FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Tomáš Boďa

AgentLang - Programming Language for Agent-based Modeling

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis: Tomáš Petříček
Study programme: Computer Science

Prague 2024

I would like to thank my supervisor Mgr. Tomáš Petříček, Ph.D., whose guidance, support, patience and expertise have been invaluable throughout this journey, shaped this thesis and contributed significantly to its completion. I would also like to give special thank you to my family, who has fully supported me during my studies.

Title: AgentLang - Programming Language for Agent-based Modeling

Author: Tomáš Boďa

Department: Department of Distributed and Dependable Systems

Supervisor: Tomáš Petříček, Department of Distributed and Dependable Systems

Abstract: With the increasing popularity of the agent-based simulation technique in various scientific fields, there is a demand for an all-in-one framework for modeling agent-based simulations. Although there are numerous agent-based tools available, these in most cases feature complex syntax and language structures or are aimed to be used in specific domains only. In response this thesis presents a new approach to modeling agent-based simulations by developing a brand new agent-based framework - AgentLang. The framework features a programming language with a unified and simple syntax for defining agents and their properties. Moreover, it provides a web-based interface with a spreadsheet module for manipulating agents and their values using the familiar spreadsheet format as well as a visualisation module for rendering the simulation in real-time. These three features of the AgentLang framework aim to introduce a new way to modeling agent-based simulations and attempt to make agent-based modeling more accessible to people of all scientific fields.

Keywords: agent-based modeling, simulation, programming language, interpreter

Název práce: AgentLang - Programovací jazyk pre agentovo orientované modelovanie

Autor: Tomáš Boďa

Katedra: Katedra Distribuovaných a Spolehlivých Systémů

Vedoucí bakalářské práce: Tomáš Petříček, Department of Distributed and Dependable Systems

Abstrakt: S rastúcou popularitou techniky agentovo orientovaného modelovania v rôznych vedeckých oblastiach vzniká dopyt po jednotnom nástroji na modelovanie simulácií. Aj keď je na trhu dostupné množstvo nástrojov na agentovo orientované modelovanie, vo väčšine prípadov sa vyznačujú zložitou jayzkovou syntaxou a štruktúrou alebo sú určené na použitie iba v špecifických oblastiach. Vzhľadom na tieto nevýhody táto práca poskytuje nový pohľad na agentovo orientované modelovanie tým, že vyvíja nový nástroj na modelovanie agentovo orientovaných simulácií - AgentLang. Tento nástroj poskytuje nový programovací jazyk s jednotnou, ucelenou a jednoduchou syntaxou na definovanie agentov a ich vlastností. Okrem toho poskytuje webovú aplikáciu s tabulkovým rozhraním na manipuláciu agentov a ich hodnôt. V neposlednom rade obsahuje vizualizačný modul na zobrazovanie simulácií v reálnom čase. Tieto tri vlastnosti nástroja AgentLang majú za cieľ poskytnúť nový spôsob modelovania agentovo orientovaných simulácií a sprístupniť agentovo orientované modelovanie používateľom zo všetkých vedeckých odvetví.

Klíčová slova: agentovo orientované modelovanie, simulácie, programovací jazyk, interpreter

Contents

Introduction

Simulations and predictions of future events play a significant role in many scientific fields nowadays. This is especially true in economics, stock markets or housing markets [1], where there is a need for accurate predictions of future developments. Moreover, simulations are also commonly used in epidemiology or sociology, where the evaluation of the dynamics of a sociological group cannot be tested on real-world entities due to ethical and practical reasons. As our understanding of complex systems deepens, the demand for sophisticated tools to model and analyse the dynamics of simulations increases. This is where agent-based modeling comes into play, offering a more straightforward, bottom-up approach for handling complex simulations.

Agent-based models are built on top of agents, which represent the fundamental units of the given system, such as people in an epidemic or birds in a flock. To build such models, there is a need for tools or languages that provide constructs for representing agents, their behaviour and decision-making logic. Such tools already exist and are widely used to model agent-based simulations [2], such as NetLogo, GAMA or AgentScript [3, 4, 5]. These tools are powerful and can handle large volumes of agents. However, with their performance comes the cost of complexity of usage and inaccessibility to people with little to none programming knowledge. They are often difficult to learn, since they are based on modern programming languages, restricting the use case to developers only. And since the demand for agent-based models can be seen in non-technical scientific fields such as epidemiology or sociology, there is a need for simpler tools and languages.

AgentLang aims to provide a new approach to modeling agent-based simulations. This is possible on the one hand due to the simplicity of the language itself, since it features straightforward syntax and only the necessary constructs for modeling a wide range of simulations. On the other hand, the language is integrated into a specialised web-based interface, which offers a code editor, a visualisation module and most importantly, a spreadsheet interface, which allows the user to edit agent properties and values manually during the runtime of the simulation. These above features of the AgentLang framework provide all the necessary tools to model agent-based simulations and get a hands-on experience with the language itself.

The structure of AgentLang is straightforward. We define the individual agents and for each agent, we define its behaviour using a set of properties.

```
agent snowflake 500 {
  const speed = random(10, 20);

  property x: random(0, width()) = x;
  property y: random(0, height()) = (y + speed) % height();
}
```

The above example generates a set of 500 snowflakes, each with a random falling speed and lets them fall on the ground by incrementing their vertical coordinates by their speed. The fact that the agent is defined as a sequence of properties recalculated in each step of the simulation makes the language conceptually simple and fits well with the spreadsheet interface discussed later in the thesis.

In conclusion, the main intention behind the creation of AgentLang is to make agent-based modeling available to all scientific domains, regardless of their technical background, allowing for a shallow learning curve and usability using the following benefits:

- simplistic language syntax designed for easy conversion to the spreadsheet representation
- spreadsheet interface for almost no-code modeling
- built-in visualisation module for better understanding of the simulation

The thesis first dives into the important theoretical concepts behind AgentLang, where it introduces the benefits and use cases of agent-based modeling as well as provides a brief look into the idea behind language interpreters and parsers. Then, we will look at and compare existing agent-based tools to define the main goals AgentLang aims to achieve. Furthermore, we will provide the AgentLang's language specification and dive deeper into the implementation of its interpreter. Then, we will briefly look at the web-based interface of the AgentLang framework and provide several simulation examples as well as compare them with other agent-based tools to see if the goals were achieved. Finally, we will reflect back on the project and depict its main limitations and talk about future work and improvements.

1 Theoretical Background

Before diving into the language specification, it's important to introduce and briefly explain the theoretical background behind the most important concepts used in AgentLang. The following chapters aim to provide an introduction to agent-based modeling, its use cases and real-world applications as well as an overview of language interpreters and parsers.

1.1 Agent-based Modeling

AgentLang is a domain-specific language designed exclusively for modeling agent-based simulations. Therefore it is necessary to understand the motivation behind the agent-based modeling technique and its increasing popularity in the recent years as well as to demonstrate its power and applications on several use cases and real-world examples.

1.1.1 Introduction

Agent-based modeling is a simulation technique used in many scientific fields to model and understand complex simulations. In an agent-based model, a system is modeled using a set of autonomous entities called agents. An agent represents a fundamental meaningful unit of a system, such as a person in an epidemic or a bird in a flock. Each agent individually assesses the current situation, both of itself as well as of other agents and makes decisions based on a set of defined rules.

It's important to note that agent-based modeling is more of a mindset than a technology. The main idea behind it is to describe a system from the perspective of its constituent units. If we can assess and understand the behaviour of a fundamental unit of the system, we can define the model or structure of this unit, generate a set of these units and let their behaviour and interaction among themselves determine the outcome of the simulation. The advantage of such models is their simplicity in terms of understanding and implementation. However, even a simple agent-based model can portray complex behavioral patterns and provide insights into the dynamics of the system that it emulates.

1.1.2 Benefits of Agent-based Models

Agent-based modeling offers numerous advantages when compared to other mathematical models and simulation techniques.

First and foremost, the approach of agent-based modeling allows us to uncover novel emergent phenomena within the system, such as complex emergent behavioral patterns in sociology or economy, phenomena that were previously unknown or considered improbable. When we establish and define the rules governing the behaviour of agents, the anticipated actions of individual agents seem straightforward, given the simplicity and predictability of these rules. However, upon executing an agent-based model involving numerous interacting agents, the system's behavior often proves to be surprisingly complex. Even slight changes to the agents' behaviour may have a profound impact on the overall outcome of the system, a phenomenon that can be seen for instance in sociology, where the collective behavior of a group can be significantly influenced by subtle changes in the behavior of an individual.

Furthermore, agent-based modeling is particularly well-suited for simulation domains involving individual behavioral entity types and their mutual interactions. This is especially true in scenarios requiring the analysis of human interaction, understanding principles behind bird flocking, or predicting traffic jam occurrences on certain city roads. These situations pose challenges for other mathematical models, which may not explicitly consider and model the fundamental units of the system, rather the system as a whole.

Last but not least, agent-based models are flexible and extensible. We can easily introduce new agents to the system or increase the quantity of existing agents. Modifying the model's behavior is as simple as remodelling the decision-making logic within one agent model, which may lead to a whole new behaviour of the whole system. This degree of flexibility and extensibility implies the usage of agent-based models in scenarios where the level of complexity of the simulation is either highly unpredictable or unknown in advance, making it easy to fine-tune the simulation without the need for making too many adjustments.

1.1.3 Use Cases

Agent-based modeling is used in numerous scientific areas and systems. Let's dive deeper into some of the major classifications of its usage, which are:

1. flows
2. markets
3. organizations

Flows

Flows are the first area of expertise suitable for closer analysis using the agent-based modeling technique [6]. To describe the idea behind flows, let's consider the evacuation of people. The behaviour of a crowd in a sudden or unexpected dangerous situation often leads to panic and chaos. These situation usually arise during mass events such as concerts, sporting events and demonstrations, where there is high density of people on a certain area. In case of fire or other disasters of such nature, people in panic are governed by short-term personal interests uncontrolled by social and cultural constraints.

To prevent uncontrolled behaviour and maximize the chances of successful evacuation, we need to design venues and their escape exits in the best way possible. This is a typical scenario suitable for agent-based modeling. *"In agent terms, collective panic behaviour is an emergent phenomenon that results from relatively complex individual behaviour patterns and interactions between individuals, such as mutual excitation of a primordial instinct, chain reaction or social facilitation"* [7]. Based on prior historical observations, statistics and sociological studies, we can fairly accurately define the behaviour of a person in such situations and observe the reaction of masses during evacuation.

Markets

Another typical usage of agent-based models is in economics, where the dynamics of the stock market or the housing market results from the behaviour of many interacting agents, leading to the aforementioned emergent phenomena that are best understood by using a bottom-up approach - agent-based modeling.

In a study published in 2018 [1], a consortium of three scientists and academics - Marco Pangallo, Jean-Pierre Nadal, Annick Vignes, studied the housing market, leveraging an agent-based model with the aim to analyse the relation between income segregation, income inequality and house prices. Through the specification of the buyer-seller dynamics and the price formation mechanisms, the study concluded two primary insights:

1. a more unequal income distribution lowers the prices globally, but implies stronger segregation
2. a spike of the demand in one part of the city increases the prices all over the city

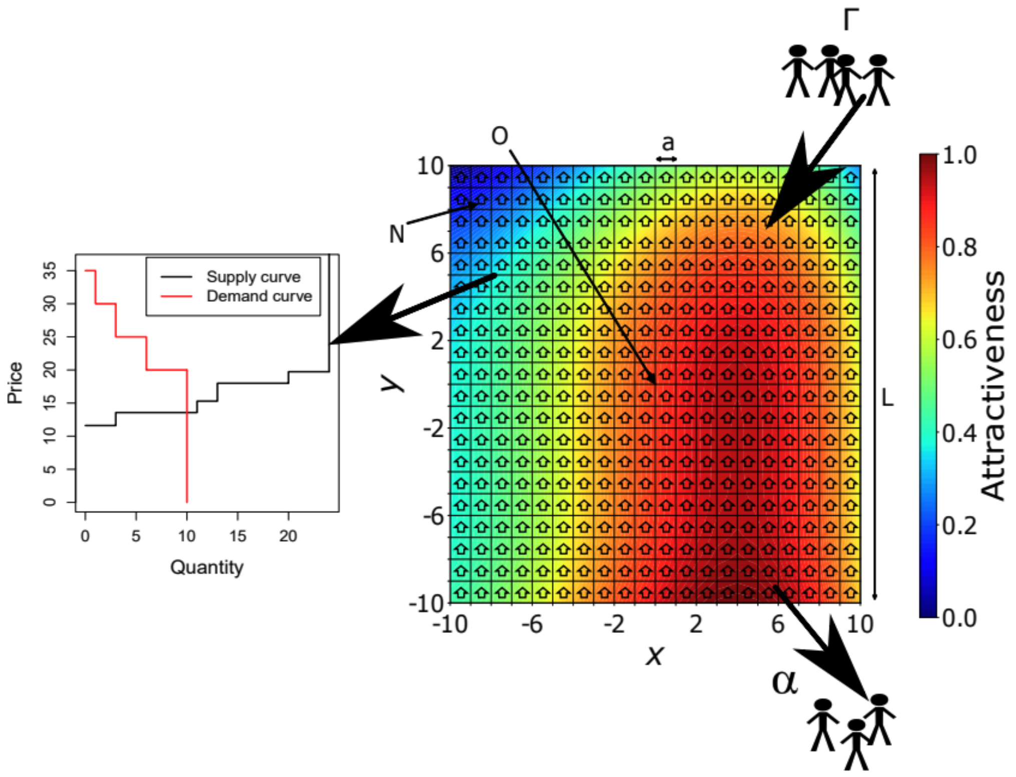


Figure 1.1 Schematic representation of the model by Marco Pangallo, Jean-Pierre Nadal and Annick Vignes

The model is a fairly simple grid on a Cartesian plane, where a specific point in the grid represents a specific location and space is defined by different levels of attractiveness, a variable subsuming exogenous intrinsic features and endogenous social characteristics. In this model, prospective buyers, drawn from external locations engage in the metropolitan housing market seeking accommodation, whereas households already living in the city decide to put their housing on sale based on a certain probability. Such households are referred to as the sellers. The buyers with heterogeneous incomes bid a certain amount of money proportional to their income in order to secure a property. The sellers, on the other hand determine the price they ask by employing an aspiration level heuristic. At each location in the grid, buyers and sellers are matched through a continuous double auction mechanism. Successful buyers take residence in the location where they searched and succeeded, whereas sellers leave the city. Finally, market prices at each location are derived as the mean value of successful transactions, thereby illuminating price dynamics.

All of the standalone entities, such as locations, buyers and sellers are represented using agents with defined set of rules and interaction logic, providing a model of the simulation capable of capturing emergent phenomena, such as the relation of income inequality and income segregation in the context of the housing market.

Organisations

Another promising area of application for agent-based modeling is organisational simulation [8]. Operational risk poses a constant threat to various sectors, such as financial institutions, stemming from potential issues such as inadequate business information systems, fraud or unforeseen catastrophes, all of which can lead to unexpected financial losses. In banking, operational risk, as defined by the Basle Committee on Banking [7], encompasses breakdowns in internal controls and corporate governance, potentially resulting in financial losses due to errors, fraud or failure to perform in a timely manner or compromises to the bank's interests caused by staff members exceeding their authority or conducting business in an unethical or risky manner. This risk is increasingly recognized as the most prominent and significant challenge faced by banks.

In contrast to market risks, operational risks predominantly arise internally within organisations, lacking a straightforward mathematical or statistical correlation between individual risk factors and the magnitude and frequency of operational losses. The lack of historical data on operational losses and their causes further complicates risk assessment, as large losses occur infrequently, leaving many banks without a sufficient time series of relevant data. This leads to uncertainty about which factors are important arising from the absence of a direct relationship between the risk factors.

Given all of these characteristics of operational risks within organisations, it is obviously difficult to quantify. Therefore, a bottom-up simulation to assess the probabilities of operational risks looks like a promising approach, since what is needed is a framework that includes the possibility of non-linear effects because of interactions among sub-units and to cascading events. Modeling the bank's agents, such as their workflow and risk factors that could potentially impact their activities and performance is the first step. Then, external factors, such as the bank's customers, markets and regulators are modeled to simulate the bank's environment. Using this approach, we can observe the emergent phenomena in form of cascading failures and analyse the operational risks of an organisation in a more straightforward way, without the knowledge of any direct relationships between failures and the bank's inner workings in advance. (see [7])

1.2 Interpreters

AgentLang is an interpreted language, which poses various advantages as well as disadvantages in terms of implementation and performance. This chapter provides a brief introduction into the inner workings of interpreters and language parsers and describe the main concepts used in their implementation.

1.2.1 Compilers vs. Interpreters

In order to better understand the main advantages and disadvantages of interpreters, let's introduce and compare the two primary approaches to developing programming languages - compilers and interpreters.

Compilers and interpreters are the fundamental building blocks of programming languages. They are responsible for reading the source code of the target language and executing its instructions. They both work in a slightly different way, each boasting its own set of strengths and weaknesses.

A compiler translates the source code of a program into machine instructions or an intermediate representation of the code before execution. The process of reading, analysing and transforming the source code involves several stages, including lexical analysis, semantic analysis, optimisations and code generation. Once compiled to machine code, the resulting executable can run independently of the original source code. Since compilers transform the source code to direct machine instructions, they tend to offer high performance, however, they are difficult to understand, implement and debug.

In contrast, an interpreter processes the source code line by line, translating and executing each instruction in real-time as the line is being read. However, interpreters also feature lexical and semantic analysis used to statically analyse the code before execution. For this, they usually produce and validate an intermediate representation of the source code known as the abstract syntax tree (AST), which is then evaluated instruction by instruction during runtime. For these reasons, interpreters usually tend to be slower than compilers, however, they provide higher flexibility and ease of debugging [9].

1.2.2 Workflow

Interpreters are usually implemented in a manner similar to the pipeline architectural style. During the process of interpretation, the source code is transformed through several processors before finally being executed by the interpreter's runtime.

Firstly, the source code is read by the lexer, which produces an array of tokens. A token is a single basic unit of the language, such as an identifier, a keyword or a numeric literal. The lexer has a set of defined, language-specific rules for producing the tokens, such as recognizing and classifying reserved language keywords or special characters. This step is called the lexical analysis.

The array of tokens is then passed to the parser, which based on a set of defined rules of the target language's grammar analyses sub-sequences of tokens, validates their integrity and produces a semantic representation of the program. This representation is also called the abstract syntax tree (AST), since it is a tree-like structure holding the semantics of the program. It can be understood as an intermediate code representation, which is validated and ready to be passed to and evaluated by the runtime module. This step is called the semantic analysis.

Finally, the abstract syntax tree is passed to the interpreter's runtime, which traverses the AST using a certain traversal algorithm, such as the depth-first search algorithm and evaluates the program instruction by instruction. Therefore, the instructions of the target language are evaluated and executed by the programming language in which the interpreter is implemented in.

1.2.3 Parsers

Although each part of the process of transforming the source code and generating the program's output is interesting and worth explaining in greater detail, the parser is one of the most interesting parts of an interpreter. The main responsibility of a parser is to evaluate the stream of tokens coming from the lexer, validate these tokens against the production rules of the target language's grammar and produce the semantic representation of the program - the abstract syntax tree.

Parser is usually implemented as a push-down automaton (PDA). This is because the syntax of a language can be defined by a set of production rules called the syntax grammar. This automaton accepts the language's grammar, therefore is able to validate the correctness of the input source code. The automaton reads a stream of tokens from the lexer. The evaluation starts at the initial state and based on the next token, it decides which state it goes to next. In case it comes across a token which cannot be pushed to a new state, since such state does not exist, it is redirected to the fallback state. This scenario is called a semantic error and in such case, the interpreter throws an exception. This aforementioned approach, however, is specific to top-down parsing only. There are various parsing techniques, each suitable for different use cases. (see [10])

Top-down vs. Bottom-up Parsing

There are two primary techniques in parsing language grammars: top-down parsing and bottom-up parsing.

In top-down parsing, the parsing starts from the root of the parse tree, which is the start symbol of the language grammar and proceeds towards the leaf nodes of the parse tree, attempting to match the current input token against the production rules of the grammar. This technique is relatively easy to implement, since it corresponds well to the way humans tend to think about parsing.

In contrast, bottom-up parsing starts from the input string and proceeds by identifying sequences of terminals and non-terminals in the input string that match the right-hand side of some production rule in the grammar. These sequences are then replaced by the corresponding non-terminal symbol. The parser iteratively attempts to match the terminals and newly created non-terminals against the production rules of the grammar, eventually resulting in the entire input string being replaced by the start symbol of the grammar. The advantage of bottom-up parsing is that it can handle a wider class of language grammars than top-down parsing, including left-recursive grammars and is in many cases more efficient than top-down parsing. However, it is much more difficult to implement and understand conceptually. (see [10] paragraph *Two Strategies*)

Recursive Descent Parsing

One of the more popular top-down parsing techniques is recursive descent parsing, also used and implemented in the AgentLang's interpreter. In recursive descent parsing, each non-terminal in the grammar corresponds to a function in the parser module. These functions call each other to parse different parts of the input source code. When parsing a program from the beginning, the parser calls the `parseProgram` function, which further calls `parseVariableDeclaration` and `parseObjectDeclaration` functions, based on the type of the token in the stream. This goes all the way to the `parsePrimaryExpression` function, which parses low-level expressions, such as numeric or string literals. (see [10] paragraph *Recursive Descent Parsing*)

The idea behind the implementation of such push-down automaton is that the stack of the automaton is implemented using the call-stack of the language in which the interpreter is implemented. When the function calls nest into each other, they are stored onto the call-stack, preserving their state. We are pushing non-terminals onto the stack, processing them, producing parts of the AST and in case of correct input eventually ending up back in the initial state with an empty stack, having the AST of the program produced and validated.

The technique of recursive descent parsing is used in the implementation of the AgentLang's parser. This is due to the simplicity of the language's syntax grammar and is a good starting point for implementing a simple parser such as that of AgentLang.

2 Analysis

The previous chapter covered the most important theoretical concepts and terms that will be used throughout the thesis and that are the main building blocks of the AgentLang framework. This chapter explores the motivation behind the creation of AgentLang, analysis and comparison with existing agent-based tools and finally outlines the main goals that this thesis strives to achieve.

2.1 Motivation

First and foremost, it is useful to distinguish two kinds of programming languages. On the one hand, general-purpose languages, renowned by their versatility and widespread adoption among developers offer a broad set of concepts, therefore developing a general-purpose programming language introducing innovation in rather challenging. On the other hand, domain-specific languages serve as more of niche tools for tackling challenges with domain-specific tasks. As technology is on the rise these days, new domains emerge that need specific tools to handle the tasks they pose. One of such domains is agent-based modeling.

Agent-based modeling, although being a simulation technique with a long history, continues to reveal variety of new applications and use cases across numerous scientific fields and domains. There are many agent-based specific tools that allow for modeling agent-based simulations, each boasting its set of strengths and weaknesses. Despite their power and extensive feature set, these tools often feature complex syntax and language constructs, limiting their usage to technical scientific domains only. Moreover, many of these tools are tailored to specific domains only, such as NetLogo for social and natural sciences. Hence the idea of AgentLang emerged, aiming to offer a unified language and environment for agent-based simulations of any nature, accessible to all scientific fields, regardless of their technical proficiency.

2.2 Existing Agent-based Tools

Before outlining the primary goals of this thesis, it is important to mention some of the existing agent-based tools to highlight the main differences between existing agent-based frameworks and AgentLang.

2.2.1 NetLogo

Perhaps one of the most wide-spread agent-based modeling software is NetLogo [3]. It is a multi-agent programmable modeling environment with its own domain-specific language as well as a web-based modeling interface. NetLogo's primary simulation domains are social and natural sciences, for which ti provides a rich library of simulation models and examples. It was first introduced and released in 1999 by Uri Wilensky, a professor at the Northwestern University in Illinois, US and is inspired by the functional programming language Logo introduced in 1967 by the company BBN. Nowadays, it is used by hundreds of thousands of students, teachers and researches all among the world.

Below is a NetLogo simulation modeling the spreading of fire in a forest [11].

```
globals [
 initial-trees
 burned-trees
]

breed [fires fire]
breed [embers ember]

to setup
 clear-all
 set-default-shape turtles "square"
 ask patches with [(random-float 100) < density]
  [ set pcolor green ]
 ask patches with [pxcor = min-pxcor]
  [ ignite ]
 set initial-trees count patches with [pcolor = green]
 set burned-trees 0
 reset-ticks
end

to go
 if not any? turtles
  [ stop ]
 ask fires
  [ ask neighbors4 with [pcolor = green]
    [ ignite ]
   set breed embers ]
 fade-embers
 tick
end

to ignite
 sprout-fires 1
  [ set color red ]
 set pcolor black
 set burned-trees burned-trees + 1
end

to fade-embers
 ask embers
  [ set color color - 0.3
   if color < red - 3.5
    [ set pcolor color
     die ] ]
end
```

NetLogo is a powerful tool capable of handling thousands of agents and their complex behavioural patterns. However, the expressivity of the language results in a steep learning curve. The language is notably specific, which may be unfamiliar for developers and data analysts, not to mention scientists of non-technical scientific fields.

2.2.2 GAMA

Another popular modeling and simulation framework is GAMA [4]. It is designed to handle spatially explicit agent-based simulations, such as urban mobility, climate change adaptation, epidemiology, disaster evacuation strategies or urban planning. Its advantages include the generality of the framework and openness to user-defined plugins as well as the possibility to use GAMA externally in custom software or in different programming languages. One of the most important strengths of GAMA is the availability to non-technical scientists. It is quite easy and straightforward to create and run the first simulation in the matter of minutes. However, GAMA is a large and rich environment with a steep learning curve if the aim is to use it regularly and in non-trivial ways. Moreover, as well as in NetLogo, its language is specific in design and structure, making it difficult to learn, even for people of technical scientific domains.

Below is a GAMA simulation model representing the behaviour of people (resting, working) [12].

```
species people skills: [moving] {

  reflex time_to_work when: current_date.hour =
    start_work and objective = "resting" {
    objective <- "working" ;
     the_target <- any_location_in (working_place);
  }

  reflex time_to_go_home when: current_date.hour =
    end_work and objective = "working" {
    objective <- "resting" ;
     the_target <- any_location_in (living_place);
  }
}
```

2.2.3 AgentScript

Finally, another agent-based framework utilising an existing programming language is called AgentScript [5]. AgentScript is more of a library rather than a framework. It provides a set of JavaScript classes and methods used to create and run agent-based models in the browser. AgentScript is heavily inspired by NetLogo, thus its main building blocks consist of three actors: turtles, patches and links. The user defines the behaviour and interaction logic of these three actors and the library takes care of the rest, such as real-time simulation visualisation or agent manipulation. Although the whole concept of AgentScript is familiar, easily understandable and usable by JavaScript developers, it poses challenges for people with limited programming and JavaScript skills. JavaScript, as many other programming languages has its structure and rules, which the user needs to know apart from the AgentScript semantics itself.

Below is an AgentScript simulation modeling bird flocking using the Boids algorithm [13].

```
export default class FlockModel extends Model {
  population = 100
  vision = 3
  speed = 0.25
  maxTurn = 3.0
  minSep = 0.75

  constructor(worldDptions) {
    super(worldDptions)
  }

  setup() {
    this.turtles.setDefault('speed', this.speed)
    this.patches.cacheRect(this.vision)
    util.repeat(this.population, () => {
      this.patches.oneOf().sprout()
    })
  }

  step() {
    this.turtles.ask(t => {
      this.flock(t)
    })
  }

  flock(t) {
    const fm = this.turtles.inRadius(t, this.vision, false)
    if (fm.length !== 0) {
      const n = fm.minOneOf(f => f.distance(t))
      if (t.distance(n) < this.minSep) {
        this.separate(t, n)
      } else {
        this.align(t, fm)
        this.cohere(t, fm)
      }
    }
    t.forward(t.speed)
  }

  separate(t, n) {
    const h = n.towards(t)
    this.turnTowards(t, h)
  }

  align(t, fm) {
    this.turnTowards(t,
      this.averageHeading(fm))
  }

  cohere(t, fm) {
    this.turnTowards(t,
      this.averageHeadingTowards(t, fm))
  }

  turnTowards(t, heading) {
    let turn = t.subtractHeading(heading)
    turn = util.clamp(turn,
     -this.maxTurn, this.maxTurn)
    t.rotate(turn)
  }

  averageHeading(fm) {
    // implementation
  }

  averageHeadingTowards(t, fm) {
    // implementation
  }

  flockVectorSize() {
    // implementation
  }
}
```

2.3 Comparison

Below is the comparison of the aforementioned agent-based tools and AgentLang. The criteria upon which the agent-based frameworks are analysed are the following:

- **general-purpose** - whether the framework can be used generally to model any type of simulation (sociological, economical, organisational, ...)
- **ease of learning** - whether the framework is easily understandable and usable by newcomers
- **language simplicity** - whether the framework provides a language that is easy to learn by people with little to none prior programming experience
- **web-based** - whether the framework provides a web-based modeling interface for use without any further manual installation

The below table represents the comparison of the NetLogo, AgentScript, GAMA and AgentLang frameworks using the above criteria.

	NetLogo	**AgentScript**	**GAMA**	**AgentLang**
General-purpose	Yes	Yes	Yes	Yes
Ease of learning	No	No	Yes	Yes
Language simplicity	No	No	No	Yes
Web-based	Yes	Yes	No	Yes

2.4 Goals

By analysing the above existing agent-based tools, we come to the following conclusions. Firstly, each of these tools is either directly based on an existing general-purpose programming language, such as AgentScript or provides its own domain-specific language, which is distinct in terms of syntax and structure, such as NetLogo or GAMA. The former requires the understanding of the underlying programming language, whereas the latter requires to learn a brand new language. Moreover, these tools are often complex and offer a wide range of features, which at one hand is useful for agent-based specialised scientists, but on the other hand discourages the rest by their steep learning curves. In response to these characteristics, AgentLang aims to fulfil the following goals:

1. provide a unified language with simple and straightforward syntax and structure
2. offer only the necessary, essential core library of features
3. allow for an alternative way of simulation modeling using a familiar approach in form of spreadsheet representation

3 Language Specification

The following chapters focus on the detailed language specification of AgentLang, describing its structure, syntax, data types as well as its standard library and core functionality.

3.1 Introduction

AgentLang is an interpreted programming language and its interpreter is written in TypeScript. Its syntax is simple and straightforward, yet it may resemble some modern general-purpose programming languages such as Python, establishing a nice balance between the ease of use for non-technical scientists as well as familiarity for developers.

The structure of AgentLang is natural in terms of the way humans tend to think about agent-based models. The user defines one or multiple agents and for each agent a set of their properties. An agent can be viewed as a class in an object-oriented programming language and a property can be understood as a member variable of this class. Apart from the declarations of agents and their properties, AgentLang supports the declarations of global variables, which are constant values that can be reused among all agents. User-defined functions with parameters are however not supported in order to keep the language simple and allow for easy conversion to the spreadsheet interface.

Each agent property has a strictly inline value defined by an expression. The language does not allow for code blocks with multi-statement definitions, except for the agent body.

Since AgentLang is an interpreted language, the program is evaluated in an incremental manner, agent by agent, property by property. Although this would imply that a property cannot be accessed unless defined earlier, it is not so, since the interpreter has a built-in topological property sorting mechanism, which determines the order in which properties are evaluated during runtime. However, more about this concept will be discussed in later chapters. The output of the interpreter is an array of agents and the current values of their properties. This gives the developer the flexibility to analyse and use the output in any way they need.

3.2 Syntax Grammar

The AgentLang's syntax grammar A.1 represents the production rules of the formal syntax grammar of the language. Terminal symbols are enclosed in double quotes and non-terminal symbols are represented by identifiers written in snake case.

3.3 Declarations

Declarations are the top-level constructs of the AgentLang language. They are used to declare agents, properties and global variables.

3.3.1 Agents

Agent is the main building block of a simulation. It represents an agent model and its properties and is used for generating a set of agents for the simulation. Agents are always declared in the top-level program scope and they cannot be nested. Defining multiple agent models is also supported.

To declare an agent, we use the following production rule:

```
agent_declaration:
    | "agent" identifier agent_count "{" "}"
    | "agent" identifier agent_count "{" agent_body "}"
```

Below is an example of an `agent` declaration.

```
agent car 20 {
    ...
}
```

AgentLang also supports multiple agent model declarations. To define multiple agent models, declare them one after another in the program scope.

```
agent car 20 {
    ...
}

agent pedestrian 60 {
    ...
}
```

Note that the `agent_count` parameter does not have to be a numeric literal explicitly. We can define a global variable with a numeric value and use this variable's identifier as the `agent_count` parameter.

```
define car_count = 20;

agent car car_count {
    ...
}
```

More about global variables will be explained later in this chapter.

After defining agent models, AgentLang will generate the corresponding number of agents for each agent model and evaluate their properties at runtime.

3.3.2 Properties

Properties are essential in defining the behaviour of an agent model. They can be understood as variables with a value defined based on some inline expression. Agents can have any number of properties, either independent or dependent on each other. AgentLang supports two types of properties, which are `property` and `const`, each having its specific use case.

Const

Property of type `const` is a special kind of property, which holds a constant value during the entire runtime of the simulation. It is calculated only once as the agent is generated.

To declare a `const` property, use the following grammar production rule:

```
property_declaration:
  "const" identifier "=" expression ";"
```

Below is an example of a `const` declaration holding a numeric value.

```
const max_speed = 260;
```

Properties of type `const` are used in cases when we need to for instance generate random initial coordinates of agents, since they are calculated only once at the beginning of the simulation.

```
const x_spawn = random(100, 200);
const y_spawn = random(100, 200);

property x: x_spawn = x + 1;
property y: y_spawn = y + 1;
```

Property

Property of type `property` is the most commonly used type of property in AgentLang. It is recalculated in each step of the simulation for every agent, based on the most current values.

To declare a `property` property, use the following grammar production rule:

```
property_declaration:
  "property" identifier "=" expression ";"
```

Below is an example of a `property` declaration that holds a random numeric value between 5 and 10 in each step of the simulation.

```
property current_speed = random(5, 10);
```

However, what if we want to set the property to some initial value and use this property's value in its own declaration?

```
property x = x + 1;
```

In the above example, it is not possible. The `x` property is incremented by 1 in each step of the simulation, however, it is not set to any default value to start with. That is why AgentLang supports default property values.

To declare a `property` with a default value, we use the following production rule.

```
property_declaration:
  "property" identifier ":" expression "=" expression ";"
```

The expression after the semicolon is used to initialise the `property` to some default value in the first step of the simulation. In each next step, the second expression is used to recalculate its value.

```
property x: 15 = x + 1;
property y: 5 = y * 2;
```

A better example would be the earlier example with the current speed. We want the agent to accelarate, so we increment its speed by one.

```
const initial_speed = 0;
property speed: initial_speed = speed + 1;
```

In this way, the speed is set to 0 in the first step and is incremented by 1 in each following step, producing values 1, 2, 3, 4 and so on.

However, this is not the only use case of default property values. Suppose the following exaggerated example, where we have two properties, where each depends on the other.

```
property a = b + 1;
property b = a + 2;
```

This example would throw an error, since there is a cycle in the property declarations. In order to fix this issue, we need to assign a default value to one of the properties, so that the interpreter knows which property will be evaluated first, so that the other property can calculate its value based on the first property.

```
property a = b + 1;
property b: 0 = a + 2;
```

The above example would work, since at least one of the properties is initialised with a default value. This topic, however, concerns the topological sorting mechanism implemented in the AgentLang's interpreter, which will be explained later in this thesis.

3.3.3 Global Variables

Apart from agent and property declarations, AgentLang supports the declaration of global variables which can be reused among all agent models as constant values. Global variables are always declared in the top-level program scope and the best practice is to declare them before all agent declarations. However, it is not necessary, since the AgentLang interpreter has a built-in mechanism for declaration reordering, so that the global variables are always evaluated before agent declarations.

To declare a global variable, we use the following production rule.

```
define_declaration:
  "define" identifier "=" expression ";"
```

Below is an example of usage of global variable declarations.

```
define person_count = 120;
define default_speed = 5;

agent person person_count {
  const speed = default_speed / 2;
}

agent car 10 {
  const speed = default_speed * 3;
}
```

Note that global variables cannot contain any identifiers or function calls in their definitions. They are plain constant values which can only hold numeric or boolean literals.

3.4 Data Types

There are five data types that AgentLang supports, which are numeric literals, boolean literals, AgentList instances, AgentObject instances and Null values.

3.4.1 Numeric Literal

Numeric literal is one of the two primitive data types in AgentLang. A numeric literal represents either an integer or a decimal number. Decimal numbers can have any number of decimal places, however, they are always rounded to up to eight decimal places in the simulation's output.

Numeric literals can be used in many ways, either as raw numeric values or in any numeric expression, such as binary or unary expressions or as parameters to function calls.

```
const integer_value = 3;
const decimal_value = 12.8;
const binary_expression = 6.5 * 2 / integer_value;
const random_value = random(10, 20);
```

3.4.2 Boolean Literal

Boolean literal is the second of the two primitive data types in AgentLang. It represents a binary value, which can either be `true` or `false`.

Boolean literals can be expressed either explicitly, using the `true` or `false` keywords, or they can be the result of some expression, such as the relational expression (more on relational expression later).

```
const is_active = false;
const is_first = index() == 0;

property temperature: 10 = temperature + random(-3, 3);
const is_cold = temperature <= 9;
```

3.4.3 AgentList

AgentList is a special data type representing an array of agent instances. This array cannot be defined explicitly, cannot be indexed and it only results from various built-in function calls.

The easiest way to retrieve an array of agent instances is to use the `agents` function, which returns the current array of agent instances of the specified type.

```
agent prey 10 {
  ...
}

agent predator 5 {
  property preys = agents(prey);
}
```

The `preys` property holds an array of agent instances of type `prey` with their most recent values.

There are numerous built-in functions used for retrieving or manipulating values of type AgentList, which will be described later in this chapter.

3.4.4 AgentObject

AgentObject is a special data type representing one specific agent instance and its properties. It can be used to retrieve the property values of an agent and use them in later calculations.

Similarly to AgentList, AgentObject can only be retrieved by using specific built-in function calls, such as `min`.

```
agent person 10 {
  property x = ...;
  property y = ...;

  property closest_person = min(agents(person) | p -> dist(p.x,
    p.y, x, y));
}
```

The above example uses the `min` function together with a set comprehension expression to retrieve an agent instance of type `person` which is closest to the current person. We can now use the `closest_person` property to retrieve the agent's property values.

```
agent person 10 {
  property x = ...;
  property y = ...;

  const is_married = prob(0.5);
  property closest_person = min(agents(person) | p -> dist(p.x,
    p.y, x, y));

  property closest_is_married = closest_person.is_married;
}
```

3.4.5 Null

Null is a special data type that represents an undefined or missing value. It is tightly bound to the AgentObject data type. Note the following example.

```
define range = 65;

agent person 10 {
  property x = ...;
  property y = ...;

  property people = agents(person);
  property close_people = filter(people | p -> dist(p.x, p.y, x
    , y) < range);
  property close_person = min(close_people | c -> dist(c.x, c.y
    , x, y));
}
```

The `close_person` property attempts to find an agent that is closest to the current agent, but is also in the visual range of 65. If there are no agents in this visual range, the `close_person` searches an empty array and cannot retrieve a specific agent instance. Therefore, it is assigned a Null value.

A problem with Null values, however, is that we cannot use it in other properties. More specifically, we cannot retrieve the agent's properties, since it does not hold any agent instance, rather a Null value. That is why AgentLang supports the `otherwise` operator, which tackles issues with Null values. The `otherwise` operator will be discussed later in this chapter.

3.5 Expressions

The following chapters introduce all expression types supported by AgentLang, from the most basic ones such as binary or relational expressions to more complex, language-specific expressions such as otherwise expressions or set comprehension expressions.

3.5.1 Binary Expression

Binary expression is the most basic expression in AgentLang. It consists of two numeric operands and one binary operator. The operator can be of type addition, subtraction, multiplication, division or modulo. These expressions can be arbitrarily nested and parenthesised.

```
const add_expr = 2 + 3;
const sub_expr = 6 - 2;
const mul_expr = 12 * 4;
const div_expr = 8 / 3;
const mod_expr = 16 % 6;

const complex_expr = 2 + 3 * 4 - 8 / 14;
const parenth_expr = (2 + 3) * 4 - (12 + 2);
```

Note that implicit conversions with other operand data types are not supported and result in a runtime error.

3.5.2 Unary Expression

Unary expressions consist of one numeric or boolean operand together with a unary operator. There are two unary operators, one for numeric unary expressions and the other for boolean unary expressions.

Numeric Unary Expression

Numeric unary expression is used to convert a positive number to a negative number using the minus ‘-‘ operator. The operand can either be a plain numeric literal or an identifier holding a numeric value.

```
const value = 12.6;
const neg_basic = -12.6;
const neg_ident = -value;
```

Boolean Unary Expression

Boolean unary expression is used to negate a boolean value using the emphasis ‘¡ operator. The operand can either be a plain boolean literal (‘true‘ or ‘false‘) or an identifier holding a boolean value.

```
const value = false;
const neg_basic = !false;
const neg_ident = !value;
```

3.5.3 Logical Expression

Logical expressions are expressions operating on boolean literals and always return a boolean value as the result. They are special types of binary expressions which use the `and` and `or` operators together with two boolean operands.

```
const bool_value = false;

const log_expr_and = true and bool_value;
const log_expr_or = true or bool_value;
```

3.5.4 Relational Expression

Relational expressions are special types of binary expressions that operate on either numeric or boolean operands and always return boolean results. They use relational operators, such as ==, !=, >, >=, < and <=.

The == and != operators can be used with either numbers and booleans, since they check for value equality. The rest of the relational operators operate on numeric values only.

```
const bool_1 = true;
const bool_2 = false;

const bool_expr_1 = bool_1 == bool_2;
const bool_expr_2 = bool_1 != bool_2;

const num_1 = 1.5;
const num_2 = 8.4;

const num_expr_1 = num_1 == num_2;
const num_expr_2 = num_1 != num_2;
const num_expr_3 = num_1 > num_2;
const num_expr_4 = num_1 >= num_2;
const num_expr_5 = num_1 < num_2;
const num_expr_6 = num_1 <= num_2;
```

Note that other types of operands are not supported and result in a runtime error.

3.5.5 Conditional Expression

Conditional expressions are used to control the flow of property evaluation. They decide between two alternatives based on some condition. The condition is always a boolean expression and the results can be of any type, based on the data type of the given property.

We use the following production rule for defining a conditional expression:

```
conditional_expression:
  | "if" expression "then" expression "else" expression
```

The `if` keyword marks the start of a conditional expression. It is followed by a boolean expression denoting the condition upon which the structure decides. Then comes the `then` keyword followed by an expression representing the value to be evaluated if the condition is met (is `true`). Finally comes the `else` keyword followed by an expression representing the value to be evaluated if the condition is not met (is `false`).

```
define max_speed = 10;

agent person 5 {
  property speed: 5 = if abs(speed) >= max_speed then speed
    else speed + random(-1, 1);
}
```

The above example controls the maximum value of `speed` using the `max_speed` global variable. If it overflows, it keeps the `max_speed` value, otherwise it is randomly incremented or decremented.

3.5.6 Otherwise Expression

The otherwise expression is a language-specific type of expression used to handle issues with Null values. It is a binary expression that uses the `otherwise` operator between the two operands.

The left-hand side of the `otherwise` expression consists of any expression containing a value of type AgentObject. The right-hand side consists of any expression that does not contain a value of type AgentObject. When the `otherwise` expression is being evaluated, AgentLang checks whether the value of type AgentObject on the left-hand side is Null. If not, it evaluates the left-hand side expression. On the other hand, if the value is Null, it instantly switches to the right-hand side of the expression and evaluates it.

Otherwise expressions serve as guards for Null values, for we can only tell if the value of type AgentObject is Null or not Null during runtime.

```
define range = 60;

agent person 120 {
  property x = x + x_move;
  property y = y + y_move;

  property people = agents(person);

  property in_proximity = filter(people | p -> dist(p.x, p.y, x
    , y) <= range);
  property closest = min(in_proximity | p -> dist(p.x, p.y, x,
    y));

  property x_move = (closest.x - x) / 10 otherwise 0;
  property y_move = (closest.y - y) / 10 otherwise 0;
}
```

The above example finds all people in some visual proximity to the current person and selects the closest person from the list. It then calculates the direction in which the current person should move in order to approach the closest person. However, we cannot be certain that we will find any people in the given proximity. If that's the case, the `in_promixity` property will be an empty array and the `closest` property will therefore result in a Null value. That is why we need to use the `otherwise` operator to ensure that if no such person is found, we will use values 0 for `x_move` and `y_move` properties.

3.5.7 Set Comprehension Expression

While being classified as expression, the set comprehension expression is rather a syntactical structure that an expression. It cannot be used on its own, only as a parameter to set comprehension-specific built-in functions. They are mainly used for traversing arrays of agent instances and manipulating them in some way. Use cases include filtering of agents, summing certain agent properties or finding a specific agent instance based on some condition.

There are several built-in functions that take set comprehension expression as their parameter, some of which are `filter`, `sum`, `min` and `max`.

To define a set comprehension expression, we use the following production rule:

```
set_comprehension_expression:
  | expression "|" identifier "->" expression
```

We start with an expression holding a value of type AgentList, followed by a divider `|`. Then, we declare the set comprehension parameter name, which is any identifier we choose, such as `item` followed again by an arrow `->`. This parameter is used to access each agent instance in the array, one by one. The final part of the set comprehension expression is an expression representing a condition based on which to manipulate the agent instances.

```
define range = 60;

agent person 120 {
  property x = x + random(-1, 1);
  property y = y + random(-1, 1);

  property people = agents(person);
  property in_proximity = filter(people | p -> dist(p.x, p.y, x
    , y) <= range);
}
```

The `filter` function takes a set comprehension expression as a parameter. We use the `p` parameter to access each individual agent instance and their properties. Finally, the right-hand side of the set comprehension expression is used for filtering the agent array based on the proximity of each agent instance to the current agent. The result of `in_proximity` property is a filtered array of agents of type `person`.

3.6 Core Library

The AgentLang's core library consists of several built-in functions necessary for agent manipulation as well as mathematical calculations. Below is the complete list of built-in functions and their usage.

3.6.1 Mathematical Functions

`sqrt(number): number` - calculates the square root of a numeric value.

`abs(number): number` - calculates the absolute value of a numeric value.

`floor(number): number` - calculates the floor value of a decimal numeric value.

`ceil(number): number` - calculates the ceil value of a decimal numeric value.

`round(number): number` - calculates the rounded value of a decimal numeric value.

`sin(number): number` - calculates the sine value of a numeric value.

`cos(number): number` - calculates the cosine value of a numeric value.

`tan(number): number` - calculates the tangent value of a numeric value.

`atan(number): number` - calculates the arc tangent value of a numeric value.

`pi(): number` - returns the value of Pi (3.14...).

3.6.2 Agent Manipulation Functions

`filter(SetComprehension): AgentList` - takes a comprehension argument with a boolean expression as its value and returns a filtered list of agents based on this value.

`sum(SetComprehension): number` - takes a set comprehension argument with a numeric expression as its value and returns a sum of these values (from all agents).

`min(SetComprehension): AgentObject` - takes a set comprehension argument with a numeric expression as its value and returns an agent object with the minimum corresponding value.

`max(SetComprehension): AgentObject` - takes a set comprehension argument with a numeric expression as its value and returns an agent object with the maximum corresponding value.

3.6.3 Utility Functions

`agents(identifier): AgentList` - returns the list of all agents of the provided type.

`count(AgentList): number` - takes an AgentList value as a parameter and returns the number of this AgentList value.

`empty(): AgentList` - returns an empty array of agents and is used primarily in defining default property values in case of topological errors.

`prob(number): boolean` - takes a decimal numeric value between 0 and 1 representing a probability ratio and returns a boolean value based on this probability. If we use `prob(0.8)`, we have a 80% chance of getting a `true` value and a 20% change of getting a `false` value as a result.

`dist(number, number, number, number): number` - calculates the distance between two points in a two-dimensional space. The parameters represent `x1`, `y1`, `x2` and `y2` values.

`find_by_coordinates(AgentList, number, number): AgentObject` - returns an AgentObject holding an agent whose coordinates match the two numeric `x` and `y` values.

3.6.4 Special Functions

`width(): number` - returns the current width of the visualisation grid, which was provided in the interpreter's configuration.

`height(): number` - returns the current height of the visualisation grid, which was provided in the interpreter's configuration.

`index(): number` - returns the numeric index of the current agent, starting from 0.

`step(): number` - returns the value of the current step, starting from 0.

4 Implementation

The following chapters describe the most important concepts in the AgentLang interpreter's implementation and point out the most interesting parts of its architecture and functioning.

4.1 Overview and Architecture

The AgentLang's interpreter is written in TypeScript. The choice of this specific language resulted from various reasons. First and foremost, we needed good compatibility and integrability with the web-based interface, which is the primary environment where AgentLang is intended to be used. Furthermore, there was no primary need for high performance, since AgentLang is intended mainly for simple simulations, as a proof of concept of the language itself. Since modern web applications are written mainly in JavaScript frameworks and we opted for the TypeScript-based Next.js framework for the web-based interface, TypeScript felt like a natural choice.

The interpreter itself follows an architecture resembling the pipeline architectural style.

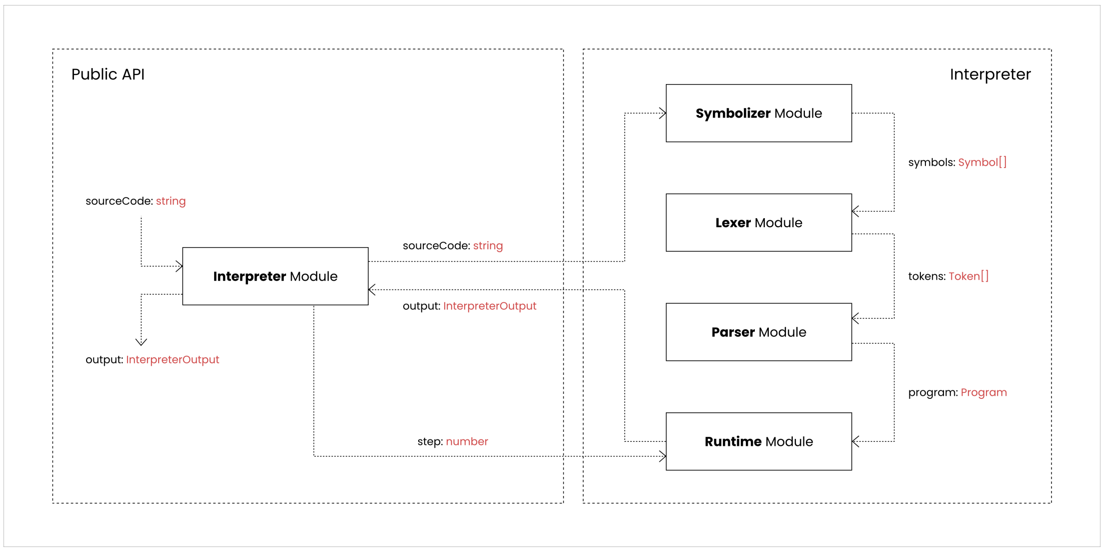


Figure 4.1 Interpreter Architecture

The interpreter consists of five main parts, which are:

1. **symbolizer** - produces an array of symbols together with their positions in the source code
2. **lexer** - produces an array of language-specific tokens together with their positions in the source code
3. **parser** - produces an abstract syntax tree holding the semantic structure of the program

4. **runtime** - traverses and evaluates the abstract syntax tree instruction by instruction and produces the program's output

5. **interpreter** - holds all of the above parts together and provides a public API for starting, pausing, resuming, stopping and resetting the simulation

4.1.1 Symbolizer

Symbolizer is a simple module responsible for one task only - to convert the source code into individual symbols and produce metadata for each symbol, such as its position in the source code. Although it is usually part of the lexer module itself, we decided to put it to a standalone module for better modularity and code readability.

The primary and only method in the symbolizer module is the `symbolize` method, which reads the source code character by character and produces an array of symbols.

```
public symbolize(): Symbol[] {
  const symbols: Symbol[] = [];
  const position: Position = { line: 1, character: 1 };

  for (const character of this.sourceCode.split("")) {
    symbols.push({
      value: character,
      position: { ...position }
    });

    position.character++;

    if (character === "\n") {
      position.line++;
      position.character = 1;
    }
  }

  return symbols;
}
```

Each symbol consists of its string value, which is strictly one character only and its position in the source code. The position contains the line number and the character number.

```
export interface Symbol {
  value: string;
  position: Position;
}

export interface Position {
  line: number;
  character: number;
}
```

4.1.2 Lexer

As opposed to the symbolizer module, the lexer module performs a slightly more complicated task. It contains language and syntax-specific logic for correctly recognizing and classifying tokens supported by the language. It groups together symbols sequentially and produces corresponding tokens with correct types. For instance, it distinguishes user-defined identifiers from language-specific, reserved keywords, so that the parser can then handle these tokens accordingly. Apart from the program's keywords, the lexer adds a special token at the end of the token array, called the end-of-file token. This token serves as an indicator to the program's end.

A token is based on a simple interface, holding its type, raw value and position in the source code. The position of a token is defined by the position of its first character.

```
export interface Token {
  type: TokenType;
  value: string;
  position: Position;
}
```

There are many token types, some of which are:

```
export enum TokenType {
  Identifier = "Identifier",
  Number = "Number",
  Boolean = "Boolean",

  OpenParen = "OpenParen",
  CloseParen = "CloseParen",

  Colon = "Colon",
  Semicolon = "Semicolon",

  // ...
}
```

The `tokenize` method is the main method of the lexer module. It traverses the input array of symbols and checks their value, based on which it decides which token type to generate. Finally, it returns the array of generated tokens.

```
public tokenize(): Token[] {
  while (this.hasNext()) {
    switch (this.getNext().value) {
      case "(":
        this.token(TokenType.OpenParen);
        break;
      case ")":
        this.token(TokenType.CloseParen);
        break;
      // ...
      default: {
        // tokenize identifiers
        // tokenize language-specific keywords
        // tokenize numeric literals
```

```
            throw new ErrorLexer("...");
        }
      }
    }

    this.token(TokenType.EOF);
    return this.tokens;
  }
```

In case the current symbol is not recognized, it throws an exception and the interpreter stops the process.

4.1.3 Parser

The parser module is without doubt one of the most interesting parts of the interpreter. It represents a push-down automaton accepting the language's grammar and producing an AST representing the semantic structure of the program. The structure of the parser module looks like the following:

```
class Parser {
  private tokens: Token[];

  constructor(tokens: Token[]) {
    this.tokens = tokens;
  }

  public parse(): Program {}

  private parseStatement(): Statement {}

  private parseObjectDeclaration(): ObjectDeclaration {}

  private parseVariableDeclaration(): VariableDeclaration {}

  private parseExpression(): Expression {};

  // rest of the parsing methods
}
```

Each `parse`... method represents one non-terminal of the language's grammar and parses the corresponding non-terminal based on the grammar rules. When we begin parsing the program, we call the `parse` method of the parser module.

```
public parse(): Program {
  const program: Program = {
    type: NodeType.Program,
    body: [],
    position: {
      line: 0,
      character: 0
    }
  };

  while (this.notEndOfFile()) {
    const statement = this.parseStatement();
    program.body.push(statement);
  }

  return program;
}
```

The method iteratively parses statements one by one until it reaches the end-of-file (EOF) token. Finally, it returns the root node of the AST called `Program`.

Let us revisit a part of the language's formal grammar for the top-level constructs:

```
program:
  | declaration
  | declaration program

declaration:
  | define_declaration
  | agent_declaration
```

The program is defined recursively and parses any number of declarations, which are statements. A declaration is either of type `define_declaration` representing a global variable declaration or `agent_declaration` representing an agent declaration. Now let's take a look at the `parseStatement` method called by the `parse` method.

```
private parseStatement(): Statement {
  switch (this.at().type) {
    case TokenType.Define:
      return this.parseDefineDeclaration();
    case TokenType.Agent:
      return this.parseObjectDeclaration();
    default:
      throw new ErrorParser("...");
  }
}
```

We can see that it abides by the grammar rules of statements (declarations). A global variable declaration always starts with the `define` keyword, whereas an agent declaration always starts with the `agent` keyword. If the current token is of any of these two types, we parse the corresponding declaration by calling the `parseDefineDeclaration` or `parseObjectDeclaration` methods respectively. In case of other token types, the parser throws an exception, since other token types do not abide by the grammar rules of the language.

These two above parsing methods are simple, since the right-hand side of their grammar rules are simple. Let us look at a more explicit example of an agent declaration, defined by the following grammar rule.

```
agent_declaration:
  | "agent" identifier agent_count "{" "}"
  | "agent" identifier agent_count "{" agent_body "}"

agent_count:
  | numeric_literal
  | identifier

agent_body:
  | property_declaration
  | property_declaration agent_body
```

The corresponding parsing function is implemented as following:

```
private parseObjectDeclaration(): ObjectDeclaration {
  const { position } = this.assert(TokenType.Agent, "...", this
    .position());
  const identifier = this.assert(TokenType.Identifier, "...",
    this.position()).value;

  this.assertMulti(TokenType.Number, TokenType.Identifier, "...
    ", this.position(), false);

  const count = this.parseExpression();

  this.assert(TokenType.OpenBrace, "...", this.position());

  const body: VariableDeclaration[] = [];

  while (this.isNotOf(TokenType.CloseBrace)) {
    switch (this.at().type) {
      case TokenType.Property:
      case TokenType.Const:
        const declaration = this.parseVariableDeclaration();
        body.push(declaration);
      break;
      default:
        throw new ErrorParser("...");
    }
  }

  this.assert(TokenType.CloseBrace, "...", this.position());

  return {
    type: NodeType.ObjectDeclaration,
    identifier, count, body, position
  };
}
```

The parsing of an agent declaration starts by checking if the current token is of type `Agent`. If yes, we save its position denoting the start of the agent declaration for future debugging purposes. The next token in the right-hand side of the grammar rule is an identifier. Therefore, we check if the current token is of type `Identifier` and if yes, we save its value. Following the identifier, we expect the next token to be either of type `Number` or `Identifier` denoting the `agent_count` non-terminal. Then, we expect an open brace. Now comes the interesting part. Based on the `agent_body` non-terminal, an arbitrary number of property declarations follows. Hence, we parse the list of property declarations until we reach the close brace token. Then we know that we reached the end of the agent declaration and can return the AST node representing the agent declaration with all its important metadata.

It is also important to mention, that the order in which these parser methods call themselves is of high importance, especially in parsing expressions. This is due to the mathematical concept of operator precedence. Let's illustrate this on an example of two parsing methods responsible for parsing additive and multiplicative binary expressions.

```
private parseAdditiveExpression(): Expression {
  let left = this.parseMultiplicativeExpression();

  while (this.at().value === "+" || this.at().value === "-") {
    const token = this.next();
    const operator = token.value;
    const position = token.position;

    const right = this.parseMultiplicativeExpression();

    left = {
      type: NodeType.BinaryExpression,
      left,
      right,
      operator,
      position
    } as BinaryExpression;
  }

  return left;
}
```

A binary expression consists of three main parts - the left operand, the right operand and a binary operator. Since the parser has no token lookahead, we can only consider the current token. Since evaluation of the AST is done using a depth-first search algorithm, we know that the lower in the AST a node is, the sooner it will be evaluated. Based on the operator precedence, more specifically the superiority of multiplication over addition, in case our expression consists of both additive and multiplicative operators, we must parse the multiplicative binary expressions first. Due to this reason, we first evaluate the left-hand side of the potential additive binary expression as a multiplicative expression, so that it lies lower in the AST than the potential additive expression we are trying to parse. Then two possible scenarios can happen - either we find an additive operator (+ or -) after the multiplicative expression or not. In the former case, we parse the right-hand side of the additive binary expression as a multiplicative expression for the same reasons that we parse the left-hand side of the additive expression as a multiplicative expression. In the latter case, however, we return the left-hand side representing a multiplicative binary expression.

To better illustrate the concept behind this way of parsing, let us look at the following two examples:

```
const a = 5 * 2;
const b = 5 * 2 + 3;
```

In the case of `a` we parse the left-hand side as a multiplicative binary expression, only to find out that it is not followed by an additive operator. In that case, we return the multiplicative binary expression from the `parseAdditiveExpression` method, which is the desired output. In the case of `b`, we again parse the left-hand side as a multiplicative binary expression. Then, we come across an additive binary operator `+`, so we parse the right-hand side of the additive binary expression. The result of the `parseAdditiveExpression` in this case is an additive binary expression with the left-hand side holding a multiplicative binary expression and the right-hand side holding a numeric literal. Since the `parseMultiplicativeExpression` works in the same way as the **parseAdditiveExpression**, it further nests into more and more low-level expressions, finally reaching the **parsePrimaryExpression** responsible for parsing numeric literals, boolean literals and identifiers. That is why the right-hand side of **b** returns a numeric literal.

Finally, to better understand the resulting AST structure of the parser module, let's look at the following AgentLang code.

```
agent person 1 {
  const age = 28;
  const is_employed = false;
}
```

It is a simple program containing one agent declaration with two `const` properties. The program is syntactically valid and after evaluation, the parser returns the following AST.

```
{
 "type": "Program",
 "body": [
  {
   "type": "ObjectDeclaration",
   "identifier": "person",
   "count": {
    "type": "NumericLiteral",
    "value": 1,
   },
   "body": [
    {
     "type": "VariableDeclaration",
     "variableType": "const",
     "identifier": "age",
     "value": {
      "type": "NumericLiteral",
      "value": 28,
     },
    },
    {
     "type": "VariableDeclaration",
     "variableType": "const",
     "identifier": "is_employed",
     "value": {
      "type": "BooleanLiteral",
      "value": false,
     },
    }
   ],
  }
 ],
}
```

The top-level unit (root node) is the program itself. It has a body, which is an array of statements (declarations). In our example program, the only statement is an agent declaration with identifier **person**. This agent declaration also has a body, which is an array of property declarations. The first property declaration has a value of a numeric literal, whereas the second property declaration has a value of a boolean literal. Each node should also have the `position` property holding the line number and character of the corresponding node in the source code, which is useful for providing the user a detailed description of potential errors. However, for the sake of this AST showcase, the `position` properties are omitted. This above AST is a real example of the AgentLang parser's output, which is then passed to the runtime module for the real-time AST evaluation.

4.1.4 Runtime

The runtime module is the final step in the process of interpreting AgentLang's source code. It's main responsibility is traversing the valid AST structure, evaluating it in real-time and producing the final program's output. This technique is, however, specific to interpreters only. Interpreters depend on the implementation language, in this case TypeScript, that merely recognizes the intentions of the AgentLang's program instructions and performs calculations of the AgentLang's program using its own data types, structures and mechanisms.

The runtime module works fundamentally in a similar way to the parser module, as far as the structure and recursive evaluation is concerned. The structure of the runtime module looks like the following:

```
class Runtime {

  private program: Program;
  private output: InterpreterOutput = {
    type: ValueType.Output,
    step: 0,
    agents: []
  };

  constructor(program: Program) {
    this.program = program;
  }

  public run(step: number): RuntimeValue {}

  private evaluateProgram(program: Program): RuntimeValue {}

  private evaluateObjectDeclaration(declaration:
   ObjectDeclaration): void {}

  private evaluateVariableDeclaration(declaration:
   VariableDeclaration): void {}

  private evaluateBinaryExpression(expression: BinaryExpression
   ): RuntimeValue {}

  // rest of the evaluating methods
}
```

Whereas in the parser module each method returned a part of the AST as the descendant of the `ParserValue` interface, the runtime module uses an abstract interface called `RuntimeValue` representing the actual value of each data type, such as a numeric literal or a boolean literal. Notice that the `evaluateObjectDeclaration` and `evaluateVariableDeclaration` methods are of type `void` and they do not return anything. That is because both object declaration and variable declaration are statements. In the `objectDeclaration` method, the final step is to save the evaluated agent to the `output: InterpreterOutput` variable, which after the evaluation of the program is returned by the `run` method.

The runtime module is not responsible for stepping through the simulation. The `run` method takes the step as the parameter and performs one evaluation of a single simulation step. The stepping functionality is implemented in the interpreter module discussed in following chapters.

To better understand the evaluation of expressions from the AST, below is the implementation of the `evaluateBinaryExpression` method:

```
private evaluateNumericBinaryExpression(
  expression: BinaryExpression
): RuntimeValue {
  const { operator, left, right } = expresssion;
  const lhs = this.evaluateExpression(left);
  const rhs = this.evaluateExpression(right);

  let result = 0;

  if (operator === "+") {
    result = lhs.value + rhs.value;
  } else if (operator === "-") {
    result = lhs.value - rhs.value;
  } else if (operator === "*") {
    result = lhs.value * rhs.value;
  } else if (operator === "/") {
    if (rhs.value === 0) {
      throw new ErrorRuntime("...");
   }

   result = lhs.value / rhs.value;
  } else {
   if (rhs.value === 0) {
     throw new ErrorRuntime("...");
   }

   result = this.customModulo(lhs.value, rhs.value);
  }

  return {
    type: ValueType.Number,
    value: result
  } as NumberValue;
}
```

First, the runtime module needs to evaluate both operands of the binary expression. The resulting values should be numeric literals. Then, based on the operator of the binary expression, we perform mathematical calculations to get the resulting value of the binary expression and we return this result as an instance of the `RuntimeValue` interface.

The result of the program is a descendant of the `RuntimeValue` interface called `RuntimeOutput`:

```
export interface RuntimeOutput extends RuntimeValue {
  type: ValueType.Output;
  step: number;
  agents: RuntimeAgent[];
}

export interface RuntimeAgent {
  id: string;
  identifier: string;
  variables: Map<string, RuntimeValue>;
}
```

The output emits an array of agents with their current property values as well as the number of the current step.

After every evaluation of the current step, the resulting array of agents is cached by the runtime module to a variable called `previousAgents: RuntimeAgent[]`. This needs to be done because the values of properties in the next step are always calculated based on their previous values (values in the last step). This is to ensure the avoidance of problems with sequential interpreting. Since agents are evaluated one by one, it could happen that an agent's property is dependent on the previous agent's property. This would result in a chain of unwanted values in the same step. Therefore, new values are always calculated based on previous values. Step 0 is therefore a special case of step, where not only `const` properties are initialised, but also default initial values are calculated and saved to the previous output.

Perhaps the most important and most used method in the runtime module is the `evaluateRuntimeValue` method, which is used every time we need to evaluate an expression of any kind. It routes the input AST node to the corresponding evaluator method and returns its result:

```
private evaluateRuntimeValue(node: ParserValue, id: string):
    RuntimeValue {
  switch (node.type) {
    case NodeType.Identifier:
      return this.evaluateIdentifier(node as Identifier, id);
    case NodeType.NumericLiteral:
      return this.evaluateNumericLiteral(node as NumericLiteral
    );
    case NodeType.BooleanLiteral:
      return this.evaluateBooleanLiteral(node as BooleanLiteral
    );
    case NodeType.BinaryExpression:
      return this.evaluateBinaryExpression(node as
    BinaryExpression, id);
    case NodeType.UnaryExpression:
      return this.evaluateUnaryExpression(node as
    UnaryExpression, id);
    // other evaluating methods
    default:
      throw new ErrorRuntime("...");
  }
}
```

4.1.5 Interpreter

The interpreter module is a place where all of the above modules are used from and controlled by. The main responsibility of the interpreter module is to provide a public API for developers to integrate and use the interpreter in external projects. The input to the interpreter module is a plain string representing the source code of an AgentLang's simulation and the output is the simulation's output itself, returned by the runtime module. For controlling the simulation's runtime, the interpreter module uses the `RxJS` [14] library for of its extensive functionality with observables and subscriptions.

The structure of the interpreter module looks as below:

```
class Interpreter {

  public get(sourceCode: string, config:
    InterpreterConfiguration): Observable<InterpreterOutput> {}

  private build(sourceCode: string, config:
    InterpreterConfiguration): void {}

  public start(): void {}
  public pause(): void {}
  public resume(): void {}
  public reset(): void {}
  public step(): void {}
}
```

The interpreter module contains a set of public control methods used for controlling the interpretation process. These include:

1. `start` - starts the interpreter
2. `pause` - pauses the interpreter
3. `resume` - resumes paused interpreter, so it will continue at the step it left off on
4. `reset` - resets the current step to 0 to start from the beginning
5. `step` - can be called when the interpreter is paused and will emit the output of the next step on each call

Then there is the `get` method, which the user subscribes to to retrieve the current output every time it is emitted by the interpreter. It accepts two parameters, which is the source code and the configuration. The source code is a plain string containing the AgentLang program's source code. The configuration is an object defined as below.

```
export interface InterpreterConfiguration {
  steps: number;
  delay: number;
  width: number;
  height: number;
}
```

When we start the simulation, a new subscription is created using the **RxJS** library:

```
private subscribe(): void {
  this.subscription = interval(this.config.delay).pipe(
    takeWhile(() => this.currentStep == this.config.steps),
  ).subscribe(() =>
    this.dataSubject.next(this.getInterpreterOutput(this.
   currentStep++)));
}
```

It utilises the **delay** and **steps** parameters to ensure the emitting of the interpreter's output for **steps** times with the delay between each step defined by the **delay** parameter. On every emit, we push a new value to the **dataSubject** variable. The **getInterpreterOutput** ensures the evaluation of the current step by the runtime module:

```
private getInterpreterOutput(step: number): InterpreterOutput {
  try {
    const value: RuntimeValue = this.runtime!.run(step);
    return this.getRuntimeOutput(value as RuntimeOutput);
  } catch (error) {
    return this.getRuntimeError(error as ErrorRuntime)
  }
}
```

The **getRuntimeOutput** and **getRuntimeError** are plain mapper functions that ensure correct mapping from the raw input to a desired output for the user.

4.2 Interesting Concepts

The following chapters describe some of the interesting, non-trivial concepts used in the implementation of the AgentLang's interpreter, such as the topological property sorting or the automatic source code formatting.

4.2.1 Topological Property Sorting

When modeling an AgentLang simulation, there are often numerous dependencies between the agents' properties. Sometimes two properties can depend on each other respectively, which poses a significant problem to the evaluation process of their values. Although the properties can be assigned a default value which is a constant calculated at the beginning of the simulation, it often does not make sense to use default values for each property. Moreover it poses bad development experience if the user needs to declare properties in the order in which they should be evaluated by the interpreter. Fortunately, there is a way to solve such issues in a general way using topological sort.

Topological sort is a sorting mechanism that operates on directed acyclic graphs (henceforth referred to as DAGs). It is a linear ordering of the graph's vertices such that for every directed edge from vertex **u** to vertex **v**, vertex **u** comes before vertex **v** in the ordering. This algorithm does not work on cyclic graphs, since there is no linear ordering. In other words, the vertices would depend on each other in a cyclic manner, not allowing the algorithm to determine which vertex to start with and which vertex to finish with.

In terms of programming language interpreters, topological sort is a useful technique for tackling issues with variable dependencies. Interpreters read and evaluate source code line by line, instruction by instruction. This mechanism implies that before we can access a variable by its identifier, we first need to define it. That is fine by most general-purpose programming languages, since they feature various other complex language structures to work around this problem. However, AgentLang is designed in a specifically variable-oriented way, therefore the issues with the ordering of variable evaluation needs to be handled. And that is where the topological sorting mechanism can be used.

In AgentLang, topological sort is used to reorder property declarations in agents so that they are evaluated in the order on which they are dependent on each other. This means, that if property `a` is dependent on property `b`, but property `b` is not dependent on any other property, we first need to evaluate `b` before evaluating `a`, even if `a` has been declared before `b`. The first step to this technique is to retrieve all property identifiers in an agent. For each such property declaration, we need to retrieve all other property identifiers that this property uses in its declaration. From these two sets of information, we can construct a directed graph, where each node represents one agent property and the edges coming from this node are directed to the property nodes on which this property is dependent on. As a result, we get a directed graph structure.

However, we do not know at this point whether this graph is cyclic or not. We need to discover this information, in order to say whether the simulation can be evaluated or not. We pass this directed graph to a topological sort algorithm, where there are two possible outputs. It either topologically sorts the graph if it is acyclic and returns an array of property identifiers in the order in which they should be evaluated, or it throws an exception if the graph is cyclic and the topological sorting of the graph's nodes is not possible. In the former case, we reorder the property declarations in the AST and can forward this updated AST to the interpreter's runtime. In the latter case, the program cannot be evaluated and the user needs to fix the cyclic property dependencies in the source code of the AgentLang simulation.

In the AgentLang's implementation, the directed acyclic graph is a simple hash map of nodes and their dependencies:

```
export type DependencyGraph = { [key: string]: Node };

export class Node {
  public identifier: string;
  public dependencies: Node[] = [];

  constructor(identifier: string) {
    this.identifier = identifier;
  }

  public addDependency(node: Node): void {
    this.dependencies.push(node);
  }
}
```

Such graph structure can be passed to the topological sorting algorithm depicted below:

```
function topologicalSort(graph: DependencyGraph): Node[] {
  const visited: { [key: string]: boolean } = {};
  const recursionStack: { [key: string]: boolean } = {};
  const result: Node[] = [];
  let containsCycle = false;

  const isSelfLoop = (node: Node): boolean =>
    node.dependencies.some(dep => dep === node);

  function visit(node: Node) {
    if (recursionStack[node.identifier]) {
      containsCycle = true;
      return;
    }

    if (visited[node.identifier]) return;

    visited[node.identifier] = true;
    recursionStack[node.identifier] = true;

    for (const dependency of node.dependencies) {
      if (dependency && !isSelfLoop(dependency))
        visit(dependency);
    }

    recursionStack[node.identifier] = false;
    result.push(node);
  }

  for (const key in graph) visit(graph[key]);

  if (containsCycle) throw new ErrorParser("...");

  return result;
}
```

This algorithm iterates over the agent's property declarations and for each declaration, it recursively visits each node to which a directed edge from the current node exists. Moreover, it saves the visited nodes in a hash map. If we come to a node which was already visited in one iteration, we know there is a cycle in the graph. In that case, we throw an exception. Otherwise, the graph is acyclic and we can return the resulting ordering of the graph's nodes representing the order in which the agent's properties should be evaluated by the interpreter.

4.2.2 Source Code Formatter

Although AgentLang's syntax is not dependent on indents or other whitespace characters, it is a good practice to follow specific syntactical rules or recommendations for the given language. Therefore, the interpreter features a source code formatter, which formats the source code in the language-specific way on the simulation startup.

The code formatter works in a straightforward way. First, it parses the source code and produces an AST representing the semantics of the program. Then, it passes this AST into a function which recursively traverses the structure and produces plain source code equivalent to the input source code, formatted to the specific way defined by a set of rules.

However, producing a semantically equivalent source code is not as straightforward as it may appear. More specifically, a great problem arises with complex parenthesised expressions, where the code formatter must correctly assess the expression tree and parenthesised it in way that the resulting evaluation of the formatted expression is semantically equivalent to the user-defined expression in the input source code.

One of the most important concepts in AgentLang source code formatting is the operator precedence in binary expressions. Each operator has a certain level of precedence represented by a numeric value. The higher this level of precedence is, the sooner the sub-expression must be evaluated. Moreover, parentheses surpass the operator precedence, rendering the whole sub-expression more important than a non-parenthesised expression.

In AST, parentheses are completely omitted. Instead, the tree structure is composed in a way it abides by the user parenthetisation. Suppose the following two examples:

```
const a = 5 + 2 * 3;
const b = (5 + 2) * 3;
```

The result of `a` is 11, since `2 * 3` is evaluated sooner. However, the result of `b` is 21, since the additive binary expression is evaluated first becuase of the parenthetisation. Let's look at the resulting AST of these two property declarations.

```
{
  "type": "VariableDeclaration",
  "variableType": "const",
  "identifier": "a",
  "value": {
    "type": "BinaryExpression",
    "left": {
      "type": "NumericLiteral",
      "value": 5,
    },
    "right": {
      "type": "BinaryExpression",
      "left": {
        "type": "NumericLiteral",
          "value": 2,
        },
      "right": {
        "type": "NumericLiteral",
        "value": 3,
      },
      "operator": "*",
    },
    "operator": "+",
  },
}
```

```
{
  "type": "VariableDeclaration",
  "variableType": "const",
  "identifier": "b",
  "value": {
    "type": "BinaryExpression",
    "left": {
      "type": "BinaryExpression",
      "left": {
        "type": "NumericLiteral",
        "value": 5,
      },
      "right": {
        "type": "NumericLiteral",
        "value": 2,
      },
      "operator": "+",
    },
    "right": {
      "type": "NumericLiteral",
      "value": 3,
    },
    "operator": "*",
  },
}
```

Each binary expression in the AST has a **`left`** and **`right`** value as well as the `operator` value. We can see that the declaration of **`a`** represents a binary expression, where the **`left`** value is of type **`NumericLiteral`** and the **`right`** value is of type **`BinaryExpression`**. That is due to the operator precedence, where multiplication has higher precedence that addition. Since ASTs are evaluated using a depth-first search algorithm, the **`right`** value will be evaluated sooner than the **`left`** value, resulting in a correct evaluation of the expression. However, in the declaration of **`b`**, the **`left`** value is of type **`BinaryExpression`** and the **`right`** value is of type **`NumericLiteral`**. This happens due to the fact that the additive sub-expression has been parenthesised. Therefore, the AST knows to evaluate the left-hand side of the expression sooner than the right-hand side.

This proves that although the AST does not explicitely hold any information about user-defined parentheses, it holds this information using its structure and semantics. This implies that we can reconstruct the parenthetisation from the AST itself, proving that the source code formatter can truly produce a semantically equivalent source code from the AST alone.

The source code formatter formats the source code recursivelly by calling one function called **`nodeToSourceCode`**. This function takes any generic AST node as its input, in our case the **`Program`** node and recursively evaluates each node of the tree, producing a new source code during its runtime. Below is an example of producing a correctly parenthesised binary expression:

```
public static nodeToSourceCode(ast: ParserValue): string {
  let sourceCode = "";

  switch (ast.type) {
    case NodeType.BinaryExpression: {
      const expression = ast as BinaryExpression;
      const { operator } = expression;

      let left = Formatter.nodeToSourceCode(expression.left);
      let right = Formatter.nodeToSourceCode(expression.right);

      // handle left parentheses
      if (expression.left.type === NodeType.BinaryExpression) {
        const needsParen = Formatter.precedence[operator] > Formatter.precedence[(expression.left as BinaryExpression).operator];
        left = needsParen ? `(${left})` : left;
      }

      // handle right parentheses
      if (expression.right.type === NodeType.BinaryExpression) {
        const needsParen = Formatter.precedence[operator] > Formatter.precedence[(expression.right as BinaryExpression).operator];
        right = needsParen ? `(${right})` : right;
      }

      sourceCode += `${left} ${operator} ${right}`;
      break;
    }
    // rest of the cases
  }

  return sourceCode;
}
```

This function checks the binary precedence defined by the below rules and based on it parenthesises the resulting expression:

```
const precedence: { [key: string]: number } = { "+": 1, "-": 1, "*": 2, "/": 2, "%": 2 };
```

The source code formatter serves as a tool to achieve code readability across all AgentLang simulations, forcing the user to abide by the syntactical rules defined by the AgentLang project.

4.3 API Reference

The AgentLang interpreter can be integrated into any TypeScript-based project and used using its public API. The public API contains three main exports:

1. `Interpreter` - the interpreter class with all its functionality
2. `InterpreterConfiguration` - an interface representing the interpreter's configuration structure
3. `InterpreterOutput` - an interface representing the interpreter's output structure

The following example demonstrates the usage of the AgentLang interpreter in a TypeScript-based project:

```typescript
import {
  Interpreter,
  InterpreterConfiguration,
  InterpreterOutput
} from "./agent-lang-interpreter";

const sourceCode = readFileSync("sourceCode.txt", "utf-8");
const interpreter = new Interpreter();
const configuration: InterpreterConfiguration = {
  steps: 1000,
  delay: 100,
  width: 500,
  height: 500
};

interpreter.get(sourceCode, configuration).subscribe((
    interpreterOutput: InterpreterOutput) => {
  const { status, output } = interpreterOutput;
  // do something with the output
});

interpreter.start();
```

The `get` method returns an `Observable` object to which the user can subscribe to to capture the outputs of individual simulation steps. The `start` method starts the interpreter and the simulation.

5 Web Interface

Apart from the AgentLang's language interpreter, the thesis provides a web-based interface serving as the main environment for trying out AgentLang in practice. The web application features a code editor for modeling AgentLang simulations, a visualisation view for quick visual rendering of the simulations and last but not least, a spreadsheet interface for manipulating agent property declarations and values during runtime.

5.1 Motivation

The motivation behind the creation of the web-based interface as the primary environment for AgentLang is deeply connected to the motivation behind the creation of the AgentLang programming language itself. As mentioned earlier, AgentLang aims to provide a brand new approach for handling and modeling agent-based simulations. On the one hand this is possible due to the simplicity of the language and its syntax itself. However, on the same level of importance lies the possibility to manipulate the simulation using a visual, interactive tool - the spreadsheet interface, which allows for quick and simple fine-tuning of the simulation during its runtime.

In order to create and use the spreadsheet interface as a tool to manipulate AgentLang simulations, a user interface of some kind is required, preferably one which is easily portable and usable everywhere without the need for any kind of manual installation. For these reasons, we opted for a web-based environment integrating both the interpreter and the spreadsheet interface, enabling to create, model and run user-defined AgentLang simulations all in one place.

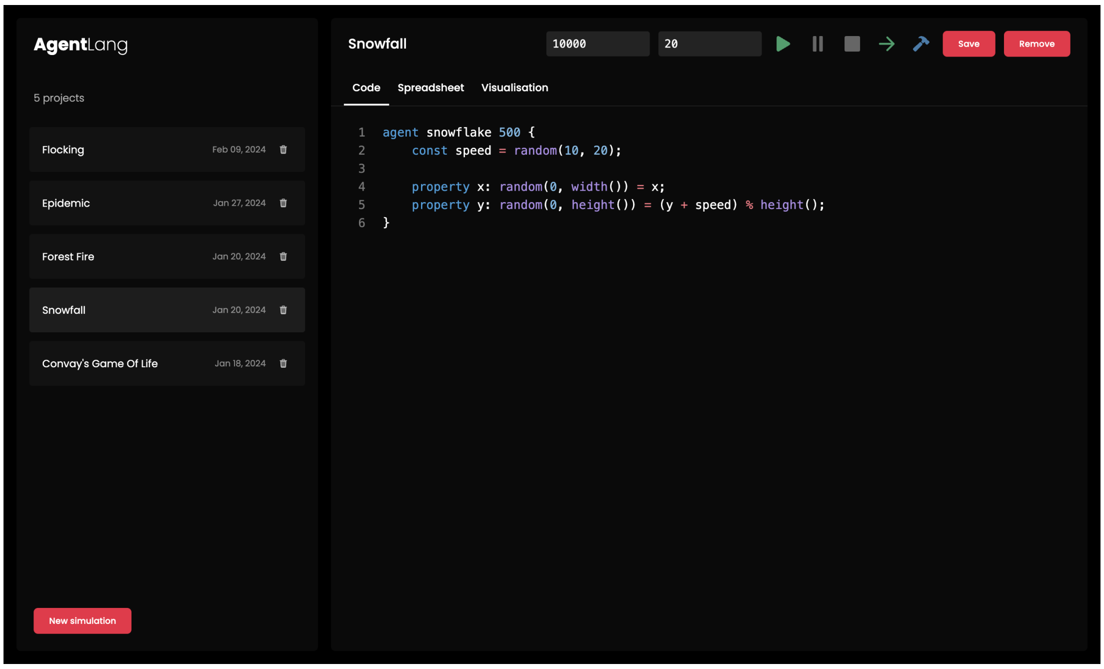
Since AgentLang interpreter is written in TypeScript, it became also the most natural choice as the implementation language for the web-based interface. More specifically, the web application is written in the TypeScript-based Next.js framework by Vercel Inc [15]. These choices allow for easy integration and seamless usage of the interpreter in the web application.

Additionally, since AgentLang is primarily intended for simple simulations as a proof of concept of the language itself, it felt natural to also provide a visualisation module for quick and easy analysis of the simulation in real-time. The simulation module consists of a panel capable of rendering simulations suitable for visualisation on a two-dimensional Cartesian plane, such as the flocking, forest fire or epidemic simulations.

These aforementioned features of the AgentLang project are complementary to each other and together provide an all-in-one suite of tools to model and analyse AgentLang simulations.

5.2 Code Sandbox

The code sandbox is the main page of the AgentLang web interface. It is a place where the user can create projects, model simulations using the AgentLang language, run the simulations and finally see their results in the visualisation module and the spreadsheet interface.



Flocking	Feb 09, 2024
Epidemic	Jan 27, 2024
Forest Fire	Jan 20, 2024
Snowfall	Jan 20, 2024
Convay's Game Of Life	Jan 18, 2024

```
agent snowflake 500 {
    const speed = random(10, 20);

    property x: random(0, width()) = x;
    property y: random(0, height()) = (y + speed) % height();
}
```

Figure 5.1 Code Sandbox Page

On the left-hand side of the screen, there is a vertical scrollable list of all user-defined projects. The user can create new projects, edit existing projects or remove existing projects. After clicking on any of the projects, the project code and the simulation configuration is loaded into the code editor on the right-hand side of the screen.

On the top-right side of the screen, there is a control panel with the current project's name, two input fields for updating the `steps` and `delay` configuration parameters of the selected simulation and a set of buttons used for handling the start, stop, pause, resume and reset of the current simulation. These buttons directly use the public API of the AgentLang interpreter, more specifically the `start`, `stop`, `pause`, `resume` and `reset` methods.

5.2.1 Code Editor

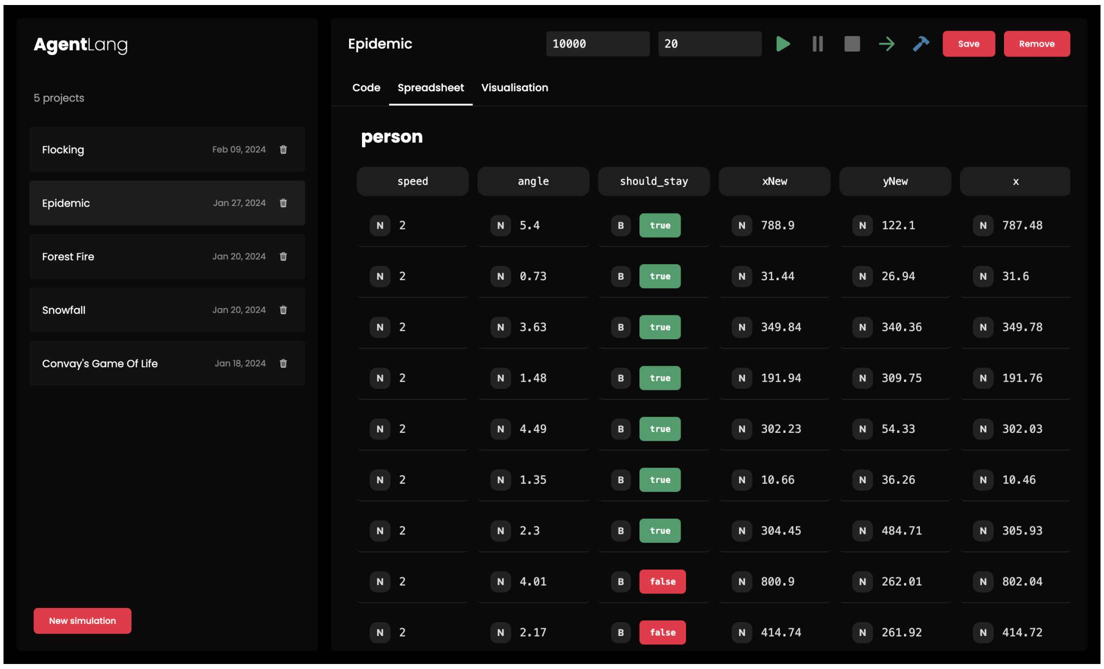
The code editor is a place where the user inputs the AgentLang source code in order to model an agent-based simulation. The code editor features basic syntax highlighting and line numbering, supporting all AgentLang syntactical constructs and concepts. However, it does not provide code completion or code suggestions.

The most recent source code updates are saved automatically to the browser's local storage on every key press without the need to press the `save` button located in the upper control panel. The `save` button serves mainly for saving the latest `steps` and `delay` parameters, which are not saved automatically upon changing. After modeling the simulation, the user can start the simulation by clicking on the `start` button located in the control panel.

On every simulation startup, the AgentLang code is parsed and run by the interpreter. In case of syntactical or semantic errors, the interpreter raises an exception, which is caught by the web interface and shown to the user using a popup message appearing on the bottom of the screen. In such case scenario, the simulation is not started and the user must fix the errors first. In case of correct source code, however, a success popup message is shown to the user and the simulation starts.

5.2.2 Spreadsheet Interface

The spreadsheet interface is initialised and filled with data as soon as the user starts the simulation. Otherwise a "No data" message is shown in the spreadsheet view.



person

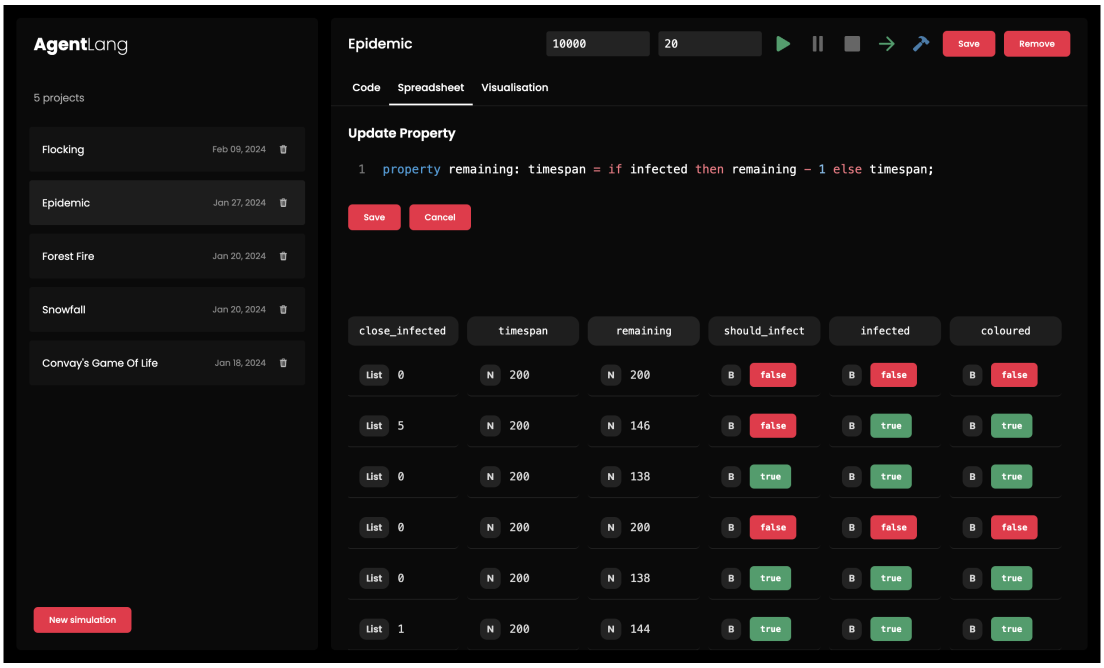
speed	angle	should_stay	xNew	yNew	x
N 2	N 5.4	B true	N 788.9	N 122.1	N 787.48
N 2	N 0.73	B true	N 31.44	N 26.94	N 31.6
N 2	N 3.63	B true	N 349.84	N 340.36	N 349.78
N 2	N 1.48	B true	N 191.94	N 309.75	N 191.76
N 2	N 4.49	B true	N 302.23	N 54.33	N 302.03
N 2	N 1.35	B true	N 10.66	N 36.26	N 10.46
N 2	N 2.3	B true	N 304.45	N 484.71	N 305.93
N 2	N 4.01	B false	N 800.9	N 262.01	N 802.04
N 2	N 2.17	B false	N 414.74	N 261.92	N 414.72

Figure 5.2 Spreadsheet Interface

The spreadsheet interface consists of one spreadsheet for each agent model. A spreadsheet contains columns representing individual agent properties defined in the source code and rows representing individual agent instances. The spreadsheet cells represent current values of the agent's properties. Moreover, the spreadsheets are recalculated and provided with new data in each step of the simulation.

Updating Property Definition

The first feature of the spreadsheet interface is the update of an agent properties' definitions. To update the definition of a property, the user needs to pause the simulation and then click on the desired property name located in the table header. A code editor with the definition of the given property is displayed above the list of spreadsheets. The user can redefine and save the property's new definition by clicking the `save` button under the code editor. The new property definition is updated, the interpreter rebuilds the simulation and the user can resume the simulation by clicking the `resume` button located in the upper toolbar.



```
property remaining: timespan = if infected then remaining - 1 else timespan;
```

close_infected	timespan	remaining	should_infect	infected	coloured
List 0	N 200	N 200	B false	B false	B false
List 5	N 200	N 146	B false	B true	B true
List 0	N 200	N 138	B true	B true	B true
List 0	N 200	N 200	B false	B false	B false
List 0	N 200	N 138	B true	B true	B true
List 1	N 200	N 144	B true	B true	B true

Figure 5.3 Agent Property Update in Spreadsheet

Updating Property Value

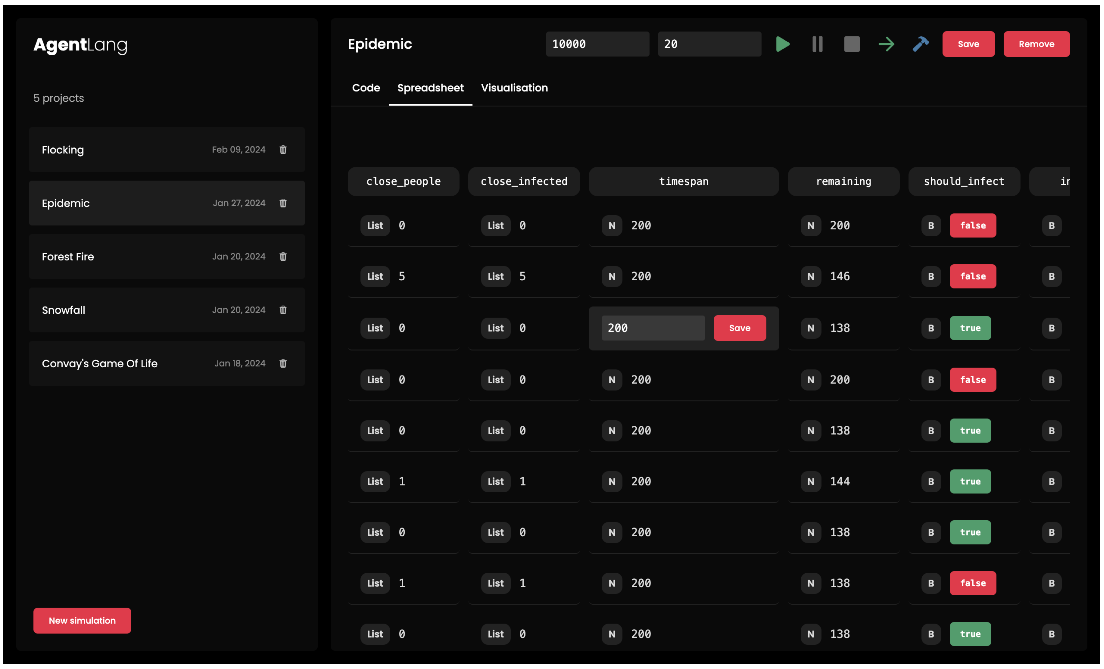
The second primary feature of the spreadsheet interface is the update of a specific property's value in an agent instance. To update a property value, click on the given cell in the spreadsheet. An input field will appear inside the corresponding cell where the user can input the new property value and save the new value by clicking the `save` button next to the input field. The new property value is updated and the user can resume the simulation by clicking the `resume` button located in the upper toolbar.

5.2.3 Visualisation

The visualisation module is used to visualise the simulations agents in real time on a two-dimensional plane.
The visualisation of agent's depends on a set of standards - each agent that should be visualised needs to have the following properties:

- `x` - a numeric value representing the x coordinate of the agent
- `y` - a numeric value representing the y coordinate of the agent
- `w` - a numeric value representing the width of the agent
- `h` - a numeric value representing the height of the agent
- `c` - an rgb value representing the colour of the agent (using the built-in function `rgb`)

After starting the simulation, the user is redirected to the visualisation panel by default, where they can observe the agent's positions, dimensions and colour.



close_people	close_infected	timespan	remaining	should_infect	ir
List 0	List 0	N 200	N 200	B false	B
List 5	List 5	N 200	N 146	B false	B
List 0	List 0	200 Save	N 138	B true	B
List 0	List 0	N 200	N 200	B false	B
List 0	List 0	N 200	N 138	B true	B
List 1	List 1	N 200	N 144	B true	B
List 0	List 0	N 200	N 138	B true	B
List 1	List 1	N 200	N 138	B false	B
List 0	List 0	N 200	N 138	B true	B

Figure 5.4 Agent Value Update in Spreadsheet

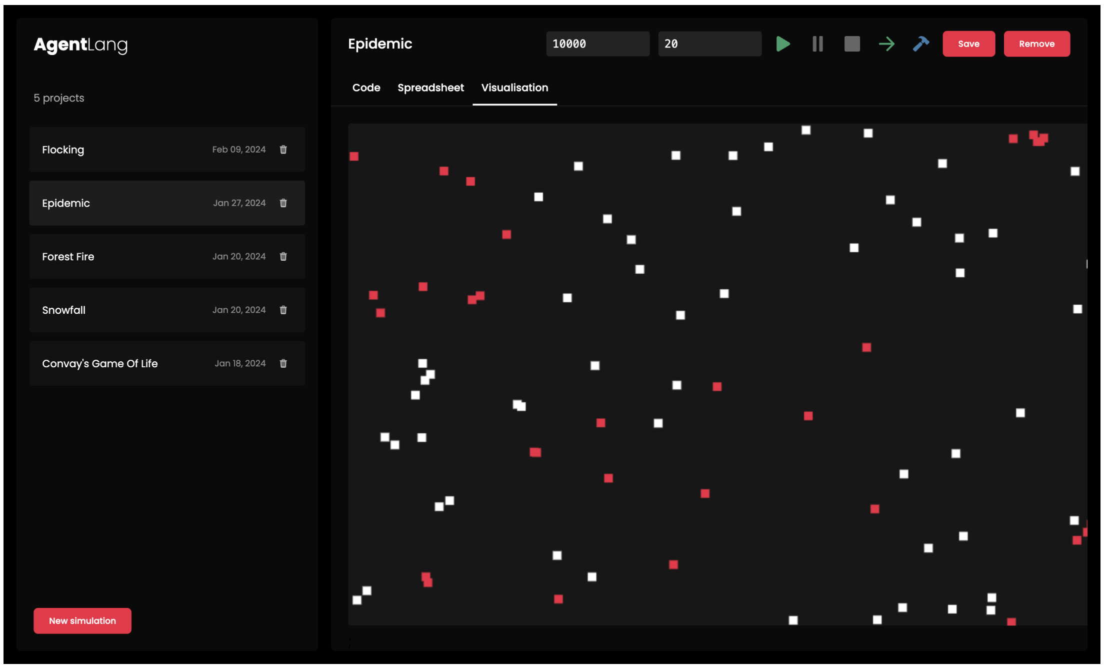


Figure 5.5 Visualisation

6 Evaluation

This chapter evaluates the AgentLang programming language, its design and ease of use using specific simulation examples and their comparison with existing agent-based frameworks. Although the below examples require rather long code blocks, their purpose is to highlight the key differences between AgentLang and other agent-based frameworks and to point out the advantages of using AgentLang to model agent-based simulations.

6.1 Flocking

The first example is the flocking simulation using the Boids algorithm. This model simulates bird flocking, where each bird in the flock steers towards a certain point using a set of three defined rules - *separation*, *alignment* and *cohesion*.

- **separation** steers the bird to avoid birds in close proximity
- **alignment** steers the bird to align their heading with the average heading of their flock mates
- **cohesion** steers the bird to fly towards the average position of their flock

Although the below examples all work in a similar way, based on the concepts used in the Boids algorithm, they may differ in details, configuration parameters or some values.

6.1.1 AgentScript

```
export default class FlockModel extends Model {
  population = 100
  vision = 3
  speed = 0.25
  maxTurn = 3.0
  minSep = 0.75

  constructor(worldDptions) {
    super(worldDptions)
  }

  setup() {
    this.turtles.setDefault('speed', this.speed)
    this.patches.cacheRect(this.vision)
    util.repeat(this.population, () => {
      this.patches.oneOf().sprout()
    })
  }

  step() {
    this.turtles.ask(t => {
      this.flock(t)
    })
  }

  flock(t) {
    const fm = this.turtles.inRadius(t, this.vision, false)
    if (fm.length !== 0) {
      const n = fm.minOneOf(f => f.distance(t))
      if (t.distance(n) < this.minSep) {
        this.separate(t, n)
      } else {
        this.align(t, fm)
        this.cohere(t, fm)
      }
    }
    t.forward(t.speed)
  }

  separate(t, n) {
    const h = n.towards(t)
    this.turnTowards(t, h)
  }

  align(t, fm) {
    this.turnTowards(t,
      this.averageHeading(fm))
  }

  cohere(t, fm) {
    this.turnTowards(t,
      this.averageHeadingTowards(t, fm))
  }

  turnTowards(t, heading) {
    let turn = t.subtractHeading(heading)
    turn = util.clamp(turn,
     -this.maxTurn, this.maxTurn)
    t.rotate(turn)
  }

  averageHeading(fm) {
    // implementation
  }

  averageHeadingTowards(t, fm) {
    // implementation
  }

  flockVectorSize() {
    // implementation
  }
}
```

In the above AgentScript flocking simulation, the `step` function is called in each step of the simulation and handles the flocking semantics of each agent. The `flock` method calls the three main methods corresponding to the rules of the Boids flocking algorithm - `separate`, `align` and `cohere`. While this approach is very straightforward and convenient for JavaScript developers, it may pose challenges to scientists of non-programming background, since the simulation is directly dependent on the JavaScript language.

6.1.2 NetLogo

```
turtles-own [
 flockmates
 nearest-neighbor
]

to setup
 clear-all
 create-turtles population
  [ set color yellow - 2 + random 7
   set size 1.5 ;; easier to see
   setxy random-xcor random-ycor
   set flockmates no-turtles ]
 reset-ticks
end

to go
 ask turtles [ flock ]
 repeat 5 [ ask turtles [ fd 0.2 ] display ]
 tick
end

to flock
 find-flockmates
 if any? flockmates
  [ find-nearest-neighbor
   ifelse distance nearest-neighbor < minimum-separation
    [ separate ]
    [ align
     cohere ] ]
end

to find-flockmates
 set flockmates other turtles in-radius vision
end

to find-nearest-neighbor
 set nearest-neighbor min-one-of flockmates [distance myself]
end

to separate
 turn-away ([heading] of nearest-neighbor) max-separate-turn
end

to align
 turn-towards average-flockmate-heading max-align-turn
end

to-report average-flockmate-heading
 let x-component sum [dx] of flockmates
 let y-component sum [dy] of flockmates
 ifelse x-component = 0 and y-component = 0
  [ report heading ]
  [ report atan x-component y-component ]
end

to cohere
```

```
 turn-towards average-heading-towards-flockmates max-cohere-turn
end

to-report average-heading-towards-flockmates
 let x-component mean [sin (towards myself + 180)] of flockmates
 let y-component mean [cos (towards myself + 180)] of flockmates
 ifelse x-component = 0 and y-component = 0
  [ report heading ]
  [ report atan x-component y-component ]
end

to turn-towards [new-heading max-turn]
 turn-at-most (subtract-headings new-heading heading) max-turn
end

to turn-away [new-heading max-turn]
 turn-at-most (subtract-headings heading new-heading) max-turn
end

to turn-at-most [turn max-turn]
 ifelse abs turn > max-turn
  [ ifelse turn > 0
    [ rt max-turn ]
    [ lt max-turn ] ]
  [ rt turn ]
end
```

The NetLogo version of the flocking simulation works fundamentally in a similar way to the AgentScript model. It contains three main procedures - `separate`, `align` and `cohere`, which are invoked by the main `flock` procedure. Although NetLogo provides a new, specific language for agent-based modeling, it is expressive and verbose and the understanding of this language and its structures poses a steep learning curve for its users.

6.1.3 AgentLang

```
define visual_range = 100;
define avoid_range = 20;
define centering_factor = 0.0005;
define avoid_factor = 0.005;
define matching_factor = 0.05;

define s_max = 8;
define s_min = 6;

agent boid 100 {

  const w = 10;
  const h = 10;

  const x_init = random(100, width() - 100);
  const y_init = random(100, height() - 100);

  property x: x_init = (x + x_vel_limit) % width();
  property y: y_init = (y + y_vel_limit) % height();

  property x_vel: choice(-2, 2) =
    x_vel + x_sep + x_align + x_coh;
  property y_vel: choice(-2, 2) =
    y_vel + y_sep + y_align + y_coh;

  property s_sqrt = sqrt(x_vel * x_vel + y_vel * y_vel);
  property s = if s_sqrt == 0 then 1 else s_sqrt;

  property x_vel_limit = if s > s_max then x_vel / s * s_max else if s < s_min then x_vel / s * s_min else x_vel;
  property y_vel_limit = if s > s_max then y_vel / s * s_max else if s < s_min then y_vel / s * s_min else y_vel;

  property boids_ar: empty() = filter(agents(boid) | b -> dist(b.x, b.y, x, y) < avoid_range);
  property boids_vr: empty() = filter(agents(boid) | b -> dist(b.x, b.y, x, y) < visual_range);
  property bvrc = count(boids_vr);

  # separation #
  property x_sep =
    sum(boids_ar | b -> x - b.x) * avoid_factor;
  property y_sep =
    sum(boids_ar | b -> y - b.y) * avoid_factor;

  # alignment #
  property x_align =
    if bvrc > 0 then
    (sum(boids_vr | b -> b.x_vel) / bvrc - x_vel) * matching_factor else 0;
  property y_align =
    if bvrc > 0 then
    (sum(boids_vr | b -> b.y_vel) / bvrc - y_vel) * matching_factor else 0;

  # cohesion #
  property x_coh =
    if bvrc > 0 then
    (sum(boids_vr | b -> b.x) / bvrc - x) * centering_factor
    else 0;
  property y_coh =
    if bvrc > 0 then
    (sum(boids_vr | b -> b.y) / bvrc - y) * centering_factor
    else 0;
}
```

Analysing the AgentLang's version of the flocking simulation, it is noticeable that the model requires less lines of code. Moreover, due to the structure of the language, it is straightforward in a way that the agent model is nothing but a sequence of property definitions, each having a human-readable name and definition, upon which the property is evaluated. The `x` and `y` coordinates are incremented by the `x_vel` and `y_vel` values, which are the sum of the *separation*, *alignment* and *cohesion* properties defined at the bottom of the agent model.

6.2 Forest Fire

The second example is a forest fire simulation which analyses the spread of fire in a forest. The trees are points in a two-dimensional Cartesian grid. Each tree can be in either of the three states - *idle*, *burning* or *dead*. If an *idle* tree has a neighbour which is *burning*, it becomes *burning* based on some probability. Moreover, if a tree is *burning* for a specified period of time, it becomes *dead*. A *dead* tree can no longer spread fire to its neighbours.

The spread of fire in a forest varies based on several factors. These include the percentage of probability based on which the fire spreads as well as the period of time a tree can be in the *burning* state. The longer a tree is *burning*, the higher the chance is for the neighbouring trees to start *burning* too.

Although the below examples all function based on the same concepts, they may differ in details, configuration parameters or some values.

6.2.1 AgentScript

```
export default class FireModel extends Model {
  density = 60

  constructor(worldDptions = World.defaultOptions(125)) {
    super(worldDptions)
  }

  setup() {
    this.patchBreeds('fires embers')

    this.patchTypes = [ /* patch types */ ]
    this.dirtType = this.patchTypes[0]
    this.treeType = this.patchTypes[1]
    this.fireType = this.patchTypes[2]

    this.patches.ask(p => {
      if (p.x === this.world.minX) this.ignite(p)
      else if (util.randomInt(100) < this.density) p.type =
    this.treeType
      else p.type = this.dirtType
    })

    this.burnedTrees = 0
    this.initialTrees = this.patches.filter(p => this.isTree(p)
    ).length
  }

  step() {
    this.fires.ask(p => {
      p.neighbors4.ask(n => {
        if (this.isTree(n)) this.ignite(n)
      })
      p.setBreed(this.embers)
    })
    this.fadeEmbers()
  }

  isTree(p) {
    return p.type === this.treeType
  }

  percentBurned() {
    return (this.burnedTrees / this.initialTrees) * 100
  }

  isDone() {
    return this.fires.length + this.embers.length === 0
  }

  ignite(p) {
    p.type = this.fireType
    p.setBreed(this.fires)
    this.burnedTrees++
  }

  fadeEmbers() {
    this.embers.ask(p => {
      const type = p.type
      const ix = this.patchTypes.indexOf(type)
      if (type === 'ember0') p.setBreed(this.patches)
      else p.type = this.patchTypes[ix + 1]
    })
  }
}
```

Similarly to the flocking simulation, the use of JavaScript results in several JavaScript-specific structures and core library functions the user needs to know in order to model AgentScript simulations. Although the code is simple and readable for JavaScript developers, its usage is limited to programmers only.

6.2.2 NetLogo

```
globals [
 initial-trees
 burned-trees
]

breed [fires fire]
breed [embers ember]

to setup
 clear-all
 set-default-shape turtles "square"
 ask patches with [(random-float 100) < density]
   [ set pcolor green ]
 ask patches with [pxcor = min-pxcor]
   [ ignite ]
 set initial-trees count patches with [pcolor = green]
 set burned-trees 0
 reset-ticks
end

to go
 if not any? turtles
```

```
    [ stop ]
  ask fires
    [ ask neighbors4 with [pcolor = green]
        [ ignite ]
      set breed embers ]
  fade-embers
  tick
end

to ignite
  sprout-fires 1
    [ set color red ]
  set pcolor black
  set burned-trees burned-trees + 1
end

to fade-embers
  ask embers
    [ set color color - 0.3
      if color < red - 3.5
        [ set pcolor color
          die ] ]
end
```

Although the NetLogo model of the forest fire simulation is clear and concise, the verbosity of the language is rather inconvenient and requires deeper understanding of the language abstractions and syntax.

6.2.3 AgentLang

```
define forest_size = 400;
define burn_time = 10;
define burn_prob = 0.08;

define IDLE = 1;
define BURN = 2;
define DEAD = 3;

agent tree forest_size {
  const fs = floor(sqrt(forest_size));

  const offset = 0;
  const size = height() - 2 * offset;
  const spacing = floor(size / fs);

  const w = spacing;
  const h = spacing;

  const x = offset + floor(index() % fs) * spacing;
  const y = offset + floor(index() / fs) * spacing;

  property trees = agents(tree);

  property tree_t =
    find_by_coordinates(trees, x, y - spacing);
  property tree_b =
    find_by_coordinates(trees, x, y + spacing);
  property tree_l =
    find_by_coordinates(trees, x - spacing, y);
  property tree_r =
    find_by_coordinates(trees, x + spacing, y);

  const ii = index() == round(forest_size / 2 + fs / 2);
  const initial_burn = if ii then BURN else IDLE;

  property burn_remaining: burn_time =
    if state == BURN then
    burn_remaining - 1
    else burn_time;

  property state: initial_burn =
    if state == IDLE and should_burn then
    BURN
    else
      if state == BURN then
      if burn_remaining == 0 then
      DEAD
      else BURN
    else state;

  property burn_t = tree_t.state == BURN otherwise false;
  property burn_b = tree_b.state == BURN otherwise false;
  property burn_l = tree_l.state == BURN otherwise false;
  property burn_r = tree_r.state == BURN otherwise false;

  property should_burn: false =
    (burn_t or burn_b or burn_l or burn_r)
    and prob(burn_prob);

  property c = if state == IDLE then
    rgb(48, 107, 64)
    else if state == BURN then
    rgb(230, 66, 41)
    else rgb(92, 53, 47);
}
```

The AgentLang forest fire simulation is quite straightforward. The agents try to find trees to the top, bottom, right and left side of themselves using the `find_by_coordinates` function. If any of these trees exists and is currently burning, the current tree starts to burn based on some probability. The code provides easy readability and understanding and is more convenient for scientists of non-technical background.

6.3 Discussion

Based on the above simulation examples, we can observe several differences between AgentScript, NetLogo and AgentLang.

First and foremost, AgentScript is built upon an existing programming language - JavaScript, which poses several advantages as well as disadvantages to its users. On the one hand, it does not require of the users to learn a new, domain-specific language, its semantics and usage. JavaScript developers can simply integrate

AgentScript into their application and model agent-based simulation with no problems. On the other hand, users with no prior knowledge of JavaScript need to learn the language and its whole functioning, even beyond the realm of agent-based modeling, in order to understand how to model an AgentScript simulation.

Although NetLogo provides its own language designed exclusively for agent-based modeling, its structure and semantics is rather challenging to understand and poses a steep learning curve for its users. The language is rather verbose and expressive and poses a challenge even for scientists with prior programming knowledge.

In AgentLang, the simulation is modeled using a sequence of properties for each agent model. Currently, the language does not offer any complex structures such as code blocks, loops or user-defined functions, each property has a strictly inline value. This is both readable and easily understandable for the users regardless of their technical background. The AgentLang programming language combines both the familiarity for developers, since its syntax resembles several widely-used programming languages and simple structure and semantics for newcomers, which is easily understandable even by users with no prior programming experience.

7 Limitations & Future Work

The following chapters describe the most significant and critical limitations of the current state of AgentLang and provide a list of possible future improvements.

7.1 Limitations

At the moment, AgentLang is a limited language serving primarily as a proof of concept of the new approach to agent-based modeling it aims to provide. Therefore, it has several significant limitations worth mentioning.

7.1.1 Performance

The most significant bottleneck of the AgentLang's interpreter is its performance. With the rising number of agents, the simulation slows down quadratically. The problem is most visible when there are numerous properties manipulating AgentList values, for those operations are costly. Since agents of the same type are evaluated the same way using the same model, with the rising number of agents also rises the number of agent instances in properties holding AgentList values. If we generate 10 agents, each of them needs to iterate over 9 other agents, resulting in 90 iterations. However, with only 10 times more agents, which is 100 total agents, the iteration count rises to 9900 (100 * 99), which is 110 times more iterations than with 10 agents.

Moreover, the evaluation of agents of the same type is handled greedily. That means that for each agent type, the entire model is reevaluated by the runtime module for each agent of that type. Although this problem seems easy to solve by caching mechanisms for example, it is defacto quite non-trivial. As far as caching is concerned, there are not many cases where caching is as straightforward as saving the intermediate results of expressions to a cache. This is due to the fact that in AgentLang, almost all properties have dependencies on other properties, whose values vary from agent to agent. Caching such values would thus be of no use to other agent instances.

The issue of low performance is also affected by the choice of TypeScript as the implementation language of the interpreter. TypeScript as such is an interpreted language, often slower than most compiled languages such as C++. The fact that the interpreter is integrated into a web-based environment running in a web browser slows down the overall performance even more, resulting in a chain of performance downfalls. Moreover, the choice of TypeScript does not allow for true parallelism. Evaluation of agent instances by running threads in parallel would help in improving the overall performance of the evaluation.

However, by experimenting with various simulations and models, it has been proven that AgentLang is able to handle simple to mid-sized simulations and a few hundreds of agents without slowing down significantly, not affecting performance to a noticable degree. With more complex simulations and higher agent volumes, however, the delay in evaluation of each step rises, rendering the simulation slower than desired.

7.1.2 Language Constructs

Currently, AgentLang is in a limited state of development, providing only the necessary set of built-in function and language constructs. Although one of the primary goals of AgentLang is simplicity and ease of use, the core library is currently not entirely sufficient for semantically more complex economical, sociological or organisational simulation needs. For instance, AgentLang does not support the generation of new agents and the deletion of existing agents during the runtime of the simulation. This would benefit the overall control over the simulations and provide the user with more flexibility. Moreover, AgentLang does not support parameterised functions with multi-statement bodies for reusing code blocks for more complex and frequently used calculations. Although this would introduce more complexity to the language, it would benefit the overall usability and extendability of the simulation models.

Although the current state of AgentLang is fully capable to model most simulations, with simulations of more complex nature, such as flows, organisations or economic markets, the lack of some complex structures and core functionality is visible and results in a more robust and less readable code base.

7.2 Future Work

This chapter describes the possible future improvements in AgentLang, such as performance optimisations or the support of additional useful language structures.

7.2.1 Performance Optimisations

One of the biggest pitfalls of the current state of AgentLang is its performance. There are numerous ways how to optimize the runtime part of the interpreter, so that it could handle greater numbers of agents and more complex mathematical calculations.

Parallel Computing

One of the most significant bottlenecks of the AgentLang's performance is the iterative evaluation of agent instances by the runtime module. The generation and evaluation of agents is implemented by a single `for` loop iterating through the number of agents defined in the source code.

```
for (let i = 0; i < count.value; i++) {
 const id = this.generateAgentId(declaration.identifier, i);
 this.evaluateObjectDeclaration(declaration, id);
}
```

For each agent, we generate its unique identifier and evaluate its entire property list. The performance of this sequential evaluation is proportional to the number of agents to be evaluated, rendering the simulation slower the more agents we generate.

One of the potential solutions to a more efficient and higher-performant runtime is to utilise parallelism. The workload of evaluating agents could be distributed among available threads, speeding up the evaluation process. This is mainly possible due to the fact that the runtime module evaluates a new generation of agents based on the values of agents from the previous step. Each agent could therefore be evaluated separately, since agents do not depend on each other during the runtime of the simulation. This results in a straightforward parallel algorithm, where each thread evaluates a portion of the agents.

Implementation in a Compiled Language

Another quite straightforward step to optimizing AgentLang's performance is choosing a compiled language for the interpreter's implementation. Compiled languages, when used correctly, tend to offer higher-performant programs.

Implementation in a compiled language is also tied to the parallel evaluation of agents, since compiled languages such as C++ or Java offer true parallelism mechanisms. Although Node.js provides parallelism using worker threads, the overhead of creating these workers and managing them is rather big, resulting in slower evaluation and less readable codebase.

7.2.2 Extended Core Functionality

At the current state of AgentLang's development, the language provides the minimal set of essential functions and language constructs to be capable of modeling a wide range of simulations. However, this set of supported functionalities is not sufficient for more complex economical or organisational simulations. These types of simulations require non-trivial complex mathematical calculations and data types. There are numerous aspects in which AgentLang can be extended of new functionality, described in the following chapters.

User-defined Functions

The most important language constructs AgentLang currently lacks are user-defined parameterised functions. In many cases, the user needs to perform the same calculations but on different sets of data. This is a typical use case of reusable code blocks with a custom set of parameters. Such functionality is especially important in economical or market simulations, where mathematics play a huge role. Therefore, a possible improvement to the AgentLang language would be a new `void` statement, which would allow for parameterised multi-line code blocks with return statements, aiming to endorse code reusability across agents.

Wider Core Library

Moreover, the core library of AgentLang's built-in set of functions is quite stripped. It consists of the essential mathematical functions to provide the necessary tools to perform calculations upon agents. Moreover, it has the necessary set of agent manipulation functions to provide agent interaction capabilities across different agent models. However, more complex calculations need to be done manually, resulting in bigger code base and thus slower performance. Therefore, a possible improvement to the AgentLang core library would be a new set of functions capable of handling frequently used and needed calculations internally, without the need to introduce multiple helper properties and overburden the runtime module with further calculations.

Conclusion

The main goal of AgentLang stated at the beginning of the thesis was to introduce a framework which provides a new approach to the agent-based modeling technique. Let us revisit the primary goals of AgentLang outlined in chapter 2.3.

1. provide a unified language with simple and straightforward syntax and structure
2. offer only the necessary, essential core library of features
3. allow for an alternative way of simulation modeling using a familiar approach in form of spreadsheet representation

First and foremost, the language simplicity was achieved by the straightforward structure of the language that aligns with the way humans tend to think about agent-based models. This was achieved by limiting the language to only the necessary declarations of agents, properties and global variables. The set of the language's syntactical constructs is also stripped down to the essential minimum necessary to be capable of modeling almost any simulation. Moreover, the core library provides all the fundamental mathematical and agent manipulation and interaction functions with human-readable names and natural usage. Last but not least, the spreadsheet interface built on top of the interpreter provides an easy and elegant way to modify the agent models and values of individual runtime agent instances. Although the AgentLang framework is limited at its current state of development, it is built in a way to be extensible to future work and improvements, allowing for the possibility to become a fully-usable agent-based modeling language.

Bibliography

1. Pangallo, Marco; Nadal, Jean-Pierre; Vignes, Annick. Residential income segregation: A behavioral model of the housing market. *Journal of Economic Behavior & Organization*. 2019, vol. 159, pp. 15–35.
2. *GAMA / Projects* [online]. [visited on 2024]. Available from: `https://gama-platform.org/wiki/Projects`.
3. Wilensky, Uri. *NetLogo - Official Website* [online]. 1999. [visited on 2024]. Available from: `https://ccl.northwestern.edu/netlogo/`.
4. *GAMA Platform - Official Website* [online]. 2024. [visited on 2024]. Available from: `https://gama-platform.org/`.
5. Densmore, Owen. *AgentScript - Official Website* [online]. 2012. [visited on 2024]. Available from: `https://agentscript.org/`.
6. Liu, Shaobo; Lo, Siuming; Ma, Jian; Wang, Weili. An agent-based microscopic pedestrian flow simulation model for pedestrian traffic problems. *IEEE Transactions on Intelligent Transportation Systems*. 2014, vol. 15, no. 3, pp. 992–1001.
7. Bonabeau, Eric. Agent-based modeling: Methods and techniques for simulating human systems. *Proceedings of the national academy of sciences*. 2002, vol. 99, no. suppl_3, pp. 7280–7287.
8. Dekker, AH. Using agent-based modelling to study organisational performance and cultural differences. In: *Proc. MODSIM 2003 International Congress on Modelling and Simulation*. 2003, pp. 1793–1798.
9. Nystrom, Robert. *Crafting interpreters*. Genever Benning, 2021.
10. Tomassetti, Gabriele. *A Guide to Parsing: Algorithms and Terminology* [online]. [visited on 2024]. Available from: `https://tomassetti.me/guide-parsing-algorithms-terminology/`.
11. Wilensky, Uri. *NetLogo - Fire* [online]. 1997. [visited on 2024]. Available from: `https://www.netlogoweb.org/launch#http://ccl.northwestern.edu/netlogo/models/models/Sample%20Models/Earth%20Science/Fire.nlogo`.
12. *GAMA - Movement of People* [online]. [visited on 2024]. Available from: `https://gama-platform.org/wiki/RoadTrafficModel_step3`.
13. *AgentScript - Flocking* [online]. [visited on 2024]. Available from: `https://agentscript.org/editor/?example=flock`.
14. *RxJS*. Available also from: `https://rxjs.dev/`.
15. *Next.js - Official Website*. Available also from: `https://nextjs.org/`.

List of Figures

A Attachments

A.1 AgentLang Syntax Grammar

```
program:
  | declaration
  | declaration program

declaration:
  | define_declaration
  | agent_declaration

define_declaration:
  | "define" identifier "=" define_value ";"

define_value:
  | numeric_literal
  | boolean_literal

agent_declaration:
  | "agent" identifier agent_count "{" "}"
  | "agent" identifier agent_count "{" agent_body "}"

agent_count:
  | numeric_literal
  | identifier

agent_body:
  | property_declaration
  | property_declaration agent_body

property_declaration:
  | "property" identifier "=" expression ";"
  | "property" identifier ":" expression "=" expression ";"
  | "const" identifier "=" expression ";"

expression:
  | "(" expression ")"
  | identifier
  | numeric_literal
  | boolean_literal
  | function_call
  | conditional_expression
  | relational_expression
  | binary_expression
  | unary_expression
  | logical_expression
  | otherwise_expression

identifier:
  | [a-zA-Z\_]+

numeric_literal:
  | ([0-9]+)|([0-9]+.[0-9]+)

boolean_literal: "true" | "false"
```

```

function_call:
  | identifier "(" ")"
  | identifier "(" argument_list ")"

argument_list:
  | argument
  | argument "," argument_list

argument:
  | expression
  | set_comprehension

set_comprehension:
  | expression "|" identifier "->" expression

conditional_expression:
  | "if" expression "then" expression "else" expression

relational_expression:
  | expression relational_operator expression

relational_operator: "==" | ">=" | "<=" | ">" | "<"

binary_expression:
  | expression binary_operator expression

binary_operator: "+" | "-" | "*" | "/" | "%"

unary_expression:
  | unary_operator identifier
  | numeric_unary_operator numeric_literal
  | boolean_unary_operator boolean_literal

unary_operator: "!" | "-"

logical_expression:
  | expression logical_operator expression

logical_operator: "and" | "or"

otherwise_expression:
  | expression "otherwise" expression
```