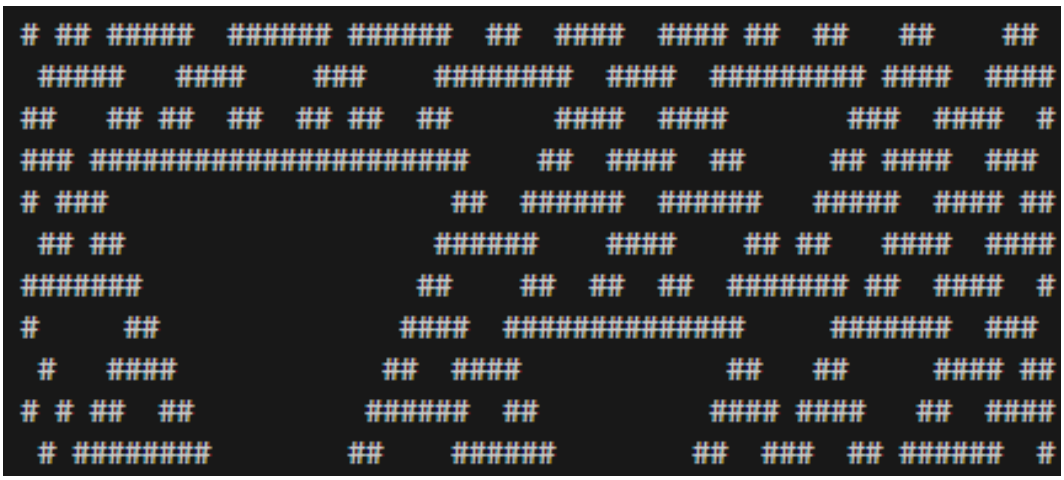
Programming assignment: Cellular automata

M. Asselbergs

July 2025



1 Introduction

Welkom! Vandaag tillen jullie jezelf naar *programming professionalism* aan de hand van deze opdracht!

We bouwen vandaag een simulatie van een cellular automaton. Die naam maakt niet per se veel uit, het gaat mij meer om hoe cool het eruit ziet! Wat je op de afbeelding op deze pagina ziet is zo'n simulatie. Iedere regel is een eigen generatie, en iedere generatie bepaalt zijn volgende generatie. Als jij de eerste generatie geeft, en de manier waarop generaties ververst worden, doet je code zo meteen het simuleren!

Ready? Maak een folder met een naam zonder spaties aan op je computer. Open deze in Visual Studio Code (via `File` links bovenin) en open een nieuwe terminal (via `Terminal` links bovenin). Typ in deze terminal `cargo init` en druk op enter om dit commando uit te voeren. Als er geen errors waren toen je dit hebt uitgevoerd, kunnen we aan de slag!

Here we go!

2 In gesprek!

In mijn optiek is het allerbelangrijkste aan programmeren dat je goed kan praten met de computer. Nou ja, dat de computer kan praten tegen jou, omdat je zo kan weten of er gebeurt wat je wil. Het is leuk als je een idee goed kan vormgeven, maar je zal op een manier moeten testen of dit idee ook goed is overgekomen bij de computer.

Als het goed is, kan je links op je scherm in een folder `src` bij een bestand genaamd `main.rs`. De `.rs` geeft aan dat we hier werken met Rust code; precies waar we voor kwamen! Open dit bestand en voer het uit door in je terminal het commando `cargo run` uit te voeren. Óf, pas eerst de text aan zodat er net een leuker bericht op je scherm verschijnt zo.

Wat je eerst ziet bij het uitvoeren lijkt op

```
   Compiling <project> v0.1.0 (<folder_van_project>)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in <tijd>
     Running `target\debug\<project>.exe`
```

en geeft aan dat je programma gecompileed (= gebouwd) wordt, en dan uitgevoerd. Hierna zie je de output van je programma.

Gefeliciteerd! Je eerste programma draait en je heb officieel een gesprek met je computer! Haal nu diep adem, want we gaan in een keer met een aantal concepten aan de slag!

3 Variabelen en types

3.1 Variabelen om data op te slaan

Oké! In het programmeren werk je met twee helften van hetzelfde geheel: data en instructies. Data kan bepalen welke instructies je uitvoert (als ... dan ...) en wat de uitkomst is van die instructies ($3 + 1 \neq 24 + 1$). Instructies hebben dus data nodig, en om iets zinnigs te kunnen doen heeft je programma instructies nodig. Tijd om het dan in ieder geval over een van de twee te hebben: data.

In Rust kunnen we data opslaan in variabelen. Dat doe je met de woorden `let` en `mut`. Je gebruikt alleen `let` voor data die je niet wil veranderen, maar die je wel nodig hebt.

```
let achternaam = "Asselbergs";  // "" is nodig om het te zien als tekst.
let voorletter = 'M';           // '' geeft een enkel karakter aan.
let lengte = 1.88;              // Gebruik punten voor kommagetallen.
let geboortejaar = 2001;
let birthday = [1, 8];          // [...] is een lijst, hier van getallen.
```

Als je er `mut` aan toevoegt, geef je aan dat je de data als *mutable* wil beschouwen en dat je deze aan wil passen.

```
let mut leeftijd = 23;
// Oh nee, wacht...
leeftijd = 24;                  // Geen `let` of `mut` nodig!
```

Je kan deze data nu al gebruiken in je programma door deze aan je `println!` toe te voegen. Dat doe je door `{}` en de namen van je variabelen toe te voegen; bijvoorbeeld als volgt.

```
let geboortejaar = 2001;
let mut leeftijd = 23;

println!(
    "Ik ben geboren in {} dus ik ben {} jaar oud.",
    geboortejaar, leeftijd
);

leeftijd = 24;

println!(
    "Wacht. Nee, ik ben inmiddels {} jaar oud!",
    leeftijd
);
```

Probeer zoiets uit in Visual Studio Code! Maak een variabele met een waarde en print 'm.

3.2 Eigen types bedenken

Iedere variabele heeft een datatype. Die types kan Rust vaak zelf uitvogelen, dus die hoef je niet in je code te schrijven. Als je dat wél wil doen, ziet het eerste blokje over variabelen er als volgt uit.

```
let achternaam: &str = "Asselbergs";     // Een string of characters; tekst
let voorletter: char = 'M';              // Een character
let lengte: f32 = 1.88;                  // 32-bit floating point kommagetal
let geboortejaar: i32 = 2001;            // 32-bit integer, een rond getal
let birthday: [i32; 2] = [1, 8];         // Een lijst van 2 i32's
```

Hier staat het datatype er direct bij voor de duidelijkheid.

In Rust is over vele soorten data al nagedacht, maar ieder programma kan baat hebben bij heel specifieke datatypes en daarom heb je zelf twee manieren beschikbaar om je eigen types te maken. De eerste optie die je hebt is een `enum` (ofwel: enumeration), waarin je zelf al de mogelijke waardes meegeeft. Een voorbeeld van een zelfgemaakt `Direction` type:

```
// De volgende regel hoef je niet te begrijpen. Gewoon kopiëren.
#[derive(Clone, Copy, Debug)]
enum Direction {
    North,
    East,
    South,
    West
}
```

Variabelen van dit type kunnen drie verschillende waardes aannemen: `Direction::North`, `Direction::East`, `Direction::South` en, unsurprisingly, `Direction::West`. Als je andere waardes in een variabele van dit type probeert te stoppen, krijg je een error. Als je precies weet welke paar opties beschikbaar zijn, kan je een `enum` maken.

Je andere tool is het bouwen van een `struct` (ofwel: structure), waarin je meerdere variabelen opslaat als één geheel. Wat dacht je van het volgende voorbeeld, bijvoorbeeld:

```
#[derive(Clone, Copy, Debug)]
struct Destination {
    direction: Direction,       // Welke kant op?
```

```
    distance: u32               // Hoe ver? 32-bit NIET NEGATIEF rond getal
}
```

Iedere variabele van het type `Destination` heeft als het ware zijn eigen variablen `direction` en `distance`. Die noemen we fields. Om bij die fields te komen gebruik je de naam van je `Destination`, gevolgd door een punt en dan de naam van een field. Als je zo'n variabele wil aanmaken en gebruiken, ziet dat er dus ongeveer zo uit:

```
let sophie_cat_cafe = Destination {
    direction: Direction::South,
    distance: 36,               // Kilometer
};

println!(
    "To reach your cat cafe, head {:?} and drive {} kilometers.",
    sophie_cat_cafe.direction,
    sophie_cat_cafe.distance
);
```

Let erop dat je voor het printen van de waarde van je `enum` een `{:?}` gebruikt. Dit heeft te maken met de `derive(Debug)` die bij diens definitie staat en is niet heel belangrijk verder. Je zou een variabele `dest` van type `Destination` dan ook kunnen printen met een `{:?}` aangezien deze ook een `derive(Debug)` heeft.

Probeer het vorige voorbeeld uit te voeren in Visual Studio Code en probeer ook de `sophie_cat_cafe` variabele in een keer te printen.

3.3 Terug naar de opdracht!

Tijd om aan de slag te gaan met onze cellular automata! En we beginnen met het modeleren van een `Cell`. Zoals je op het voorblad kan zien, kan een cel actief zijn of niet.

En laten we vervolgens een `Generation` type maken die een regel voorstelt zoals op het voorblad. Een lijst met 5 cellen, zou een goed onderdeel kunnen zijn van zo'n generatie. In ieder geval voor nu. Die lijst kunnen we later altijd groter maken.

Gebruik `Direction` en `Destination` als inspiratie voor je `Cell` en `Generation`. Misschien is het ook slim om nog een keer te kijken naar het voorbeeld aan het begin van 3.2 voor als je lijsten wil gebruiken.

Nog niet per se *heel* indrukwekkend, maar onthoud dat we tot nu toe één van de puzzelstukjes gebruikt hebben: data. Hoog tijd om de andere helft erbij te betrekken. Tijd voor gedrag!

4 Gedrag

4.1 Gedrag voor specifieke types

Laten we beginnen bij onze eigen types: `Cell` en `Generation`. Die types kunnen we eigen gedrag geven in de vorm van een `impl` (ofwel: implementation). Dat wordt jullie opdracht zo. Ik geef eerst een voorbeeld met een ander bekend type: `Direction`.

Een voorbeeld van een functie voor het `Direction` type zou de functie `print_as_arrow` kunnen zijn. Een implementatie daarvoor zou eruit kunnen zien als volgt.

```
impl Direction {
    fn print_as_arrow(self) {
        match self {
            Direction::North => print!("^"),
            Direction::East  => print!(">"),
            Direction::South => print!("V"),
            Direction::West  => print!("<")
        }
    }
}
```

Hier geeft `impl Direction {...}` aan dat de omvatte functies bij het type `Direction` horen. Het stukje `fn print_as_arrow(self) {...}` geeft aan dat we een functie maken die met als input `self`. Omdat deze implemetatie geldt voor een `Direction`, is `self` een `Direction`. De `match self {...}` gaat `self` vergelijken met alle opties voor het type van `self`; met alle opties voor een `Direction` dus. Voor elk van deze opties wordt in dit geval een ander stukje code uitgevoerd zodat we een pijltje tekenen in de juiste richting. We gebruiken hier trouwens `print!` in plaats van `println!` omdat `print!` niet een regel naar beneden springt na het printen van diens inhoud. Dit wordt later belangrijk.

Als je deze code wil testen, kan je in je `main` functie een `Direction` aanmaken en die op de volgende manier printen:

```
fn main() {
    let dir: Direction = // Dit mag je zelf doen ;P

    dir.print_as_arrow();       // Alsof je fields gebruikt, maar met ().
}
```

Dan is het nu hoog tijd om een functie te schrijven die hoort bij het `Cell` type. Zorg ervoor dat deze functie op basis van de waarde van een cell een spatie of een hekje print. Als je iets anders wil gebruiken dan een hekje en een spatie, dan kan dat ook. Wees creatief!

4.2 Herhaling

We komen in de buurt van het printen van een eerste generatie! You're almost there!!!

Nu dat we gedrag kunnen toevoegen aan types (`fn` en `impl`), en nu dat we kunnen kiezen wat we uitvoeren op basis van data (`match`) is het tijd om het te hebben over herhaling. Waarom zou je immers dezelfde code meerdere keren schrijven als je aan Rust kan vragen om code te herhalen?

Introducing: de `for`-loop! Zo'n loop schrijf je als `for ding in lijst {...}` met code in `...` die je gaat herhalen voor ieder `ding` in lijst `lijst`. Dat maak ik duidelijker met een voorbeeld.

Stel: je komt net terug van een concert van Mother Mother (een geweldige naam in een stuk over herhaling). Je hebt een nummer van ze in je hoofd en het gaat er maar niet uit, en je wil je waardering ervoor eigenlijk wel graag uitdrukken. In een programma. In Rust. Natuurlijk :) Hoe begin je daar aan? Kijk met me mee.

```
fn main() {
    print!("Artiest:");

    // Herhaal de volgende code voor ieder element in de lijst
    // van nummers vanaf 0 tot aan en ZONDER 2, dus 0 en 1; 2 herhalingen.
    // Geef deze getallen geen naam want ongebruikt (vandaar de _).
    for _ in 0..2 {
        print!(" Mother");
    }

    println!("");
    println!("Lyrics:");

    println!("...");
}
```

Deze code kan je zo in Visual Studio Code gooien om te zien wat'ie doet, maar ik denk dat het ook slim is om te zien of je al kan bedenken hoe je output eruit gaat zien. De herhaling scheelt hier toch zeker één hele `print!` en da's niet niks! Dat is een goed begin, maar de voordelen worden groter als ik begin aan de lyrics van het nummer. Let op.

```
fn main() {
    print!("Artiest:");

    for _ in 0..2 {
        print!(" Mother");
    }

    println!("");
    println!("Lyrics:");

    // 4 herhalingen.
    // Maak het getal beschikbaar in de lijst-code als variabele `n`.
    for n in 0..4 {
        print!("On");

        for _ in 0..4 { print!(" and on"); }

        println!(" (n = {})", n);
    }

    println!("You're my best friend. I'd have your kid.");
    println!("...");
}
```

Kijk, nu komen we ergens! Dit scheelt een aantal lange `print!`s en daarmee een hoop typwerk. En hier hebben we zelfs een voorbeeld van het gebruiken van een element uit de lijst in een variabele voor de loop-code: `n`. Dat gebruik van de waardes in de lijst kan heel handig zijn als de lijsten over iets anders gaan dan nummers. Laatste voorbeeld, dat beloof ik:

```rust
fn main() {
    let genodigden = [
        "Achlys", "Cody",  "Myrthe",
        "Roos",   "Sarah", "Sem"
    ];

    for naam in genodigden {
        println!("Thank you so much for attending, {}!", naam);
    }
}
```

Ook dit scheelt veel typen, maar dit geeft voornamelijk flexibiliteit. Als ik 13 nieuwe mensen wil uitnodigen, dan kan ik hun namen in de lijst zetten en dan werkt de code nog steeds zoals ik wil! Zonder loop zou dat me 13 extra `println!`s kosten, allemaal met mogelijke spelfouten. Knap staaltje programmeerwerk! Maar wacht, gebruikten wij niet ook een lijst voor onze `Generation`s..?

Yes we do! Aan jou nu de taak om een functie te schrijven die hoort bij het `Generation` type en die een hele generatie print (de lastigste opdracht van de dag). Je hebt al een functie geschreven voor het printen van een enkele cel en je weet nu hoe je herhaling kan gebruiken om dingen uit te voeren voor een hele lijst. Bovendien heb je gezien hoe je een waarde van een `struct` type in een variabele kan stoppen en hoe je `enum` waardes en lijsten kan maken, duuuuusss... Ik denk dat je het voor elkaar kan krijgen om deze functie te schrijven én dat je in `main` handmatig een `Generation` kan maken om te printen. Vraag vooral elkaar en mij om hulp als je dat graag wil, maar onderschat jezelf niet!

Hint: we gebruiken voor de functie hier een `fn` in een `impl`, met in die functie een `for`-loop met als lijst een field van een `self`. Da's een hoop stof in een keer samen, maar ik heb er alle vertrouwen in dat je er zelf of met de anderen uit kan komen. Zet 'm op!

5 Een cadeautje!

Wait. Je hebt de vorige opdracht voor elkaar gekregen?!?! Holy crap! Well done!!!

Dat verdient een cadeautje! En wat is er nu een beter cadeau dan diefstal?

Wij kunnen andermans code gebruiken voor ons eigen project! En met behulp van de code in het "crate" `rand` kunnen we een `Generation` vullen met willekeurige cellen. Dat scheelt suuuuper veel schrijfwerk bij het bouwen van een `Generation` met meer cellen. Ik presenteer, als cadeau, een nieuwe functie om in de `impl` voor `Generation` type te zetten.

```rust
// Deze functie heeft geen input (want `random()` ipv `random(<iets>)`) en
// geeft een `Generation` terug als output (want `-> Self`).
fn random() -> Self {
    use rand;                          // Gebruik gestolen code

    let mut states = [false; 5];       // Maak een lijst van 5 keer `false`

    for i in 0..5 {
        states[i] = rand::random(); // Zet hier en daar `true` ipv `false`
    }

    return Generation {                // Vertaal t/f naar Active/NotActive
        cells: states.map(|b| if b {Cell::Active} else {Cell:NotActive}),
    };
}
```

Hoe deze code precies werkt is niet al te belangrijk. Ik zou 'm lekker kopiëren en plakken. Wat wel handig is, is weten hoe je 'm kan gebruiken. Als er in een functie in een `impl` geen `self` wordt meegegeven als input, roep je de functie aan met een dubbele dubbele punt: `Generation::random()`. Deze functie retourneert een `Generation` die je in een variabele kan zetten als je dat wil. Zo kan je nu bijvoorbeeld je `main` functie als volgt schrijven.

```
fn main() {
    let genn = Generation::random();
    genn.print();
}
```

En als je dan `cargo run` meerdere keren uitvoert, krijg je steeds een andere generatie te zien! Dat effect is misschien wel makkelijker te zien met een `Generation` van 60 cellen in plaats van 5... En misschien is het ***nóg*** makkelijker te zien als je keer een `Generation` onder elkaar print met een loop. :)

Zet jij dat nog even recht? Misschien is een `println!("")` ergens handig om ze *onder elkaar* te zetten.

6 Almost there!

Je bent aangekomen bij het eenvoorlaatste deel van de opdracht! Hier staat een kleine nieuwe functie voor je `Generation` op de planning.

We gaan zo meteen een nieuwe generatie opbouwen op basis van de huidige generatie en een regel voor het vernieuwen van cellen. Daarvoor schrijven we eerst een functie `cell_neighborhood(...)` voor `Generation`s om de waardes van cellen op te vragen. Voor diens invulling moeten we echter wel even kijken naar `if` en het opvragen van waardes uit lijsten. Gelukkig zijn deze concepten niet al te lastig.

6.1 What `if` ...

Als je code alleen uit wil voeren als er aan een bepaalde voorwaarde voldaan wordt, gebruik je `if voorwaarde {...}`. Als je meerdere voorwaarden wil testen kan je een of meerdere keren `else if voorwaarde2 {...}` toevoegen. In dat geval worden de voorwaardes een voor een bekeken en wordt alleen de code uitgevoerd die hoort bij de eerste voorwaarde waaraan voldaan wordt. Gedrag dat uitgevoerd moet worden als er aan geen van de eerdere voorwaardes voldaan wordt, vat je in `else {...}`. Een voorbeeld voor de vorm:

```
fn main() {
    let fave_nr = 415;

    if fave_nr > 0 {            // `fave_nr` is groter dan 0
        if fave_nr >= 20 {      // `fave_nr` is groter dan of gelijk aan 20
            println!("BIG favorite number: {}", fave_nr);
        }

        println!("Favorite number is positive.");
    } else if fave_nr == 0 {
        println!("Favorite number is not positive, nor negative.");
    } else {
        println!("Favorite number is negative.");
    }
}
```

6.2 Wacht, tellen begint bij 0?!

Ja, dat doet het zeker! In ieder geval in de meeste programmeertalen en zo ook in Rust. Dit is belangrijk bij het opvragen van waardes uit een lijst. Daar gebruiken we namelijk een lijst en een index om een element uit die lijst te plukken: `lijst[index]`. Om het eerste element uit een lijst te plukken gebruiken we dus `lijst[0]`, niet `lijst[1]`. Lijsten kan je dan als volgt gebruiken:

```
fn main() {
    let genodigden = [
        "Achlys", "Cody",  "Myrthe",
        "Roos",   "Sarah", "Sem"
    ];

    println!(
        "Extra stoutpunt voor {}, {} en {}!",
        genodigden[2],      // Myrthe
        genodigden[3],      // Roos
        genodigden[5]       // Sem
    );
}
```

Hmmmm. Sounds about right. Maar wat nu als ik vraag om `genodigden[10]`, of `genodigden[-3]`, of, als ik het écht erg maak, `genodigden["appelboom"]`? Goeie vraag! De laatste twee geven een probleem in Rust door hun types: een `i32` en `&str` (string). Een index moet altijd als type `usize` hebben. Dat is een rond getal dat 0 is of groter en dat zoveel bits heeft als de computer gebruikt voor addressering (komt uit de vorige les en is nu niet belangrijk).

Maar sure, een rond getal groter dan of gelijk aan 0, dan mag 10 dus? Jazeker, qua type mag je 10 gebruiken! Maar Rust kan zelf (vaak) bedenken dat 10 als index buiten `genodigden` ligt, dus krijg je er bij `cargo run` een error over.

Dan nu jullie opdracht: schrijf een functie die voor een `Cell` in een `Generation` teruggeeft wat zijn waarde is én wat de waarde van diens buurcellen is. Als een van die buren een index heeft buiten de lijst van cellen mag je zelf kiezen of deze als default actief zijn of niet. Ik kies vaak voor inactief.

De functie ziet eruit als `fn cell_neighborhood(self, index: usize) -> [Cell; 3]` Je kan deze functie ook met redelijk gemak testen in `main` om het gesprek met de computer gaande te houden. Dat helpt mij vaak.

7 The final result

Tijd voor de laatste functie: `fn update(self) -> Self`! En voor deze functie heb je niet zo gek veel nieuwe kennis nodig. Het enige dat je moet weten is dat je `match` ook kan gebruiken om een waarde te kiezen voor in een variable. Dat wil zeggen dat je eem `match` kan schrijven aan de rechterkant van een `=` voor een vullen van een variabele.

```
enum Nummer {
    Een,
    Twee,
    Drie
}
```

```rust
fn main() {
    let enum_num = Nummer::Drie;

    let num_num = match enum_num {
        Nummer::Een  => 1,
        Nummer::Twee => 2,
        Nummer::Drie => 3
    };  // Als je match gebruikt voor een variabele schrijf je een ;
}
```

Dit werkt zelfs met lijsten (hint hint)!

```rust
fn main() {
    let enum_nums = [Nummer::Een, Nummer::Twee];

    let plus = match enum_nums {
        [Nummer::Een,  Nummer::Een ] => 2,
        [Nummer::Een,  Nummer::Twee] => 3,
        [Nummer::Een,  Nummer::Drie] => 4,
        [Nummer::Twee, Nummer::Een ] => 3,
        // -- snip --
        [Nummer::Drie, Nummer::Drie] => 6
    }
}
```

Dan is nu de laatste opdracht om de functie `update(...)` te schrijven voor het type `Generation`. Deze functie krijgt als input én als output een `Generation` en baseert iedere cel in de nieuwe generatie op de "neighborhood" van de oude cel op dezelfde index. Ik zou hiervoor beginnen met een nieuwe variabele `let mut new_cells = [Cell::Active; 60];`. Hierna kan je iedere cel afgaan om diens neighborhood op te vragen om die te gebruiken voor het vinden van de waarde waarmee je `new_cells[...]` kan overschrijven.

Als dit allemaal gelukt is, kan je in `main` een loopje schrijven dat begint met een `Generation::random()` en bijvoorbeeld 100 updates daarvan laat zien. Hierbij kan je gebruiken dat je variabelen kan hergebruiken: `cel = cel.update();` is toegestaan.

Gefeliciteerd! Je hebt hiermee de opdracht afgerond! Als je dat wil, kan je nog wat kleine updates maken. Zo kan je bijvoorbeeld je `for _ in 0..100 {...}` in `mail` vervangen door `loop {...}` (een eindeloze loop). Dan zou ik je wel aanraden om ook de regel `std::thread::sleep(std::time::Duration::from_millis(50));` toe te voegen aan wat er in de loop staat. Ook kan je ervoor kiezen om de `match` in functie `update(...)` te veranderen door de rechterkanten van de `=>`-pijlen te veranderen. Zo kan je variëren!

Dankjewel voor het meedoen! Daarmee maak je me blij!