Table of Contents

Dictionnaire

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

Etat de l'art

Format de données existant orienté humainement éditable

Ces recherches ignorent les formats de données largement supporté et répandu tel que le XML, JSON, YAML et TOML. Ils sont tout à fait adapter pour des configurations, de la sérialisation et de l'échange de donnée et sont pour la plupart facilement lisible. Cependant la quantité de séparateurs et délimiteurs en plus du contenu qu'ils n'ont pas été optimisé pour la rédaction par des humains. Le YAML et le TOML, bien que plus léger que le JSON, inclue de nombreux types de données autre que les strings, des tabulations et des guillemets, ce qui rend la rédaction plus fastidieuse qu'en Markdown.

On cherche quelque chose du niveau de simplicité du Markdown en terme de rédaction, mais avec une validation poussée customisable par le projet qui définit le schéma.

TODO: continuer markdown inspiration + besoin

Ces recherches se focalisent sur les syntaxes qui ne sont pas spécifique à un domaine ou qui seraient complètement déliée de l'informatique ou de l'éducation. Ainsi, l'auteur ne présente pas Cooklang [1], qui se veut une langage de balise pour les recettes de cuisines, même si l'implémentation du parseur en Rust [2] 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) [3], de par leur nécessité de connaître un peu la syntaxe du Rust et surtout parce qu'elle ne simplifie pas vraiment l'écriture comparé à du YAML. On ignore également les projets dont la spécification ou l'implémentation est en état de « brouillon » et n'est pas encore utilisable en production.

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

KHI - Le langage de données universel

D'abord nommée UDL (Universal Data Language) [4], cette syntaxe a été inventée pour mixer les possibilités du JSON, YAML, TOML, XML, CSV et Latex, afin de supporter toutes les structures de données modernes. Plus concrètement le markup, les structs, les listes, les tuples, les tables/matrices, les enums, les arbres hiérarchiques sont supportés. Les objectifs sont la polyvalence, un format source (fait pour être rédigé à la main), l'esthétisme et la simplicité.

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

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

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

Snippet 1. – Un exemple simplifié de KHI de leur README [5], décrivant un exemple d'article d'encyclopédie.

Une implémentation en Rust est proposée [6]. Son dernier commit sur ces 2 repositorys date du 11.11.2024, le projet a l'air de ne pas être fini au vu des nombreux `todo!()` présent dans le code. La large palette de structures supportées implique une charge mentale additionnelle pour se rappeler, ce qui en fait une mauvaise option pour PLX.

Bitmark - le standard des contenus éducatifs digitaux

Bitmark est un standard open-source, qui vise à uniformiser tous les formats de données utilisés pour décrire du contenu éducatif digital sur les nombreuses plateformes existantes [7]. Cette diversité de formats rend l'interropérabilité très difficile et freine l'accès à la connaissance et restreint les créateurs de contenus et les éditeurs dans les possibilités de migration entre plateformes. La stratégie est de définir un format basé sur le contenu (Content-first) plus que basé sur son rendu (layout-first) permettant un affichage sur tous type d'appareils incluant les appareils mobiles [7]. C'est la Bitmark Association en Suisse à Zurich qui développe ce standard, notamment à travers des Hackatons organisés en 2023 et 2024 [8].

Le standard permet de décrire du contenu statique et interactif, comme des articles ou des quiz de divers formats. 2 formats équivalents sont définis: le bitmark markup language et le bitmark JSON data model [9]

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

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

Snippet 2. – Un exemple de question à choix multiple tiré de leur documentation [10]. 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 3. – Equivalent de Snippet 2 dans le Bitmark Json data model [10]

Open Taskpool, projet qui met à disposition des exercices d'apprentissage de langues [11], fournit une API JSON utilisant le Bitmark JSON data model.

Demander à Open Taskpool des exercices d'allemand vers anglais autour du mot `school` de format `cloze` (texte à trou), se fait avec cette simple requête: `https://taskpool.taskbase.com/exercises?translationPair=de→en&word=school&exerciseType=bitmark.cloze`.

```
...
"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 4. – Extrait simplifié de la réponse JSON, respectant le standard Bitmark [12]. 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 [13], on voit que le système de création d'exercices est basé sur des formulaires. Ces 2 exemples donnent l'impression que la structure JSON est plus utilisée que le markup. Au vu de tous séparateurs et symboles de ponctuations à se rappeler, la syntaxe n'a peut-être pas été imaginée dans le but d'être rédigée à la main directement. Finalement, Bitmark ne spécifie pas de type d'exercices programmation nécessaire à PLX.

NestedText — Un meilleur JSON

NestedText se veut human-friendly, similaire au JSON mais pensé pour être facile à modifier et visualiser par les humains. Le seul type de donnée scalaire supporté est la chaîne de caractères, afin de simplifier la syntaxe et retirer le besoin de mettre des guillemets. La différence avec le YAML, en plus des types de données restreint est la facilité d'intégrer des morceaux de code sans échappements ni guillemets, les caractères de données ne peuvent pas être confondus avec NestedText [14].

```
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 5. – Exemple tiré de leur README [14]

Ce format a l'air assez léger visuellement et l'idée de faciliter l'intégration de blocs multi-lignes sans contraintes de charactères 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 librairies externes qui vérifient le schéma [15]. De plus, l'implémentation officielle est en Python et il n'y a pas d'implémentation Rust disponible; il existe une crate réservée mais vide [16].

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 [17]). SDLang définit également différents types de nombres (32bit, 64bit, entier, flottant, …), 4 valeurs de booléans (`true` , `false` , `on` , `off`) comme en YAML, différents formats de dates et un moyen d'intégrer des données binaires encodées en Base64.

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

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

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

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

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

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

Snippet 6. – Exemple tiré de leur site web [17]

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

KDL - Le « 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 7. – Exemple simplifié tiré de leur site web [18]

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

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

Conclusion

En conclusion, au vu du nombre de tentatives/variantes trouvées, on voit que la verbosité des formats largement répandu du XML, JSON et même du YAML est un problème qui ne touche pas que l'auteur. Le gain de verbosité des syntaxes listées est réel mais reste ciblé sur un usage plus avancé de structure de données et types variés. L'auteur pense pouvoir proposer une approche encore plus légère et plus simple, inspirée du style du Markdown en évitant une partie des charactères non explicites.

TODO finish + merge intro above

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 [19], j'ai trouvé la liste de librairies suivantes:

- `winnow` [20], fork de `nom`, utilisé notamment par le parseur Rust de KDL [21]
- `nom` [22], utilisé notamment par `cexpr` [23]
- `pest` [24]
- `combine` [25]
- `chumsky` [26]

A noter aussi l'existance de la crate `serde`, un framework de sérialisation et desérialisation très populaire dans l'écosystème Rust (selon lib.rs [27]). 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 [28], qui supporte 29 types de données, ce projet paraît à l'auteur apporter plus de complexités qu'autre chose pour trois raisons:

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

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

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

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

```
exo Dog struct
Consigne très longue

en *Markdown*
sur plusieurs lignes

xp 20
checks
...
```

Snippet 8. – Exemple d'un début d'exercice de code, on voit que la consigne se trouve après la ligne `exo` et continue sur plusieurs lignes jusqu'à qu'on trouve un autre préfixe (ici `xp` qui est optionnel ou alors `checks`).

Systèmes de surglignage de code

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

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

Textmate

Textmate est un IDE pour MacOS qui a inventé un système de grammaire Textmate. Elles permettent de décrire comment tokeniser le code basée sur des expressions régulières. Ces expressions régulières viennent de la librairie Oniguruma [29]. VSCode utilise ces grammaires Textmate [30]. Intellij IDEA l'utilise également pour les langages non supportés par Intellij IDEA [31].

Tree-Sitter

Tree-Sitter [32] se définit comme un « outil de génération de parser et une librairie de parsing incrémentale. Il peut construire un arbre de syntaxe concret (CST) pour depuis un fichier source et efficacement mettre à jour cet arbre quand le fichier source est modifié. » [32] (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 librarie C sans dépendance dans n'importe quelle type d'application [32], [33].

Etant donné Snippet 9, le défi est d'arriver à coloriser les préfixes et les flags pour ne pas avoir cette affichage noir sur blanc qui ne facilite pas la lecture.

```
// Basic MCQ exo
exo Introduction
opt .multiple
- C is an interpreted language
- .ok C is a compiled language
- C is mostly used for web applications
```

Snippet 9. – Un exemple de question choix multiple, décrite avec la syntaxe DY. Les préfixes sont `exo` (titre) et `opt` (options). Les flags 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 régle construites via des fonctions fournies par Tree-Sitter et des expressions régulières.

```
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, /.+/)),
  },
});
```

On observe dans cet exemple un fichier source, découpé en une répétition de ligne. Il y a 4 types de lignes qui sont chacunes décrites avec des plus petits morceaux. `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 flags insérés dans les tokens de `prefix` et `property`. La documentation The Grammar DSL de la documentation explique toutes les options possibles en détails [34].

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 charactères associée au noeud. Par exemple, on voit que l'option `C is a compiled language` a bien été extraite à la ligne 4, entre le byte 6 et 30 (`4:6 - 4:30`) en tant que `content`. Elle suit un token de `property` avec notre flag `.ok` et le tiret de la règle `dash`.

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

La tokenisation fonctionne bien pour cette 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 ressembles à `@variable`, `@constant`, `@function`, `@keyword`, `@string` etc... et des versions plus spécifiques comme `@string.regexp`, `@string.special.path`. Ces noms sont ensuite utilisés par les thèmes pour appliquer un style.

```
> cat queries/highlights.scm
(prefix) @keyword
(commented_line) @comment
(content) @string
(property) @property
(dash) @operator
```

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.

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

```
// 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. 1. – Résultat final surligné par `tree-sitter highlighting mcq.dy`

Tree-Sitter est supporté dans Neovim [35], dans le nouvel éditeur Zed [36], ainsi que d'autres. Tree-Sitter a été inventé par l'équipe derrière Atom [37].

Semantic highlighting

Choix final

Si le temps le permet, une grammaire développée avec Tree-Sitter permettra de supporter du surglignage dans Neovim. Le choix de ne pas explorer plus les grammaires Textmate se justifie également par l'intégration en cours de Tree-Sitter dans de VSCode

Les serveurs de language et librairies Rust existantes

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

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

La communication entre l'éditeur et un serveur de langage démarré pour le fichier en cours, se fait via le `Language Server Protocol (LSP)`. Ce protocole inventé par Microsoft pour VSCode, résoud le problème des développeurs de langages qui doivent supporter chaque éditeur de code indépendamment avec des APIs légèrement différentes pour faire la même chose. Le projet a pour but également de simplifier la vie des nouveaux éditeurs pour intégrer rapidement des dizaines de langages via ce protocole commun et standardisé [38].

Fig. 2. – Exemple d'auto-complétion dans Neovim, générée par le serveur de language `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 optionnelement `params` [39]. 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 10
- Un serveur de langage n'a pas besoin d'implémenter toutes les fonctionnalités du protocole. Un système « Capabilities » est défini pour annoncer les méthodes implémentées [40].

- Le transport des messages JSON-RPC peut se faire en `stdio` (flux standard entrée et sorties), sockets TCP ou même en HTTP.

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

Snippet 10. – 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 [41]

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 language officiel du langage Rust
- `asm-lsp` [42], 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 language pour Java implémenté en Java [43]
- `tailwindcss-language-server`, le serveur de langage pour le framework CSS TailwindCSS, implémenté en TypeScript [44]
- `typescript-language-server` et pour finir celui pour TypeScript, implémenté en TypeScript également [45]
- et beaucoup d'autres projets existent…

Une crate commune à plusieurs projet est `lsp-types` [46] qui définit les structures de données, comme `Diagnostic`, `Position`, `Range`. Ce projet est utilisé par `lsp-server`, `tower-lsp`, `lspower` et d'autres [47]

L'auteur a modifié et exécuté l'exemple de `goto_def.rs` fourni par la crate `lsp-server` [48]. Il a aussi créé un script `demo.fish` permettant de lancer la communication en stdin et attendre entre chaque requête. Cet exemple minimaliste mais clair démontre la communication qui se produit quand on clique sur un `Aller à la définition` dans un IDE. L'IDE va lancer le serveur de langage associé au fichier édité en lancant 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":{"char
acter":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. 3. – Exemple de discussion en LSP une demande de `textDocument/definition`, output de `fish demo.fish` dans le dossier `pocs/lsp-server-demo`.
Les lignes après `CLIENT:` sont envoyés en stdin et celles après `SERVER` sont reçues en stdout.

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

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

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

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

Adoption
Selon la liste sur le site de la spécification [49], 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 [50] quand à elle, contient plus de 200 projets, dont 40 implémentés en Rust! Ce large support et ces nombreux exemples va grandement faciliter le développement de ce serveur de language et son intégrations dans différents IDE.

Librairies disponibles
En cherchant à nouveau sur `crates.io` sur le tag `lsp`, on trouve différent projets dont `async-lsp` [51] utilisée dans `nil` [52] (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 [53]. En continuant la recherche on trouve encore un autre `tower-lsp` et un fork `tower-lsp-server` [54]… `rust-analyzer` a également extrait une crate `lsp-server`.

Choix final
L'auteur travaillant dans Neovim, l'intégration ne sera faite que dans Neovim pour ce TB, l'intégration dans VSCode pourra être fait dans le futur et devrait être relativement simple.

Les 2 projets les plus utilisés (en terme de reverse dependencies sur crates.io) sont `lsp-server` [55] (56) et `tower-lsp` (85) [56]. 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 de ce travail, il n'est pas sûr qu'elle puisse aboutir.

Protocoles de synchronisation existants

gRPC

Websockets

Choix final

Bibliographie

[1] A. Dubovskoy, « Cooklang – Recipe Markup Language ». [En ligne]. Disponible sur: https://cooklang.org/

[2] A. Dubovskoy, « Canonical Cooklang parser in Rust ». [En ligne]. Disponible sur: https://github.com/cooklang/cooklang-rs

[3] C. de Ron, « Rusty Object Notation ». [En ligne]. Disponible sur: https://github.com/ron-rs/ron

[4] Torm, « udl v0.3.1 - Parser for UDL (Universal Data Language) ». [En ligne]. Disponible sur: https://crates.io/crates/udl

[5] Torm, « The Khi data language ». [En ligne]. Disponible sur: https://github.com/khilang/khi

[6] Torm, « Rust Khi parser & library ». [En ligne]. Disponible sur: https://github.com/khilang/khi.rs

[7] bitmark Association, « bitmark Association website ». [En ligne]. Disponible sur: https://www.bitmark-association.org/

[8] bitmark Association, « bitmark Hackathon ». [En ligne]. Disponible sur: https://www.bitmark-association.org/bitmarkhackathon

[9] bitmark Association, « bitmark Documentation ». [En ligne]. Disponible sur: https://docs.bitmark.cloud/

[10] bitmark Association, « Quizzes - .multiple-choice, .multiple-choice-1 ». [En ligne]. Disponible sur: https://docs.bitmark.cloud/quizzes/#multiple-choice-multiple-choice-1

[11] Taskbase, « open-taskpool - 12,000 UK 🇺🇦 → DE 🇩🇪 & DE 🇩🇪 → EN🇬🇧 learning tasks ready for you to use. ». [En ligne]. Disponible sur: https://github.com/taskbase/open-taskpool

[12] bitmark Association, « Quizzes - .cloze (gap text) ». [En ligne]. Disponible sur: https://docs.bitmark.cloud/quizzes/#cloze-gap-text

[13] Classtime, « Créer la première question / le premier jeu de questions ». [En ligne]. Disponible sur: https://help.classtime.com/fr/comment-commencer-a-utiliser-classtime/creer-la-premiere-question-le-premier-jeu-de-questions

[14] K. Kundert, « NestedText — A Human Friendly Data Format ». [En ligne]. Disponible sur: https://github.com/KenKundert/nestedtext

[15] Ken et K. Kundert, « NestedText documentation - Schemas ». [En ligne]. Disponible sur: https://nestedtext.org/en/latest/schemas.html

[16] bob22z, « docs.rs - Crate nestedtext ». [En ligne]. Disponible sur: https://nestedtext/latest/nestedtext/

[17] S. Ludwig, « SDLang, Simple Declarative Language ». [En ligne]. Disponible sur: https://sdlang.org/

[18] K. M. (zkat) et contributeurs, « KDL, a cudlly document language ». [En ligne]. Disponible sur: https://kdl.dev/

[19] « All Crates for keyword 'parser' ». [En ligne]. Disponible sur: https://crates.io/keywords/parser

[20] E. P. (epage), « winnow v0.7.8 A byte-oriented, zero-copy, parser combinators library ». [En ligne]. Disponible sur: https://crates.io/crates/winnow

[21] « Dependencies of kdl crate ». [En ligne]. Disponible sur: https://crates.io/crates/kdl/6.3.4/dependencies

[22] G. C. (Geal), « nom v8.0.0 A byte-oriented, zero-copy, parser combinators library ». [En ligne]. Disponible sur: https://crates.io/crates/nom

[23] « Reverse dependencies of nom crate ». [En ligne]. Disponible sur: https://crates.io/crates/nom/reverse_dependencies

[24] « pest v2.8.0 The Elegant Parser ». [En ligne]. Disponible sur: https://crates.io/crates/pest

[25] M. W. (Marwes), « combine v4.6.7 Fast parser combinators on arbitrary streams with zero-copy support. ». [En ligne]. Disponible sur: https://crates.io/crates/combine

[26] R. T. (. e. c. Joshua Barretto (zesterer), « chumsky v0.10.1 A parser library for humans with powerful error recovery ». [En ligne]. Disponible sur: https://crates.io/crates/chumsky

[27] « Most popular Rust libraries ». [En ligne]. Disponible sur: https://lib.rs/std

[28] « Serde data model ». [En ligne]. Disponible sur: https://serde.rs/data-model.html

[29] [En ligne]. Disponible sur: https://macromates.com/manual/en/regular_expressions

[30] [En ligne]. Disponible sur: https://code.visualstudio.com/api/language-extensions/syntax-highlight-guide

[31] [En ligne]. Disponible sur: https://www.jetbrains.com/help/idea/textmate.html

[32] C. de Tree-sitter, « Introduction - Tree-sitter ». [En ligne]. Disponible sur: https://tree-sitter.github.io/tree-sitter/

[33] C. de Tree-sitter, « Creating Parsers - Getting Started - Tree-sitter ». [En ligne]. Disponible sur: https://tree-sitter.github.io/tree-sitter/creating-parsers/1-getting-started.html

[34] C. de Tree-sitter, « The Grammar DSL - Tree-sitter ». [En ligne]. Disponible sur: https://tree-sitter.github.io/tree-sitter/creating-parsers/2-the-grammar-dsl.html

[35] « Neovim Documentation - Treesitter ». [En ligne]. Disponible sur: https://neovim.io/doc/user/treesitter.html

[36] « Language Extensions - Grammar ». [En ligne]. Disponible sur: https://zed.dev/docs/extensions/languages?#grammar

[37] « Creating a Grammar ». [En ligne]. Disponible sur: https://flight-manual.atom-editor.cc/hacking-atom/sections/creating-a-grammar/

[38] M. et contributeurs, « Language Server Protocol ». [En ligne]. Disponible sur: https://microsoft.github.io/language-server-protocol/

[39] J.-R. W. Group, « JSON-RPC 2.0 Specification ». [En ligne]. Disponible sur: https://www.jsonrpc.org/specification

[40] M. et contributeurs, « Language Server Protocol Specification - 3.17 - Capabilities ». [En ligne]. Disponible sur: https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/#capabilities

[41] M. et contributeurs, « Language Server Protocol Specification - 3.17 - Content part ». [En ligne]. Disponible sur: https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/#contentPart

[42] bergercookie et contributeurs, « asm-lsp v0.10.0 Language Server for x86/x86_64, ARM, RISCV, and z80 Assembly Code ». [En ligne]. Disponible sur: https://crates.io/crates/asm-lsp

[43] eclipse-jdtls organisation et contributeurs, « GitHub - eclipse-jdtls/eclipse.jdt.ls: Java language server ». [En ligne]. Disponible sur: https://github.com/eclipse-jdtls/eclipse.jdt.ls

[44] T. et contributeurs, « GitHub - tailwindlabs/tailwindcss-intellisense: Intelligent Tailwind CSS tooling for Visual Studio Code ». [En ligne]. Disponible sur: https://github.com/tailwindlabs/tailwindcss-intellisense

[45] typescript-language-server organisation et contributeurs, « GitHub - typescript-language-server/typescript-language-server: TypeScript & JavaScript Language Server ». [En ligne]. Disponible sur: https://github.com/typescript-language-server/typescript-language-server

[46] C. de lsp-types, « lsp-types v0.97.0 Types for interaction with a language server, using VSCode's Language Server Protocol ». [En ligne]. Disponible sur: https://crates.io/crates/lsp-types

[47] O. gluon-lang et contributeurs, « Reverse dependencies of lsp-types crate ». [En ligne]. Disponible sur: https://crates.io/crates/lsp-types/reverse_dependencies

[48] rust-lang organisation et contributeurs, « rust-analyzer/lib/lsp-server/examples/goto_def.rs at master · rust-lang/rust-analyzer · GitHub ». [En ligne]. Disponible sur: https://github.com/rust-lang/rust-analyzer/blob/master/lib/lsp-server/examples/goto_def.rs

[49] M. et contributeurs, « Implementations - Tools supporting the LSP ». [En ligne]. Disponible sur: https://microsoft.github.io/language-server-protocol/implementors/tools/

[50] M. et contributeurs, « Implementations - Language Servers ». [En ligne]. Disponible sur: https://microsoft.github.io/language-server-protocol/implementors/servers/

[51] oxalica et contributeurs, « async-lsp v0.2.2 Asynchronous Language Server Protocol (LSP) framework based on tower ». [En ligne]. Disponible sur: https://crates.io/crates/async-lsp

[52] oxalica et contributeurs, « nil/crates/nil/Cargo.toml - NIx Language server, an incremental analysis assistant for writing in Nix. ». [En ligne]. Disponible sur: https://github.com/oxalica/nil/blob/577d160da311cc7f5042038456a0713e9863d09e/crates/nil/Cargo.toml#L11

[53] M.-D. et contributeurs, « sync-ls - Synchronized language service inspired by async-lsp, primarily for tinymist. ». [En ligne]. Disponible sur: https://crates.io/crates/sync-ls

[54] tower-lsp-community organisation et contributeurs, « tower-lsp-server v0.21.1 Language Server Protocol implementation based on Tower ». [En ligne]. Disponible sur: https://crates.io/crates/tower-lsp-server

[55] rust-lang organisation et contributeurs, « Reverse dependencies of lsp-server crate ». [En ligne]. Disponible sur: https://crates.io/crates/lsp-server/reverse_dependencies

[56] E. K. et contributeurs, « Reverse dependencies of tower-lsp crate ». [En ligne]. Disponible sur: https://crates.io/crates/tower-lsp/reverse_dependencies