippet 1, dans une version préliminaire de la syntaxe DY ... | 15 |
| Fig. 5 | Aperçu de l'expérience imaginée de rédaction imaginée dans un IDE | 15 |
| Fig. 6 | Un exemple de Rustlings en haut dans le terminal et VSCode en bas, sur un exercice de fonctions | 17 |
| 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> | 28 |
| 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. | 30 |
| Fig. 9 | Liste de symboles générées par Tree-Sitter, affichés à droite du code sur GitHub pour un exemple de code Rust de PLX | 33 |
| 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. (69) | 34 |
| Fig. 11 | Concrete Syntax Tree généré par la grammaire définie sur le fichier <code>mcq.dy</code> | 37 |
| Fig. 12 | Screenshot du résultat de la commande <code>tree-sitter highlight mcq.dy</code> avec notre exercice surligné | 38 |
| 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 | 38 |
| 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. | 44 |
| 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> | 44 |
| Fig. 16 | Architecture haut niveau décrivant les interactions entre les clients PLX et le serveur de session live | 47 |
| Fig. 17 | Aperçu des possibilités de DY sur un exercice plus complexe | 49 |

Tables

Annexes

Outils utilisés

Usage de l'intelligence artificielle

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, en raison d'une faible utilisation ou décrits avec d'autres mots-clés
- pour trouver la raison de certaines erreurs d'exécution ou de compilation dans les POC fait en Rust
- pour mieux comprendre les règles de précédence de Tree-Sitter et avoir des exemples
- avec LanguageTool pour trouver les fautes d'orthographe ou de frappe et les corriger, basé sur des règles logiques et sur l'IA (101)

Outils techniques

- Neovim pour l'édition du rapport et l'écriture du code
- Template Typst `HEIG-VD typst template for TB` (102)
- Convertisseur de BibTex vers Hayagriva (103)

Logo

Le logo de PLX utilisé sur la page de titre a été créé par l'auteur de ce travail au commencement du projet PLX, bien avant ce travail de Bachelor (104).

Cahier des charges original

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

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 l'idée 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 transformer les longs moments de théorie en session d'entraînement dynamique, et redéfinir l'expérience des étudiants sur ces exercices ainsi que les laboratoires. L'ambition est qu'à terme, cela génère un apprentissage plus profond de modèles mentaux solides, pour que les étudiants aient moins de difficultés avec ces cours.

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. 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.

Défis

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.

Pour faciliter l'adoption de ces outils et la rapidité de création/transcription d'exercices, 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.

Voici un exemple préliminaire de la syntaxe DY qui permettra de décrire un exercice de programmation dans PLX. Elle contient 2 checks pour vérifier le comportement attendu. Le premier cas décrit un check de succès et le deuxième cas décrit une situation d'erreur.

```
exo Just greet me

checks
name Can enter the full name and be greeted
see What is your firstname ?
type John
see Hello John, what's your lastname ?
type Doe
see Have a nice day John Doe !
exit 0

name Stops if the name contains number
see What is your firstname ?
type Alice23
see Firstname cannot contain digits.
exit 1
```

Ces 2 défis impliquent :

1. Une partie serveur de PLX, gérant des connexions persistantes pour chaque étudiant et enseignant connecté, permettant de recevoir les réponses des étudiants et de les renvoyer à l'enseignant. Une partie client est responsable d'envoyer le code modifié et les résultats après chaque lancement des checks.
1. Le but est de définir une syntaxe et de réécrire le parseur en Rust en s'aidant d'outils adaptés (TreeSitter, Chumsky, Winnow, ...).

Le projet, les documents et les contributions de ce TB, seront publiés sous licence libre.

Objectifs et livrables

1. Livrables standards : rapport intermédiaire ; rapport final ; résumé ; poster.
1. Un serveur en Rust lancé via le CLI plx permettant de gérer des sessions live.
1. Une librairie en Rust de parsing de la syntaxe DY.
1. Une intégration de cette librairie dans PLX.

Objectifs fonctionnels

Les objectifs fonctionnels posent l'hypothèse du cas d'utilisation où un professeur lance une session live pour plusieurs étudiants. Il n'y a cependant pas de rôle spécifique attribué au professeur par rapport aux étudiants, il y a seulement une distinction des permissions entre le créateur de la session et ceux qui la rejoignent.

1. Les professeurs peuvent lancer et stopper une session live via PLX liée au repository actuel, via un serveur défini dans un fichier de configuration présent dans le repository. Il peut exister plusieurs sessions en même temps pour le même repository (afin de supporter plusieurs cours en parallèle dans plusieurs classes). Ils donnent un nom à la session, afin que les étudiants puissent l'identifier parmi les sessions ouvertes. Un code de vérification unique est généré par session permettant de distinguer 2 sessions du même nom dans le même repos.
1. En tant qu'étudiant, une fois le repository cloné, il est possible de lancer PLX, de lister les sessions ouvertes et de rejoindre une session en cours en s'assurant du code de vérification. Un numéro unique incrémental est attribué à chaque étudiant pour la session.
1. Le professeur peut choisir une série d'exercices parmi ceux affichés par PLX, lancer un exercice et gérer le rythme d'avancement de la classe. Cet exercice sera affiché directement chez les étudiants ayant rejoint.

1. Une vue globale permet au professeur d'avoir un aperçu général de l'état des checks sur tous les exercices. En sélectionnant un exercice, il est possible de voir la dernière version du code édité ainsi que les résultats des checks pour ce code, pour chaque étudiant.
1. L'intégration de la librairie `dy` dans PLX permet de décrire les informations d'un cours, des compétences et des exercices. Elle détecte les erreurs spécifiques à PLX.
1. L'intégration dans PLX permet d'utiliser uniquement des fichiers `.dy` pour décrire le contenu. Elle doit aussi afficher les erreurs dans une liste sur une commande dédiée (par ex. `plx check`).

Objectifs non fonctionnels

1. Une session live doit supporter des déconnexions temporaires, le professeur pourra continuer à voir la dernière version du code envoyé et le client PLX essaiera automatiquement de se reconnecter. Le serveur doit pouvoir supporter plusieurs sessions live incluant au total 300 connexions persistantes simultanées.
2. Une session live s'arrête automatiquement après 30 minutes après déconnexion du professeur, cela ne coupe pas l'affichage de l'exercice en cours aux étudiants
3. Pour des raisons de sécurité, aucun code externe ne doit être exécuté automatiquement par PLX. Seule une exécution volontaire par une action dédiée peut le faire.
4. Le temps entre la fin de l'exécution des checks chez l'étudiant et la visibilité des modifications par l'enseignant ne doit pas dépasser 3s.
5. Le code doit être le plus possible couvert par des tests automatisés, notamment par des tests end-to-end avec de multiples clients PLX.
6. Le parseur DY doit être assez capable de parser 200 exercices en $< 1s$.
7. Retranscrire à la main un exercice existant du Markdown en PLX DY ne devrait pas prendre plus d'une minute.

Objectif nice to have

1. La librairie `dy` permettrait d'intégrer le parseur et les erreurs spécifiques à un langage server permettant une expérience complète d'édition dans VSCode et Neovim.
2. La librairie `dy` serait également capable de générer des définitions TreeSitter pour supporter le syntax highlighting via ce système.

Calendrier du projet

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.