Rust roest

Inleiding

"Rust roest" en dan heb ik het niet over de televisieserie, maar dan heb ik het over het spreekwoord. Het spreekwoord is ontleent aan ijzer, dat wanneer het gebruikt word blank blijft, maar wanneer het stil ligt gaat roesten. Dit gaat natuurlijk ook op in de wereld van programmeurs. Er blijft een continue stroom van nieuwe informatie komen, nieuwe technieken, andere talen, meer gespecialiseerde frameworks, handige tools en ga zo maar door. Want zwem je niet met de stroom mee, dan ga je roesten. Gedoemd tot een leven lang leren? Of steeds een nieuwe uitdaging om enthousiast van te worden. In mijn blog wil ik jullie meenemen in een nieuwe uitdaging die ik mezelf gegeven heb.

De uitdaging

In mijn dagelijks werkzaamheden gebruik ik hoofdzakelijk Java, en daarnaast af en toe wat C#, maar veel variatie daarnaast is er niet. Dit zijn beide talen vanuit het objectgeoriënteerd programmeren paradigma. Mijn nieuwe uitdaging is dus ook, om uit dit paradigma te stappen en een compleet andere programmeer taal te kiezen uit een ander paradigma. Maar voordat we dat doen zoomen we heel even in wat nu een programmeerparadigma is, en welke gangbare paradigma's er zijn.

Programmeerparadigma's

> *In de informatica zijn programmeerparadigma's denkpatronen of uitgesproken concepten van programmeren, die voornamelijk verschillen in de wijze van aanpak om het gewenste resultaat te kunnen behalen.*

Gangbare paradigma's

De belangrijkste programmeerparadigma's zijn:

- Imperatief programmeren
- Functioneel programmeren
- Logisch programmeren
- Objectgeoriënteerd programmeren

In **imperatief programmeren** is een programma in essentie een reeks instructies die het geheugen manipuleren en die op volgorde door de computer worden uitgevoerd. Deze stijl van programmeren staat dicht bij de werking van een computer en werden derhalve als eerste praktisch gerealiseerd.

Functioneel programmeren is gebaseerd op formalismen zoals de theorie van recursieve functies of de lambda calculus-programma's. Hieronder worden wiskundige functies gedefinieerd die invoer naar uitvoer transformeren.

Bij **logisch programmeren**, gebaseerd op (doorgaans) predicatenlogica, zijn het definities van predicaten die een bepaalde relatie tussen objecten in het geheugen uitdrukken.

Objectgeoriënteerd programmeren (object oriented programming) verenigt berekening en de gegevens: deze worden verpakt in objecten, waarbij de details worden verborgen achter een algemene interface, vaak gerangschikt in een hiërarchie van klassen. Objecten sturen elkaar berichten (Smalltalk) of roepen elkaars methoden aan (C++, Java); alleen zo hebben ze toegang tot elkaars gegevens. De methoden/reacties op berichten zijn procedures die de interne gegevens van een object manipuleren.

De keuze

Om makkelijker te kunnen kiezen heb ik een shortlist gemaakt van potentiële talen, en een overzicht gemaakt van paradigma's waarin deze talen vallen.

Language and paradigma's

Language	Intended use	Imperative	Object-oriented	Functional	Procedural	Generic	Reflective	Event-driven	Other paradigm(s)	Standardized?
Clojure	General			Yes					concurrent	No
Erlang	Application, distributed			Yes				Yes	concurrent, distributed	No
F#	Application	Yes	Yes	Yes	Yes	Yes	Yes	Yes		No
Haskell	Application			Yes		Yes			lazy evaluation	2010, Haskell 2010[27]
Occam	General	Yes			Yes				concurrent, process-oriented	No
R	Application, statistics	Yes	Yes	Yes	Yes		Yes			No
Rust	Application, system	Yes	Yes	Yes	Yes	Yes		Yes	concurrent	No
Scala	Application, distributed, web	Yes	Yes	Yes		Yes	Yes	Yes		De facto standard via Scala Language Specification (SLS)

Bron: Comparison of programming languages. (2019, October 10)

Wat direct opvalt, is dat er een aantal talen zijn die heel puur in één paradigma vallen zoals Clojure en Erlang. Maar dat er veel talen zijn die onder meer paradigma's vallen. Dan is het dus aan de programmeur om het op de juiste manier te gebruiken.

Maar een keuze maken is, en blijf ik nog steeds lastig met de huidige gegevens. Uiteindelijk heb ik er voor gekozen om me iets meer te verdiepen in de populariteit van de talen van dit moment, deze heb ik in een shortlist gezet.

Ik ben op onderzoek uitgegaan en vond een aantal indices die de populariteit van programmeertalen weergegeven. Elke index berekent de populariteit zijn eigen manier, maar als je alles samenvoegt geeft het wel een goede indicatie hoe populair een taal is. Je kunt je natuurlijk de vraag stellen of het belangrijk is, of een taal populair is. Maar de populariteit impliceert wel dat de community die de taal ondersteund groot is. En dat de kans dat je de taal ooit op het werk gaat gebruiken ook groter is.

Tiobe Index Oktober 2019

Programming Language	Rank	Ratings
Clojure	51 – 100	-
Erlang	51 – 100	-
F#	32	0.391%
Haskell	43	0.209%
Occam	-	-
R	15	1.261%
Rust	34	0.356%
Scala	29	0.442%

Bron: Tiobe Index. (2019, October)

PyPL Index Oktober 2019

Programming Language	Rank	share
Clojure	-	-
Erlang	-	-
F#	-	-
Haskell	21	0.29 %
Occam	-	-
R	7	3.82 %
Rust	18	0.64 %
Scala	16	1.15 %

Bron: PyPL

Stackoverflow Survey 2019 - Mosted used

Programming Language	Rank	share
Clojure	24	1.50%
Erlang	-	-
F#	-	-

Programming Language	Rank	share
Haskell	-	-
Occam	-	-
R	16	5.60%
Rust	21	3.00%
Scala	20	4.20%

Stackoverflow Survey 2019 - Mosted loved

Programming Language	Rank	share
Clojure	7	68.30%
Erlang	22	47.40%
F#	15	61.70%
Haskell	-	-
Occam	-	-
R	20	51.70%
Rust	1	83.50%
Scala	17	58.03%

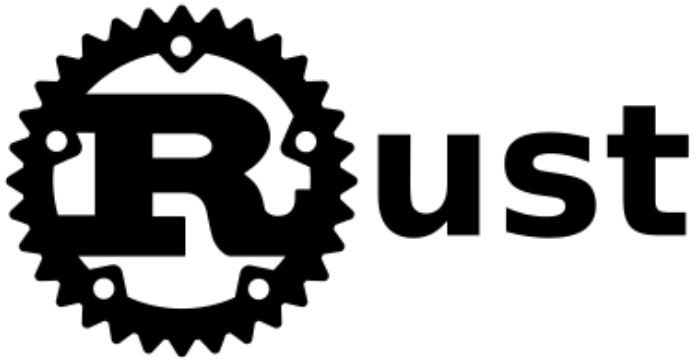
Bron: Stackoverflow Surves 2019

Resultaat overzicht

Ranking	Programming Language	Tiobe	PyPL	SO - MU	SO – ML
1	R	15	7	16	20
2	Rust	34	18	21	1
3	Scala	29	16	20	17
4	Clojure	50		24	7
5	F#	32			15
6	Haskell	43	21		

Ranking	Programming Language	Tiobe	PyPL	SO - MU	SO – ML
7	Erlang	50			22
8	Occam				

Nadat ik alle resultaten naast elkaar heb gezet blijken R, Rust en Scala de meest populaire talen van de shortlist. Uiteindelijk heb ik gekozen, zoals de titel waarschijnlijk al wel doet vermoeden, voor Rust. De reden hiervoor is dat het als "mosted loved" taal uit de Stackoverflow survey komt, en het stond samen met GO al eerder op mijn lijst als interessante talen, dus een win win.



Afbeelding 01: Rust logo

Zoals we al in language en paradigma's hebben gezien zit valt Rust in meerdere paradigma's. Maar grotendeels word het gebruikt als een functionele programmeertaal die zich focust op veiligheid.

En nu .. Rust

Wat nu, gekozen voor Rust, maar hoe nu verder. Eerst maar eens rondneuzen op de website van Rust , en daar blijkt tot mijn verbazing dat de documentatie redelijk goed en uitgebreid is, dit tegenover de Java documentatie waar ik altijd met moeite vind wat ik nodig heb. En ik merk dat ik langzaam steeds enthousiaster word.
De avonden er op kijk ik elke avond een gedeelte van een Pluralsight cursus over Rust Fundamentals gegeven door Dmitri Nesteruk, en dit geeft me steeds meer inzicht in de taal. Tijdens de video's typ ik mee door wat code snippets te runnen in https://play.rust-lang.org/.

Tijd voor het echte werk, en moet een één commando installeer ik de laatste versie van Rust op mijn laptop.

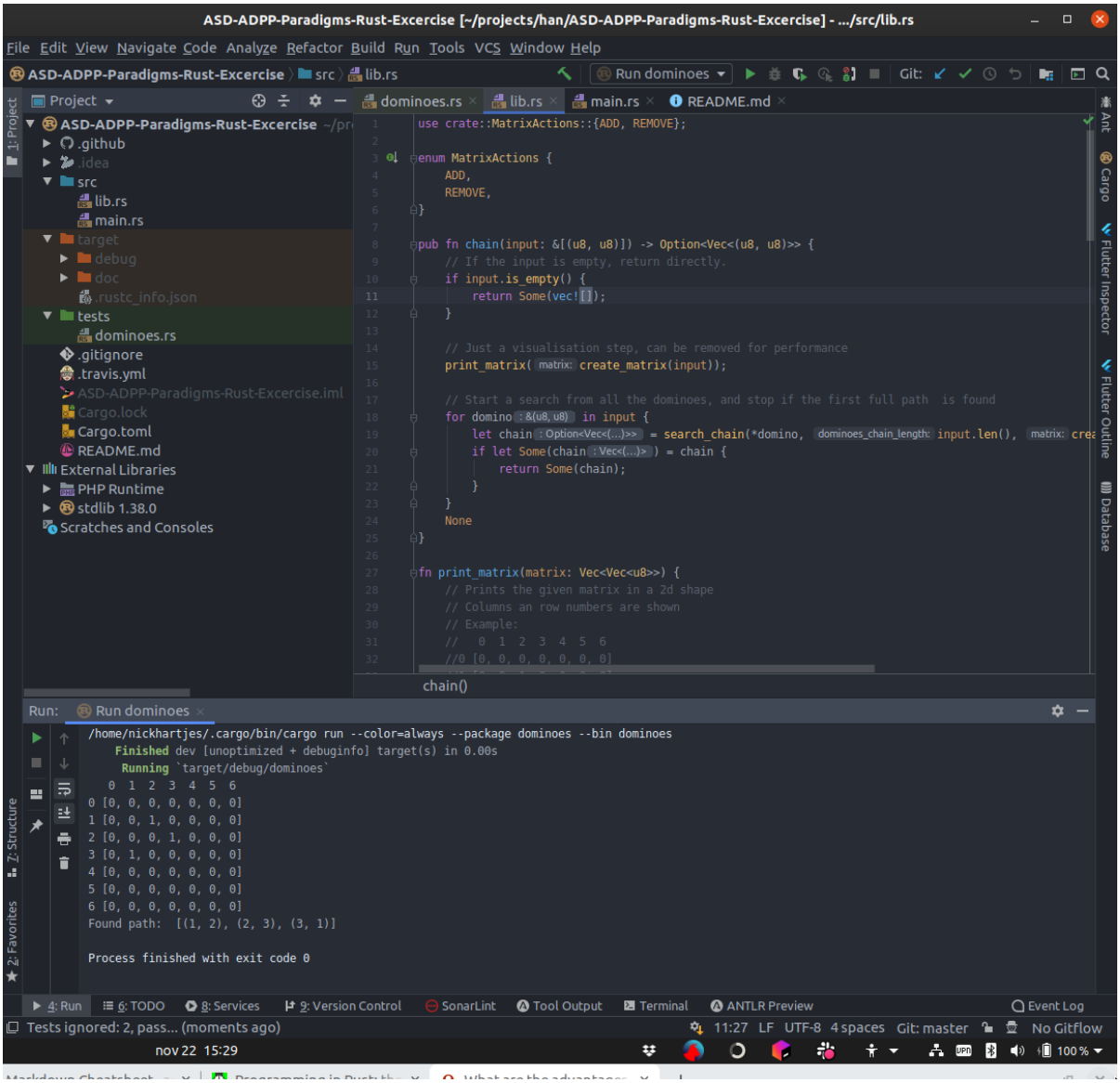
```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

En met rustup kun je rust up to date houden, meer info kun je krijgen door het onderstaande commando in je terminal te typen.

```
nickhartjes@XPS-15 ~ rustup show
Default host: x86_64-unknown-linux-gnu
rustup home:  /home/nickhartjes/.rustup

stable-x86_64-unknown-linux-gnu (default)
rustc 1.38.0 (625451e37 2019-09-23)
```

En voeg de Rust plugin toe aan mijn Intellij. Door de syntax highlighting en de codecompletion maakt het werk toch een stukje aangenamer.



Afbeelding 02: Jetbrains Intellij met Rust plugin

Syntax

Elke taal heeft een syntax, een set van regels, principes en processen die de structuur aangeven.Ik wil nu een aantal voorbeelden van Rust geven, en deze vergelijken met Java.

variabele binding

In rust kun je een let gebruiken, terwijl het een statische getypte taal is, het type past zich dan automatisch aan. Dit word in Rust 'type interference' genoemd. In Java 9 en lager moet je het type van te voren defineren, in Java 10 en hoger heb je ook de mogelijkheid om van een generieke variabele gebruikt te maken.

Rust:

```rust
let x = 50;
```

Java:

```java
int x = 50;

// Optioneel in Java versie 10 en hoger
var x = 50;
```

control flows

De control flows zijn redelijk identiek, de rust syntax gebruikt geen haakjes, terwijl dit in Java wel gebruikt word. Een overzicht van de control flows.

Rust	Java
break / break 'label	break / break label
continue / continue 'label	continue / continue label
for i in 0..n { _ } [1]	for (int i = 0; i < n; i++) { _ }
for i in _ { _ }	for (X i : _) { _ }
if _ { _ } else { _ }	if (_) { _ } else { _ }
if let _ = _ { _ } else { _ }	if (_ = _) { _ } else { _ }
loop { _ }	while (true) { _ }
loop { .. ; if _ { break; }}	do { .. } while (_);
match _ { .. }	switch (_) { .. } [2]
return _	return _
while _ { _ }	while (_) { _ }
while let _ = _ { _ }	while (_ = _) { _ }

Je kunt zien dat Rust een veel nieuwere taal is dan Java, ze hebben met kleine verbeteringen de taal veel effectiever gemaakt. Enkel voorbeelden zijn bijv.

De for-loop

Rust:

```rust
for x in 0..10 {
    println!("{}", x);
}
```

Playground

Java:

```java
for (int x = 0; x < 10; x++) {
    System.out.println( x );
}
```

Semantiek

Mutability

Standaard zijn variabelen inmutable, de waardes kunnen niet veranderen. Dit is gedaan vanuit het veiligheidsoogpunt. Het moet dus een bewuste keuze zijn om iets mutable te maken. Dit kun je uiteindelijk voor elkaar krijgen door het keyword `mut` te gebruiken.

Rust:

```rust
let x = 50;
x = 100;
```

```
error[E0384]: cannot assign twice to immutable variable `x`
 --> src/main.rs:4:5
  |
3 |     let x = 50;
  |         -
  |         |
  |         first assignment to `x`
  |         help: make this binding mutable: `mut x`
4 |     x = 100;
  |     ^^^^^^^ cannot assign twice to immutable variable

error: aborting due to previous error

For more information about this error, try `rustc --explain E0384`.
error: could not compile `playground`.
```

Playground

```rust
let mut x = 50;
x = 100;
```

Playground

Bij Java is alles mutable, en kan direct gewijzigd worden.

Java:

```java
int x = 50;
x = 100;
```

De match

Een groot verschil tussen de match in Rust en de switch in Java is dat de match in Rust exhaustive moet zijn. Dit betekent dat hij alle cases die er zijn moet afdekken. Hierdoor heb je zekerheid dat er altijd een branch word gekozen. Bij Java hoeft dit niet, dit levert vooral bij het refactoren van code nog wel eens de nodige issues op.

Rust:

```
let x = 5;

match x {
    1 => println!("one"),
    2 => println!("two"),
    3 => println!("three"),
    4 => println!("four"),
    5 => println!("five"),
    _ => println!("something else"),
}
```

Playground

Java:

```
switch(x){
    case 1:
        System.out.println("one");
        break;
    case 2:
        System.out.println("two");
        break;
    case 3:
        System.out.println("three");
        break;
    case 4:
        System.out.println("four");
        break;
    case 5:
        System.out.println("five");
        break;
    case default:
        System.out.println("something else");
        break;
}
```

Pragmatiek

enumarate

In plaats van extern een counter bij te houden in een loop is het mogelijk een geïntegreerde counter van enumerate te gebruiken.

Voorbeeld zonder:

```
for (index, value) in (5..10).enumerate() {
    println!("index = {} and value = {}", index, value);
}
```

Playground

Voorbeeld Java

```
int index = 0;
for (int i = 5; i < 10; i++) {
  System.out.println("index =" + index + "and value = " + value);
}
```

De if-let

Voordeel van de if-let is dat je geen variabele in de main scope hoeft de definiëren. Als de variabele aan de voorwaarde voldoet word hij gevuld.

Rust:

```
if let Some(x) = option {
    foo(x);
}
```

Java:

```
var b;
if ( x == y){
    b = foo(x)
}
```

De coding challenge

Afbeelding 03: Een chain of dominoes

Iedereen kan wel even wat loopjes schrijven, een struct maken en daarna wijzigen of natuurlijk als eerste

```
fn main() {
  println!("Hello World!");
}
```

in de rust playground uitvoeren. Maar je leert de taal pas echt wanneer je het doelgericht gaat gebruiken. Omdat dit te bereiken zocht ik een leuke coding challenge, om mezelf uit te dagen. Uiteindelijk kwam ik terecht op de website van excercism.io. Daar hebben ze meer dan 3000 coding challenges voor 52 verschillende talen. En ook Rust staat er tussen met 92 excercises. De exercises zijn gecategoriseerd met een moeilijkheidsgraad van easy, medium en hard.

Na wat oefeningen bekeken te hebben, valt mijn oog op de volgende opdracht

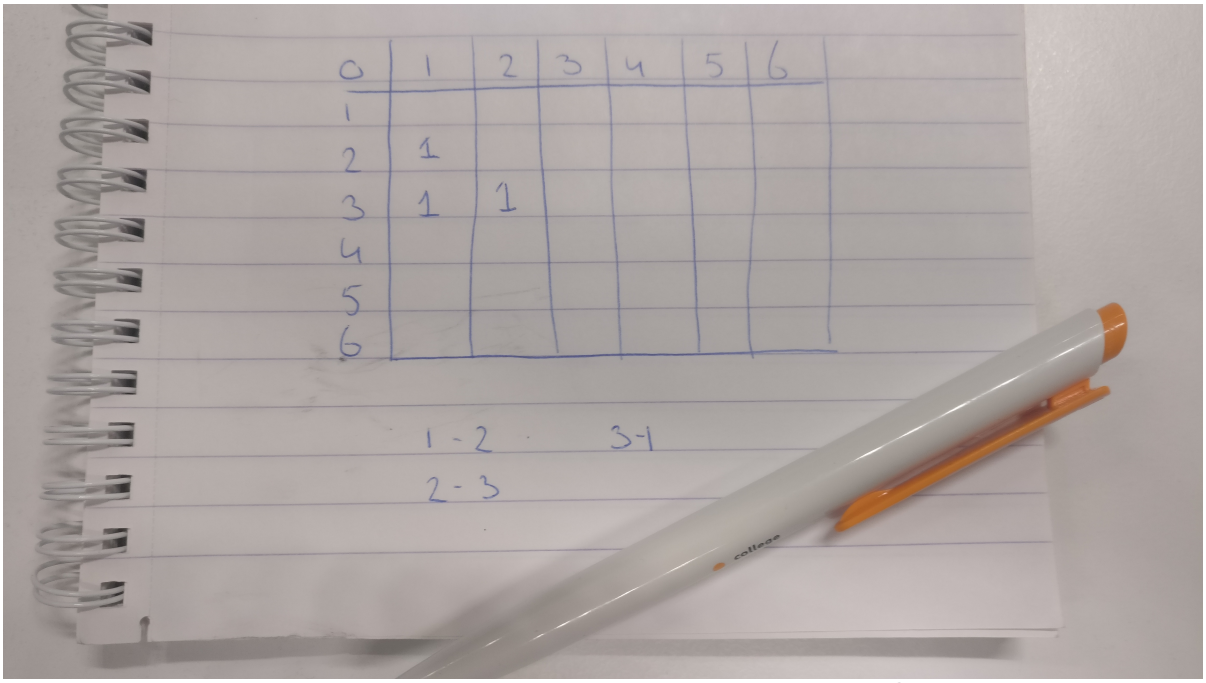
Make a chain of dominoes.

Compute a way to order a given set of dominoes in such a way that they form a correct domino chain (the dots on one half of a stone match the dots on the neighbouring half of an adjacent stone) and that dots on the halves of the stones which don't have a neighbour (the first and last stone) match each other.
For example given the stones `[2|1]`, `[2|3]` and `[1|3]` you should compute something like `[1|2] [2|3] [3|1]` or `[3|2] [2|1] [1|3]` or `[1|3] [3|2] [2|1]` etc, where the first and last numbers are the same.
For stones `[1|2]`, `[4|1]` and `[2|3]` the resulting chain is not valid: `[4|1] [1|2] [2|3]`'s first and last numbers are not the same. 4 != 3
Some test cases may use duplicate stones in a chain solution, assume that multiple Domino sets are being used.

Deze gaat het worden, leuk en toch uitdagend. Binnen 5 uur wil ik het oplossen, veel ruimer dan ik het met Java zou inschatten, maar bij voorbaat neem ik de tijd wat ruimer om de taal te kunnen ontdekken. De uitdaging wil ik met een graph gaan oplossen, dit heb ik vaker gedaan in Java, maar ik snap het datastructuur dus wil het nu implementeren in Rust. Tevreden ben ik wanneer ik na afloop alle unit testen op groen heb staan. Dat is wel 1 van de voordelen van exercism.io, bij elke opdracht krijg je een opzet van een project en daarbij behorende unittesten om te valideren of je opdracht succesvol is.

- **S**pecifiek - De opdracht is duidelijk omschreven.
- **M**eetbaar - Succesvol wanneer alle unittesten groen zijn
- **A**cceptabel - De coding challenge is goedgekeurd door de Docent
- **R**ealistisch - Na een kleine analyse haalbaar, ondanks het gebrek van kennis over de taal Rust.
- **T**ijdgebonden - Binnen 5 uur

Ik wil gebruik gaan maken van een adjacency matrix om de domino stenen in bij te houden. Dit geeft me een eenvoudige manier om te zoeken of er domino stenen zijn die aangrenzend zijn.



Afbeelding 04: Matrix op papier, met insert van domino's [1-2, 2-3, 3 -1]

Als ik bijvoorbeeld een dominosteen zoek die aan een 2 gelegd kan worden, dan hoef ik alleen maar de 2e rij en de 2e kolom na te lopen. En dan vind ik alle stenen die aangelegd zouden kunnen worden.
Dit zou theoretisch ook met een adjacency list kunnen, alleen zou ik dan een dubbele boekhouding moeten bijhouden. Dan moet `[2, 1]` in de linkedlist van `1` toevoegen en ook in de linkedlist van `2`. Haal ik de domino weg, moet ik ze ook weer beide gaan verwijderen. Dus het brengt extra complexiteit met zich mee. En geen extra voordelen zie, ook niet ten opzichte van tijdscomplexiteit.

De eerste stappen

De eerste stappen zijn het maken van de adjacency matrix en het toevoegen van de domino stenen in de matrix. Initieel zat ik heel erg in de denkwijze van het objectgeoriënteerd paradigma. "Oow.. dan moet ik een Struct maken, en daar functionaliteit aan toevoegen zodat ik de state van de struct kan wijzigen." Maar realiseerde dat dat misschien wel kon, maar niet het doel was van de opdracht, en heb er toen bewust voor gekozen om te proberen om het in het functionele paradigma te houden. Het doel is ook om alle functies idempotent te maken. Simpel gezegd, de uitkomst van een functie veranderd niet, ook al roep je deze ontelbaar vaak achter elkaar aan.

```rust
use crate::MatrixActions::{ADD, REMOVE};

enum MatrixActions {
    ADD,
    REMOVE,
}

pub fn chain(input: &[(u8, u8)]) -> Option<Vec<(u8, u8)>> {
    // If the input is empty, return directly.
    if input.is_empty() {
        return Some(vec![]);
    }

    create_matrix(input));
}

fn create_matrix(input: &[(u8, u8)]) -> Vec<Vec<u8>> {
```

```
    // Creates a 2d Vector matrix from given domino slices
    let matrix_size: usize = 7;
    let mut matrix: Vec<Vec<u8>> = vec![vec![0; matrix_size]; matrix_size];

    // Loop the slice input, that contains tuples of u8
    for domino in input {
        matrix = update_matrix(*domino, matrix, ADD);
    }
    matrix
}

fn update_matrix(domino: (u8, u8), matrix: Vec<Vec<u8>>, action:
MatrixActions) -> Vec<Vec<u8>> {
    let mut tmp_matrix = matrix.clone();
    let row_location = domino.0 as usize;
    let column_location = domino.1 as usize;
    match action {
        ADD => tmp_matrix[row_location][column_location] = 1,
        REMOVE => tmp_matrix[row_location][column_location] = 0,
    };
    tmp_matrix
}
```

Toen liep ik eigenlijk tegen het eerste probleem aan, als IDE gebruik IntelliJ van Jetbrain, met de Rust plugin. Dat werkte tot nu toe perfect.. alleen werkt de debugger niet. Hoe moest ik nu controleren of mijn matrix wel correct gevuld werd. Dus uiteindelijk maar een print functie toegevoegd die de 2d Vector weergeeft in de output.

```
fn print_matrix(matrix: Vec<Vec<u8>>) {
    // Prints the given matrix in a 2d shape
    // Columns an row numbers are shown
    // Example:
    //   0  1  2  3  4  5  6
    //0 [0, 0, 0, 0, 0, 0, 0]
    //1 [0, 0, 1, 0, 0, 0, 0]
    //2 [0, 0, 0, 1, 0, 0, 0]
    //3 [0, 1, 0, 0, 0, 0, 0]
    //4 [0, 0, 0, 0, 0, 0, 0]
    //5 [0, 0, 0, 0, 0, 0, 0]
    //6 [0, 0, 0, 0, 0, 0, 0]
    for (row_count, row) in matrix.iter().enumerate(){
        if row_count == 0 {
            print!("  ");
            for x in 0..row.len() {
                print!(" {} ", x);
            }
            println!();
        }
        print!("{} ", row_count);
        println!("{:?}", row);
    }
}
```

De matrix representeert welke domino's nog beschikbaar zijn. Normaal zou ik niet direct naar de print of prinln wat schrijven, eerder naar een logger. Maar voor deze case vond ik het eigenlijk goed genoeg.

Het zoek algoritme

Het idee is om een recursieve functie te maken, die 1 domino als variabele krijgt. Vervolgens word in de matrix gecontroleerd welke domino's er op de tweede waarde van de domino variabele kunnen worden aangesloten.
Deze domino's worden worden dan vervolgens gebruikt in de recursive functie. Zo word er stap voor stap een Vector gecreëerd met een lijst van aaneensluitende domino's.
Wanneer de recursive functie detecteert dat het einddoel is bereikt, dus een path van de volledige lengte van de input slice, en dat de eerste waarde van de eerste domino en laatste waarde van de laatste domino in de Vector ook op elkaar aansluiten, dan stopt de recursive functie. En word de path teruggestuurd.

```
pub fn chain(input: &[(u8, u8)]) -> Option<Vec<(u8, u8)>> {
    // If the input is empty, return directly.
    if input.is_empty() {
        return Some(vec![]);
    }

    // Just a visualisation step, can be removed for performance
    print_matrix(create_matrix(input));kan

    // Start a search from all the dominoes, and stop if the first full path
is found
    for domino in input {
        let chain = search_chain(*domino, input.len(), create_matrix(input),
Vec::new());
        if let Some(chain) = chain {
            return Some(chain);
        }
    }
    None
}

fn search_chain(
    domino: (u8, u8),
    dominoes_chain_length: usize,
    matrix: Vec<Vec<u8>>,
    mut path: Vec<(u8, u8)>,
) -> Option<Vec<(u8, u8)>> {
    // Remove the current domino from the matrix
    let tmp_matrix = update_matrix(domino, matrix, REMOVE);
    // Create the result path
    let mut result: Vec<(u8, u8)> = Vec::new();

    // To match
    let mut first = 0;
    let mut last = u8::max_value();
    if !path.is_empty() {
        first = path[0].0;
        last = path[path.len() - 1].1
    }

    if last == domino.0 || path.is_empty() {
        path.push(domino);
        first = path[0].0;
        last = path[path.len() - 1].1
    } else {
        // insert reverse domino
        path.push((domino.1, domino.0));
        last = domino.0;
```

```
    }

    // Check correct path
    if path.len() == dominoes_chain_length && first == last {
        println!("Found path:  {:?} ", path);
        Some(path)
    } else {
        // Search both sides of the domino
        result.append(search_in_row(domino.1, tmp_matrix.clone()).as_mut());
        result.append(search_in_column(domino.1,
tmp_matrix.clone()).as_mut());

        // Recursive call to the nodes found
        if !result.is_empty() {
            for x in result.iter() {
                let row = x.0;
                let column = x.1;
                return search_chain(
                    (row, column),
                    dominoes_chain_length,
                    tmp_matrix.clone(),
                    path.clone(),
                );
            }
        }
        None
    }
}

fn search_in_column(column_number: u8, matrix: Vec<Vec<u8>>) -> Vec<(u8, u8)>
{
    let mut result: Vec<(u8, u8)> = Vec::new();
    for (row_counter, row) in matrix.iter().enumerate() {
        if let 1 = row[column_number as usize] {
            let domino = (row_counter as u8, column_number);
            result.push(domino);
        }
    }
    result
}

fn search_in_row(row_number: u8, matrix: Vec<Vec<u8>>) -> Vec<(u8, u8)> {
    let mut result: Vec<(u8, u8)> = Vec::new();
    for (column_counter, value) in matrix[row_number as
usize].iter().enumerate() {
        if let 1 = value {
            let domino = (row_number, column_counter as u8);
            result.push(domino);
        }
    }
    result
}
```

[Domino's uitwerking op repl.it]
[Source code op Github.com]

Helaas kon ik `cargo test` niet runnen op repl.it. Dus repl.it is wel ideaal om de applicatie te delen en online uit te voeren, maar niet ideaal om geautomatiseerd mee te bouwen en te testen. Dus daarom heb ik ook gebruik gemaakt van de build runner van Github en Travis-CI om geautomatiseerd de code te builden, testen en uit te voeren.
Build runner / action in Github.com
Build runner in Travis-CI

Tijd op

En.. doel gehaald ? Nee, helaas ben ik gestruikeld net voor de finishlijn. Van de 12 testen zijn er 10 geslaagd. Op dit moment gaat het nog mis met dubbele domino's waardes zoals `[1,2] & [1,2]`
Terwijl er het volgende duidelijk in de opdracht staat:

> *Some test cases may use duplicate stones in a chain solution, assume that multiple Domino sets are being used.*

Op dit moment maak ik gebruik van een `0` of een `1` in mijn matrix. Een `0` wanneer er geen domino is, en een `1` wanneer er wel een domino is. Geen echte binaire waarde, maar wel het idee.
Dus bij het toevoegen van een 2e domino met een identieke waarde word nu de `1` overschreven met een nieuwe `1`.

Uiteindelijk zou dit nog wel redelijk makkelijk te repareren zijn door het toevoegen van een domino de matrixwaarde met 1 te verhogen. En met verwijderen de matrixwaarde met 1 te verlagen.

```
    match action {
        ADD => tmp_matrix[row_location][column_location] += 1,
        REMOVE => tmp_matrix[row_location][column_location] -= 1,
    };
```

Zo zou je meerdere domino's met dezelfde waarde in de matrix kunnen bijhouden. Waarschijnlijk zou je nog wel wat andere code moeten aanpassen om het werkende te krijgen. Maar gelukkig hebben we de testen om te zien of er dan niets breekt.

Conclusie

Rust heeft een hele fijne indruk op me achtergelaten. Niet alleen ben ik redelijk snel gewend geraakt aan de syntax en semantiek, maar aan het hele ecosysteem. Versies van Rust die makkelijk te updaten zijn, en out of the box word `cargo` meegeleverd.
En `cargo` is echt een Zwitsers zakmes. Waar je in Java meerder tools nodig hebt, zoals Maven, Gradle, Sonarqube om een bepaalde kwaliteitsstandaard te halen, zitten al veel features bij default al in `cargo`.

- cargo test = om de testen te runnen
- cargo build = om de build te starten
- cargo doc = om documentatie te genereren
- cargo fmt = een code formater
- cargo clippy = een code linter

Wat ik ook een grote plus vond, is dat de error meldingen van de compiler heel duidelijk en leesbaar worden weergegeven. Geen rare stacktrace, maar een goede en vaak duidelijke melding en advies.
Hierdoor is de instap een stuk prettiger.

Moet er dus binnenkort wat gebouwd worden in een een low level snelle taal, dan is Rust ten zeerste aan te bevelen.