

Софтуер за генериране на изпитни билети

Даниел Василев

14 април 2023 г.

Съдържание

1	Увод	2
2	Използвани езици и технологии	3
2.1	Rust	3
2.1.1	Отличаващи се особености на езика	3
2.1.2	Enum	4
2.1.3	Option типа	4
2.1.4	Result типа	4
2.2	egui	6
2.2.1	Минимален пример	6
2.3	genpdf	7
2.3.1	Елементи	7
3	Подготовка	9
3.1	Инсталиране на Rust	9
3.2	Избор на IDE	10
3.2.1	Visual Studio Code	10
3.2.2	Vim	10
3.2.3	Neovim	11
3.3	Git	12
4	Разработка	13
4.1	Създаване на проекта	13
4.2	Добавяне на библиотеки	14
4.2.1	Конзолни аргументи	14
4.2.2	Serde	14
4.2.3	Toml	14
4.2.4	genPDF	15
5	Заклучение	16

1 Увод

В настоящата дисертация ще бъде представен проект, който има за цел да улесни генерирането на изпитни билети за учебни заведения. Програмата е изградена на езика Rust и предлага лесен и ефективен начин за създаване на изпитни билети.

За да може един софтурен продукт да бъде завършен на време, трябва да задачите и целите на продукта да бъдат разделени на по-малки под задачи.

2 Използвани езици и технологии

2.1 Rust

Rust е програмен език от високо ниво създаден през 2006 година от Graydon Hoare, който по това време работи за Mozilla. През 2009 година разработката на езика бива спонсорирана от Mozilla, а през 2010 езика е обявен публично. [1]

2.1.1 Отличаващи се особености на езика

Езици като C#, Python и JavaScript използват система за освобождаване на паметта наречена Garbage Collector (GC). За да може да се освободят неизползваните променливи, изпълнението на програмата трябва да бъде спряно на пауза и да се провери дали има заделени региони от паметта, към които вече не се използват или са маркиране за освобождаване от програмиста [2].

Rust използва система наречена borrow checker, която проверява, по време на компилация, дали програмата следва следните принципи:

- Ресурсите (отделената памет за стойността) могат да имат само един собственик и това е самата променлива. Когато променлива вече не може да бъде достъпена ресурсите биват освободени.
- Когато една променлива бъде подадена към някоя функция, собственик на ресурсите става функцията. Ако се пробваме подадем отново променливата, компилатора ще ни каже, че променливата е била преместена (Use of moved value).

```
9 fn print_string(data: String) {
8     println!("{}", data);
7 }
6
5 fn main() {
4     let owner_of_data = String::from("Hello, World!"); => String
3
2     print_string(owner_of_data);
1
10    print_string(owner_of_data);
1 }
Diagnostics:
1. use of moved value: `owner_of_data`
   value used here after move
```

Фигура 1: Модела на собственик в Rust

2.1.2 Enum

Enum е един от основните типове в Rust. Всеки вариант на enum-а може да има съдържа информация от различен вид [3]. Така са имплементирани някои от най-важните типове: `Option<T>` и `Result<T, E>`.

```
1  enum Option<T> {  
2      Some(T),  
3      None,  
4  }  
5  enum Result<T, E> {  
6      Ok(T),  
7      Err(E),  
8  }
```

Фигура 2: Стандартната имплементация на `Option<T>` и `Result<T, E>`

2.1.3 Option типа

В повечето езици съществува идеята за NULL пойнтери. Когато един pointer е Null това означава, че той сочи към нищо. Идеята за Null на теория е много добра, но на практика създава повече проблеми. Ако се пробваме да достъпим pointer който е Null, програмата ще крашне или в някои езици като C# ще хвърли `NullReferenceException`.

Разработчиците на Rust са намерили много добър заместител на Null и това е `Option` enum-а, който има два варианта. Това са `Some(T)` когато имаме някаква стойност и `None` когато нямаме нищо.

2.1.4 Result типа

Когато програмираме на C# много често ни се случва да хвърляме `Exception`-и и съответно да ги хващаме с `try/catch` блока. `Exception`-ите се ползват когато в една функция възникне грешка.

Във Фигура 3 е даден код който на пръв поглед изглежда добре, но има скрити бъгове. Какво ще стане ако потребителя въведе дума вместо число? Ще получим `Exception` който ни казва: "Input string was not in a correct format".

```

1  int ReadNumberFromUser()
2  {
3      string user_input = Console.ReadLine();
4      int parsed_int = int.Parse(user_input);
5      return parsed_int;
6  }
7  int number = ReadNumberFromUser();
8  Console.WriteLine($"{number} * 2 = {number * 2}");

```

Фигура 3: Пример за скрит Exception

```

~/Programming/C#/Scratchpad
> dotnet run
word
Unhandled exception. System.FormatException: Input string was not in a correct format.
   at System.Number.ThrowOverflowOrFormatException(ParsingStatus status, TypeCode type)
   at System.Number.ParseInt32(ReadOnlySpan`1 value, NumberStyles styles, NumberFormatInfo info)
   at System.Int32.Parse(String s)
   at Program.<<Main>>g__ReadNumberFromUser|0_0() in /home/daniel/Programming/C#/Scratchpad/Program.cs:line 4
   at Program.<Main>$(String[] args) in /home/daniel/Programming/C#/Scratchpad/Program.cs:line 8

```

Фигура 4: Изход на кода от Фигура 3

Проблема е че ние като програмисти не знаем, че `int.Parse` може да хвърли `Exception` без да се консултираме с документацията [4]. Същият код написан на Rust би изглеждал по следния начин [Фигура 5].

Разликата между C# и Rust е че Rust кода ни кара ни показва типовете при успех и грешка. Функцията връща променлива от тип `Result<T, E>` където `T` е променливата от тип `i32` (`int`) ако всичко се и изпълнило без проблем, а `E` е от тип `ParseIntError`.

За да използваме резултата от функцията, какъвто и да е той, можем да използваме `match`. С `match` можем да проверим дали резултата е `Ok` или `Err`.

```

1 use std::num::ParseIntError;
2
3 fn read_number_from_user() -> Result<i32, ParseIntError> {
4     let mut user_input = String::new(); => String
5     std::io::stdin().read_line(&mut user_input).unwrap(); <- (buf)
6     return user_input.trim().parse();
7 }
8
9 fn main() {
10     let number = read_number_from_user(); => Result<i32, ParseIntError>
11
12     match number {
13         Ok(number) => { => i32
14             // NOTE принтира резултата
15             println!("{}", number, number * 2);
16         }
17         Err(err) => { => ParseIntError
18             // NOTE принтира грешката в конзолата, но не crash-ва!
19             println!("{}", err);
20         }
21     }
22 }

```

Фигура 5: Кода от Фигура 3 написан на Rust

2.2 egui

egui е проста, бърза и много преносима библиотека за графични потребителски интерфейси (GUI). Egui работи на много платформи включително: уеб браузъри, като обикновено приложение и в някои game engine-a. Написана е на Rust и има много лесен и интуитивен API за разработване.

Главните цели на проекта са:

- Най-лесната за използване GUI библиотека
- Отзивчив: цели поне 60 FPS при компилация с Debug опциите
- Преносим: кода да работи в браузър и като собствено приложение
- Лесен за интегриране във всяка среда
- Разширяем: лесно да пишете свои собствени джаджи за egui
- Модулен: можете да използвате малки части от egui и да ги комбинирате по нови начини
- Минимален брой зависимости (библиотеки)

2.2.1 Минимален пример

egui библиотека ни дава достъп до App интерфейса. Този интерфейс съдържа една функция update. Тя се извиква всеки път когато потребителския интерфейс се е променил или се получи някакъв евент от мишката или клавиатурата. [Фигура 6]

```

1
2 impl eframe::App for MyApp {
3     fn update(&mut self, ctx: &egui::Context, _frame: &mut eframe::Frame) {
4         egui::CentralPanel::default().show(ctx, |ui| {
5             ui.heading("My egui Application");
6             ui.horizontal(|ui| {
7                 let name_label = ui.label("Your name: ");
8                 ui.text_edit_singleline(&mut self.name)
9                     .labelled_by(name_label.id);
10            });
11            ui.add(egui::Slider::new(&mut self.age, 0..=120).text("age"));
12            if ui.button("Click each year").clicked() {
13                self.age += 1;
14            }
15            ui.label(format!("Hello '{}', age {}", self.name, self.age));
16        });
17    }
18 }
19

```

Фигура 6: Имплементация на egui интерфейса

Библиотеката е достатъчно умна сама да прецени дали се нуждае от повторно изобразяване.

egui е GUI библиотека от незабавен режим (Immediate Mode). Това означава че начина по който искаме да изглежда графичния интерфейс се описва извиквайки методи. По този начин се упростира разработката. За да покажем един прост бутон се нуждаем от един if оператор.

2.3 genpdf

genpdf е библиотека която абстрахира създаването на PDF файлове. Тя се грижи за оформлението на страницата, подравняването на текста и изобразяването на структурата на документа в PDF файл. Библиотека както и всичките и зависимости са написани на Rust и следват добрите практики на езика [5].

2.3.1 Елементи

genpdf използва елементи за да опише оформлението на документа. Всеки елемент имплементира Element интерфейса. Интерфейса съдържа функция render която бива извикана всеки път когато елемента трябва да бъде показан в PDF файла [6].

Използвайки Element интерфейса, разработчиците на genpdf са ни предоставили най-често използваните елементи:

- Контейнери:

LinearLayout: подрежда елементите си последователно

TableLayout: подрежда елементите си в колони и редове

OrderedList/UnorderedList: подредете елементите им последователно с bullet-и

- Текст:

Text: един ред текст

Paragraph: подравнен параграф

- Обвивки:

FramedElement: елемент с рамка

PaddedElement: добавя разстояние между елементите

StyledElement: задава стил по подразбиране за обвития елемент и неговите деца

- Други:

Image: снимка с описание

Break: добавя прекъсвания на редове като разделител

PageBreak: добавя принудително прекъсване на страницата

3 Подготовка

Преди да започнем с разработката на софтуера трябва да настроим нашата среда за разработка. Тя включва Rust компилатора, текстов редактор (IDE) и Git за контрол на версиите.

3.1 Инсталиране на Rust

Rust е език който използва LLVM за обръщането на код в машинни инструкции. LLVM е набор от технологии за компилиране, който позволява да бъдат написани различни frontend-ове за всеки език и backend-ове за всяка хардуерна архитектура. Благодарение на този факт Rust може да работи на всички модерни операционни системи като Windows, MacOS, Linux, OpenBSD и още много други.

За операционните системи базирани на UNIX принципите, като MacOS и Linux можем да инсталираме Rust с една проста команда:

```
curl --proto 'https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

Или през package manager-а на операционната система. В MacOS можем да използваме Homebrew. За да инсталираме Homebrew трябва да изпълним тези команди:

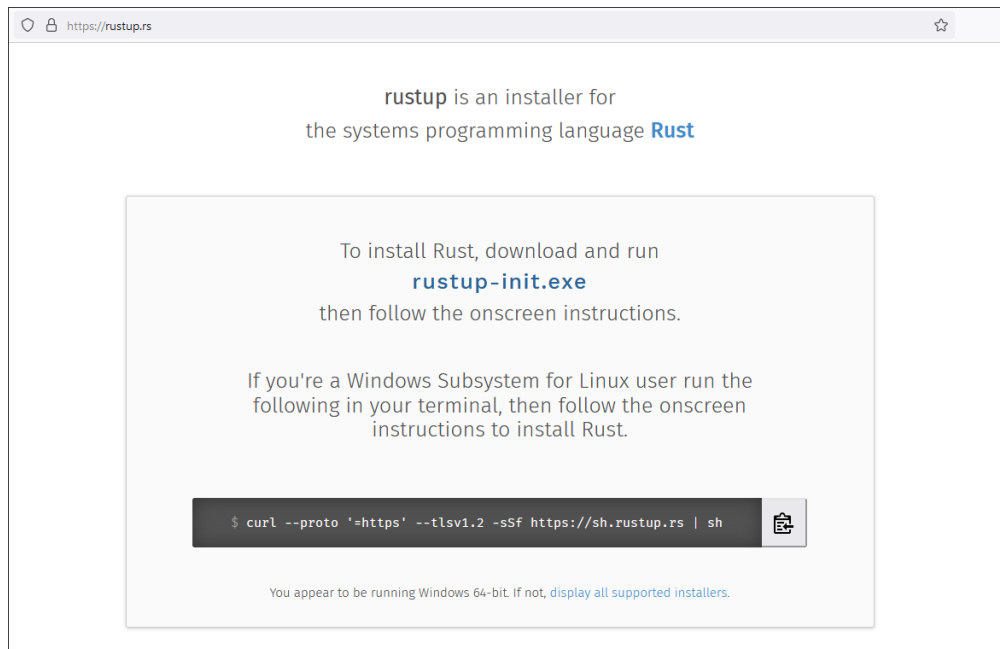
```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

```
brew install rust
```

А в Linux зависи от дистрибуцията:

- Arch Linux - pacman -S rustup
- Debian Linux - apt-get install rustup
- Fedora Linux - dnf install rust cargo

За да инсталираме Rust на Windows трябва да изтеглим 64 или 32 битовия инсталационен файл от сайта на Rust: <https://rustup.rs/>.



Фигура 7: Сайт за изтегляне на инсталационния файл на Rust за Windows

3.2 Избор на IDE

За да можем да създаваме софтуер на Rust, по най-ефективния начин се нуждаем от текстов редактор който поддържа LSP. LSP или Language Server Protocol е протокол създаден от Microsoft за Visual Studio Code и служи за комуникация между текстовия редактор и специализирани програми, които анализират кода който пишем и показват къде има грешки, предложения как да бъдат поправени, допълване на код, подчертаване на синтаксиса [7].

3.2.1 Visual Studio Code

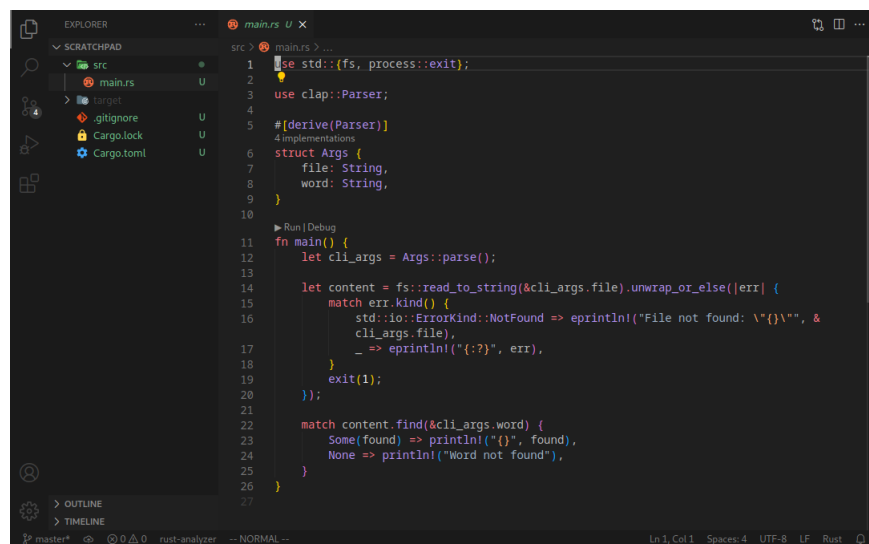
Visual Studio Code е текстов редактор с отворен код създаден от Microsoft. Той използва Electron за графична библиотека и работи на Windows, Linux и MacOS. През 2022 в допитване до потребителите на Stack Overflow, Visual Studio Code е класиран като най-популярният текстов редактор сред 71 010 респонденти [8].

Visual Studio Code със заводските си настройки не може да прави почти нищо. За да получим всички полезни функционалности на LSP, трябва да инсталираме така наречените плъгини.

Един от недостатъците на VSCode обаче е високите системни изисквания за нормална работа и моментално време за реакция при въвеждане на текст.

3.2.2 Vim

Vim е текстов редактор с отворен код първоначално написан за настолният компютър Amiga през 1991 година от Брам Моленар. За разлика от Visual Studio Code, vim е



Фигура 8: Visual Studio Code

текстов редактор, който е бил замислен да работи не само в графични среди, но и в терминални среди [9].

Най-привлекателната част от Vim е начина за навигация. В повечето редактори се навигира чрез мишката и няколко клавишни комбинации, докато Vim използва само клавишни комбинации. По този начин ръцете ни остава на клавиатурата и няма нужда да отделяме време за навигиране с мишка.

Vim разполага с 3 режима за работа и това са:

- Normal - В нормалният режим можем само да манипулираме текст
- Visual - В визуалният режим можем да избираме по-големи региони от текст
- Command - В командният режим можем да изпълняваме команди за манипулиране на текст, настройване на редактора и изпълнение на команди в операционната система

Vim успява да много малко ресурси от VSCode, без да прави компромиси от към функционалности. За да може да бъде постигнато Vim е написан на C и потребителя трябва ръчно да си настрои редактора, използвайки специално направеният език VimScript. Този процес е прекалено сложен за повечето потребители затова те предпочитат да използват VSCode, дори и да използва повече ресурси.

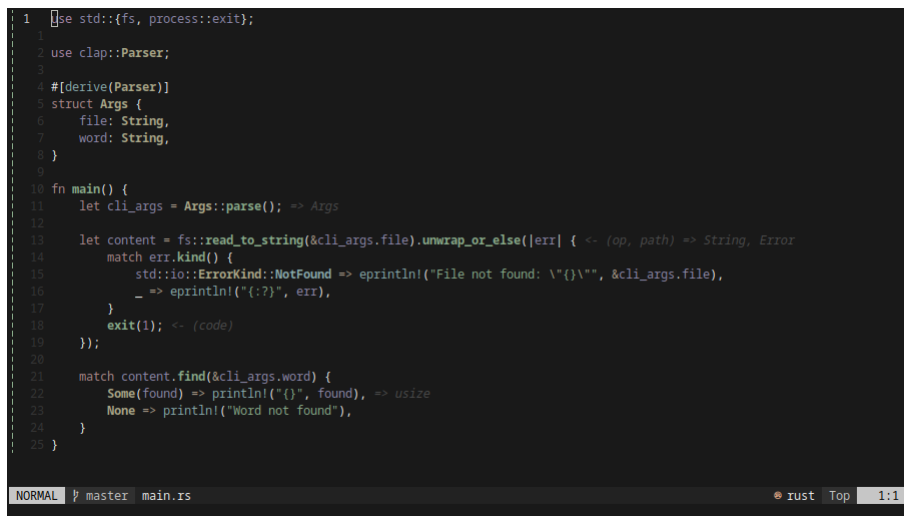
3.2.3 Neovim

Neovim е копие на Vim, което се стреми да подобри скоростта и поддръжката на Vim. Някои добавени функционалности на копието включват вградена поддръжка на LSP и поддръжка за Lua скриптове като заместител на VimScript [10].

Проектът Neovim стартира през 2014 година, като някои членове на Vim общността предлагат помощ в усилията за основно рефакториране, за да осигурят по-добри езици

за скриптове, плъгини и интеграция с модерни графични потребителски интерфейси. Проектът е с отворен код и, който е достъпен в GitHub. [11]

От към производителност Neovim е малко по-бавен от предшественика си Vim, но все пак е в пъти по-бърз от главния си конкурен Visual Studio Code, затова аз се спрях на Neovim.



```
1 use std::{fs, process::exit};
2
3 use clap::Parser;
4
5 #[derive(Parser)]
6 struct Args {
7     file: String,
8     word: String,
9 }
10
11 fn main() {
12     let cli_args = Args::parse(); => Args
13
14     let content = fs::read_to_string(&cli_args.file).unwrap_or_else(|err| { <- (op, path) => String, Error
15         match err.kind() {
16             std::io::ErrorKind::NotFound => eprintln!("File not found: \"{}\"", &cli_args.file),
17             _ => eprintln!("{:?}", err),
18         }
19         exit(1); <- (code)
20     });
21
22     match content.find(&cli_args.word) {
23         Some(found) => println!("{}", found), => usize
24         None => println!("Word not found"),
25     }
26 }
```

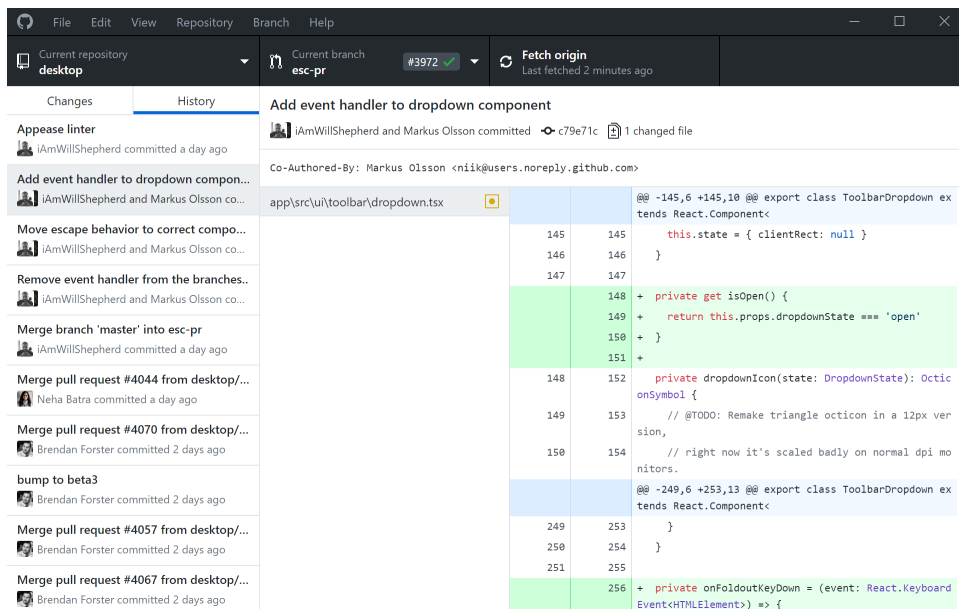
Фигура 9: Neovim

3.3 Git

Git е система за контрол на версиите, която проследява промените във всеки набор от компютърни файлове, обикновено се използва за координиране на работата между програмистите, които съвместно разработват изходния код по време на разработката на софтуер. Целите на системата включват скорост, цялост на данните и поддръжка за разпределени, нелинейни работни потоци (хиляди паралелни клонове, работещи на различни системи).

Git първоначално е създаден от Линус Торвалдс, през 2005 година за разработване на ядрото на Linux, като инструмент за други разработчици на ядрото да допринасят за първоначалното му развитие. От 2005 година Junio Hamano е основният разработчик. Както при повечето други разпределени системи за контрол на версиите и за разлика от повечето системи клиент-сървър, всяка Git директория на всеки компютър е пълноценно хранилище с пълна история и пълни възможности за проследяване на версиите, независимо от достъпа до мрежата или централен сървър. Git е безплатен софтуер с отворен код, разпространяван само под лиценз GPL-2.0. [12]

Git е конзолно приложение, но съществуват и приложение с графичен интерфейс като GitHub Desktop. Това приложение ни дава възможност да управляваме разработката на проекта, създаване на потребителски истории и тяхното менижиране. [Фигура 10] [13]



Фигура 10: Github Desktop в Windows

4 Разработка

4.1 Създаване на проекта

За да създадем нов Rust проект, трябва да отворим конзолата и да изпълним следната команда:

```
cargo new test-generator
```

Тя ще генерира папка със името `test-generator` в която се намира проекта. В `src` се намират файловете в които пишем кода, а в `Cargo.toml` файла се намира конфигурацията на проекта.

За да отворим проекта в текстов редактор може да напишем командата `code .` за да го отворим във VSCode или `nvim .` за да го отворим във Neovim.

```
→ ~ cargo new test-generator
    Created binary (application) `test-generator` package
→ ~ tree test-generator
test-generator
├── Cargo.toml
└── src
    └── main.rs
→ ~
```

Фигура 11: Файловата структура на Rust проекта

4.2 Добавяне на библиотеки

4.2.1 Конзолни аргументи

Аргументите на конзолата са параметри, предавани на програма преди изпълнение на поргамата в командния ред. В Rust аргументите могат да бъдат достъпени чрез функцията `std::env::args()`, която връща итератор над аргументите като списък от низове.

За да вземем подходящата информация за приложението може да напишем наш собствен анализатор или да използваме една от многото различни библиотеки за работа с конзолни аргументи. Една от най-използваните библиотеки е `clap` (Console Line Argument Parser).

`Clap` ни предоставя с `clap::Parser` макрото, което при компилирането на програмата анализира структурата от данни и автоматично търси командните аргументи при екзекуция. Също така проверява кои аргументи са маркирани като задължителни или такива със стойност по подразбиране.

При въвеждане на грешни аргументи или при липсата на задължителните такива, `Clap` показва автоматично генерираното помощно съобщение на потребителя.

4.2.2 Serde

`Serde` е framework за ефективно сериализиране и десериализиране на структури от данни в Rust.

Екосистемата на `Serde` се състои от структури от данни, които знаят как да сериализират и десериализират себе си заедно с формати на данни, които знаят как да сериализират и десериализират други неща. `Serde` предоставя слой, чрез който тези две групи взаимодействат помежду си, позволявайки всяка поддържана структура от данни да бъде сериализирана и десериализирана с помощта на всеки поддържан формат на данни.

Докато много други езици разчитат на runtime среда (като `Dotnet`) за сериализиране на данни, `Serde` вместо това е изградена върху много добрата интерфейс система на Rust.

Структура от данни, която знае как да сериализира и десериализира сама себе си, е тази, която използва интерфейсите на `Serde` за сериализиране и десериализиране (или използва атрибута `derive` на `Serde` за автоматично генериране на интерфейси по време на компилация). По този начин се избягват забавянето от употребата на runtime среда.

Всъщност в много ситуации взаимодействието между структурата на данните и формата на данните може да бъде напълно оптимизирано от Rust компилатора, оставяйки сериализацията на `Serde` да се изпълни със същата скорост като ръно написан сериализатор в езици от по-ниско ниво като C.

4.2.3 Toml

`Toml` (Tom's Obvious Minimal Language) е файлов формат за съхранение на софтуерни конфигурации. Този формат ще бъде използван за съхраняване на настройките, въпросите и друга информация за тестовете. За да добавим поддръжка за този формат трябва в Rust, трябва да инсталираме `Toml` библиотека която използва `Serde` са преобразуването на файл в обект и обратно.

4.2.4 genPDF

genPDF е библиотека за генериране на PDF документи на високо ниво, изградена от две по-малки библиотеки от по-ниско ниво. Тя се грижи за оформлението на страницата и подравняването на текста и изобразява дървовидна структура на документа в PDF документ.

5 Заключение

Литература

- [1] Wikipedia contributors. Rust (programming language) — Wikipedia, The Free Encyclopedia. 2023. URL: [https://en.wikipedia.org/w/index.php?title=Rust_\(programming_language\)&oldid=1146879721#Origins_\(2006%E2%80%932012\)](https://en.wikipedia.org/w/index.php?title=Rust_(programming_language)&oldid=1146879721#Origins_(2006%E2%80%932012)).
- [2] Wikipedia contributors. Garbage collection (computer science) — Wikipedia, The Free Encyclopedia. 2023. URL: [https://en.wikipedia.org/w/index.php?title=Garbage_collection_\(computer_science\)&oldid=1146816153](https://en.wikipedia.org/w/index.php?title=Garbage_collection_(computer_science)&oldid=1146816153).
- [3] Rust contributors. Defining an Enum. 2021. URL: <https://doc.rust-lang.org/book/ch06-01-defining-an-enum.html>.
- [4] Microsoft. Int32.Parse Method. 2023. URL: <https://learn.microsoft.com/en-us/dotnet/api/system.int32.parse?view=net-8.0>.
- [5] ireas. genpdf-rs - User-friendly PDF generator written in pure Rust. 2021. URL: <https://git.sr.ht/~ireas/genpdf-rs>.
- [6] ireas. docs.rs - Trait genpdf::Element. 2021. URL: <https://docs.rs/genpdf/0.2.0/genpdf/trait.Element.html>.
- [7] Wikipedia contributors. Language Server Protocol — Wikipedia, The Free Encyclopedia. 2023. URL: https://en.wikipedia.org/w/index.php?title=Language_Server_Protocol&oldid=1146435481.
- [8] Wikipedia contributors. Visual Studio Code — Wikipedia, The Free Encyclopedia. 2023. URL: https://en.wikipedia.org/w/index.php?title=Visual_Studio_Code&oldid=1147584690.
- [9] Wikipedia contributors. Vim (text editor) — Wikipedia, The Free Encyclopedia. 2023. URL: [https://en.wikipedia.org/w/index.php?title=Vim_\(text_editor\)&oldid=1147168005](https://en.wikipedia.org/w/index.php?title=Vim_(text_editor)&oldid=1147168005).
- [10] Wikipedia contributors. Neovim — Wikipedia, The Free Encyclopedia. 2023. URL: [https://en.wikipedia.org/w/index.php?title=Vim_\(text_editor\)&oldid=1147168005#Neovim](https://en.wikipedia.org/w/index.php?title=Vim_(text_editor)&oldid=1147168005#Neovim).
- [11] Neovim contributors. neovim - GitHub User. 2023. URL: <https://github.com/neovim/neovim/>.
- [12] Wikipedia contributors. Git — Wikipedia, The Free Encyclopedia. 2023. URL: <https://en.wikipedia.org/w/index.php?title=Git&oldid=1145875787>.
- [13] Microsoft. GitHub Desktop | Simple collaboration from your desktop. 2023. URL: <https://desktop.github.com>.