Simulanka
Language Specification & User Manual

Alexandru Bianca Ana-Maria

Academic Year 2025-2026

Contents

Part I
Language Specification

0.1 Overview

Simulanka is a domain-specific programming language designed for simulating genetic algorithms represented on a matrix. This language can:

- Define species with genetic properties

- Create fitness functions that evaluate individuals

- Specify mutation and crossover rules

- Visualize the evolution of generations

- Run parallel simulation instances

The language handles all the complexities of genetic algorithms: selection, reproduction, and population management.

0.2 Implementation Language Choice

I chose to implement Simulanka in **Rust** for its extraordinary performance. Genetic algorithms run thousands of steps and generations, so execution time is one of the most important things to consider. Each millisecond saved in a single step can amount to minutes in the long term.

Another great benefit of Rust is its ownership model preventing the vast majority of memory leaks, although caution had to be advised when using Arc with reference cycles, and manual clearing had to be applied there to ensure memory was freed.

The egui crate also provided an efficient way to implement graphics without much low-level work, keeping the visualization fast and responsive.

0.3 Code Organization

The interpreter has been split into modular components:

Module	Purpose
`types.rs`	Core data structures (Value, Environment, Program, etc.)
`lexer.rs`	Tokenization of source code
`parser.rs`	Parse tokens into Abstract Syntax Tree
`eval.rs`	Expression and command evaluation logic
`world.rs`	World simulation (stepping, fitness, mutation)
`evolution.rs`	Evolutionary algorithm (selection, crossover, generations)
`semantic.rs`	Static semantic analysis and validation
`gui.rs`	GUI visualization using egui/eframe
`main.rs`	Entry point and program orchestration

1 Lexical Structure

1.1 Character Set

- **Letters**: `a-z`, `A-Z`, `_`

- **Digits**: `0-9`

- **Whitespace**: spaces, tabs, newlines (ignored except as separators)

- **Operators**: `+ - * / % = > < !  & |  .`

- **Delimiters**: `{ } ( ) [ ] :  ; ,`

1.2 Comments

Single-line comments begin with `//` and extend to the end of the line:

```
1        //this is a single-line comment
2        x = 5; //inline comment
```

1.3 Keywords

The following identifiers are reserved keywords. Keywords are **case-insensitive** (`SPAWN`, `Spawn`, and `spawn` are all valid):

ENVIRONMENT	SPECIES	SPAWN	FITNESS
MUTATE	EVOLVE	VISUALIZE	ROUTINE
if	else	while	for
in	return	print	at
true	false	random	environment

1.4 Literals

1.4.1 Integer Literals

Integer literals consist of one or more decimal digits:

⟨*integer*⟩ ::= ⟨*digit*⟩
    | ⟨*integer*⟩ ⟨*digit*⟩

⟨*digit*⟩ ::= '0'
    | '1'
    | '2'
    | '3'
    | '4'
    | '5'
    | '6'
    | '7'
    | '8'
    | '9'

Examples: `0`, `67`, `1000`

1.4.2 String Literals

String literals are enclosed in double quotes:

$\langle string \rangle ::= $ '' `<char-list>` ''

$\langle char\text{-}list \rangle ::= \langle character \rangle$
     | $\langle char\text{-}list \rangle \, \langle character \rangle$

Examples: `"hello"`, `"Species A"`

1.4.3 List Literals

List literals are enclosed in square brackets with comma-separated elements:

$\langle list \rangle ::= $ '[' ']'
     | '[' $\langle exp\text{-}list \rangle$ ']'

Examples: `[1, 2, 3]`, `[random(0, 50), random(0, 50)]`

1.5 Operators

Category	Operators	Description
Arithmetic	`+ - * / %`	Addition, subtraction, multiplication, division, modulo
Comparison	`== != > < >= <=`	Equality, inequality, relational
Logical	`&& ||`	Conjunction, disjunction
Assignment	`=`	Value assignment
Access	`. [ ]`	Field access, indexing

2 Formal Syntax - BNF Grammar

2.1 Program Structure

A valid Simulanka program consists of a series of blocks. The following blocks are **obligatory** (in any order): ENVIRONMENT, SPECIES, EVOLVE, FITNESS, MU-TATE, SPAWN. The VISUALIZE block is optional. In the case it is not provided, the gui will run without a visual representation of the best generations.

$\langle program \rangle ::= \langle block\text{-}list \rangle$

$\langle block\text{-}list \rangle ::= \langle block \rangle$
     | $\langle block\text{-}list \rangle \, \langle block \rangle$

$\langle block \rangle ::= \langle env\text{-}block \rangle$
     | $\langle species\text{-}block \rangle$
     | $\langle spawn\text{-}block \rangle$
     | $\langle fitness\text{-}block \rangle$
     | $\langle mutate\text{-}block \rangle$
     | $\langle evolve\text{-}block \rangle$
     | $\langle visualize\text{-}block \rangle$

2.2   Environment Block

⟨*env-block*⟩ ::= 'ENVIRONMENT' '{' ⟨*env-prop-list*⟩ '}'

⟨*env-prop-list*⟩ ::= ⟨*env-prop*⟩
    | ⟨*env-prop-list*⟩ ⟨*env-prop*⟩

⟨*env-prop*⟩ ::= ⟨*identifier*⟩ ':' ⟨*value*⟩ ⟨*opt-comma*⟩

⟨*opt-comma*⟩ ::= ','
    | ε

⟨*value*⟩ ::= ⟨*integer*⟩
    | ⟨*identifier*⟩
    | 'true'
    | 'false'

2.3   Species Block

⟨*species-block*⟩ ::= 'SPECIES' '{' ⟨*species-item-list*⟩ '}'
    | 'SPECIES' ⟨*identifier*⟩ '{' ⟨*species-prop-list*⟩ '}'

⟨*species-item-list*⟩ ::= ⟨*species-item*⟩
    | ⟨*species-item-list*⟩ ⟨*species-item*⟩

⟨*species-item*⟩ ::= ⟨*routine-def*⟩
    | ⟨*species-def*⟩

⟨*species-def*⟩ ::= ⟨*identifier*⟩ '{' ⟨*species-prop-list*⟩ '}'

⟨*species-prop-list*⟩ ::= ⟨*species-prop*⟩
    | ⟨*species-prop-list*⟩ ⟨*species-prop*⟩

⟨*species-prop*⟩ ::= ⟨*identifier*⟩ ':' ⟨*exp*⟩ ⟨*opt-sep*⟩
    | 'routine' ':' ⟨*identifier*⟩ ⟨*opt-sep*⟩

⟨*opt-sep*⟩ ::= ';'
    | ','
    | ε

⟨*routine-def*⟩ ::= 'ROUTINE' ⟨*identifier*⟩ ⟨*cmd-block*⟩

2.4   Spawn Block

⟨*spawn-block*⟩ ::= 'SPAWN' ⟨*cmd-block*⟩

2.5   Fitness Block

⟨*fitness-block*⟩ ::= 'FITNESS' ⟨*cmd-block*⟩

2.6 Mutate Block

⟨*mutate-block*⟩ ::= '`MUTATE`' '`{`' ⟨*mutation-rule-list*⟩ '`}`'

⟨*mutation-rule-list*⟩ ::= ⟨*mutation-rule*⟩
    | ⟨*mutation-rule-list*⟩ ⟨*mutation-rule*⟩

⟨*mutation-rule*⟩ ::= ⟨*identifier*⟩ '`:`' ⟨*cmd-block*⟩ ⟨*opt-comma*⟩

2.7 Evolve Block

⟨*evolve-block*⟩ ::= '`EVOLVE`' '`{`' ⟨*evolve-prop-list*⟩ '`}`'

⟨*evolve-prop-list*⟩ ::= ⟨*evolve-prop*⟩
    | ⟨*evolve-prop-list*⟩ ⟨*evolve-prop*⟩

⟨*evolve-prop*⟩ ::= '`generations`' '`:`' ⟨*integer*⟩ ⟨*opt-comma*⟩
    | '`instances`' '`:`' ⟨*integer*⟩ ⟨*opt-comma*⟩

2.8 Visualize Block

⟨*visualize-block*⟩ ::= '`VISUALIZE`' ⟨*cmd-block*⟩

2.9 Commands

⟨*cmd-block*⟩ ::= '`{`' ⟨*command-list*⟩ '`}`'

⟨*command-list*⟩ ::= ⟨*command*⟩
    | ⟨*command-list*⟩ ⟨*command*⟩

⟨*command*⟩ ::= ⟨*assign-cmd*⟩
    | ⟨*if-cmd*⟩
    | ⟨*while-cmd*⟩
    | ⟨*for-cmd*⟩
    | ⟨*return-cmd*⟩
    | ⟨*print-cmd*⟩
    | ⟨*spawn-cmd*⟩
    | ⟨*exp-cmd*⟩

⟨*assign-cmd*⟩ ::= ⟨*target*⟩ '`=`' ⟨*exp*⟩ ⟨*opt-semi*⟩

⟨*opt-semi*⟩ ::= '`;`'
    | ε

⟨*target*⟩ ::= ⟨*identifier*⟩
    | ⟨*exp*⟩ '`.`' ⟨*identifier*⟩
    | ⟨*exp*⟩ '`[`' ⟨*exp*⟩ '`]`'

⟨*if-cmd*⟩ ::= '`if`' '`(`' ⟨*bexp*⟩ '`)`' ⟨*cmd-block*⟩ ⟨*else-opt*⟩

⟨*else-opt*⟩ ::= ε
      | 'else' ⟨*else-body*⟩

⟨*else-body*⟩ ::= ⟨*cmd-block*⟩
      | ⟨*if-cmd*⟩

⟨*while-cmd*⟩ ::= 'while' '(' ⟨*bexp*⟩ ')' ⟨*cmd-block*⟩

⟨*for-cmd*⟩ ::= 'for' ⟨*identifier*⟩ 'in' 'environment' ⟨*cmd-block*⟩

⟨*return-cmd*⟩ ::= 'return' ⟨*exp*⟩ ⟨*opt-semi*⟩

⟨*print-cmd*⟩ ::= 'print' '(' ⟨*exp-list-opt*⟩ ')' ⟨*opt-semi*⟩

⟨*spawn-cmd*⟩ ::= 'spawn' ⟨*identifier*⟩ '@' '(' ⟨*exp*⟩ ',' ⟨*exp*⟩ ')' ⟨*opt-semi*⟩

⟨*exp-cmd*⟩ ::= ⟨*exp*⟩ ⟨*opt-semi*⟩

2.10 Expressions (exp)

⟨*exp*⟩ ::= ⟨*term*⟩
      | ⟨*exp*⟩ '+' ⟨*term*⟩
      | ⟨*exp*⟩ '-' ⟨*term*⟩

⟨*term*⟩ ::= ⟨*primary*⟩
      | ⟨*term*⟩ '*' ⟨*primary*⟩
      | ⟨*term*⟩ '/' ⟨*primary*⟩
      | ⟨*term*⟩ '%' ⟨*primary*⟩

⟨*primary*⟩ ::= ⟨*integer*⟩
      | ⟨*string*⟩
      | ⟨*list*⟩
      | ⟨*identifier*⟩
      | ⟨*identifier*⟩ '(' ⟨*exp-list-opt*⟩ ')'
      | '(' ⟨*exp*⟩ ')'
      | ⟨*primary*⟩ '.' ⟨*identifier*⟩
      | ⟨*primary*⟩ '[' ⟨*exp*⟩ ']'

⟨*exp-list-opt*⟩ ::= ε
      | ⟨*exp-list*⟩

⟨*exp-list*⟩ ::= ⟨*exp*⟩
      | ⟨*exp-list*⟩ ',' ⟨*exp*⟩

2.11 Boolean Expressions (bexp)

⟨*bexp*⟩ ::= ⟨*and-exp*⟩
      | ⟨*bexp*⟩ '||' ⟨*and-exp*⟩

⟨*and-exp*⟩ ::= ⟨*cmp-exp*⟩
      | ⟨*and-exp*⟩ '&&' ⟨*cmp-exp*⟩

$\langle cmp\text{-}exp \rangle ::= \langle exp \rangle \ \langle cmp\text{-}op \rangle \ \langle exp \rangle$

$\langle cmp\text{-}op \rangle ::= \text{`=='}$
$\qquad | \quad \text{`!='}$
$\qquad | \quad \text{`>'}$
$\qquad | \quad \text{`<'}$
$\qquad | \quad \text{`>='}$
$\qquad | \quad \text{`<='}$

3 Type System

3.1 Overview

Simulanka uses a **dynamic type system** with **static validation**. Variables don't need type declarations, the interpreter figures out types automatically. Before running, it checks for common mistakes like using undefined variables.

3.2 Types

Type	Description
`Int`	signed integers (e.g., `67`, `-5`)
`Bool`	Boolean values (`true` or `false`)
`String`	Text enclosed in quotes (e.g., `"hello"`)
`List`	Ordered collection of values (e.g., `[1, 2, 3]`)
`Object`	An individual with named properties (e.g., `self.x`)
`Environment`	The 2D grid containing all individuals

3.3 Type Coercion

- **Bool to Int**: `true` becomes 1, `false` becomes 0

- **Object to Int**: A valid object becomes 1, missing/null becomes 0

- **List to Int**: Non-empty list becomes 1, empty list becomes 0

3.4 Static Validation

Before running a program, Simulanka checks for:

- **Undefined Variables**: Using a variable before assigning it

- **Missing Properties**: Accessing `self.speed` when `speed` isn't defined in the species

- **Undefined Routines**: A species referencing a routine that doesn't exist

3.5 How Types Are Determined

The interpreter figures out types based on context:

- A number like `42` is always an `Int`

- Text in quotes like `"hello"` is always a `String`

- Square brackets like `[1, 2, 3]` create a `List`

- Using `self.property` accesses an `Object`

- Math operations (like `a + b`) produce an `Int`

4 Operational Semantics

Simulanka uses **big-step semantics** to describe how programs execute. This section explains how expressions produce values and how commands modify program state.

4.1 Expression Evaluation

Expressions are evaluated to produce values. The notation $\langle e, \sigma \rangle \Downarrow v$ means "expression e in state σ produces value v".

4.1.1 Literals

$$\frac{}{\langle 67, \sigma \rangle \Downarrow 67}$$

$$\frac{}{\langle "hello", \sigma \rangle \Downarrow "hello"}$$

4.1.2 Variable Lookup

Variables are looked up in the current scope. If not found locally, parent scopes are searched:

$$\frac{\sigma(x) = v}{\langle x, \sigma \rangle \Downarrow v}$$

4.1.3 Arithmetic Operations

$$\frac{\langle e_1, \sigma \rangle \Downarrow v_1 \qquad \langle e_2, \sigma \rangle \Downarrow v_2}{\langle e_1 + e_2, \sigma \rangle \Downarrow v_1 + v_2}$$

4.1.4 Field Access

$$\frac{\langle e, \sigma \rangle \Downarrow \text{obj} \qquad \text{obj}.f = v}{\langle e.f, \sigma \rangle \Downarrow v}$$

4.1.5 List and Grid Indexing

Lists start from index 0. The environment grid uses double-indexing `environment[x][y]`:

$$\frac{\langle list, \sigma \rangle \Downarrow [v_0, v_1, \ldots] \qquad \langle i, \sigma \rangle \Downarrow 1}{\langle list[i], \sigma \rangle \Downarrow v_1}$$

4.2 Command Evaluation

4.2.1 Assignment

Evaluate the right side, store the result:

$$\frac{\langle e, \sigma \rangle \Downarrow v}{\langle x = e, \sigma \rangle \Rightarrow \sigma[x \mapsto v]}$$

4.2.2 Conditional (if-else)

If condition is true, execute first branch; otherwise execute second:

$$\frac{\langle b, \sigma \rangle \Downarrow \text{true} \qquad \langle c_1, \sigma \rangle \Rightarrow \sigma'}{\langle \textbf{if } (b)\{c_1\} \textbf{ else } \{c_2\}, \sigma \rangle \Rightarrow \sigma'}$$

4.2.3 While Loop

Check condition; if true, execute body and repeat:

$$\frac{\langle b, \sigma \rangle \Downarrow \text{false}}{\langle \textbf{while } (b)\{c\}, \sigma \rangle \Rightarrow \sigma}$$

$$\frac{\langle b, \sigma \rangle \Downarrow \text{true} \qquad \langle c, \sigma \rangle \Rightarrow \sigma' \qquad \langle \textbf{while } (b)\{c\}, \sigma' \rangle \Rightarrow \sigma''}{\langle \textbf{while } (b)\{c\}, \sigma \rangle \Rightarrow \sigma''}$$

4.2.4 For Loop

Binds each element to the loop variable and executes the body:

$$\frac{\text{individuals} = [ind_1, \ldots, ind_n] \qquad \text{for each } ind_j \colon \langle c, \sigma[x \mapsto ind_j] \rangle \Rightarrow \sigma_j}{\langle \textbf{for } x \textbf{ in environment } \{c\}, \sigma \rangle \Rightarrow \sigma_n}$$

4.2.5 Spawn Command

Creates a new individual at the given position:

$$\frac{\langle e_x, \sigma \rangle \Downarrow x \qquad \langle e_y, \sigma \rangle \Downarrow y}{\langle \textbf{spawn } S \textbf{ @ } (e_x, e_y), \sigma \rangle \Rightarrow \text{add individual of species } S \text{ at } (x, y)}$$

5 How Evolution Works

5.1 Overview

1. **Create Worlds**: Start n parallel world instances, each with the same initial setup

2. **Simulate**: Run all worlds for a fixed number of steps

3. **Evaluate**: Calculate fitness for each individual

4. **Select**: Keep the top 50% best-performing worlds

5. **Reproduce**: Create new children using crossover from survivors

6. **Mutate**: Apply random mutations to all children

7. **Repeat**: Go back to step 2 for the next generation

5.2 World Instances and Parallel Execution

Each generation consists of multiple **world instances**—independent copies of the simulation that run in parallel. This serves two purposes:

- **Exploration**: Different random seeds lead to different behaviors

- **Speed**: Using the Rayon library, all instances run simultaneously on multiple CPU cores. Although Rayon complicated the development and introduced many issues that needed to be fixed, the execution time improvements were worth the debugging time.

In Rust, parallel execution looks like this:

```
1    // Run all worlds in parallel using Rayon's
         par_iter_mut
2    self.instances.par_iter_mut().for_each(|world| {
3        world.generation = g;
4        for _ in 0..env_steps {
5            world.step();   // Simulate one time step
6        }
7        world.calculate_total_fitness();
8    });
```

The `par_iter_mut()` function from Rayon automatically distributes work across CPU threads.

5.3 Fitness Evaluation

After simulation, the `FITNESS` block runs for each individual. This block has access to:

- `self` : the individual being evaluated

- `environment` : the entire world grid

To return a fitness score, use one of:

```
FITNESS {
    return self.food_collected;   // Explicit return
}
// OR
FITNESS {
    score = self.x * 2;           // Implicit 'score'
        variable
}
```

If neither is provided, the fitness defaults to 0.

5.4   Selection: Keeping the Best

After all instances finish, they are sorted by their best individual's fitness. The top half survive to become parents:

```
// Sort instances by fitness (highest first)
indices.sort_by_key(|&i| -self.instances[i].fitness);

// Keep only the top 50%
let keep_count = (num_instances / 2).max(1);
```

The bottom 50% of worlds are discarded and their genetic traits will not pass on.

5.5   Crossover: Creating Children

For the non-elite children (new instances that replace the discarded ones), crossover combines traits from two parents. The user can define how this works in the MUTATE block:

```
MUTATE {
    crossover: {
        // parent1, parent2, and child are
            automatically provided
        child.speed = (parent1.speed + parent2.speed) /
            2;
        child.direction = parent1.direction;
    }
}
```

In Rust, the interpreter sets up the crossover environment:

```
// Bind parent1, parent2, and child for the crossover
    code
env_mut.store.insert("parent1", Value::Object(
    parent1_env));
env_mut.store.insert("parent2", Value::Object(
    parent2_env));
env_mut.store.insert("child", Value::Object(child_env))
    ;

// Execute the user's crossover code
```

```
7            for cmd in body {
8                cmd.execute(crossover_env, ...);
9            }
```

5.6   Mutation: Adding Variation

After crossover, mutation adds random changes. The `mutation` rule receives `self`:

```
1        MUTATE {
2            mutation: {
3                if (random(0, 100) < 5) {   // 5% chance
4                    self.speed = random(1, 10);
5                }
6            }
7        }
```

Important: Both crossover and mutation are user-defined. If you don't define them, nothing happens automatically.

5.7   Memory Management

Each generation can creates many objects, and even reference cycles which can prevent Arc from dropping the reference counter to 0. To prevent memory leaks, Simulanka performs **manual state cleanup** between generations:

```
1        // Only keep persistent state (position + species
              properties)
2        // Temporary variables from FITNESS/ROUTINES are
              dropped
3        let mut store = HashMap::new();
4
5        // Copy position
6        store.insert("x", parent_env.get("x"));
7        store.insert("y", parent_env.get("y"));
8
9        // Copy only species-defined properties (the "DNA")
10        for key in species_def.properties.keys() {
11            store.insert(key, parent_env.get(key));
12        }
13        // Temporary variables like 'queue', 'visited', etc.
              are NOT copied
```

This ensures that only the "genetic" information passes to the next generation.

5.8   Simultaneous Update Semantics

During simulation, all individuals see a **snapshot** of the environment from the beginning of each step. This means:

- The order in which individuals are processed doesn't affect results

- Two individuals can "see" each other at the same time

14

- No race conditions between individuals

```
// In world.rs: Take snapshot before processing
fn step(&mut self) {
    let mut snapshot = Vec::new();
    for ind in &self.individuals {
        snapshot.push(ind.deep_clone());
    }

    // Now all individuals use this frozen snapshot
    for ind in &mut self.individuals {
        // ind sees the snapshot, not real-time changes
    }
}
```

5.9 Timeline Summary

Phase	What Happens
Generation 0	SPAWN runs once to create initial individuals
Each Generation	Simulate → Fitness → Select → Crossover → Mutate
Selection	Top 50% worlds survive
Crossover	Only for new children (not elite survivors)
Mutation	Applied to ALL individuals after crossover

6 Built-in Functions

6.1 Random Number Generation

```
random(min, max)
```

Returns a random integer between min (inclusive) and max (exclusive).

6.2 List Operations

```
len(list)      // Returns the number of elements
push(list, v)  // Adds value to the end of the list
pop(list)      // Removes and returns the last element
```

6.3 Spatial Query

```
get_at(x, y)
```

Returns the individual at grid position (x, y), or 0 if the cell is empty.

6.4 Distance Calculation

```
1         dist(obj1, obj2)
```

Returns the distance between two objects, calculated from their x and y properties. Returns an integer (the floor of the actual distance).

6.5 Drawing Functions

These functions are used inside the VISUALIZE block to draw shapes:

```
1         draw_rect(x, y, w, h, r, g, b)
2         draw_line(x1, y1, x2, y2, r, g, b, thickness)
3         draw_circle(x, y, radius, r, g, b)
```

RGB values range from 0-255. The RGB parameters and thickness are optional and default to white (255, 255, 255) and 1.0 respectively.

7 Error Handling

7.1 Lexical Errors

The lexer tracks line and column numbers for all tokens for precise error localization.

7.2 Syntax Errors

Parse errors report:

- Expected token type

- Found token type

- Line and column number

Example:

```
Error at line 15:8: Expected RBrace, found Identifier("x")
```

7.3 Semantic Errors

The static semantic pass catches:

- **Undefined variables**: Variable 'x' is not defined!

- **Undefined properties**: Species property 'speed' is not defined!

- **String operator misuse**: Cannot use operator '*' on a String

7.4 Runtime Errors

- **Division by zero**: Returns 0

- **Index out of bounds**: Returns 0 for reads, no-op for writes

- **Missing file**: Displays error message and exits

8 Design Decisions

8.1 Why Simulanka?

This project arose from my personal curiosities, having an interest in genetic algorithms and how something so simple could seemingly improve on its own. Maze generation, specifically, alongside mini-world simulations, were long-time dreams I hoped to achieve on my own instead of seeing them online.

8.2 Block-Based Structure

I designed the syntax in an intuitive way, each block with a different role being in its own separate place, making the code easier to read and forcing the user to separate the logic clearly in their mind, leading to a clearer understanding of the algorithm. Debugging each block is thus easier, and isolating bugs will be faster than if the code were an entire wall of unbroken text.

8.3 Grid-Based Environment

A 2D grid was chosen because it offers better opportunities in both understanding the environment that individuals are placed in, and for making the visualization process easier. By forcing everything on a grid, writing code that will let you see exactly how your species evolve becomes trivial. Additionally, many simulations involve spatial relationships (mazes, flocking, resources), so the decision to be grid-based only does not limit the user too much.

Thanks to fast $O(1)$ lookup using grid coordinates, the environment also helps with the much-needed time optimization.

8.4 No type declarations

Variables don't need type declarations, but the interpreter still catches common errors (undefined variables, missing properties) before running. This gives flexibility without sacrificing safety. I believe that the most difficult part of writing Simulanka code should be thinking about the logic of the algorithm, not implementing that logic. Thus, declaring the variable type would just lead to unnecessary friction.

8.5 Multi-threading

Time is of the essence, so multi-threading was a necessary hurdle I had to pass. The interpreter uses Rust's `rayon` library for parallelism:

- `Arc`: Shared ownership across threads

- `RwLock`: Safe read/write access to shared data

- `par_iter_mut()`: Distributes work across CPU cores

8.6   Future Improvements

While Simulanka is currently a functional language, future improvements could include supporting parameters for routines to allow for more modular code. In the rock-paper-scissors example all 3 species share nearly identical logic, but it was still necessary to write 3 routines, while a generalized routine with parameters could have handled for all 3.

Additionally, user-defined functions would allow more complexity and reduce code duplication.

9   Complete Examples

This section documents all included example simulations with screenshots showing early and evolved states.

9.1   Example 1: Maze Evolution (`maze.txt`)

Description: Evolves a 21×21 maze where the goal is to create a solvable but challenging path from the top-left corner (0,0) to the bottom-right corner (20,20). The fitness function uses BFS to find the shortest path and rewards longer, more complex paths with more reachable cells.

Key Features:

- Each cell has genes controlling wall connections to neighbors

- BFS-based pathfinding in the FITNESS block

- Rewards: reachability, path length, wall count

- Visual path highlighting from start to goal
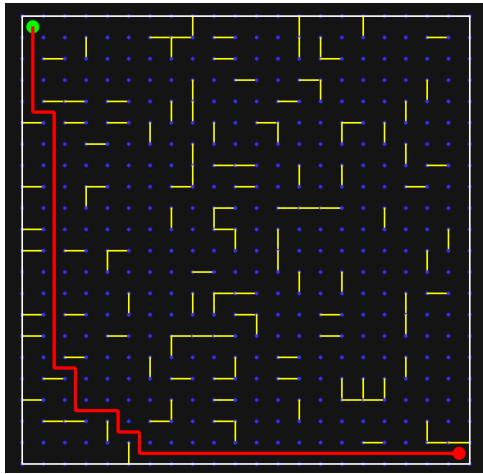
Figure 9.1: Maze at early generations
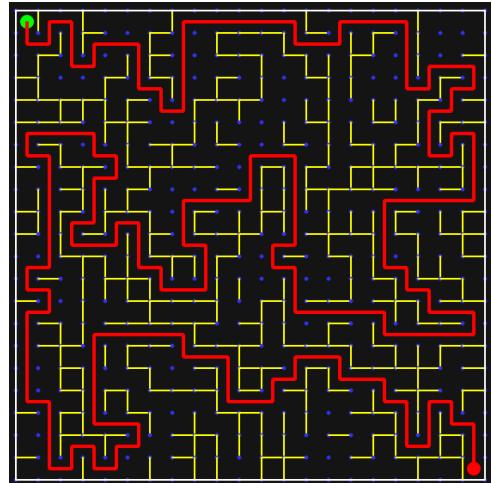
Figure 9.2: Maze at later generations

9.2 Example 2: Map Generator (`map.txt`)

Description: Generates a 20×20 terrain map with smooth transitions between terrain types. Terrains range from deep water (0) through shallow water, sand, grass, forest, hills, to mountains (6). The fitness penalizes abrupt transitions (e.g., water next to mountains).

Key Features:

- 7 terrain types with color-coded visualization

- Adjacency rules: same terrain +20, adjacent type +30, 2-step +5, 3+ step -50

- Diversity bonus for including multiple terrain types

- Island-like shape encouraged (water at edges, land in center)
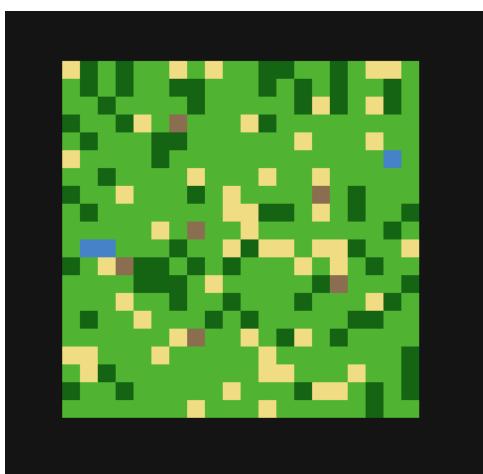
Figure 9.3: Map at early generations
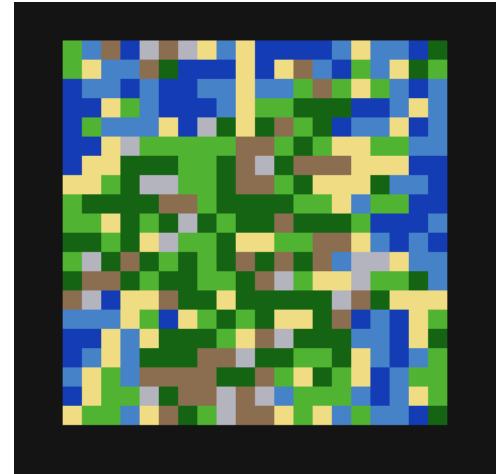
Figure 9.4: Map at later generations

9.3 Example 3: Flocking Simulation (`flock.txt`)

Description: Simulates bird flocking behavior (Boids algorithm). Each bird has genetic parameters controlling cohesion (stay with flock), separation (avoid collisions), and alignment (match flock velocity). Evolution optimizes these parameters for tight but collision-free flocking.

Key Features:

- 20 birds with velocity and steering behavior

- Three genetic parameters: cohesion, separation, alignment

- Fitness rewards tight grouping, penalizes collisions

- Direction indicators show bird velocity
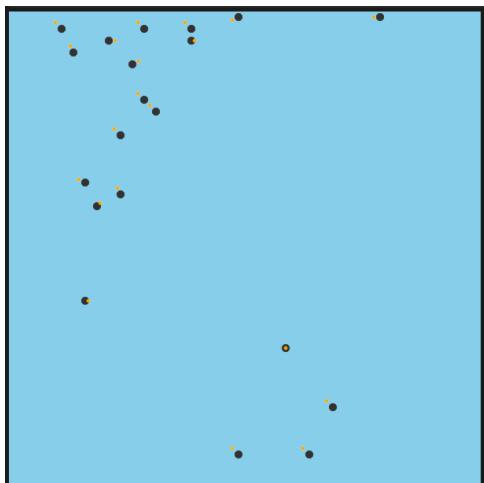
Figure 9.5: Flock at early generations
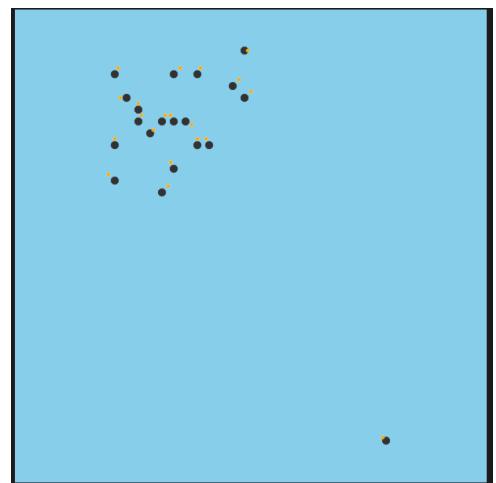
Figure 9.6: Flock at later generations

9.4 Example 4: Wolf Pack Hunting (`wolves.txt`)

Description: A predator-prey simulation where wolves hunt sheep. When a wolf catches a sheep, the sheep "respawns" at a random edge (teleporting to safety). Evolution optimizes spawn positions and hunting strategies. Fitness rewards successful catches.

Key Features:

- 12 sheep (white) and 4 wolves (gray)

- Sheep run away when wolves are close and respawn after being killed by a wolf

- Fitness tracks total catches
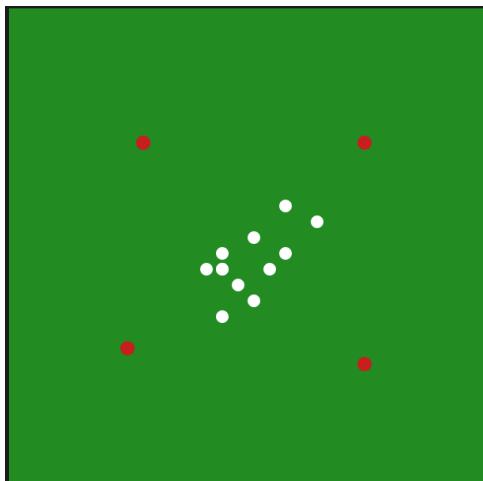
Figure 9.7: Wolves at early generations
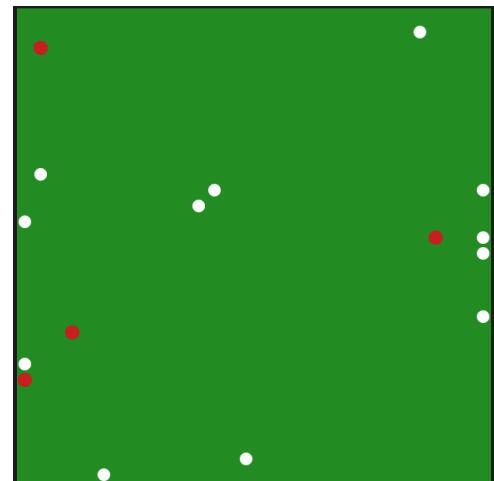
Figure 9.8: Wolves at later generations

9.5   Example 5: Rock-Paper-Scissors Ecosystem (`rps.txt`)

Description: A multi-species ecosystem where Rock beats Scissors, Scissors beats Paper, and Paper beats Rock. When a predator touches its prey, the prey *converts* to the predator's type.

Key Features:

- 45 individuals: 15 Rock (brown), 15 Paper (white), 15 Scissors (blue)

- Chase prey / flee predator behavior

- Conversion mechanic on contact

- 10 generations of 50 steps each, no mutation, just pure simulation
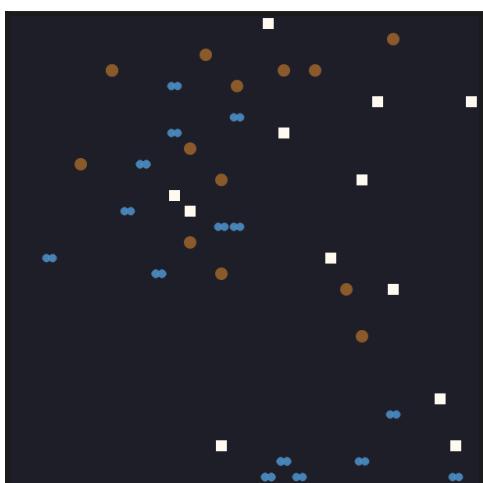
Figure 9.9: RPS at early steps
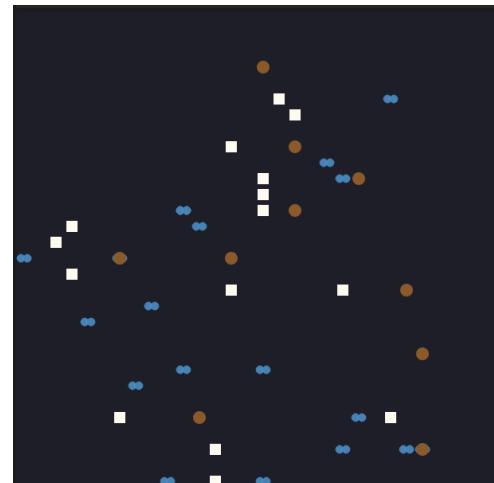
Figure 9.10: RPS at later steps

9.6 Example 6: Simple Tutorial (`example.txt`)

Description: A minimal example demonstrating basic Simulanka concepts. Agents have a "power" property that increases each step. Fitness simply returns the power value. Ideal for learning the language structure.

Key Features:

- Single species with one property

- Trivial routine (increment power)

- Direct fitness function (return self.power)

- Shows mutation mechanism

9.7 Summary of Included Examples

File	Type	Description
`maze.txt`	Optimization	Evolves solvable mazes with complex paths
`map.txt`	Generation	Creates terrain maps with smooth transitions
`flock.txt`	Behavior	Evolves bird flocking parameters
`wolves.txt`	Predator-Prey	Wolf pack hunting with sheep respawn
`rps.txt`	Ecosystem	Rock-Paper-Scissors with conversion
`example.txt`	Tutorial	Minimal example for learning

Run any example with: `cargo run -release - <filename>`

10 Conclusion

Simulanka is a language built specifically for evolutionary simulations. By using a block-based structure and easy-to-understand commands, it allows users to focus on the logic of their genetic algorithms and see the results instantly, without writing a lot of complicated code.

The main technical parts that make this project work are:

- An intuitive grammar where evolutionary steps (like mutation and fitness) are built directly into the language syntax.

- A type system that lets you write code quickly while still catching common errors before the simulation starts.

- Optimizations such as a grid-based environment that uses $O(1)$ lookups to keep the simulation running smoothly and multi-threading to share the computational load.

- Visualization using simple drawing commands that let you watch your species evolve frame by frame.

Part II
User Manual

11 Introduction

11.1 What is Simulanka?

Simulanka is a domain-specific programming language designed for simulating genetic algorithms represented on a matrix. With Simulanka, you can:

- Define species with genetic properties

- Create fitness functions that evaluate individuals

- Specify mutation and crossover rules

- Visualize the evolution of your species

- Run parallel simulation instances

The language handles all the complexities of genetic algorithms: selection, reproduction, and population management.

12 Installation & Setup

12.1 Prerequisites

Before installing Simulanka, ensure you have:

- **Rust toolchain** (version 1.70 or later) - Install from https://rustup.rs

- **Cargo** (included with Rust)

- A text editor (VS Code, etc.)

12.2 Building from Source

1. Clone or download the Simulanka source code

2. Open a terminal in the `simulanka` directory

3. Build the release version:

```
1    cd simulanka
2    cargo build --release
```

The executable will be created at `target/release/simulanka.exe` (Windows) or `target/release/simulanka` (Linux/macOS).

12.3 Running a Simulation

To run a Simulanka program:

```
cargo run --release -- myprogram.txt
```

Or with the compiled executable:

```
./target/release/simulanka myprogram.txt
```

Optional parameters:

```
simulanka <file> [generations] [instances]
# Example: simulanka maze.txt 100 20
```

13 Quick Start Tutorial

13.1 Your First Simulanka Program

Let's create a simple simulation where individuals try to maximize a "color" property.
Create a file called simple.txt:

```
ENVIRONMENT {
    width: 10
    height: 10
    steps: 1
}

SPECIES {
    ROUTINE do_nothing {
        // Creatures don't move in this example
    }

    Creature {
        color: random(0, 255),
        routine: do_nothing
    }
}

SPAWN {
    spawn Creature @ (5, 5);
}

FITNESS {
    return self.color;
}

MUTATE {
    mutation: {
        self.color = self.color + random(0, 21) - 10;
        if (self.color < 0) { self.color = 0; }
        if (self.color > 255) { self.color = 255; }
    }
```

```
32          }
33
34          VISUALIZE {
35              for cell in environment {
36                  draw_rect(100, 100, 400, 400, cell.color, 100,
                         100);
37              }
38          }
39
40          EVOLVE {
41              generations: 100,
42              instances: 10
43          }
```

Run it:

```
1           cargo run --release -- simple.txt
```

A window will open showing the evolution progress. Click **Start** to begin!

13.2   Understanding the Structure

Every Simulanka program consists of blocks:

Block	Purpose
ENVIRONMENT	Define world dimensions and simulation parameters
SPECIES	Define species types and their properties
SPAWN	Create initial individuals
FITNESS	Calculate fitness scores
MUTATE	Define mutation and crossover rules
VISUALIZE	Draw the simulation state
EVOLVE	Configure evolution parameters

14   Language Reference

14.1   Environment Block

The ENVIRONMENT block configures the simulation world:

```
1           ENVIRONMENT {
2               width: 100      // Grid width (default: 50)
3               height: 100     // Grid height (default: 50)
4               steps: 10       // Steps per generation (default:
                    10)
5           }
```

Note: The visualization window is always 600x600 pixels. Plan your drawing coordinates accordingly.

14.2 Species Block

Define species with properties and behavior routines:

```
1    SPECIES {
2        // First, define routines
3        ROUTINE move_random {
4            dx = random(0, 3) - 1;   // -1, 0, or 1
5            dy = random(0, 3) - 1;
6            self.x = self.x + dx;
7            self.y = self.y + dy;
8        }
9
10       // Then define species
11       Agent {
12           speed: 5,
13           health: 100,
14           genes: [random(0, 100), random(0, 100)],
15           routine: move_random
16       }
17   }
```

14.2.1 Built-in Properties

Every individual automatically has these properties:

Property	Type	Description
x	Int	X-coordinate on the grid
y	Int	Y-coordinate on the grid
species	String	Name of the species
fitness	Int	Fitness score (set after FITNESS block runs)

14.2.2 Using Lists for Genes

Lists are essential for complex genetic encodings:

```
1    SPECIES {
2        Creature {
3            // Create a list of 10 random genes
4            genes: [
5            random(0, 100), random(0, 100), random(0, 100),
6            random(0, 100), random(0, 100), random(0, 100),
7            random(0, 100), random(0, 100), random(0, 100),
8            random(0, 100)
9            ],
10           routine: behavior
11       }
12   }
```

14.3   Spawn Block

The SPAWN block runs once at the start to create individuals:

```
1      SPAWN {
2          // Spawn a single creature at the center
3          spawn Agent @ (50, 50);
4
5          // Spawn a grid of creatures
6          i = 0;
7          while (i < 10) {
8              j = 0;
9              while (j < 10) {
10                 spawn Cell @ (i, j);
11                 j = j + 1;
12             }
13             i = i + 1;
14         }
15     }
```

Warning: The spawn position (x, y) sets the self.x and self.y properties. If you define x or y in your species properties, they will be overwritten by the spawn position.

14.4   Fitness Block

The FITNESS block calculates a score for each individual:

```
1      FITNESS {
2          // Simple fitness: distance from origin
3          dist = self.x + self.y;
4          return dist;
5      }
```

```
1      FITNESS {
2          // Complex fitness: reward reaching a goal
3          if (self.x == 19 && self.y == 19) {
4              return 10000;
5          }
6          // Penalty for being far from goal
7          dx = 19 - self.x;
8          dy = 19 - self.y;
9          return 100 - dx - dy;
10     }
```

Tip: Higher fitness scores are better. The algorithm selects the top 50% of instances by their best individual's fitness.

14.5   Mutate Block

The MUTATE block defines genetic operators:

```
1      MUTATE {
2          // Crossover: combine genes from two parents
```

```
3            crossover: {
4                i = 0;
5                while (i < len(parent1.genes)) {
6                    if (random(0, 2) == 0) {
7                        child.genes[i] = parent1.genes[i];
8                    } else {
9                        child.genes[i] = parent2.genes[i];
10                   }
11                   i = i + 1;
12               }
13           }
14
15           // Mutation: randomly modify genes
16           mutation: {
17               if (random(0, 100) < 10) {  // 10% chance
18                   idx = random(0, len(self.genes));
19                   self.genes[idx] = random(0, 101);
20               }
21           }
22       }
```

14.5.1   Available Variables in MUTATE

Rule	Variables	Description
crossover	parent1, parent2, child	First parent, second parent, and the new offspring
mutation	self	The individual to mutate

Tip: In each generation, half of the new population is formed by your crossover logic (mixing traits from two winners), while the other half are clones. The mutation rules are then applied to the entire population to introduce random diversity. If you omit the crossover body, the evolution will rely entirely on mutated clones.

14.6   Visualize Block

The VISUALIZE block draws the simulation state:

```
1        VISUALIZE {
2            cell_size = 30;
3
4            for cell in environment {
5                // Draw each cell as a colored rectangle
6                vx = cell.x * cell_size;
7                vy = cell.y * cell_size;
8
9                // Color based on fitness
10               r = cell.fitness % 256;
11               g = 100;
12               b = 150;
13
```

```
14              draw_rect(vx, vy, cell_size - 1, cell_size - 1,
                    r, g, b);
15          }
16      }
```

14.6.1 Drawing Functions

Function	Description
draw_rect(x, y, w, h, r, g, b)	Draw a filled rectangle at (x,y) with width w, height h, and RGB color
draw_line(x1, y1, x2, y2, r, g, b, thickness)	Draw a line from (x1,y1) to (x2,y2) with specified color and thickness
draw_circle(x, y, radius, r, g, b)	Draw a filled circle at (x,y) with given radius and RGB color

Note: The visualization canvas is 600x600 pixels. RGB parameters and thickness are optional (default to white and 1.0).

14.7 Evolve Block

Configure the evolutionary algorithm:

```
1   EVOLVE {
2       generations: 1000,   // Number of generations to run
3       instances: 30        // Parallel simulation
            instances
4   }
```

Tip: More instances provide better exploration of the solution space but require more computation. Start with 10-20 instances and increase if needed.

15 Control Flow

15.1 Conditional Statements

```
1   if (condition) {
2       // executed if true
3   }
4
5   if (condition) {
6       // executed if true
7   } else {
8       // executed if false
9   }
10
11  if (x > 10) {
12      // ...
13  } else if (x > 5) {
```

```
14            // ...
15        } else {
16            // ...
17        }
```

15.2   While Loops

```
1        i = 0;
2        while (i < 10) {
3            print(i);
4            i = i + 1;
5        }
```

15.3   For Loops

Iterate over all individuals in the environment:

```
1        for cell in environment {
2            // 'cell' is bound to each individual
3            print(cell.x, cell.y);
4        }
```

Warning: `for` loops only support iterating over `environment`. To iterate over a list, use a `while` loop with an index variable.

16   Built-in Functions

16.1   Random Numbers

```
1        // random(min, max) - returns integer in [min, max]
2        x = random(0, 100);    // 0 to 99
3        y = random(-5, 6);     // -5 to 5
4        coin = random(0, 2);   // 0 or 1
```

16.2   List Operations

```
1        mylist = [1, 2, 3];
2
3        // Get length
4        n = len(mylist);  // n = 3
5
6        // Access by index
7        first = mylist[0];  // first = 1
8
9        // Modify by index
10       mylist[0] = 10;
11
```

```
12          // Add to end
13          push(mylist, 4);   // mylist = [10, 2, 3, 4]
14
15          // Remove from end
16          last = pop(mylist);   // last = 4, mylist = [10, 2, 3]
```

16.3   Spatial Queries

```
1          // Get individual at specific coordinates
2          neighbor = get_at(self.x + 1, self.y);
3
4          if (neighbor != 0) {
5              // There is someone at that position
6              print("Neighbor health:", neighbor.health);
7          }
```

```
1          // Alternative: use environment indexing
2          neighbor = environment[self.x + 1][self.y];
```

Note: Environment indexing uses wrap-around coordinates. If you query outside the grid boundaries, it wraps to the other side. `get_at` does not wrap.

16.4   Distance Calculation

```
1          // Calculate the distance between two objects
2          d = dist(self, neighbor);
3
4          if (d < 5) {
5              print("Too close!");
6          }
```

16.5   Print Function

```
1          print("Hello");
2          print("x =", self.x, "y =", self.y);
3          print("Fitness:", self.fitness);
```

Tip: You can use print() for debugging in any block that contains logic (like ROUTINE, SPAWN, or FITNESS), but it is not allowed in purely configuration-based blocks like ENVIRONMENT or EVOLVE which only accept specific settings.

17 Operators

17.1 Arithmetic Operators

Operator	Description	Example
+	Addition	5 + 3 = 8
-	Subtraction	5 - 3 = 2
*	Multiplication	5 * 3 = 15
/	Integer Division	7 / 3 = 2
%	Modulo (remainder)	7 % 3 = 1

17.2 Comparison Operators

Operator	Description	Example
==	Equal to	x == 5
!=	Not equal to	x != 0
>	Greater than	x > 10
<	Less than	x < 10
>=	Greater than or equal	x >= 10
<=	Less than or equal	x <= 10

17.3 Logical Operators

Operator	Description	Example
&&	Logical AND	x > 0 && x < 10
||	Logical OR	x == 0 || x == 10

18 Example Showcase

This section showcases two representative examples to demonstrate Simulanka's capabilities.

18.1 Maze Evolution (`maze.txt`)

Evolves a 21×21 maze with a solvable path from (0,0) to (20,20). The fitness function uses BFS pathfinding to reward longer, more complex paths.
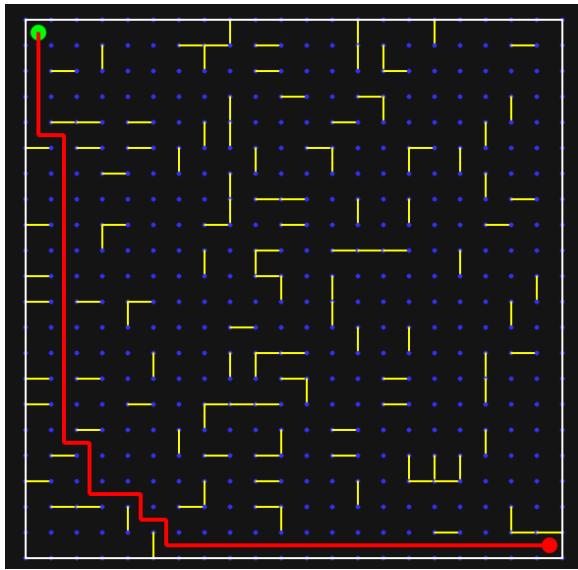
Figure 18.1: Maze at early generations
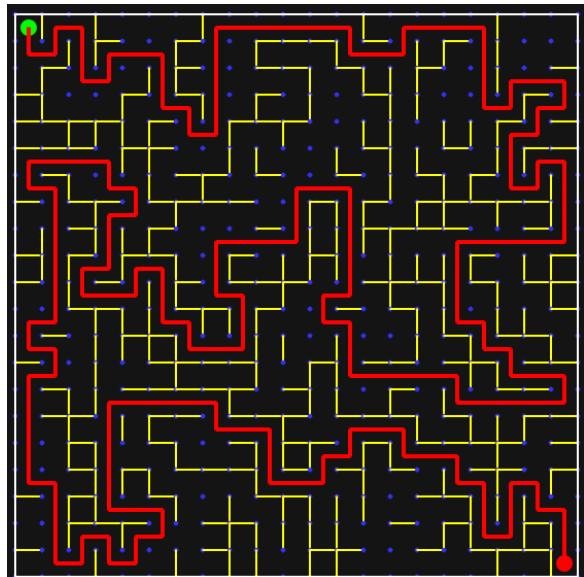
Figure 18.2: Maze at late generations

Run with: `cargo run -release - maze.txt`

18.2 Flocking Simulation (`flock.txt`)

Evolves bird flocking parameters (cohesion, separation, alignment) using the Boids algorithm. Fitness rewards tight grouping without collisions.
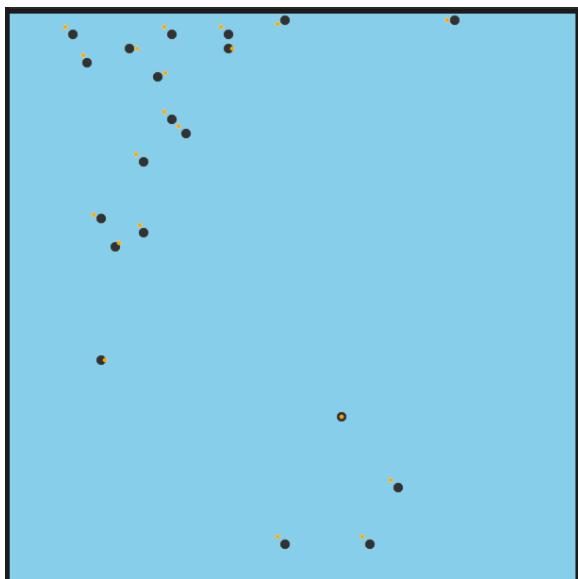
Figure 18.3: Flock at early generations
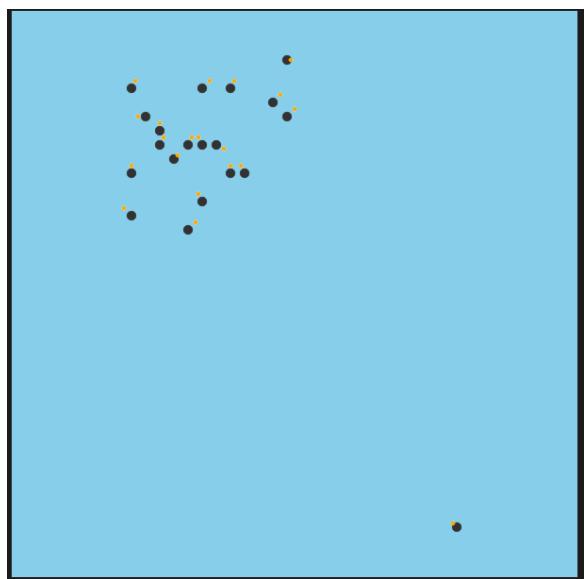
Figure 18.4: Flock at late generations

Run with: `cargo run -release - flock.txt`

More examples: See Part I, Section "Complete Examples" for detailed documentation of all included simulations (maze, map, flock, wolves, rps, example).

19 GUI Controls

When running a simulation, a graphical window appears with these controls:

Control	Description
Start/Stop	Begin or pause the evolution
Next	Run a single generation
Reset	Restart the simulation from scratch
<-Prev/Next->	Navigate through history
View Gen slider	Jump to a specific generation in history

Note: To conserve memory, the GUI stores a maximum of 100 generations of history. As new generations are computed, the oldest entries are removed from the timeline.

The display shows:

- **Avg Fitness**: Average fitness across all instances

- **Gen Best**: Best fitness in the current generation

- **Global Best**: Best fitness ever achieved

20 Troubleshooting

20.1 Common Errors

20.1.1 Variable 'x' is not defined!

Cause: You're using a variable before assigning it.

```
// Wrong
print(x);  // Error: x not defined

// Correct
x = 5;
print(x);
```

20.1.2 Species property 'speed' is not defined!

Cause: You're accessing a property that doesn't exist in the species definition.

```
// In your SPECIES block, make sure to define 'speed':
Agent {
    speed: 5,  // Add this line
    routine: move
}
```

20.1.3 Species 'Agent' calls undefined routine 'move'

Cause: The routine specified doesn't exist.

```
1         SPECIES {
2             // Define the routine BEFORE the species that uses
               it
3             ROUTINE move {
4                 self.x = self.x + 1;
5             }
6
7             Agent {
8                 routine: move   // Now this works
9             }
10        }
```

20.1.4 Expected RBrace, found Identifier

Cause: Missing closing brace or semicolon.

```
1         // Wrong - missing semicolon
2         x = 5
3         y = 10;
4
5         // Correct
6         x = 5;
7         y = 10;
```

20.1.5 Error: No instances defined in EVOLVE block

Cause: Missing or zero `instances` value.

```
1         EVOLVE {
2             generations: 100,
3             instances: 10   // Must be at least 1
4         }
```

20.2 Performance and Parallelism

The Simulanka interpreter is built for speed and utilizes modern hardware:

1. **Parallel Execution**: Simulation instances are automatically distributed across all CPU cores using the `rayon` library. This means that a simulation with 50 instances will run significantly faster on a multi-core processor compared to a sequential execution.

2. **Thread-Safe Core**: The runtime uses `Arc<RwLock<T>>` primitives, allowing for safe concurrent access to individual environments during the simulation phase.

3. **Release Mode**: Always run with `cargo run -release` to enable full optimizations.

20.3   Debugging and Error Reporting

The interpreter provides detailed feedback to help you fix issues:

1. **Row-Level Reporting**: Every error message includes the line number in your source `.txt` file where the issue occurred.

2. **Semantic Analysis**: Before execution, Simulanka checks for variables out of scope, undefined functions, and syntax errors, reporting exactly where they are.

3. **Runtime Console**: Use `print()` statements to debug complex logic; the output will appear in your system terminal.

20.4   Performance Tips

1. **Reduce grid size**: Smaller grids run faster

2. **Fewer individuals**: Each individual is evaluated every step

3. **Simpler fitness functions**: Avoid nested loops when possible

4. **Use release mode**: Always run with `cargo run -release`

A Language Keywords & Tokens

Keyword / Token	Context	Description
`ENVIRONMENT`	Top-level	Defines world dimensions and step count
`SPECIES`	Top-level	Defines species properties and routines
`SPAWN`	Top-level	Defines initial population (Generation 0)
`FITNESS`	Top-level	Defines the fitness scoring logic
`MUTATE`	Top-level	Defines crossover and mutation rules
`EVOLVE`	Top-level	Configures simulation parameters (instances, generations)
`VISUALIZE`	Top-level	Defines how to draw the simulation
`ROUTINE`	In SPECIES	Defines a reusable behavior function
`if / else`	In blocks	Conditional branching logic
`while`	In blocks	Loops while condition is true
`for`	In blocks	Iterates over `environment` (all individuals)
`in`	In for	Syntax marker for loops
`return`	In blocks	Returns a value from FITNESS
`print`	In blocks	Outputs text to the console (debugging)
`spawn`	In SPAWN	Creates a new individual instance
`@`	In spawn	Position marker (e.g., `spawn Agent @ (x,y)`)
`true / false`	Anywhere	Boolean literals
0-9 (Digits)	Anywhere	Integer literals
`" "` (String)	Anywhere	String literals
`A-Z, a-z, _`	Anywhere	Identifiers (variable or function names)

B Built-in Functions Reference

Function	Arguments	Returns
`random(min, max)`	Int, Int	Random Int in [min, max]
`dist(o1, o2)`	Object, Object	Int (Euclidean distance)
`len(list)`	List	Int (length of list)
`push(list, value)`	List, Any	Void (appends to list)
`pop(list)`	List	Last element (removes it)
`get_at(x, y)`	Int, Int	Individual at (x,y) or 0
`draw_rect(x, y, w, h, r, g, b)`	7 args	Void (draws rectangle)
`draw_line(x1, y1, x2, y2, r, g, b, th)`	8 args	Void (draws line)
`draw_circle(x, y, rad, r, g, b)`	6 args	Void (draws circle)