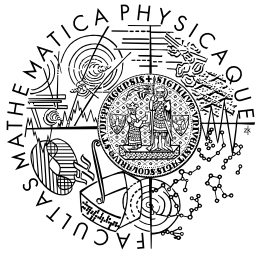


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Emanuel Buzek

Web Platform for Parallel Programming Tutorials

Department of Software Engineering

Supervisor of the master thesis:	RNDr. Martin Kruliš, Ph.D.
Study programme:	Computer Science
Study branch:	Software Systems

Prague 2017

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

Title: Web Platform for Parallel Programming Tutorials

Author: Emanuel Buzek

Department: Department of Software Engineering

Supervisor: RNDr. Martin Kruliš, Ph.D., Department of Software Engineering

Abstract: This thesis presents a novel approach to introducing programmers into parallel and distributed computing. The main objective of this work is to develop an online coding environment which contains tutorials in form of simple parallel programming tasks. The online application simulates and visualizes multiple agents which cooperate on a task in virtual environment. These agents are programmed in a custom procedural language similar to JavaScript. A significant part of this thesis focuses on the design of this language. The client-side compiler is built using tools similar to Bison and Flex. The parallel simulator supports different scheduling algorithms including lock-step mode simulating computation on a GPU. An important aspect of the platform is extensibility; therefore, the tutorials and the packages for the programming language can be added as plugins. The final part of this thesis is dedicated to the implementation of sample packages and tutorials which demonstrate that the key goals of this thesis have been accomplished.

Keywords: parallel concurrent programming tutorial compilers

Dedicated to students who wish to enter the world of parallel programming.

I would like to sincerely thank my thesis supervisor RNDr. Martin Kruliš, Ph.D. for inspiration and valuable feedback. I would also like to mention Johanne Kjærsgaard, Jiří Grunwald, Martin Kvěš, Vojtěch Žárský, and Benni Lichtner who tested the application. Special thanks go to Cedar Attanasio for editing. Finally, I thank my family and friends for supporting me.

Contents

Introduction

Parallel programming is a notoriously demanding discipline. Its purpose is to achieve computation speedup by splitting up workload to multiple tasks that can be processed simultaneously. The hardware necessary for parallel computing is cheap and available; these days common laptops and smartphones are equipped with multicore CPUs. Therefore, parallel programming is applicable in many areas of software engineering, and this makes the skills related to parallelism and concurrency highly relevant for wide range of programmers.

There are several factors that make the learning processes of parallel programming challenging.

First, humans are sequential thinkers. Arriving at a solution of a problem is typically described through a sequence of instructions which naturally corresponds with the paradigm of sequential algorithms. Of course, there are sequential algorithms that are extremely difficult to comprehend, but implementing a parallel solution to a given problem will always require *at least* the same level of understanding as the sequential approach.

Second, programmers are typically exposed to sequential programming first, and often accumulate years of experience before learning about parallelism. The first step of successful parallelization of a problem is to identify the potential for parallelism, which often requires the programmer to switch into different mindset. Discovering parallelism can be a challenge for a programmer who is only used to sequential way of thinking.

Correct and efficient implementation of a parallel solution can be a challenge even for experts. Inexperienced programmers will stumble into the pitfalls unique to concurrency, such as deadlocks, livelocks, and race conditions. Inherently unpredictable delays caused by cache misses, preemption and failures causes parallel programs to be practically non-deterministic[1], and that makes the debugging of concurrency-related issues even more difficult. Furthermore, efficient parallel implementations often require the programmer to have a deep knowledge of the underlying architecture.

Programming skills cannot be achieved without *practice*. Hands-on experience is essential for internalizing learned concepts, and seeing immediate results provides instant feedback (and gratification). The accessibility of interactive programming tutorials can be boosted by creating platforms that can visualize the executed program. Such beginner-oriented platforms also shield the users from complexity that is not essential for learning a particular concept.

Parallel programming is not different in this aspect–mastering it also requires intense hands-on experience. However, real-life parallel platforms (for example, MPI, CUDA, or Java Threads to name a few) have a steep learning curve[2]. Such technologies do not necessarily provide the easily approachable learning environment that allows the programmer to focus on learning *how to think* about parallelism without worrying about subtle technical details. Tackling precisely this problem is the main motivation behind this thesis.

Objectives

The objective of this thesis is to eliminate at least some of the difficulties that beginner programmers face when entering the world of parallel programming. This should be achieved by creating a platform for parallel programming tutorials, which will provide an intuitive, user-friendly learning environment.

Although the proposed platform simulates certain aspects of parallel systems, such as non-deterministic scheduling and variable instruction cost, the platform provides an abstraction that will allow the students to focus on parallel thinking without demanding the knowledge of the underlying architecture.

The following three areas are crucial for accomplishing the main objective:

1. **Programming language** suitable for solving the parallel programming tutorials is an essential component of the platform. Significant part of the thesis is focused on the design of the grammar and the compiler.

2. **The application** is designed and implemented to provide an accessible, user-friendly, and maintainable foundation for the programming tutorials.

3. **Example tutorials** are created to verify that the implemented platform meets the requirements. The by-product of this effort is a collection of reusable packages serving as a framework for the creation of other tutorials.

Outline

Chapter 1 contains a thorough analysis of the requirements for the platform. It discusses the aspects of the programming language, its runtime environment, and the required platform features. It also contains an overview of the existing work in the field.

Chapter 2 is dedicated to the details of the programming language designed for the tutorials. It contains the full language reference explaining all features of the language, and it explains some of the interesting issues related to the design of the grammar and parser.

The architecture of the proposed platform is described in Chapter 3. Finally, Chapter 4 explains the technical details of the implementation and presents the implemented tutorials together with the evaluation of preliminary user testing.

1. Analysis

This chapter discusses the fundamental features of the proposed web platform. It outlines the key functional requirements and evaluates the technical challenges that will need to be solved. Section 1.1 focuses on the nature of the parallel programming tutorials. The target platform for the proposed platform is discussed in Section 1.2. Section 1.3 discusses the programming language that will be used for writing the solutions to the tutorials. Sections 1.4 and 1.5 examine the compiler for the programming language, its runtime, and the simulator. Section 1.6 is dedicated to the extensibility of the platform. Finally, Section 1.7 provides an overview of the existing work in the field of parallel programming tutorials.

1.1 Parallel Programming Tutorials

The purpose of the programming tutorials is to demonstrate the challenges related to parallel computing and to teach proper techniques to address them. The aim of this section is not to present the exact content of the programming assignments; given the broad scope of this field, it must be possible to create and install new tutorials on the existing platform. The goal of this section is to discuss the fundamental concepts of parallel programming paradigms and how these concepts will be reflected in the tutorials. This will determine the basic primitives the tutorials will work with and the core functionality the platform will need to implement.

It is important to keep in mind the target audience of the tutorials. This project aims to teach parallel programming to users with no prior exposure to this field; however, certain basic knowledge of programming is expected. The most probable user group will consist of highschool or university students with basic knowledge of some procedural programming language. In the remainder of this thesis, the end-users completing the tutorials will be referred to as *students*.

The underlying process of parallel programming, regardless of platform, consists of identifying the potential for parallelism, finding the right algorithm structure, and implementing the algorithm with the available tools on the given platform[3]. In order to achieve some speedup from the serial approach, the programmer needs to identify the parallelizable part of the problem and divide it between the available processing elements. The tutorials should range from trivially parallelizable tasks to difficult ones, which are not only hard to parallelize, but also difficult to implement correctly. Students should become familiar with concepts like deadlock and race condition, and they should observe that splitting up tasks between more workers does not always lead to a speedup.

When designing a platform for experimenting with parallel algorithms, it is essential to decide on which level the parallelism will be implemented. This will determine the set of primitives in the programming language available to the students for implementing the tutorial solutions. Broadly, there are two options[4]. The first one entails writing actual parallel code in a language with support for that; the second option is to write serial programs and execute them simultaneously. With the first approach, the programmer uses API for spawning parallel tasks, such as threads, and uses synchronization primitives to ensure correct in-

teraction between these threads. This approach shifts most responsibilities to the programmer, by the same token, it gives the programmer greater control.

The second approach shifts the parallelism from the program to the platform; the programmer writes inherently serial code, and the platform is responsible for parallel (or pseudo-parallel) execution.

Most common programming languages provide mechanisms for writing parallel code; this could be an argument for choosing the first approach, because it would better prepare the students for real-life parallel programming platforms. However, writing parallel code is challenging for beginner programmers. The programming primitives are not easy to grasp, and the code becomes more complex and harder to debug. Also, seemingly straight-forward code can behave erratically due to mistakes leading to race conditions, which are especially frustrating and difficult to spot for inexperienced programmers.

The difficulties of writing and debugging parallel code are the main motivation for developing this platform. In order to remove some of the complexity of writing parallel code, the students will write serial code which will be simulated in parallel on multiple workers. Since the parallel workers are, from the perspective of the user, autonomous entities behaving according to the programmed algorithm, they are referred to as *agents*.[1] This approach should be intuitive to grasp for the students[5]. The agents will be visualized as some sort of independent programmable entities (robots, animals...) interacting in a virtual environment. The students will be presented with a clearly defined task that the agents need to collaboratively solve. The tasks should be simple and thought-provoking at the same time; they should feel more like puzzles or games than programming exercises. Challenges related to parallelism will naturally arise from the agents interacting in the shared space.

1.1.1 Tutorial Examples

The platform should provide sufficient means for creating tutorials which will help the students to become familiar with concepts related to parallelism, such as

- unit of execution, processing element
- speedup, Amdahl's law
- load balancing, work distribution, task
- shared resources, communication
- race condition, synchronization, deadlock

The examples of programming tutorials listed in this section are based on the following assumptions:

- the total number of agents is known (may be changed in the tutorial options)

[1]The proposed platform is not a Multi-Agent System, nor it is aimed at implementing Agent-Based Models

- the agents operate in a 2-dimensional grid of fixed size (grid dimensions may be changeable as part of the tutorial options)
- each field has a certain state (this can be represented, for instance, by the color of the field)
- each field on the grid may be occupied by at most one agent
- each agent has a unique identifier
- the agents provide an interface that can be accessed via calls from the program written by the student. The exact list of available function calls of the interface may vary for the specific agent type or tutorial; unless stated otherwise, the agents code can take advantage of the following function calls:
  - moving the agent on the grid
  - accessing the current coordinates of the agent
  - accessing and/or altering the field state (with certain limitations depending on the exact tutorial needs; the following tutorials assume agents can access the state of neighboring fields, but they can alter only the field they are currently on)
  - accessing the unique identifier of the agent and the total count of all agents
  - communication between agents; the agent programs are executed in a completely separated context, so they do not share any global variables; however, they can exchange data (with certain limitations, for example, only neighboring agents can exchange messages)
- each function call has an associated cost that can be either deterministic or random (it can also depend on the specific agent); this cost affects the scheduling of the parallel simulation
- the parallel execution is simulated using an algorithm that is transparent and known to the user

The following examples are intended to give a general idea about the parallel programming tutorials. These tutorial proposals should serve as foundation to specify the feature requirements of the platform. They will also serve as a foundation for evaluation of the implemented application.

Counting

Initially, the tiles of the grid are colored randomly (white and black). The goal is to use the agents to to count the number of black tiles. At the end, each agent must know the result, for example, each agent must print the total count to the console. This exercise should demonstrate simple division of work between multiple agents and parallel reduction.

Coloring

The grid is initially colored all black and the goal is to color all fields white. Initial positions of the agents are set randomly. The agents can color only the field they are currently standing on. This assignment assumes that the instruction cost is different for each agent; therefore, some agents work significantly faster than others. This property makes the task more interesting, because the agents have to allocate the work dynamically.

Row Coloring

This assignment assumes that the parallel simulation is done in lock-step; meaning in a single cycle, all agents execute the same instruction. The number of agents is the same as the number of rows of the grid. The grid is initially colored randomly, and all agents are positioned in the first grid column. Agents are supposed to traverse the grid from left to right, and in each step, they must color the current column with the color prevailing in that column. Assignments of this kind introduce students to parallel programming architectures using lock-step (GPUs).

This tutorial assumes a special function call *broadcast()*, which allows the agents to communicate using a fixed number of channels. On each channel, only one agent can send, the number of receiving agents is unlimited.

Masters and Workers

The task in this assignment is the same as in *Coloring*, but in this case, there are two types of agents–masters and workers. Masters can move fast and they can access the color of all fields, but they cannot alter the color of a field. Workers can color fields, but move slowly and may access only the color of its current position. The motivation behind this assignment is to simulate the communication and load balancing in a simple distributed system.

Mining

This assignment requires an extension that will provide additional objects the agents will interact with. These objects are visualized as gold nuggets on the grid. The API of the agents is also extended to provide additional functionality to carry these nuggets. The gold is concentrated at the end of narrow passages (gold mines); only one agent at a time may enter this passage, and it may pick up only one nugget. The goal is to move all the gold to a specified destination.

1.1.2 Requirements

From the analysis and examples listed above, it follows that the tutorial platform requires support for the following features:

Programmable Agents

The most fundamental idea of the platform are agents that can execute code written by the users in a parallel fashion. From the user perspective, the agents

are the units of execution; their code runs in an isolated context and they may interact with the application (and with each other) only through special function calls connected to the platform API.

Agent Types

In some tutorials agents need to execute the same code, the user writes just one program that is executed multiple times simultaneously. In other cases, agents need to be programmed differently. For instance, in the *Master–Worker* example, Master agents must execute different code than Workers. Moreover, these agents should have access to a different set of function calls and they need to be visualized differently. This suggests the need for introducing the concept of *Agent Types*; all agents of a given type execute the same code and have access to the same API.

Variable Number of Agents and Agent Types

The set of agent types is defined by the tutorial; although there should be practically no upper limit on the number of agent types, it will be typically less than 10 (for most tutorials, a single agent type will suffice.) Since the students must write a separate program for each agent type, it seems infeasible to have higher numbers of agent types, although it should be technically possible.

The number of agents for each agent type may be fixed for the given tutorial, or it can be adjusted by the user. This is essential so that the students can experiment by varying the number of workers and observing the effectiveness of their parallel algorithms. Typically, there will be several agents per agent type; it seems unlikely that a tutorial may require more than a hundred agents in total.

Extensible Agent API

The programmable interface available in the agent code should be extensible, reusable, and modular. The API should be encapsulated in some reusable modules; it may contain helper functions (for example for mathematical computations), and I/O functions that can trigger changes in the GUI of the application.

API for Tutorials

Tutorials should be in the form of plugins, so that they can be added to the platform without the need of re-building the application. In order to make the tutorial definitions concise, the platform must provide general API. The tutorial definition may list package extensions as its dependencies; these should be loaded prior to loading the tutorial itself.

Initial and Terminal State of Tutorial

The initial and terminal state can be arbitrarily defined for each tutorial; the terminal state needs to differentiate between successful and failing configurations.

Variable Instruction Cost

The instruction cost may vary not only for different instructions and function calls, but also for different agents. The instruction cost needs to be reflected by the scheduler and it will be used by the visualization packages (for example, the duration of animations must match the scheduling delay determined by the cost of an API call.)

Elapsed Time and Number of Instructions

The total elapsed time and total number of instructions performed should be recorded to calculate the speedup; these metrics should be displayed to the user.

Arbitrary Settings Dialogs

The tutorials may require a fairly complex settings dialog; this can be used, for example, to pick the number of agents or the dimensions of the grid. The dialog and its logic should be part of the tutorial definition.

Scheduling Policies

The parallel code execution needs to support different scheduling algorithms. Typically, the simulation will be done round-robin or randomly with taking into account the instruction cost. Agents of the same type executing identical code may be simulated in lock-step.

Debugging Features

The platform should provide tools for interactive debugging. The user should be able to place **breakpoints** in the agent code; the simulation should pause prior to executing the instruction matching the breakpoint, and the corresponding source code should be highlighted. Furthermore, the user should be able to **step** through the simulation manually. When stepping through the agent programs manually, the time intervals between instructions are irrelevant; however, the scheduling order of agents and instructions must remain the same. Finally, the debugger should allow watching **variable values** and indicate which agent is currently being simulated.

Error Handling

The user should be confronted only with user-friendly error messages when encountering compilation and runtime errors. However, since the package and tutorial extensions will be fairly complex, the error handling for loading the extensions must be robust as well.

1.2 Platform Discussion

The intention is to make the application easily accessible without platform restrictions. Although it may be used in parallel programming courses (i.e., in computer labs with computers running a specific OS where it could be installed

by an administrator) students should have the opportunity to use it outside of classroom on their own devices. Ideally, it should be available with little or no installation on all mainstream desktop operating systems.

Given the nature of the application, it is not intended to be used on devices with small screens (i.e., smart phones); writing code, debugging, and displaying a sophisticated visualization are not tasks suited for pocket devices. However, it is imaginable that users may want to try it out on their mobile phones; also, some users may wish to use the platform on a tablet. In other words, availability of the application on mobile OSs is not necessary, but it would be a valuable advantage.

The above considerations suggest that the proposed system is a good fit for a web application. Assuming the application is built using modern web technologies supporting cross-browser compatibility, it can be used in all modern browsers without the need of installing any extensions. There are several advantages of designing the platform as a web application:

- accessible on all mainstream operating systems (Windows, MacOS, and Linux)
- usable without installation and configuration
- easily maintainable; new versions and bug fixes are easily delivered to the users
- practical for adding new tutorials without the need to download and reinstall the application
- available in computer labs where users are not allowed to install new software

Regarding the use on mobile devices, web applications are usually less ergonomic than native mobile apps. However, the main target are desktop devices and mobile support is secondary. As a rich web application, the platform should be fully functional (although less comfortable to use) in browsers on mobile devices.

For future development, it is plausible to imagine adding support for user accounts, leader boards, and possibly also tools for creating and sharing new tutorials. For such network-dependent features, a web-based platform is also advantageous.

The above considerations are made with respect to the objective of this thesis. Desktop applications have unquestionable advantages for certain use cases; for example, desktop applications may have better performance (no browser overhead), and they can leverage direct access to system resources. Also, developing for a single desktop OS may be easier (and thus faster) without worrying about cross-browser compatibility. However, for the purpose of this project, these aspects are not so essential. Performance is not expected to be an issue[6], and cross-platform support is more important than development time.

To conclude, there seems to be enough reasons to design the platform as a web application (rather than a desktop application). There are no known technical issues that should make the implementation of such application infeasible.

1.3 Programming Language

The programming language used for the tutorial solutions is essential for an overall positive user experience and effective learning. It needs to provide sufficiently powerful tools for solving the exercises, at the same time, it needs to be simple and familiar, so that students can start using it quickly. This section discusses the most important aspects of the language, while the full language specification is provided in Chapter 2.

1.3.1 Language Requirements

In order to satisfy the objectives discussed in the previous sections, the programming language will ideally meet the following requirements:

- **Imperative**. The key idea of this project is that students will instruct the agents to solve certain task. There should be a very intuitive correspondence between the steps of the agents and the sequence of instructions in the program code; therefore the agents should be programmed using an imperative language.
- **Procedural**. The intended exercises fit well the procedural programming paradigm; they will be easily solved with a few subroutines and functions containing a series of computational steps. More complex language features, such as classes, are unnecessary and out of scope of this thesis.
- **Concise**. The language is not intended for large projects; it will be used for writing programs of tens or hundreds lines of code. Therefore, rather than robust it should be lightweight and concise; i.e. it could be weakly typed.
- **Familiar**. The objective of this project is to teach parallel programming concepts, and the programming language should make this task as easy as possible. Users familiar with mainstream C-like languages should be able to pickup this language effortlessly and focus solely on the parallel aspect of the exercise.
- **Easy debugging**. Debugging parallel code is already challenging, and this platform should make this part of the user experience as easy as possible.
- **Executable in browser**. Since the platform is intended to be accessed in a common web browser, it needs to be technologically feasible to translate the program to instructions that can be simulated in the virtual machine implemented in JavaScript. Compilation or other pre-processing (like sanitization) may be performed on a server side.

1.3.2 Language Considerations

A crucial decision of this project is whether to choose an existing language or design a new one. The above requirements are met by many existing, widely-known procedural languages. Considering how laborious it can be to design a

custom grammar, compiler, and simulator, the use of an existing language seems more appropriate.

There are many advantages in picking an existing language over designing a new one. First, students could be already familiar with, and learning this language could be useful for them in the future. Second, there could be an existing solution to execute the code directly in browser, such as Skulpt for Python (www.skulpt.org). The tutorials would only provide additional functions that the user code could invoke to interface with the platform environment. Calls to the API would have a defined cost to simulate the variable instruction cost. The agent code could be sanitized and its execution could be sandboxed to prevent any unwanted invocations.

Although choosing an existing language has several advantages, there are important reasons why it is necessary to design a custom compiler and simulator. Most importantly, the nature of the tutorials require precise control in execution of the code in an instruction-by-instruction manner. In order to support the required scheduling policies, the agent code must be compiled to atomic instructions that are interleaved according to the chosen scheduling algorithm.

Special attention requires the execution of the agent code in lock-step. In this scheduling mode, all agents perform identical instruction in every execution step. Conditional statements and irregular iteration blocks require special treatment (instructions masking) to satisfy this condition. There is no straight-forward way to achieve this behavior with existing tools to execute programs in browser; the only feasible way to implement lock-step execution is to parse and compile the user program to instructions, and then simulate these instructions in a custom-built VM supporting parallel execution in lock-step.

Variable instruction cost is another reason why it is necessary to compile the user code. It is not just external API calls that have some associated cost; arithmetic operations, invocations of user-declared procedures, and other elementary instructions (even loads and stores) performed by the agent code need to reflect the instruction cost defined for the particular agent type.

There are few more reasons why it is necessary to build a custom simulator for the user code, for instance, to compute precise metrics such as the total number of instructions executed by one or all agents. Implementing a debugger that will allow to place breakpoints in the code and step through the agent code step-by-step also requires absolute control of the code execution. It would be difficult, if not impossible, to achieve this by simply evaluating the programs in a black box.

It follows that it is necessary to parse and compile the user programs to a custom instruction set in order to have total control over the simulation. This does not require inventing a new language; an existing, freely available parser for JavaScript, C, or any other language could be used to output the desired code. However, to implement a full-fledged simulator for a real world programming language is complicated and possibly out of scope of this thesis. Moreover, for the purpose of the proposed tutorials, only a subset of features of the mentioned programming languages are necessary.

The most suitable solution appears to be building a parser and a compiler for *a subset* of an existing language; this approach will ensure that the language will seem familiar to many users yet it will provide implementation freedom to meet all the discussed requirements.

1.3.3 Parascript

The proposed programming language, referred to as *Parascript*, in many aspects resembles JavaScript. JavaScript was chosen as primary source of inspiration for the following reasons:

- JavaScript is very popular [7] [8], and due to its adoption for server [9], mobile [10], and desktop [11] application developments, its popularity is likely to grow further. According to the 2016 Developer Survey on StackOverflow.com, it is currently the most popular language [12].
- C-like syntax of common JS constructs will be easy to adopt for programmers knowing some of the mainstream languages, such as C/C++, C#, or Java.
- Soft typing in conjunction with Object and Array literal initializers is practical for concise implementations; functions can easily accept and return structured data without the need of declaring classes (as in Java) or structs (as in C).

Semantically, Parascript programs define the behavior of the agents. Since the tutorial solutions are intended to be relatively short (in the oder of hundreds of lines of code), each Parascript program consists of a single independent text unit. However, some tutorials will require different code for different agents, so a complete solution to a tutorial exercise might consist of more independent programs.

1.3.4 Deviations from JavaScript

Since Parascript is designed to fit a very specific purpose, many JavaScript features are intentionally left out. Based on the discussion in Section 1.3.2, Parascript is roughly framed as a subset of JavaScript. However, for the purposes of this project there is no reason to keep it strictly a subset–there is no requirement that a valid Parascript program should be a valid JavaScript program. Deviating from JavaScript as little as possible has only the benefit of keeping the language familiar to JavaScript programmers. Given the complexity of the Ecma Script standard considered here as the JavaScript language reference [13], the following list of deviations is by far not exhaustive. The important differences worth mentioning are:

Not Object-oriented

An important difference is that Parascript is not an **object-oriented** language; there is no **new** keyword, functions do not have the role of constructors, and there is no support for prototyping. However, it is worth noting that structured data types (Map and Array) are passed by reference like in JavaScript.[2] While in JavaScript all built-in types are objects (and thus contain properties and methods), Parascript primitive types do not contain properties and methods. Parascript is obviously missing all of the JavaScript global constructors, such as Object, Number, Date, Map, etc...

[2]Technically, maps and arrays are passed by value which is a reference

Absence of Closure

Parascript does not support **closure**, meaning that the local context of a function exists only during the time of execution of that function. This is partly related to the limitation of Parascript requiring that functions must be declared in the global scope, and that functions are not a data type, i.e. they cannot be assigned to a variable. Therefore, it is impossible to create a factory function that would return another function and form a closure.

Parascript also does not support **anonymous functions**; this limitation implies that functions cannot be passed as arguments, and this prevents the use of the classic JavaScript callback pattern. This is an intentional design decision; Parascript programs are intended to be synchronous and procedural. The students should focus on programming the agents in the simplest way possible and observe what happens when multiple synchronous programs are executed simultaneously.

Block-scoped Variables

Parascript requires **declaration of variables** and does not allow accidental creation of global variables by assigning undeclared identifiers, like in common implementations of JavaScript (JavaScript enforces declaration of variables in strict mode [14]). Local variables in Parascript declared using `var` block-scoped.

No Exceptions

Parascript also does not support **try...catch** error-handling mechanism. Given the nature of the expected Parascript programs, it is regarded as unnecessary to introduce such constructs; the tutorial solutions will be relatively short and they do not need to be exceptionally robust.

Undefined Merged with Null

The primitive value **undefined** is omitted in Parascript; unassigned values that are loaded to memory are initialized to `null` to avoid working with undefined values.

No Built-in Functions

The core implementation of Parascript does not contain any native functions; all functions are added by linking external packages. The packages may implement global functions like **isNaN()** or **typeof()** identically as in JavaScript, but it is not required by the language and it could differ from tutorial to tutorial. Also, functions calls in Parascript (both user-defined and external package functions) must match the number of arguments in the function declaration; this is enforced at compilation time.

Enforced Semicolons

Finally, Parascript **semicolons** are not optional (with the exception of semicolon after a block statement.) The compiler does not add any required semicolons;

however, since empty statements are legal, superfluous semicolons are tolerated.

1.3.5 Proposed Language Features

The following section summarizes the decisions made about the programming language, and outlines the key properties the proposed programming language should have.

C-like Control Statements

Common statements including `if...else`, `for`, and `while` should have the same syntax as in other C-like languages. For the purposes of this thesis is not necessary to adhere to the syntax of `for` statement completely and support all short-hand variations, such as `for(;;)`.

Primitive Data Types

The primitive data types needed for the tutorials are **null**, **boolean**, **number**, and **string**. These data types are immutable and passed by value.

Structured Data Types

Supported structured data types are **array** and **map**. These data types are mutable and passed as references. Array contains a fixed number of elements in a defined order, and map contains key-value pairs with unique keys.

Global and Local Variables

All variables must be declared and block-scoped. Variables declared in the top-most scope are referred to as global, and they are accessible in all nested scopes. However, global variables are still private for the given agent, i.e., variables cannot be shared among more execution contexts.

Dynamic Typing

Parascript type system closely resembles JavaScript; choosing a weak, dynamic type system seems appropriate for this project for a number of reasons. First, deviating in this aspect from JavaScript would make the two languages too different and Parascript would cease to be easily adoptable for JavaScript programmers. Second, it seems useful given the intended use of the language to support structured data types array and map as outlined above. Returning maps from functions is very practical, for example, when requiring to return structured data like coordinates. Similarly, for such short, puzzle-solving programs, it is practical to allow mixed-type elements in arrays. However, dynamic typing has the downside of possible runtime errors, notably invalid access (for example, indexing a non-array type or member access on a non-map type.) Errors caused by accidental implicit type conversions can also be confusing for beginner programmers. To conclude, the disadvantages of dynamic typing are acceptable given that error messages caused by runtime errors will be specific and clear.

Implicit conversions in Parascript will be implemented directly using implicit conversions in JavaScript. Although this introduces many quirks from the JavaScrip weak typing system, it seems more appropriate to keep this behavior consistent with JavaScript.

User-declared Functions

User functions must be declared using the `function` keyword in the global scope. They can accept zero or more arguments, and they may return a value. The return value of functions without an explicit return statement is `null`. The exact place and order in which functions are declared is irrelevant, because user function declarations are hoisted to the top of the program. This legalizes function calls before a function is actually declared, and by the same token, it enables recursive function calls between two functions.

Package Functions and Constants

Package functions are implemented in JavaScript and from the programmer's point of view, they are invoked just like user-defined functions. They need to be loaded before compilation since the declaration and argument count is checked at compile-time. It is left to the authors of Parapple packages to ensure that the package functions return values are of data types supported by Parascript.

Besides functions, packages may define constants. These are accessible like global variables; however, the compiler prevents assigning a value to an identifier belonging to a constant. Re-declaring (overloading) package functions and constants in user code should not be allowed.

1.4 Compiler

The application module responsible for translation of Parascript program to instruction list is called compiler. The core of this module is a parser generated from the grammar rules defining the Parascript language.

The compilation process consists of the following phases:

1. **Lexical analysis**. First, the raw source is stripped of unnecessary characters (for example, whitespace and comments) and processed a stream of tokens recognized by the grammar. This phase is also referred to as *tokenization*.

2. **Syntactic analysis**. Next, the tokens are parsed using the grammar rules to a *parse tree*. Unexpected tokens cause **syntax errors**.

3. **Code generation**. Finally, the parse tree is traversed to emit the executable code.

As input, the compiler takes the raw Parascript source code and the agent type definition specifying the API (package functions and constants.) This will enable the compiler to link the generated instructions with the actual implementation of the package functions and constant values.

The compiler needs to provide clear, user-friendly error messages to account for the fact that the users will be often programmers with little experience. Syntax errors should be highlighted in the source code and accompanied with a relevant error message.

1.4.1 Source Locations

In order to make possible precise highlighting of syntax error, source locations (i.e. row and line numbers for the first and the last character of each token) must be passed through all compilation phases. Source locations are also written to the emitted instructions, which is necessary to implement debugging and runtime error highlighting. Actually, it is necessary to construct a two-way mapping between the source code and the generated code. From one side, instructions are mapped to the locations in the source code; this is used to highlight the relevant line (or token) when the simulation hits an instructions with a breakpoint, or when a runtime error occurs. From the other side, the user needs to be able to place a breakpoint on a particular line of source code; to implement this, a reverse mapping associating code lines with instructions is built after the code generation phase finishes.

1.4.2 Client-side Compiling

A standard way to build a parser is using a parser generator tool; among the most commonly used ones is *GNU bison*, usually used together with Flex for tokenization. This tool generates from the context-free grammar a LALR parser–a program in C or C++ (Java is supported experimentally) that can parse the language specified by the grammar. In order to use Bison, the compilation would need to be done on server-side (the web application would execute the C/C++/Java program on the server). Alternatively, the parser generated by Bison could be transpiled to JavaScript. Tools like *emscripten*[3] can transpile C/C++ programs to JavaScript that can be ran directly in a mainstream browser. However, this approach is rather complicated; it would introduce many dependencies in the project, and a small change to the grammar would require re-generating the parser and then transpiling over to JavaScript. Moreover, such transpiled JavaScript module of a generated C program would be impractical to debug.

Compilation could be done on remote server, but it is worth considering to implement the compiler in browser. This approach brings several advantages; first, removing the network upload and download time would improve the user experience. Second, this approach decreases the load imposed on the central server. The second reason becomes relevant when the application is deployed publicly on a single server. Distributing the load on the machines of the users elegantly decreases the hardware requirements for the central server. However, implementing the parser in JavaScript manually (as opposed to generating it from grammar rules) would be tedious. Such parser would be also impossible to maintain, because the language grammar would be scattered in the program code.

[3] http://kripken.github.io/emscripten-site/

A convenient solution is to use a parser generator that supports targeting JavaScript; this approach would allow client-side compiling, yet the parser could be generated from a well-defined grammar. An existing tool satisfying these requirements is Jison (JavaScript port of Bison and Flex, [15]). Jison generates a JavaScript parser from bison-like grammar, and the parser can directly utilize other JavaScript modules, which makes it easy to integrate in a web application. After evaluating the advantages and disadvantages of the discussed methods, Jison was chosen as the most appropriate tool for the purposes of this project.

1.5 Simulator and Runtime

Parascirpt programs are compiled to a series of instructions (referred to as *Paracode*). Paracode is executed by an **executor**, which is a model of a Harvard-style stack machine (data and code are kept separately; programs are not intended to modify themselves). There will be as many executor instances (VMs) as there are agents; every agent has its own isolated context. The interleaving of the executors is orchestrated by a **scheduler**.

1.5.1 Instructions

Every instruction consists of the **instruction code**, **source location** and **parameters**. Instructions are executed atomically, and instructions from multiple agents are serialized by the scheduler; therefore, execution of instructions from different agents cannot interfere with each other. For example, a hypothetical instruction *INCREMENT* may consists of three steps: load, increment, and store. This could possibly lead to race conditions if instructions were not executed atomically. Atomic execution is obvious for the elementary instructions (like *JUMP* or *LOAD*), but instructions invoking package functions can trigger arbitrarily complex JavaScript computations.[4] Some actions (package calls performed withing the same simulation cycle) may be considered as conflicting actions in the context of the particular tutorial. This aspect is discussed in Section 1.5.3.

Each performed instruction will have an associated **instruction cost**. This cost, a non-negative number, is may depend not just on the instruction type, but it can vary for different agents. In fact, it can be defined as function and computed at runtime. The role of instruction cost is two-fold. It is proportional to the real time the instruction requires before the next instruction (of the same agent) may be executed. For example, package call triggering an animation must return instruction cost corresponding to the duration of the animation. The second role of instruction cost is the simulation of various issues from real-world parallel platforms; for example, cache misses and bus conflicts. The scheduler will take the instruction cost into account, therefore, agents will be penalized according to their accumulated cost. Some instructions (namely auxiliary instruction required for the stack machine) may have instruction cost set to 0; this will instruct the scheduler to perform another instruction immediately without delay.

[4] It is expected that a single package call will perform some calculations, change data in the shared memory, and perhaps trigger an animation

1.5.2 Simulation

The term *simulation* is used in this thesis for the process of the parallel execution of the agent programs. A simulation **step** is the execution of one instruction of a given agent. When the simulation runs freely, the scheduler periodically performs several simulation steps in an event referred to as **tick**. Typically, in a single tick the scheduler invokes one step for each agent. The speed of the simulation, which may be controlled by the user, determines the time between the ticks. The simulation can be also ran manually step-by-step (using the debugger); in this case, the simulation speed becomes irrelevant, but it is important that the algorithm determining the order of the steps remains the same.

1.5.3 Simulating Parallelism

Actual parallel execution of the agent programs is technically possible. It could be achieved with HTML5 Web Workers which are supported by modern web browsers [16]. Spawning scripts using Web Workers has several advantages; besides the possibility of boosting performance by using multiple CPU cores of the host machine, the scripts run in the background without blocking the user interface.

However, Web Workers (nor any other method for achieving native parallelism) are not useful for this project. Paradoxically, the agent programs should not run physically at the same time; the serial order of the instruction interleaving should be transparent for the students because it plays an important role in the tutorials. Performance is not also an issue; the number of agents is expected to be small enough that a single simulation tick will not present a significant performance issue for the browser. Since the simulation will be artificially slowed down by the period between the ticks (this can be implemented using `window.setInterval()`), there is no issue with blocking the user interface by long-running scripts.

Conflicting Operations

There is a possibility of two or more agents attempting to perform a conflicting operation in one simulation tick. For example, two agents might make a move to the same position on the grid. One possibility to solve this would be to first perform all instructions in a given cycle in *shadow mode*–changes (both to the VM and the GUI) would be performed only when no conflict within a cycle occurs; in case of a conflict, neither the conflicting instructions is performed. Although this approach fits better the paradigm that a given simulation tick is one atomic instant of reality, it would be more complex to implement, and perhaps also more frustrating to program with (agents would often deadlock, i.e. there would be no progress, because two agents would just keep attempting conflicting operations.)

The solution for conflicting operations chosen for this project is that *first agent succeeds*; in other words, a simulation tick is not considered to be an atomic event happening at one instant, instead, it is simply considered a sequence of independent serial steps. This approach seems also more intuitive for running the simulation in debug mode when agents are executed step-by-step manually and the concept of a simultaneous tick disappears.

It is important to keep in mind that elementary instructions are private to the agent; therefore, instructions like auxiliary stack operations cannot cause conflicts; the only instruction that can hypothetically cause a conflict is *package call* invoking an external package function. In the first version of this project, there will be no support for reverting actions of other agents within the same simulation cycle; therefore, the default behavior will be also that the first agent will succeed. Support for more sophisticated solutions of conflicting operations could be possibly added in the future.

1.5.4 Scheduling Policies

The scheduler must implement several scheduling policies, because some tutorial assignments will simulate a different parallel model requiring a specific scheduling algorithm. In some exercises the students will be encouraged to experiment with various scheduling algorithms.

- **Round robin**. In each simulation tick, the scheduler invokes all agents in an order that remains constant throughout the simulation. Agents are penalized according to the cost of their instructions; an agent with non-zero accumulated penalty is skipped and a fixed sum is deduced from its penalty.
- **Random**. Instruction cost is reflected just like in the round robin case, but the order of agents is random (in a given tick, all agents are invoked in a random order without repetition.) This scheduling policy simulates certain hardware aspects of parallel systems.
- **Lock-step**. This approach simulates parallel execution of code on GPUs. All agents must execute identical code, and they perform the same instruction in a given tick. This requires instruction masking to account for conditional statements and irregular loops. The instruction cost is irrelevant (all agents would be penalized equally), except to set the tick period to account for the real time needed for animations.

Special Considerations for Lock-step Scheduler

Tutorials designed for lock-step simulations have a special requirement for calls that need to behave as if they happened at the same time. For example, imagine a hypothetical call `broadcast(channel, mode, message)`, in which the `mode` (boolean) argument determines whether the agent is receiving or sending. In a given simulation tick, all agents execute the same call, and they may listen or transmit on a given channel. However, as explained before, the scheduler serializes all agents even within a single simulation tick. This means that some receiving agents may execute broadcast before the message was sent to the channel. The exact solution to this problem is an implementation detail; however, it is important to design the lock-step scheduler to support such functionality.

1.5.5 Runtime Error Handling

Program errors that happen during the simulation are referred to as **runtime errors**; due to the nature of the programming language (notably dynamic typing),

certain errors cannot be prevented by the compiler. For example, given the soft typing of variables, it is impossible to ensure that a function will return an array; it is perfectly legal to use functions that return sometimes an array, and in other cases a primitive value.

From the perspective of the platform, runtime errors can be divided to *expected* and *unexpected* errors. For the first group, the Parascript runtime explicitly checks certain properties and artificially throws an error if certain conditions are met. On the other hand, unexpected errors should never occur; however, there could be flaws in the platform implementation that could possibly lead to illegal states of the runtime. To ensure the application stability, the implementation must account for unhandled exceptions to allow for graceful error reporting (the simulation should pause rather than crash.)

Explicitly Checked Runtime Errors

- **Arrays and maps** are checked prior to accessing them using an index or a field reference; this is done to ensure that the variable name (from the parascript source) is included in the error message. Without checking the reference prior to access, the native JavaScript *Reference Error* message triggered by indexing an undefined variable would not contain the variable name declared by the user. Also, as mentioned in section 1.3.4, Parascript primitive values do not contain properties; therefore, it is illegal to index primitive values. While JavaScript allows expressions like `true[1]`, this causes a runtime error *Illegal Access* in Parascript.
- **Exceptions** from JavaScripts package functions must be handled like unexpected errors discussed above.
- **Return value** of package functions must contain the actual result and the instruction cost; failing so will result in runtime error. (This approach will make it easier for package developers to ensure that all return statements returning valid values.)

Ignored Runtime Errors

The following events may be considered runtime errors in common programming languages; however, in Parascript they are ignored, which is consistent with JavaScript.

- Array **out of bounds access** is legal and reading such value returns `null`.
- Similarly, reading an **undefined field of a map** results in `null`.
- Arithmetics is implemented in JavaScript, and no special checks are applied to prevent **arithmetic errors** (e.g. division by zero results in `infinity`; undefined operations such as `sqrt(-1)` result in `NaN`).

1.6 Extensibility: Packages and Tutorials

An important aspect of the platform is extensibility; adding new tutorials needs to be possible without rebuilding the application. This section focuses on the

requirements for the plugins, and the requirements for the API these plugins will interface with.

Each tutorial will require a slightly different framework that will facilitate visualizations and other support functions. Including such engine as a built-in module of the platform could be too restrictive. In order to make the platform as general as possible, all these support functions will be encapsulated in reusable plugins referred to as **packages**.

1.6.1 Agent Types and API

The Parascript programming language will not contain any native built-in functions or constants; these will be specified in the definition of the agent provided by the tutorial. The set of functions and constants is referred to as **Agent API**. It would be extremely impractical to define such API in every single tutorial; therefore, packages should be able to expose modular agent APIs that can be reused in many tutorials. These APIs can be further composed together to specify **agent types**; all agents of the same type share the same API. The idea behind this is to design such environment that the tutorial definitions can remain focused only on the actual tutorial assignment, whereas the engine for the tutorial environment stays encapsulated in the packages.

1.6.2 Packages

Packages contain implementations of the agent API functions and other support code used for creating the tutorial environment. The core of a package will be a JavaScript module exposing methods to other packages and tutorials. However, packages will likely require other resources, such as 3rd party JavaScript libraries, images, or HTML templates. These resources must be bundled and loaded together with the package code. In order to make packages as reusable as possible, it would be practical if some packages could be referenced as dependencies by other packages. Therefore, packages need to be loaded and initialized in correct order according to their dependencies.

Packages are allowed to maintain a state while they are loaded; however, auxiliary data structures should be kept within the package module in order to prevent collisions in the global name space. While allowing the packages to be stateful is necessary to support certain features (for example, visualization is stateful by its nature), every package must support **reset** and **cleanup**. The platform will invoke these calls when needed, but the correct implementation of these functions is left to the authors of the packages.

Package Function Caller

The package functions will often require the information about the caller. For example, a function `myId()` must return the ID of the agent that initiated this call. By convention, the first argument of all package functions will be the definition of the invoking agent. This parameter will be populated automatically by the executor; therefore, in the parascript program, the corresponding function call will always have one less argument than its native counterpart. From the

perspective of the programmer (student), the program will be running within the agent context.

1.6.3 Tutorials

Tutorials can be thought as **supersets of packages**; besides features specific to tutorials, they inherit all properties of packages. However, they cannot be used as dependencies by other tutorials or packages–there is always only one tutorial loaded at a time. A tutorial defining zero or more packages as its dependencies acts as the root of the extension dependency tree.

Apart from the properties required for packages, tutorials must include the following:

- **Options dialog**. This dialog will be displayed to the user before initializing the tutorial (and the whole IDE); This dialog should be probably a template bundled with the tutorial; however, some settings will be common for most tutorials (e.g. a toggle for choosing the scheduler type). For such common controls, tutorials should just specify whether they should be displayed or not.

- **Tutorial guide**. The tutorial must contain clear step-by-step information to instruct the students to complete the tutorial.

- **Agents and agent types**. In order to initialize the IDE, the platform needs to get the definition of the agent types (containing the agent API), and the list of all agents (each agent must have a unique ID, name, and an associated agent type.)

- **Tutorial state and assignment objective**. What makes each tutorial unique is the actual objective of the assignment the agent programs need to solve. To keep this as general as possible, the tutorial module should expose a function returning the current tutorial status code. The simulator will query this function after each step to check whether the tutorial has reached a terminal state. Note that the simulation can also finish in a failing state; this happens when the tutorial objective is not met, but the simulation cannot continue. Some tutorials could be also limited by time or the number of allowed instructions, or specify various conditions that must be true at every step of the simulation.

1.6.4 Platform API

The implementation of the packages will often need to interface with the platform to reflect the package calls in the user interface; such action should be done exclusively via an API designated for this purpose to avoid compatibility issues when upgrading the platform or packages. The most important features of the platform that the packages will need access are listed below.

- **Console** - packages may print messages to the system console; this may happen at runtime, but also in the loading and initialization phase. Logging should be also accessible for the programmer using Parascript functions;

therefore, some package should provide agent API including functions like `print(message)` that will print text to the console.

- **Scheduler** - certain events in the simulation may require the simulation to pause or halt, and this can be achieved by exposing an API for the scheduler. Another use of such API could be an on-click callback on the image of the agent; once this callback is invoked, the package code would forward the call to `focusAgent()` method on the simulator, which would for example focus the code window of this agent and highlight the current instruction.

- **Simulator canvas** - the platform needs to provide a way to insert the main element that the visualization packages will use as a drawing canvas.

1.7 Related Work

The existing work on the topic of parallel programming tutorials was examined from two perspectives: first, the analysis looks at existing coding tutorials on the web, without necessarily limiting the scope to tutorials on parallelism. As there are numerous articles and guides about programming that can be considered tutorials, attention is given especially to tutorials with interactive coding platforms; these could serve as inspiration for the technical solution of this project, or they could be extended to serve as basis for Parapple. Second, the analysis examines tutorials specifically on parallel programming.

Attention is also given to gamification of learning that are becoming increasingly popular especially for programming tutorials [17]. Research is being done also on interactive in-class games that should introduce concepts from distributed and parallel programming [18].

The conclusion of this section is that there is little or no intersection between interactive online platforms and tutorials on parallelism; therefore, it is worth building the proposed web platform for parallel programming tutorials.

1.7.1 Web Platforms for Coding Tutorials

Several well-known websites offer tutorials covering various topics of computer programming; Khan Academy (www.khanacademy.org) has a rich curriculum of computer science courses, and their coding tutorials contain a very interesting feature–video merged with an interactive coding editor. The student can change and run the code directly in the narrated video tutorial. Another website featuring broad range of coding tutorials is Code School (www.codeschool.com).

Some web platforms focus on the *gamification* of coding. Notable examples are CodeCombat (codecombat.com) and CodinGame (www.codingame.com). Both platforms feature graphically appealing game environment, and they offer several programming languages the students can use. In CodeCombat, the students learn basic programming topics by writing programs for characters in an RPG game. The code changes are reflected real-time, and the students can choose from Python and JavaScript. Similarly, CodinGame users solve short programming puzzles by writing a program in any of the 20 supported programming languages.

However, neither of these platforms features tutorials covering parallelism. More importantly, although they seem very extensible in terms of graphical representations of the game, they are not suited for processing of multiple programs in parallel; therefore, these platforms do not seem suitable for implementing the tutorials outlined in this chapter. Another elaborated puzzle-solving platform for programming exercises worth mentioning is CodeWars (www.codewars.com); it focuses on peer competitions, and allows users to create new challenges. Popular platform for learning Python and JavaScript through exercises in browser is CheckiO (checkio.org) and HackerRank (www.hackerrank.com); however, all these mentioned platforms focus on teaching a particular programming language, and not on the topic of parallel programming.

There are several online projects that focus on so called *code battles*; the main principle is that the users write a program that competes against programs form other users. For example, on Robocode[5] users program robot tanks in Java or .NET, and these tanks then engage in battles. Similar principle uses Fight Code (fightcodegame.com), where users create fighting robots by writing a JavaScript program. These battling platforms are especially interesting to look at in the context of this thesis for their touch on parallel programming. The robots act as independent programs; however, these games do not focus on solving parallel programming tasks.

1.7.2 Platforms for Parallel Programming Tutorials

Deadlock Empire (deadlockempire.github.io) is an online project that teaches concurrency topics by giving the user the role of a scheduler. In each tutorial, the user is presented with two or more short programs in C#. The user then steps through them concurrently by choosing which program executes next in every step. The goal is to make the programs crash–for example, cause a deadlock or enter a critical section in more threads simultaneously. While such platform is interesting for demonstrating some topics in concurrency, it does not really teach solving programming tasks with parallel approach. Also, it is not as extensible as the platform proposed in this thesis.

GPU Platforms

GPU programming used to be inaccessible without actually having a GPU; nVidia changed this with an online CUDA tutorial platform. This platform, fully available in browser, allows users to explore topics including OpenACC, Multi-GPU programming, and GPU Memory Optimizations [19].

Summary

There online programming tutorial platforms that provide user-friendly environment for beginner programmers. They prove that there is a demand for gamified programming tutorials, and that the concept of online programming games or puzzles works. However, none of these platforms focuses on parallelism nor provides sufficient means to create tutorials as proposed in this thesis. Therefore, it seems that the proposed project is original.

[5]Robocode battles are executed in a desktop simulator, so it is not an online platform

2. Programming Language

This chapter focuses on the programming language designed for solving the parallel programming tutorials. It starts by description of the fundamental language features in Section 2.1. Section 2.2 explains the details of extending the language with libraries that can provide functions and constants. Finally, Section 2.3 is dedicated to the grammar rules defining the language, and it discusses some of the challenges that had to be solved while designing the language.

The programming language is called **Parascript**. The name intentionally rhymes with JavaScript to suggest that it is a mutation of JavaScript tweaked for the purposes of the presented online platform named *Parapple*. The previous chapter discusses in detail the requirements for the language and the reasons for inventing a new language at all. The important point is that programmers with experience with JavaScript (or similar language with a C-like syntax) should be able to write simple programs in Parascript without struggling with the language syntax.

2.1 Language Reference

Parascript is a dynamically-typed procedural language. The sole purpose of Parascript programs is to represent the instructions for the Parapple agents. Such program is composed of a single text unit, which is complied together with the implicitly referenced packages to executable code. Although the user might write several programs (one per agent type) and execute them in parallel, these programs cannot directly reference each other.

2.1.1 Lexical conventions

Parascript programs consists of a series of statements. Statements must be terminated with semicolon; semicolon after a code block is optional. Empty statements are allowed, so the shortest valid Parascript program is just an empty string.

Whitespace and Indentation

Whitespace is used to delimit individual tokens; other than that, whitespace characters are ignored unless they appear within String literals. Newline and indentation characters are completely optional and are generally used to increase code readability.

Comments

Parascript supports line and multi-line comments (Listing 1.)

```
// Line comment

/*
 * This is a multi-line comment
 */
```

Listing 1: Comments in Parascript

Reserved keywords

The following keywords are reserved for special purposes in Parascript:
if, **else**, **var**, **for**, **while**, **function**, **return**, **break**, **continue**, **true**, **false**, **null**

2.1.2 Identifiers

Identifiers are used for naming functions, function parameters, variables, and properties. A valid identifier can be any string matching the following regular expression: `[_a-zA-Z][_a-zA-Z0-9]*`

Parascript identifiers are treated as case-sensitive. All identifiers must be unique within their scope.[1] Reserved keywords cannot be used as identifier names.

2.1.3 Variables

All variables in Parascript programs must be declared, and their name must be a valid identifier. Referencing an undeclared variable results in a compile-time error.

Variables are declared using the **var** statement:

```
var identifier;
```

Single **var** statement may contain multiple declarations separated with a comma. Optionally, declarations may contain initializer.

```
var variable1, variable2 = 42, variable3;
```

Scope of Variables

Parascript variables are block-scoped, where a *block* is a region of code enclosed in curly braces:

```
{
  // block of Code
  var foo; // Variable is visible only in this block
}
```

[1] Parascript identifiers are generally block-scoped, but the scoping details are explained for variables and functions separately

The root scope of the program is defined implicitly (without the need to enclose everything within a code block) and all identifiers defined there are considered **global**.[2] Constants provided by external packages (refer to Section 2.2 for details) can be referenced just like global variables, but they cannot be assigned.

Local variables are declared in function bodies. These variables are also block-scoped (the block can be the function body itself), but they are allocated in the function *local context* to enable nested function calls.

Function parameters act effectively as local variables declared in the function; although they are technically declared above the code block of the function body, their scope is the function body. Similarly, variables declared in the **`for`** statement initializer belong to the scope of the iteration body.

Finally, variables in Parascript are not hoisted, which means they cannot be referenced from the beginning of their scope, but only from the point of the relevant **`var`** statement onwards.

2.1.4 Data Types

The following data types are supported in Parasript: **Null**, **Boolean**, **Number**, **String**, **Array**, and **Map**. This section describes all types in detail together with literals that can be used to represent values of these types in the program code.

A notable difference from the JavaScript type system is the absence of the `Undefined` type. This was done deliberately for the sake of simplicity. Undefined was practically merged with null, meaning that all undefined values are always converted to null.

Parascript also does not treat functions as values. This implies that maps cannot map keys to functions.[3] Since Parascript has no notion of objects and functions cannot be assigned to properties, values of all types do not have methods. Useful utility methods of certain types, such as `Number.isNaN()` or `String.indexOf()`, need to be provided as functions by external packages.

Null

Null is a special type containing only one value–null. Unassigned variables are null by default; similarly, the return value of functions without a return statement (or with an empty return statement) is also null. The literal **`null`** is the most straight-forward way to represent a null value in the program code.

Boolean

Boolean is a logical type containing two binary values represented by literals **`true`** and **`false`**.

[2]Note that global variables are still private for the particular agent since all Parascript programs run in an isolated context

[3]For this reason, the type is called "map" rather than "object", although it uses the JavaScript object literal notation

Number

Number type holds numerical values; its range is determined by the underlying JavaScript representation of *Number*. Parascript does not support hexadecimal literals; only decimal literals may be used to represent numeric values. The rules for decimal literals conform to the ECMA Script Language Specification [13]. Examples are shown in Listing 2.

```
-0.0;    // 0
10e5;    // 10^5
-0.1e2; // -10
```

Listing 2: Examples of valid decimal literals

Parascript does not contain JavaScript global object properties **`Infinity`** and **`NaN`**; however, these values are valid results of some (arithmetic) expressions. Constants representing these values may be provided by packages to compensate for this. By the same token, constants may provide references to properties on the JavaScript `Number` objects such as `Number.MAX_VALUE` or `Number.NEGATIVE_INFINITY`.

String

The String holds string values that can be initialized using **string literals**. These literals can be either single-quoted or double-quoted[4] as illustrated in Listing 3.

```
"Double-quoted literals may contain text in 'quotes'";
'Single-quoted literals may contain text in "quotes"';
```

Listing 3: Examples of string literals

Array

Array type represents an ordered list of elements; the individual elements may be of various types. Arrays may also contain **`null`** values as well as nested arrays and maps.

Array literal is a comma-separated list of 0 or more expressions enclosed in brackets. An example is shown in Listing 4.

```
      [ true, 2, "three", [ ], null ]
```

Listing 4: Array Literal

Array elements are 0-indexed and may be accessed using bracket notation. Accessing an array out of its bounds does not cause a runtime error; reading

[4] At the time of writing, Parascript does not support escape sequences to escape quotes in strings

from a position greater than array length returns null, and writing to such position stretches the array to that length.

Arrays also contain a property `length` that can be accessed to get the number of elements. The property can be also set to directly change the array length as illustrated by Listing 5.

```
var array = [];        // Initialize empty array
print(array.length);   // Prints 0
// Stretch the array to [null, null, null]
array.length = 3;
// Empty the array
array.length = 0;
```

Listing 5: Length property of arrays

Map

Map is a data structure holding key-value pairs of arbitrary types. The value of a key stored in a map can be accessed in two ways:

- dot notation: `map.identifier`
- bracket notation: `map[expression]`

The dot notation is limited only for referencing string keys which are valid identifiers, whereas the bracket notation provides a way to get and set computed keys of arbitrary data types including null.[5]

Maps are instantiated using the **map literal** which syntax is modelled after the JavaScript object literal. The syntax is shown in Listing 6. In the map literal notation, keys must be valid identifiers. Keys in a map are unique; if a key is specified in the map literal repeatedly, it is mapped to the last assigned value.

```
{
  key1: true,
  key2: "two",
  key3: { },
  key4: null
}
```

Listing 6: Map literal example

Underlying JavaScript Data Types

For all of the listed data types there is corresponding underlying JavaScript data type used to store the value. Except for map, which is represented by JavaScript `Object`, the underlying data types have the same name. The native JavaScript data types are used for interfacing with package functions.

[5]Null keys are often useful in practice; the same can be hardly said about array and map keys, which are technically allowed as well

Determining Variable Types at Runtime

There is no built-in function `typeof()`[6] to determine the variable type; however, this function can be implemented trivially as a package function. This is possible due to the fact that there is a one-to-one mapping between types Parascript and their underlying types in JavaScript.

Type Conversion and Coercion Rules

Parascript is very lenient when it comes to data types, and when possible, values are converted implicitly at runtime as required by context. The coercion rules are determined by the underlying JavaScript types and the fact that binary operations are also implemented directly in JavaScript without any additional type checking. This makes the type coercion rules work the same as in JavaScript,[7] although these rule are known to be confusing [20] [21] as illustrated by example in Listing 7. However, tweaking these rules would probably result in even more surprising behavior for experienced JavaScript programmers.

```
1 // + operator may act as string concatenation
2 // so the number 222 is converted to a string
3 print("111" + 222); // prints 111222
4 // - operator expects number operands
5 // so the string "111" is converted to a number
6 print("111" - - 222); // prints 333
```

Listing 7: Example of type coercion in Parascript

Parascript does not provide any built-in functions for explicit type conversion, however, these functions can be trivially implemented as package functions.

Truthy and Falsy Values

Like in JavaScript [22], Parascript operates with the concept of *truthy* and *falsy* values. They play an important role in evaluation of boolean expression.

Parascript behavior depends on the JavaScript implementation; therefore, the values are defined as:

- **falsy**: `false`, `0`, `""`, `null`, `NaN`
- **truthy**: everything else

2.1.5 Expressions

A valid Parascript expression evaluates to a value belonging to one of the data types. Expressions are composed from operators and primary expressions.

[6] Actually, there are no built-in functions in Parascript at all.
[7] Except that undefined is treated as null

Primary Expressions

Primary expressions is one of:

1. **Literal** explicitly representing a value
2. **Identifier** referencing a declared variable
3. **Function call** effectively representing the return value of the function
4. **(Expression)** expression enclosed in parenthesis

Operators

Operators in Parascript can be roughly classified according to the number of operands they require–these are **unary**, **binary**, and **ternary**. Another classification could be based on the data types of their operands (and outcome). Finally, operators can be divided according to their associativity (left-to-right, right-to-left, and non-associative.) The following sections group the operators by their semantic meaning which roughly corresponds with the classification according to the data types. Furthermore, the operators are sorted with respect to their precedence; operators with higher precedence number are evaluated first.

The presented operator system is mimicking ECMA script in terms of operator semantics and precedence. Since the actual operations are implemented in JavaScript using the same operators, the same coercion rules apply.

Assignment Operators

Assignment operators listed in Table 2.1 assign value to the left operand, which must be assignable (in other words, it must be a identifier or member expression; this is checked at compilation time). These operators have the lowest precedence, and they are right-to-left associative, which can be exploited for chained assignments as in `a = b = 1;`. Assignment expressions always evaluate to the value that is assigned.

Precedence	Operator	Semantics	Associativity
1	`a = b`	assignment	right-to-left
	`a += b`	`a = a + b`	
	`a -= b`	`a = a - b`	
	`a *= b`	`a = a * b`	
	`a /= b`	`a = a / b`	

Table 2.1: Assignment operators

Conditional Operator

The only conditional operator listed in Table 2.2 is also called *ternary* since it is the only operator that requires three operands. The expression returns the second operand when the first expression is *truthy*, otherwise it returns the third.

Precedence	Operator	Associativity
2	`a ? b : c`	right-to-left

Table 2.2: Conditional operator

Logical Operators

Logical operators are listed in Table 2.3). Note that the logical NOT operator is also listed among the unary operators since ito has the same precedence as other unary operators. The binary operators return the actual value of one of the operands without necessarily converting it to boolean.

Short circuit evaluation is used for expressions with binary logical operators:

- If `a` is *truthy*, then `a || b` evaluates to `a` without evaluating `b`
- If `a` is *falsy*, then `a && b` evaluates to `a` without evaluating `b`

This is an important language feature that is used by JavaScript programmers as part of common programming patterns. The fact that the right-hand side of the expression is not evaluated at all when short circuit evaluation is applied is not just an optimization issue. Although short circuit evaluation does not affect the result of the logical expression, it has a significant effect on the program flow. For example, in expression `true || foo()`, the function `foo()` is not invoked at all. A common use of short circuit evaluation is to implement property access with an inline null reference check: the expression `foo && foo.property` does not throw a runtime error when `foo` is null because in that case, the right-hand side is not evaluated at all.

Precedence	Operator	Semantics	Associativity
3	`a || b`	OR	left-to-right
4	`a && b`	AND	left-to-right
13	`!a`	NOT	right-to-left

Table 2.3: Logical operators

Bitwise Operators

Parascript bitwise operators are in (Table 2.4); bit-shift operators are not implemented.

Precedence	Operator	Semantics	Associativity
5	`a | b`	OR	left-to-right
6	`a & b`	AND	left-to-right
7	`a ^ b`	XOR	right-to-left

Table 2.4: Bitwise operators

Comparison Operators

Comparison operators (Table 2.5) are comprised of logical equality and relational operators. Equality operators, like in JavaScript, have also a *strict version* which takes into account the data types of the operands.

All comparison operators are left-to-right associative and no special measures are taken for their chaining. In some languages,[8] the intuitively true expression `4 > 3 > 2` would be interpreted as `4 > 3 AND 3 > 2`; in Parascript, this expression is legal, although it is evaluated to false as `(4 > 3) > 3`. This is due to the fact that the outcome of the first parenthesis (true) is coerced to 1.

Precedence	Operator	Semantics	Associativity
8	`a == b`	equals	left-to-right
	`a != b`	not equals	left-to-right
	`a === b`	strict eq.	left-to-right
	`a !== b`	strict not eq.	left-to-right
9	`a < b`	less than	left-to-right
	`a <= b`	less than or eq.	left-to-right
	`a > b`	greater than	left-to-right
	`a >= b`	greater than or eq.	left-to-right

Table 2.5: Comparison operators

Arithmetic Operators

Arithmetic operators are listed in Table 2.6.

Precedence	Operator	Semantics	Associativity
10	`a + b`	addition	left-to-right
	`a - b`	subtraction	left-to-right
11	`a * b`	multiplication	left-to-right
	`a / b`	division	left-to-right
	`a % b`	modulo	left-to-right

Table 2.6: Arithmetic operators

Unary Operators

Unary operators are listed in Table 2.7. The postfix/prefix increment/decrement operators change the operand value; therefore, they require the operand to be *assignable*.[9] Since the result of these operators is the augmented value (which is unassignable, these operators are non-associative.

[8] For example Python

[9] Identifier referring to a variable

Precedence	Operator	Semantics	Associativity
12	`--a`	pre-decrement	n/a
	`++a`	pre-increment	n/a
	`a++`	post-decrement	n/a
	`a--`	post-increment	n/a
13	`+a`	unary +	right-to-left
	`-a`	unary −	right-to-left
	`!a`	logical NOT	left-to-right

Table 2.7: Unary operators

Member Access and Grouping Operators

Among the operators with the highest order of precedence listed Table 2.8 are member access operators (dot and bracket notation for accessing maps and arrays), and the grouping operator–parenthesis.

Precedence	Operator	Semantics	Associativity
14	`a.b`	member access	left-to-right
`a[b]`	computed member access	left-to-right	

Table 2.8: Member access and grouping operators

2.1.6 Program Elements

A valid Parascript program is a series of *program elements*, which can be either a *statement* or a *function declaration*.

2.1.7 Statements

A *Statement* can be one of:

- **Block statement**: `{ Statement1; ... }`
- **Selection statement**: **`if`**, **`if-else`**
- **Iteration statement**: **`for`**, **`while`**
- **Expression statement**: `expression;`
- **Return statement**: **`return;`**, **`return`** `expression;`
- **Break/continue statement**: **`break`**, **`continue`**

All statements except for the block statement must be terminated with a semicolon.

Selection Statement

IF and IF-ELSE statements with syntax shown in Listing 9 provide a basic mechanism for program flow control. The statement immediately after the IF clause is executed only when the expression in the IF clause evaluates to a *truthy* value. Otherwise, the statement of the ELSE clause is executed (if present). The syntax of the IF-ELSE statement introduces an ambiguity known as "the dangling else" explained in Section 2.3.

```
1 // IF statement
2 if (expression)
3   statement1;
4
5 // IF-ELSE statement
6 if (expression)
7   statement1;
8 else
9   statement2;
```

Listing 8: Syntax of IF and IF-ELSE statements

Iteration Statements

There are two iterations statements, **for** and **while**, shown in Listing 9.

```
1 for (exprStatement_initializer; expression_condition; expression_iteration)
2   statement;
3
4 while (expression_condition)
5   statement;
```

Listing 9: Syntax of iteration statements

This initializer is an expression statement, which may be either a declaration (**var** statement), an expression, or empty. The statement in the iteration loop body repeats as long as $\texttt{expression}_{condition}$; evaluates to a *truthy* value.

Statements **break;** and **continue;** may be used within iteration statements to break or resume the iteration of the nearest loop.

2.1.8 Functions

Functions provide a tool for simple reuse of Parascript code. They can be invoked recursively, and they may return a value with an optional **return** statement. Functions without a return statement implicitly return **null**. Functions may accept 0 or more parameters of any type; overloading functions is not possible, every function regardless of its parameters must have a unique name.

The **number of arguments** of a function declaration must be the same as the number of parameters provided in the function call; compile-time error is thrown if the argument count does not match.

User functions

Functions declared in the Agent Unit are referred to as *User functions*. These functions are visible only in the Parascript program in which they are declared. User functions must be declared in the root scope of a Parascript program. The function declaration syntax is shown in Listing 10.

```
function functionName(arg1, arg2, ... argN) {

  // function body

  // return statement is optional
  return expression;
}
```

Listing 10: Syntax of a function declaration

Function hoisting

The place where a function is declared is irrelevant (as long as it is in top-most scope). It is legal to invoke a function before it was declared, i.e., from a place in the program code above the function declaration. This is essential for allowing two functions to invoke each other. Similarly, it is legal to invoke a function from the function itself.

Package Functions

Package functions are essentially library functions implemented in JavaScript; they are described in Section 2.2.

2.2 API modules

Parascript programs are compiled together with *API modules* provided by external packages (refer to Section 3.3.) These modules are JavaScript objects containing nested objects **functions** and **constants**. The structure of an API module is in Listing 11. The function and constant names are determined by the key names. Note that function and constant names are not namespaced.[10] When multiple modules with the functions or constants of the same names are loaded, the identifier that is processed later is used.

[10]Nothing prevents module authors to use a naming convention, i.e., prefix the identifiers with package name

```
{
  name: "module name",
  doc: template,
  /* Functions */
  functions: {
    function1: function(agent) { ... },
    function2: function(agent, arg1, ...) { ... },
    ...
  },
  /* Constants */
  constants: {
    constant1: value,
    constant2: value,
    ...
  }
}
```

Listing 11: Example of an API object

Package functions are arbitrary JavaScript functions meeting the following requirements:

- **first argument** is required, and it is always populated with the definition of the calling agent; other arguments are optional
- **return value** must be an object with the following structure:

```
{
  cost: Number,
  result: Any
}
```

  `cost` determines the cost of the performed function call instruction. `result` is passed as the return value to the caller.

- **return without blocking**–package functions will often cause changes in the GUI which might take hundreds of milliseconds to execute. Such actions must be asynchronous, i.e., wrapped in `window.setTimeout()`.

These function provide an interface to invoke JavaScript code directly from Parascript programs. They are implemented in JavaScript. This code is assumed to be trustworthy and harmless; the code is executed without special security precautions such as sanitization, escaping or sandboxing. JavaScript functions may be added only through external packages and cannot be declared in the solution programs.

There are some inherent limitations to native functions: it is not possible to invoke Parascript functions from native functions. The source code of the package function is hidden to the user, and calls to package functions are, from

the user's perspective, atomic calls to a black box. This means that package functions cannot be stepped-trough with the Parascript debugger.

Package constants are essentially global read-only variables. They can be mapped to arbitrary values of any of the underlying supported data types. Mapping constants to objects (translated as maps in Parascript) is a good way to implement *enumerations* as suggested in Listing 12. The numeric color codes can be used in a Parascirpt programs like `COLOR.RED`.

```
{
  name: "colors",
  /* Constants */
  constants: {
    COLORS: {
      RED: 0xFF0000,
      GREEN: 0x00FF00,
      BLUE: 0x0000FF
    }
  }
}
```

Listing 12: Example: API constants as enumerations

2.3 Language Grammar

The Parascript language is defined as a LALR(1) context-free grammar description for the Jison parser generator. The grammar description can be found in the source code of the attached application.[11] This file combines both the lexer rules for tokenization, and the grammar rules in a bison-like syntax.

The following paragraphs explain some of the interesting issues that the Parascript grammar definition addresses.

2.3.1 Dangling else

The ambiguous grammar rules of the if-else statement lead to the well-known *dangling else* problem. Consider the grammar excerpt in Listing 13 showing the rules for IF statement.

[11] Located in /src/parascript/parser/parascript.jison

```
Statement
  : SelectionStatement
  | // Other statement productions
  ;

SelectionStatement
(1) : "IF" "(" Expression ")" Statement
(2) | "IF" "(" Expression ")" Statement "ELSE" Statement
    ;
```

Listing 13: Example: API constants as enumerations

The ambiguity of these grammar rules can be illustrated by the code in listing 14. Suppose there are two IF statements immediately following each other, followed by an ELSE clause.

```
1 if (expr1)
2 if (expr2)
3   statement1;
4 else
5   statement2;
```

Listing 14: Example of the dangling else problem

The indentation of the second IF statement is left out in order to highlight the ambiguity of this code. Without the use of curly braces to enclose the statements (which is not mandatory), it is unclear whether the ELSE clause belongs to the first or the second IF statement. However, like in other common C-like languages, this code is perfectly legal. It is a common convention that the dangling ELSE clause attaches to the nearest IF statement. The above code would be parsed as

```
"IF" "(" Expression ")"
"IF" "(" Expression ")" Statement "ELSE" Statement
```

When the parser reaches the ELSE token, it can either shift (using rule 2) or reduce (using rule 1). In the grammar of Parascript, this conflict is solved using *precedence rules*. The parser needs to be instructed that shifting the ELSE token has a higher priority. In Jison, this is done by defining precedence for some tokens and specifying the precedence in the conditional statement rule as in Listing 15. This removes the shift-reduce conflict and ensures that dangling ELSE clauses are attached correctly to the nearest IF.

```
// Precedence is increasing with order of declaration,
// "ELSE" has higher precedence than NO_ELSE
%left NO_ELSE
%left "ELSE"

%%

SelectionStatement
  : "IF" "(" Expression ")" Statement %prec NO_ELSE
  | "IF" "(" Expression ")" Statement "ELSE" Statement
  ;
```

Listing 15: Solving the dangling else problem

Another common way of solving the dangling else problem consists of duplicating the selection statement rule and splitting up the statement rule to a branch with else and without else. However, this approach unnecessarily clutters the grammar with duplicated rules.

2.3.2 Expression Statement and Map Literal

In Parascript, an expression is a valid statement (referred to as expression statement). It may seem counter-intuitive why all lines in Listing 16 are accepted as valid statements. One reason why it makes sense to structure the grammar in such way is to allow function calls as statements. Clearly, `myFunction();` needs to be a valid statement; at the same time, it is an expression. A conflict arises when considering the statement on line 4. It is unclear whether it should be parsed as an empty object literal (i.e. expression statement) or a block statement.

```
1  1;                  // 1
2  (1);                // 2
3  { key: "value" };   // 3
4  { };                // 4
```

Listing 16: Expression statement vs. map literal

Block statements are a necessary in many constructs as it is showed in Listing 17. On the other hand, a map literal *used as a statement* is unnecessary. To solve this issue, the grammar rules were adjusted so that a statement may be any expression except for map literal (relevant rules are in Listing 18).

```
1  while (true) {
2    // This is a perfectly legal empty block statement
3  }
```

Listing 17: Empty block statement

```
Statement
    : ExpressionStatement
    | // Other statements
    ;

// Expression MAY contain map literal
Expression
    : ExpressionNoMapLiteral
    | MapLiteral
    ;

// Expression statement MAY NOT contain map literal
ExpressionStatement
    : ";" // Empty expression statement also legal
    | VarStatement
    | ExpressionNoMapLiteral ";"
    ;

// Remaining production of expression
ExpressionNoMapLiteral
    : AssignmentExpression
    ;
```

Listing 18: Expression statement solution

Summary

The present version of the grammar implements all Parascript features described in the language reference (Section 2.1). It does not contain any shift-reduce conflicts, nor does it produce any warnings when processed by the parser generator. The JavaScript parser for the Parascript programming language generated by Jison proves stable and integrates well with the rest of the platform.

3. Architecture

This chapter provides solutions to the requirements and challenges presented in Chapter 1. It gives an overview of the design of the platform, and presents the general architectural decisions. The technical solutions are explained in detail to implementation level.

Modules

Figure 3.1 shows high-level module decomposition of the proposed platform. The largest module, named *Workbench*, represents the sort of *Integrated Development Environment* integrating together the most important submodules: *Simulator* (explained in detail in Section 3.1) contains the runtime for the Parascript language, and *Compiler* (Section 3.2) is responsible for translating of Parascript programs to executable code. The Simulator module also consists of UI components (the design of the GUI is presented in Section 3.4).

A separate module, referred to as *Tutorial Picker*, is responsible solely for browsing the available tutorials, and loading the tutorials and their dependencies (called *packages*). The structure of tutorials and packages is presented in Section 3.3.

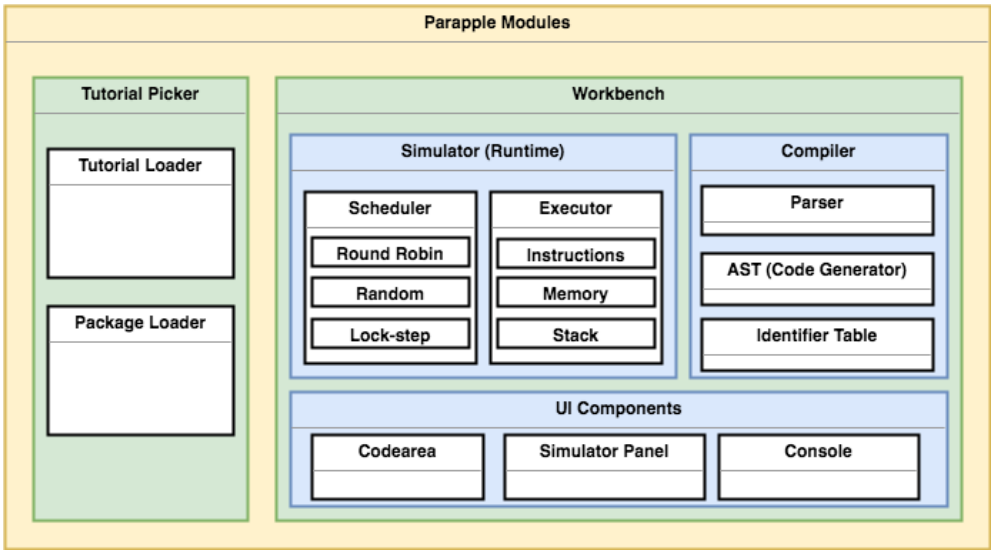


Figure 3.1: Module decomposition of the platform

3.1 Simulator

This section describes the architecture of the simulator module; this module enables the execution; debugging and visualization of the code written by the student in the context of the chosen programming tutorial.

The simulation process can be started by the user once all Parascript units (code for all agent types) have been successfully compiled. This simulation can be paused, reset and debugged from the simulator GUI. The simulation terminates when the simulator reaches a terminal state; that is, either the tutorial assignment

has been completed (this indicating by a return value from the tutorial instance which the simulator periodically queries), or the simulator cannot proceed with execution, i.e. all agent instances have been removed or have finished execution. Handling of runtime errors is described in detail at the end of this section.

3.1.1 Executor

Executor is the core submodule providing the runtime environment for compiled Parascript programs. It implements the virtual machine which simulates the execution of instructions produced by the compiler. Typically, there are multiple executor instances in a simulation, each corresponding to a particular agent. Separate executor instances ensure total isolation of of individual agent programs; the only way they can interact is through package calls that use shared data structures.

The executor processes a list of instructions (generated by compiler as explained in Section 3.2). The index position of an instruction in the list is referred as *address*.

The executor maintains an instruction pointer (IP) which corresponds to the address of the current instruction. When invoked by the scheduler, the executor performs the instruction and increases the IP by 1 (unless the instruction is a JUMP in which case it directly changes the IP to the JUMP address.)

Stacks

The stack machine utilizes three separate stacks:

1. **Main stack** contains only values teh stack machine uses for calculations; it is used for performing arithmetic operations, passing arguments to function calls, and return values

2. **Address stack** is used solely for addresses to return from function calls

3. **Memory stack** used for stacking od local contexts of function calls (encapsulated in the **Memory** module explained later)

The reasons for using more stacks rather than just one are purely practical. Splitting up addresses and values makes the executor easier debug (it avoids mixing up addresses and other values on the stack); also, it requires slightly less instructions in the final code (PUSH instruction for the return address prior to placing function parameters onto the main stack is not needed, because the return address can be placed on the address stack inside of the function call instruction).

The memory and address stacks exist only to enable separate local contexts for function calls; only the main stack is used for computations of the stack machine. These stacks could be easily implemented as one (function return address and function context are pushed/popped at the same time); keeping these stacks separated is purely an implementation decision.

Memory

The memory is a data structure where the executor stores values of variables.[1] It is effectively a key-value store where the keys are identifier names and the values are variable values.

Values in memory can be both primitive data types and complex types, arrays or maps. This implies that these values can be nested and reference each other; correct behavior of referencing pointers to complex data types is implemented using references to JavaScript objects on the level of the JavaScript engine the executor runs on. Table 3.1 shows an illustration of how map instances can reference each other. The first variable, `foo`, points to the same object in memory as `bar.key`. Similarly, arrays can also contain maps and arrays.

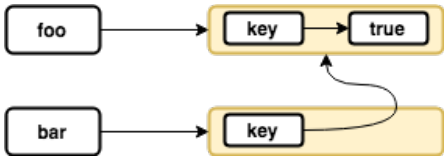
Command	Memory contents
`var foo = { key: true };`	
`var bar = { key: foo };`	

Table 3.1: Memory structure and complex data types

When entering a function, the executor creates a new *context* (map) and pushes it onto the memory stack; after exiting the function, the top-most context is popped and discarded. Local variables are always read from the top-most map, whereas global variable are retrieved from the first element in the stack. The first element in the memory stack can be called the global scope and it is created implicitly before the start of the simulation.

It should be stressed that this does not interfere with the fact that Parascript variables are *block-scoped*. However, the block visibility of identifiers is handled at compile time. The mechanism described above deals purely with the context of local variables in function calls *at runtime*. Local variables are also block-scoped from the perspective of the programmer.

Function Calls

The mechanics of function calls differ for local (user-defined) and package (external) functions. They both start by placing the fixed number of arguments onto the (main) stack. In the case of a package call, the arguments are simply taken from the stack, the executor invokes the function and places the return value back on the stack. The invocation of a local function requires more operations:

1. place arguments on the main stack
2. create new map and push it on the memory stack

[1] Memory also holds values of constants provided by external packages

3. push the current address on the address stack
4. jump to the subroutine address
5. subroutine starts by taking argument values from the stack and assigning them as local variables to the memory
6. return statement pushes the result on the main stack (void return statements return `null`)
7. pop the return address from the address stack and jump back

Runtime Errors

All runtime errors are handled by the executor by logging the error message, highlighting the relevant agent and code location, and pausing the simulation. All runtime errors are listed in Table 3.2.

Note that all calls to the external API are wrapped in `try...catch` to ensure that the simulator does not crash when exception in the API module occurs. Similarly, the scheduler catches unexpected runtime errors that may happen in the executor.

Runtime Error	Meaning
Null reference access	member access of a null value
Illegal member access	member access of a value that is not type array or map
Package function return value	external function call did not return object containing cost
Package function exception	external function call threw an exception
Unexpected runtime error	executor threw an exception or an assertion failed

Table 3.2: Runtime errors types

3.1.2 Instruction Set

The instructions executed by the stack machine contain the following information:

- **Code** uniquely identifying the instruction type
- **Function** that performs the operation (see below)
- **Cost function** used for calculating the instruction cost[2]
- **Source location** containing the line and column numbers of the first and last character of the relevant tokens from the source code
- **Parameters** specifying additional data for the instruction (for example, the address of a JUMP instruction)

[2]For package functions, the instruction cost is obtained from the result

- **Breakpoint** flag instructing the executor to pause before executing this instruction

The key property is the *function* which points to the actual piece of JavaScript code that performs all operations when the instruction is executed. All instruction of a given type point to the same function. The instruction types with the description of their functions are listed in Table 3.3.

3.1.3 Scheduler

The scheduler is in charge of the simulation progress by invoking executor instances. The exact order and timing of calling the executors varies with the chosen scheduling strategy. The instruction cycle is the single iteration of the scheduler; the actual duration of the instruction cycle in milliseconds is determined by a constant multiplied by a factor referred to as the **simulation speed**.

Two different scheduling strategies are supported by the **standard scheduler** and **lock-step scheduler**. The normal scheduler treats all executors independently, whereas lock-step scheduler executes in one instruction cycle the same instruction on all executors.

All schedulers with the following two concepts:

- **step: the execution of a single instruction of an executor**
- **tick: one step for all (active) executors is performed**

Simulation Speed and Instruction Cost

The instruction cost is a value associated with every instruction; for common instructions, it is a constant determined by the tutorial for each agent type (by default, all common instructions have cost 1). Regarding the calls to the external API, the cost is variable and it is determined from a component contained the return value of the call. The cost is an integer and it represents the number of cycles the instruction takes. If the cycle duration is t, an instruction of cost c will take approximately tc ms to execute. Instruction cost of 0 means that another instruction of that executor can be performed without additional delay.

The delay between instructions is not possible to implement precisely in browser; in the current implementation, the timing is approximated using JavaScript calls to `window.setTimeout()`. Precise timing is not so important; the scheduling order of instructions is still determined by the scheduler from the instruction cost. The purpose of the timing interval is just to represent the instruction cost to the user and thus make the simulation easier to follow for the user.

The value of the instruction cycle duration is essential for the external packages providing visualizations. Calls to these packages will often trigger animations and the duration of these animations must be set accordingly to ensure that the scheduler will not invoke another instruction for the particular agent before the animation completes.

Code	Parameters	Meaning
JUMP	address	jump to address
COND_JUMP	address	pop value from the stack; if the value is negative, jump to address
CALL	function name	invoke a external JavaScript function and push the return value on the stack
RETURN		return form user-defined function; push return expression or null on the stack; pop return address from address stack and jump there
DECLARE	identifier	create variable in the current scope
ASSIGN	identifier	peek a value from the stack and assign it to the specified variable
PUSH	identifier	get variable value from the current or global scope and push it onto the stack
POP		pop a value off the stack (and discard it)
PUSH_LITERAL	literal	push a value on the stack
ARRAY	length	pop the specified number of elements from the stack, construct an array with these elements, and push the array on the stack
MAP	list of keys	pop the specified number of elements from the stack, construct a map using the property names, and push the map on the stack
MEMBER	field	pop an array or map from the stack and accesses the specified field (pop computed index from the stack if field is null); push result on the stack
MEMBER_ASSIGN	field	assign value from the stack to field/index in map or array (accessing the array/map like in MEMBER)
BINARY_OP	operator	take two operands from the stack, perform binary operation specified by operator, and store the result on the stack
UNARY_OP	operator	take a single operand from the stack, performs the specified unary operation, and store the result on the stack

Table 3.3: The instruction set of Paracode

Simulation States

The scheduler is always in one of the states listed in Table 3.4. Figure 3.2 displays the state transitions that correspond either to user actions (dashed arrows) or program events that occur in one of the executors (solid arrows). The scheduler is initialized to READY when all agent programs are compiled successfully.

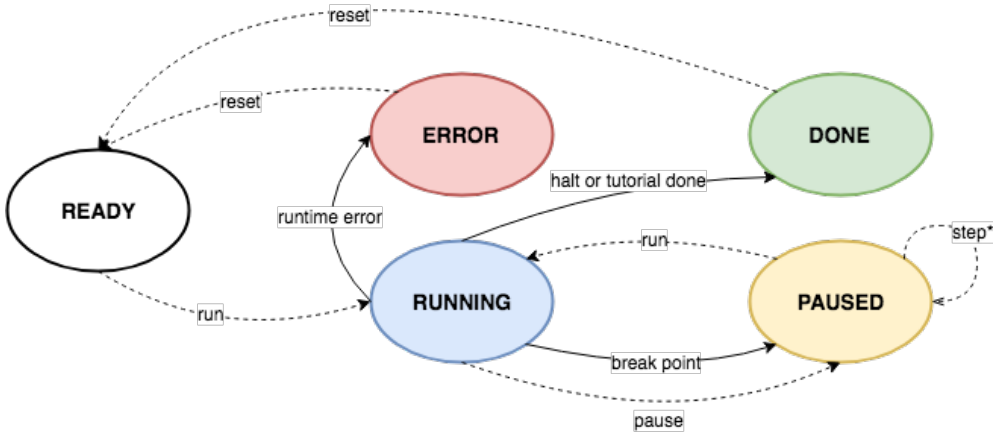


Figure 3.2: State transitions of the scheduler

State	Description
READY	**Initial state** indicating that all executors are at $IP = 0$
PAUSED	Simulation loop is paused (user action or an executor hit a breakpoint) and may be resumed
RUNNING	Simulation loop is running
DONE	**Final state** indicating that all programs have halted or the tutorial is done
ERROR	**Final state** indicating that a runtime error occurred

Table 3.4: Types of runtime errors

Standard Scheduler

The algorithm of the standard scheduler is outlined by the pseudocode in Listing 19. The instruction cost c is used as *a penalty*, so each executor must wait c cycles to be invoked again. The order in which the executors are iterated is deterministic for the **round robin scheduler** and random (without repetition) for the **random scheduler**. A sample execution of 4 agents is showed in Table 3.5.

```
function tick() {
    // Iterate over all active executors
    for (executor : executors) {
        // Decrement penalty (and keep it >= 0)
        executor.penalty = MAX(0, executor.penalty - 1);

        // Perform instructions while penalty is 0
        while (executor.penalty == 0) {
            executor.nextStep();
            executor.penalty = lastInstruction.cost;
        }
    }
}
```

Listing 19: Simplified algorithm of the standard scheduler

	Instruction cost			
t	Agent 1	Agent 2	Agent 3	Agent 4
0	1	2	10	3
1	1	x	x	x
3	1	2	x	x
4	1	x	x	1
5	1	2	x	1
6	1	x	x	2
7	1	2	x	x
8	1	x	x	1
9	1	2	x	1

Table 3.5: Example of instruction scheduling for 4 agents

3.1.4 Lock-step Scheduler

The lock-step scheduler requires significantly more complex approach then the standard one. The objective is to ensure that in every simulation step, all executors have equal IP, meaning all agents perform the same instruction. Although the use of this scheduler assumes that all executors are running the same code (in other words, there is just a single user program being simulated), individual executors may deviate from ech other due to *conditional jumps* generated by selection and iteration statements (**if**, **for** and **while**). The instruction set (refer to Table 3.3) contains only one instruction that performs a conditional jump: COND_JUMP.

The key insight to accomplish this behavior comes from the fact that conditional jumps, in all cases, *increase* the IP. Therefore, the executors that perform the conditional jump skip several instructions ahead. This set of executors can simply wait for the other executors to catch up. By *active* are meant executors that are currently making progress, and *waiting* refers to executors that are temporarily on hold until they are joined with the active set. Note that conditional

and iteration statements can be *nested*; therefore, there is not a single set of waiting executors, but it is a *stack of sets*. The general approach to this algorithm is outlined in Listing 20.

```
var activeSet; // set of active executors
// Perform next instruction on all active executors
instruction = nextStep(activeSet);
// instruction is now the last executed instruction
// Check last performed instruction
if (instruction.type == "COND_JUMP") {
    // Filter out executors that jumped
    jumpSet = selectJumped(activeSet);
    // Remove jumpSet from activeSet
    activeSet.removeAll(jumpSet);
    // Executors that jump will wait
    pushWaiting(jumpSet, instruction.address);
}

// handle else, break, and continue
if (instruction.type == "JUMP") {
    handleElseBreakContinue(instruction);
}
// Get current address
currentIP = getIP(activeSet);
// Check if there a waiting executors
if (hasWaiting(currentIP)) {
    // Join last waiting set and active executors
    activeSet.addAll(popWaiting());
}
```

Listing 20: Pseudocode of a simplified lock-step scheduler

The two calls for adding and retrieving the waiting sets, `pushWaiting()` and `popWaiting()`, are detailed in listing 21. The algorithm requires (due to the complications explained later) also associating the waiting sets with the *address* they are waiting on, in other words, the address the conditional jump pointed to. To satisfy this, there is also a *wait map* mapping address to the executor sets. The function `hasWaiting(address)` is then trivially implemented using this map–it simply checks whether the *address* is contained as a key.

```
function pushWaiting(set, address) {
    if (waitMap.hasKey(address)) {
        waitMap[address].addAll(set);
    } else {
        waitStack.push(set);
        waitMap[address] = set;
    }
}

function popWaiting() {
    waiting = waitStack.pop();
    address = getIP(waiting);
    waitMap.remove(address);
    return waiting;
}
```

Listing 21: Manipulation of waiting sets in lock-step

Complications of Lock-step

There are few issues that make the lock-step scheduler complicated:

1. **else** statements: the conditional jump makes the waiting executors wait at the first instruction of the ELSE clause; however, the active executors never step on this exact address since they must obviously jump over the ELSE clause

2. **break** and **continue** statements can be nested under IF statements; therefore, the active executors can be joined with a set on any index on the waiting stack

In both cases (represented as `handleElseBreakContinue()` in the pseudocode), the problematic JUMP instruction can be marked by the compiler as *ELSE*, *BREAK*, or *CONTINUE*. This will allow the scheduler to handle each of these cases explicitly.

For the ELSE jump, the scheduler simply swaps the latest waiting set with the active set. Figure 3.3 shows a sample execution of two executors, E_1 (red) and E_2 (blue). E_2 enters the IF clause, whereas E_1 jumps over to the ELSE clause. The special handling is required at the JUMP at address 3: the active set (containing just E_2) must be swapped with the waiting set. The join at address 5 happens normally as outlined in the pseudocode earlier.

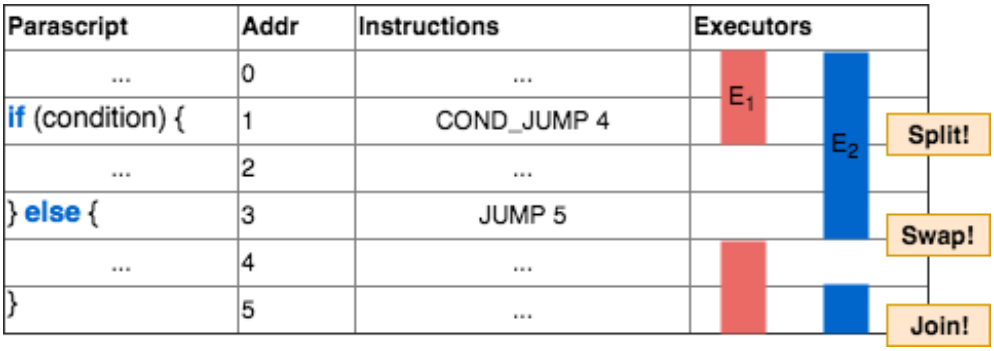


Figure 3.3: Possible execution of IF-ELSE in lock-step

The BREAK/CONTINUE jump requires even more attention. Although BREAK and CONTINUE statements always relate to the nearest iteration cycle (in Parascript), they can be nested in (possibly many) IF statements; therefore, when the current active executors perform BREAK or CONTINUE jump, they may well become a waiting set. More specifically, they are joined with the waiting set that mapped to the beginning or the end of the loop. To make this work for CONTINUE as well, the code in Listing 20 would have to push an empty waiting set with the address of the CONTINUE address.

The pseudocode in Listing 22 should clarify what exactly the lock-step scheduler does after the active executors perform a JUMP that is marked ELSE, BREAK or CONTINUE.

```
function handleElseBreakContinue(instruction) {
    if (instruction.isElse) {
        // Swap active and last waiting
        temp = activeSet
        activeSet = popWaiting();
        pushWaiting(temp, instruction.address);
    } else if (last.isBreak || last.isContinue) {
        pushWaiting(activeSet, last).address;
        activeSet = popWaiting();
    }
}
```

Listing 22: Special handling of JUMP in lock-step

Lock-step Callback

Finally, to fully simulate lock-step behavior, the scheduler needs to provide means for package calls to appear simultaneous within one simulation tick. For example, it should be possible to implement function for exchanging values between agents such as `broadcast(mode, value)` (mode determines whether the agent is sending or receiving). The lock-step scheduler will provide a hook for callbacks that packages can take advantage off. Such callbacks would be invoked at the end of every lock-step cycle (tick), and it enables the package to arbitrarily change the latest return values on the stacks of the active executors.

3.2 Compiler

The compiler module is responsible for translating Parascript programs to executable code. As input, it receives the source and the agent type definition containing the API modules with package functions and constants. The compiler uses a parser to build the parse tree which is then traversed to emit the output of the compiler–sequence of instructions referred to as *Paracode*.

3.2.1 Parser and Abstract Syntax Tree

The compilation process starts by parsing the raw source code into an abstract syntax tree (AST). This is accomplished with the parser generated form the grammar description of Parascript using the Jison tool. The result of the parsing phase is the root node of the AST.

Building the AST instead of emitting the instructions directly from the parser has several advantages. First, the grammar definition is decoupled from the code generation logic; it remains concise and easier to read. Each grammar rule is in charge of instantiating a node in the syntax tree. The only dependency of the generated parser is just on the JavaScript module exporting the AST nodes classes.

Another advantage of the syntax tree is that it simplifies the logic of the **code generation** process. The code is generated by recursively traversing the tree starting at the root; every visited node contributes to the final instruction list by either directly appending some instructions, or by recursively visiting its children. Most node types actually do both, and the logic is often nontrivial. Some instructions may require special checks (for example, a *Return AST node* must be nested under a *Function AST node*) and backward fixing of JUMP addresses.[3] A good example of this is the code generation function of *IF AST node* in Listing 23.

[3]Address refers to the index of the instruction in the instruction sequence

```
IfASTNode {
    append(instructionList) {
        // Recursively add the instructions
        // to evaluate the boolean expression
        this.condition.append(instructionList);
        var condJmp = new ConditionalJumpInstruction();
        instructionList.push(condJmp);

        // Recursively add IF body
        this.body.append(instructionList);
        // JUMP over ELSE (if any)
        if (this.elsebody) {
            var elseJmp = new JumpInstruction();
            instructionList.push(elseJmp);
        }
        // Fix the address of the ELSE jump
        condJmp.setAddress(instructionList.length);
        // ELSE block
        if (this.elsebody) {
            this.elsebody.append(instructionList);
            // Fix the address of the ELSE block jump
            elseJmp.setAddress(instructionList.length);
        }
    }
}
```

Listing 23: Instruction emitting for IF AST node

3.2.2 Identifier Table

The identifier table is a submodule of the compiler containing data structures and logic helping to keep track of the declared variables and functions. Besides the identifiers declared in the user code, the identifier table is initialized with the constants and functions defined for that particular agent type API modules. The identifier table has two main roles. First, it checks wether a referenced identifier is declared in the current scope; if not, an undeclared identifier error is thrown. Second, it serves as a lookup table for the JUMP addresses of user-declared functions, references to the package functions, and constant values.

Scopes of Identifiers

Identifiers declared in a Parascript program are block-scoped, and the compiler must keep track of the scopes of the identifiers to correctly signal undeclared or re-declared identifiers. Therefore, the identifier table provides methods `pushScope()` and `popScope()`. These methods are called `ListASTNode` representing code blocks before and after entering the nested scope.

Function Hoisting

The compiler performs an initial traversal of the AST to register all function names before it starts the main code generation phase. This ensures that the language is oblivious to the place and order of function declarations; functions can invoke themselves and call functions that are declared later. In the initial tree traversal, the addresses of the function declarations are unknown (the addresses become available only once the instructions of all function bodies are generated); for this reason, the compiler needs to iterate over the instruction list at the end to set the addresses to the relevant JUMP instructions.

3.2.3 Compile-time Errors

When the compiler encounters a problem, the compilation process stops and a special type of error object is thrown. The error always contains an error message, and usually also the source code location pointing to the line and position of the token causing this error. The error message is showed to the user and if the source location is present, the relevant line is highlighted. Stopping the compilation process after encountering the first error implies that only one compilation error at a time is reported.

There are two main sources of compile time errors. They can be either thrown directly from the parser; in that case, they are referred to as **parse errors**. Parse errors are caused when the parser is unable to build the parse tree from the source code given the rules defined in the grammar, like in the case of encountering an unexpected token. The other class of compile-time errors are **code generation errors**; these error are thrown by the AST in the code generation phase when the compiler performs checks of declared identifiers and matching argument counts of function calls. From the user perspective, these two classes of compile-time errors are indistinguishable.

3.3 Packages and Tutorials

The platform needs to provide sufficient means for adding functionality in three areas. First, there needs to be a way to add an implementation of the rendering engine that will enable to visualize the simulation. Second, the programming language needs support for loading external functions and constants that will provide powerful-enough programming tools for the students to write the tutorial solutions. Finally, there has to be a way to define the actual programming tutorials.

3.3.1 Structure of Packages

Packages are organized in a directory structure outlined in Listing 24. The *packages* directory is in the static folder of the application. This makes the downloading of package files straight-forward, and installation of new packages is done by simply copying them to this directory.

```
packages/
    packageA/
        index.json
        ...
    packageB/
        index.json
        ...
    ...
```

Listing 24: Package directory structure

Package Index File

Each package must contain a file called **index.json**. This file contains the metadata describing the package; namely the name of the file containing the main module, resources (list of file names) and dependencies (list of package names that this package depends on). An example index file is shown in Listing 25.

```
{
    "main": "main.js",
    "resources": [
        "template.html",
        "image.png",
        "sound.mp3",
        "module1.js",
        "module2.js"
    ],
    "dependencies": [ "foo", "bar" ],
    "modules": [ "module1.js", "module2.js" ]
}
```

Listing 25: Example index file of a package

The main file must contain JavaScript with a function (see example in listing 26) that is used as constructor to instantiate the package. The downloaded JavaScript modules are processed using `eval()`, it is expected that the eval call returns the function. The function is invoked by the package loader. This main instance exposes methods to be used by other packages. The package may require any number of resources–JavaScript files, HTML templates, or media files. These resources are downloaded prior to the initialization of the package and are passed to the constructor along with the instances of the packages listed as dependencies.

```
(function (application, dependencies, resources) {
    // Initialize the package instance here...
    console.log("Package initialized...");
})
```

Listing 26: Example of a package main module

3.3.2 Tutorials

Tutorials can be considered supersets of packages. Tutorials are also placed in the static directory of the application; the structure of the *tutorials* folder is outlined in Listing 27. The **intro** folder is compulsory for every tutorial; it contains the template (and its JavaScript) that is displayed in the tutorial selection modal window. This makes arbitrarily complex user interface for the tutorial settings possible.

```
tutorials/
    tutorialA/
        main.js
        index.json
        intro/
            index.html
            index.js
```

Listing 27: Tutorial directory structure

Required Interface for Tutorials

The tutorial main object must expose the following functions that are invoked on the tutorial instance by the platform.

- `getConfig(options)` - create agent types based on the tutorial options
- `initialize()` - setup the tutorial environment; the tutorial is responsible for calling initialize on its dependencies
- `reset()` - set tutorial to initial state; again, tutorial must call reset on its dependencies
- `getStatus()` - return status code indicating whether the tutorial is done

Tutorial Selection, Loading, and Initialization

The initialization of packages must be done in correct order; specifically, if package A depends on package B, package B must be initialized first and passed in the dependencies list to the constructor of package A.

In order to initialize all necessary dependencies for the selected tutorial, the *package loader* module must load all index files of all dependent packages, build a dependency graph, and finally download all packages and instantiate them in correct order. This is achieved by assuming that the dependency graph is a directed acyclic graph (DAG). The DAG is sorted topologically and the packages are downloaded and instantiated in reverse order.

The tutorial loading sequence is modelled in using a sequence diagram in Figure 3.4.

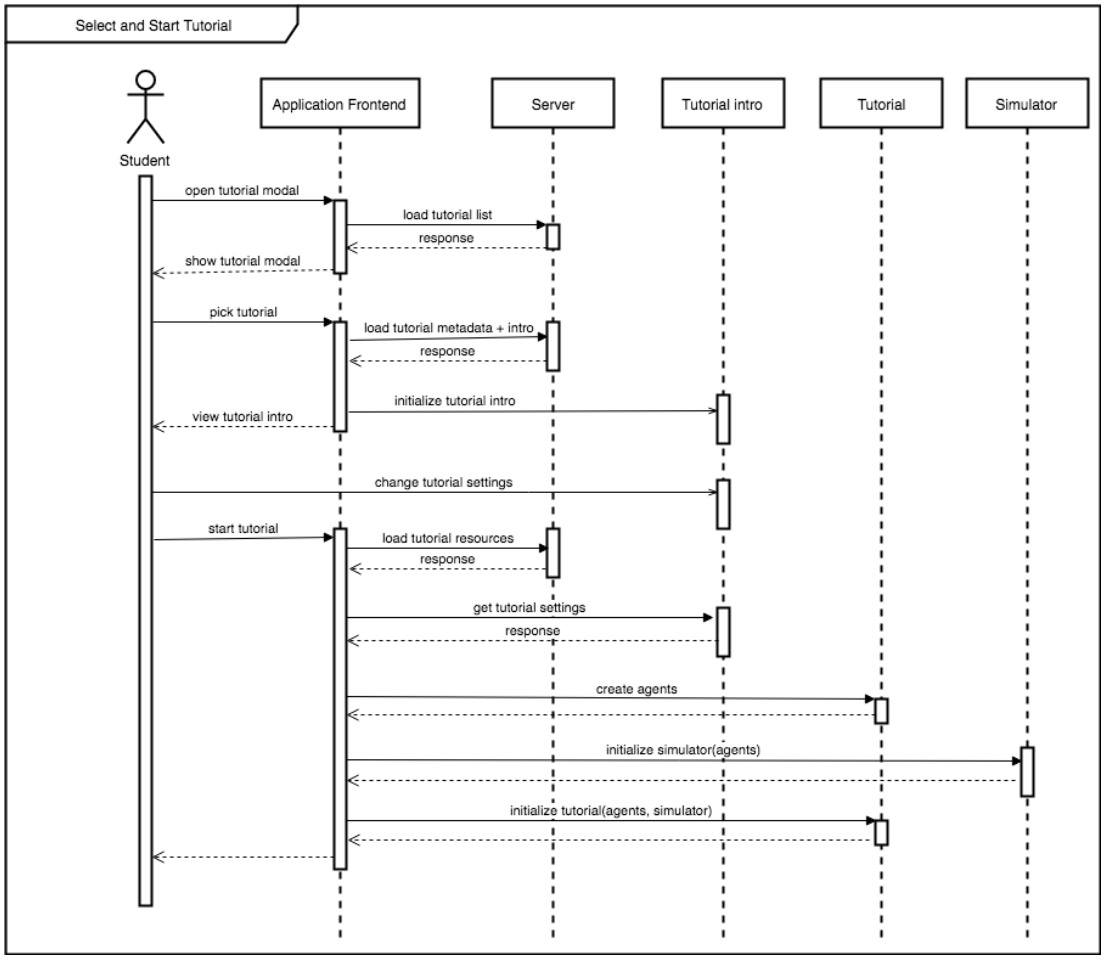


Figure 3.4: Sequence diagram of tutorial selection and loading

3.4 User Interface Design

This section briefly discusses the proposed graphical user interface (GUI). It is expected the the appearance of the implemented application will diverge from this design as new features will be added. The diagrams in this section served merely as a guideline for the implementation to help envision the layout of the application and to discuss the main GUI components.

3.4.1 Workbench

The main view of the application is called the *workbench*; it contains the core features of the application for completing the programming tutorials.

The workbench UI is displayed in Figure 3.5. The bottom part of the screen is occupied with a tab panel; for each agent type provided by the tutorial there is a tab with a *coding editor*. This implies that these coding tabs are initialized only once the tutorial is loaded. The right panel of the coding editor is dedicated to quick API reference summarizing the available programming interface for the particular agent type. There is also a tab with the main *console* showing log messages (from application, packages, and agents) and error messages.

The left panel of the main view is intended to show the assignment of the

tutorial. This assignment is going to be a template provided by the tutorial, so the content can be arbitrary. It is assumed to contain static HTML explaining the tutorial objective.

The central view is of the workbench is dedicated to the *simulator*. The bar on the top contains essential simulator controls (for example, to run, pause, and reset the simulation). The largest element is the *canvas* of the simulator; this is where the tutorial (with the help of packages) can visualize the simulation.

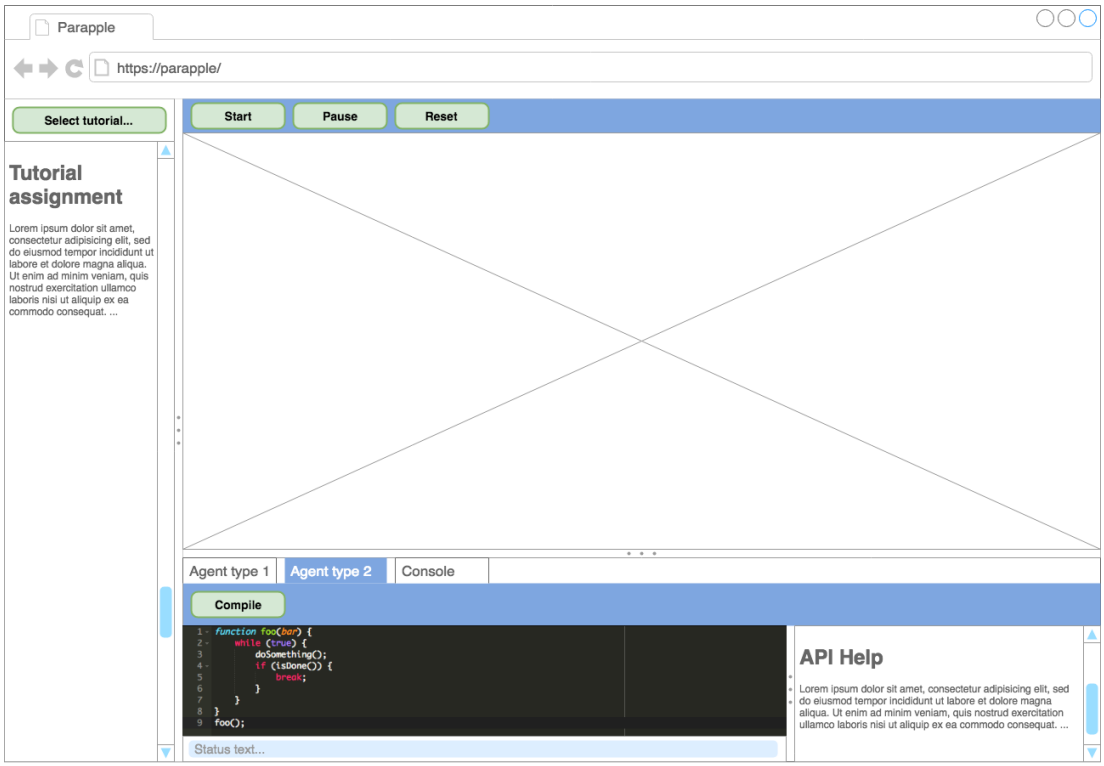


Figure 3.5: User interface of the Parapple workbench

3.4.2 Tutorial Picker

The UI component for browsing and loading tutorials is designed as a modal window. It is depicted in Figure 3.6. The modal window is divided vertically to a narrow left panel and the main panel. The left panel provides links (or similar controls) to all available tutorials. Choosing a tutorial in the list results in loading the tutorial intro template which is then shown in the main panel. The UI controls in the intro template of the tutorial can be arbitrary. Clicking on the *Start tutorial* button starts the tutorial loading sequence, closes the modal window, and initializes the workbench

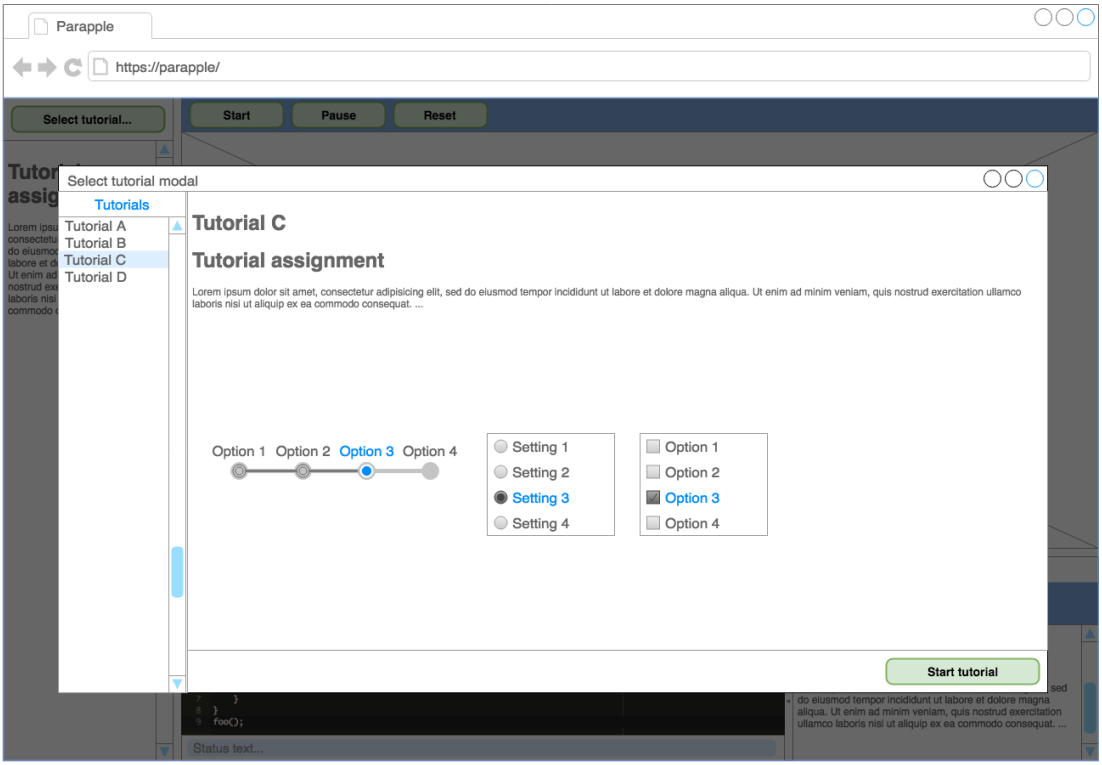


Figure 3.6: User interface of the tutorial picker

4. Implementation

This chapter contains high level description of the implementation of the platform. Section 4.1 explains key properties of the presented web application. It discusses the framework selection, and gives an overview of the structure of the application source code. Section 4.2 lists the implemented tutorials and packages. The platform was evaluated by preliminary user testing and the findings are presented in Section 4.3. Finally, Section 4.4 discusses the opportunities for improvements and further development.

The software project is called *Parapple*, and its current version is attached as an electronic appendix to this thesis. The project is maintained in GitLab (gitlab.com) which integrates the project *git*[1] repository (containing the source code of the application), and an issue tracker.

User documentation is an important part of the application; therefore, it is part of the electronic attachment. Documentation for developers who should modify the application, or create new packages and tutorials, is attached as Appendix A.

4.1 Web Application

It was determined in the first chapter that the application will be developed as a web application. Given the nature of the main use case (programming agents and running parallel simulations) it is clear that the application requires a complex and interactive user interface. Execution of the agent code invoking Parascript packages (which are implemented in JavaScript) dictates that the application frontend is powered by JavaScript.

Single-page Application

The nature of the proposed platform seems a good fit to be implemented as a single-page application (SPA). There is no set definition of the term *SPA*, but it is commonly used for websites that do not perform a full page reload for (most) actions [23]; instead, SPAs load the page only once, and user actions are handled completely on client side or processed via AJAX requests on the server. Common property of SPAs is that they also perform URL routing in browser. The frontend[2] of SPA requires non-trivial program logic typically implemented in JavaScript.[3]

There are certain **disadvantages of SPAs** [24] that should not be overlooked while choosing this design approach:

1. **Refresh behavior**: refreshing an SPA (with the browser refresh button) typically leads to loss of state which is not encoded in the URL; this may be confusing for users and can potentially lead to loss of work.

[1] A common version control system, git-scm.com

[2] Part of the application running in the web browser

[3] The implementation can be done in another language (i.e., TypeScript), and then transpiled to JavaScript

2. **Unsaved changes**: SPAs often lead to accumulating unsaved work in the browsers; accidentally closing the browser window (or browser crash) can potentially lead to loss of work.

3. **Page load**: an SPA has to load at least the core JavaScript bundles to initialize; this can cause longer load times. However, repeated loads of the application are faster due to caching.

4. **Memory leaks**: SPAs are prone to memory leaks just like normal websites with JavaScript, but lack of page reloads (which allows the browser to free all references) makes this more symptomatic.

5. **Routing and History**: performing navigation (changing URL in address bar) in JavaScript bypasses the native browser behavior, and it can break the expected back button behavior.

6. **SEO and Analytics**: as the final HTML displayed to user is only composed in browser based on the user interaction, search engine crawlers may not be able to index website content. This issue is not regarded as particularly important for the discussed web application. Separate, SEO-optimized page(s) with information about the project could easily address this if needed.

Although some of the disadvantages of SPAs are relevant to the Parapple project as discussed above, the SPA approach still seems appropriate for accomplishing the project objectives. The reasons for implementing the platform as SPA are:

- **Routing** - the application does not require complicated URL routing; it is mostly comprised from one main view (the workbench); this also means that the back button behavior discussed above is not particularly important for this application

- **Interactivity** - SPA paradigm is suitable for implementing highly interactive GUI; the application should provide programming and debugging experience similar to desktop IDEs and that will inevitably require client-side scripting

- **Rapid responses** - it seems reasonable to sacrifice *page load time* for more rapid responses; the advantage of SPA is that the application is loaded in advance and individual user actions are fast since there is no or very little communication with the server

The problem of *unsaved changes* can be also addressed for SPA; this is discussed at the end of the chapter as an opportunity for future development. The decision of framing the platform as SPA was reflected while choosing the main frontend framework (discussed later).

No Backend Requirements

While the proposed application requires a sophisticated frontend, there are no features that would require a complex backend part of the application. As there is no need for a persistance layer, all logic can be implemented on the client side. This allows the server part of the application to be completely stateless, and it can be served as a static web application. While the initial implementation does not require application backend, the possibilities for future enhancements (listed at the end of this chapter) were known from the beginning of the development; therefore, the application is built with the possibility of adding a backend.

4.1.1 Web Technologies

A mature frontend framework is essential for a successful development of SPA. There are several reasons that motivate for the use of such framework:

- writing **less code** since frameworks provide general purpose skeleton
- ensuring cross-browser **compatibility** by providing an abstraction over browser implementations
- improving application **maintainability** by imposing application structure and coding conventions

There are many admissible JavaScript frameworks that would fit the purpose of this application, and choosing between them often depends on the personal preference of developers. The frameworks considered for this project are AngularJS (angularjs.org) and its successor Angular (angular.io), Ember (emberjs.com), React (facebook.github.io/react), and, the framework that was finally selected, vue.js (vuejs.org).

Vue Framework

Selecting one framework does not imply that the other ones would not fit the purpose as well. Vue.js compares well to the other frameworks [25], and it has the following benefits that led to its adoption for this project:

- **Performance**: Vue.js offers very good rendering times due to its lightweight virtual DOM implementation [26]
- **Open-source**
- **Active community**: although relatively new (initial release in 2014), Vue.js has a vibrant community and its adoption is rapidly increasing; there are also many freely available and actively developed 3rd party Vue components
- **Appropriate tooling**: there is a special *Chrome devtools* extension for debugging Vue.js applications in Google Chrome, and a command-line tool (`vue-cli`) for generating application scaffolding

- **Gentle learning curve**: part of Vue.js philosophy is *approachability*; it requires the programmer to write as little code as possible; getting started with a Vue application requires very little configuration and bootstrapping; also, it is in some aspects similar to widely-adopted AngularJS

Application Bundling

Parapple is built using the *vue/webpack* boiler plate; this application setup provides the basic configuration of the following tools:

- **webpack** - *module bundler* running on Node.js[4] that processes all application resources and dependencies into *bundles* that can be loaded in web browser (webpack.js.org)
- **babel** - transpiles ES6 JavaScript modules to ES5
- **uglify** - minifies and optimizes JavaScript modules to minimize file size
- **vue loader** - enables single-file Vue components loading

With this setup, the application can be launched in *dev mode* (without transpiling, minification, and bundling) or built for production use. The production distribution contains essentially just a single static `index.html` file and several JavaScript modules (and static resources).

Single-file Vue Components

Vue.js provides support for interesting approach to code organization. User interface modules, referred to as components, can be fully contained in a single `.vue` file. This file includes the HTML template, JavaScript controller and CSS rules. The CSS rules are (optionally) *scoped* to make them applicable only for the particular component. Scattering of templates, JavaScript and CSS is a common problem of other frameworks that typically encourage placing these resources in separate source folders.

4.1.2 Project structure

The important components and modules of the application are organized to the following source folders: **components**, **parascript**, and **runtime**. The following paragraphs briefly explain the responsibilities of each of the key modules.

Components

This folder contains custom Vue.js components and some auxiliary JavaScript modules that the components use.

- `Codearea.vue` - main editor component wrapping an instance of AceEditor (https://ace.c9.io/) that supports code highlighting and other important features for writing code such as indentation. The Codearea componenet also contains inspection panels for displaying the compiled code, and debugging information such as the callstack and memory contents.

[4]Node.js environment is only needed for development and building

- `TutorialPicker.vue` - component for browsing and loading the tutorials (and packages they depend on). The tutorial browser is implemented in a modal window; the key module that takes care of loading and initializing the packages is contained in (`packageLoader.js`). It uses resource-loader[5] for the actual loading of the static files comprising the tutorials, packages, and their resources.

- `Simulator.vue` - this component contains the GUI elements to control the scheduler; that allows the user to run or step through the simulation. This component also contains an empty DIV element that the packages may use to display arbitrary content to the user. Typically this element will be initialized with a canvas visualizing the agents.

- `Workbench.vue` - component combining together the `Simulator` and multiple instances of `Codearea` (one for each agent type in the tutorial).

Parascript

The compiler for Parascript is implemented in this folder.

- `parser/parascript.js` - parser for the Parascript language; this module is generated from `parser/parascript.jison` using Jison; it produces an instance of the abstract syntax tree from Parascript source code

- `ast.js` - classes representing the nodes of the abstract syntax tree (all nodes extend class `ASTNode`); invoking `generate()` method on the root node traverses the tree and emits instructions that can be fed into an instance of `Executor`

- `compiler.js` - encapsulates the parser and the syntax tree; also handles compile-time errors

- `identifierTable.js` - helper module that the compiler uses for keeping track of identifier declaration, scopes, and package function implementations

Runtime

The following modules implement the runtime environment for the parascript programs.

- `session.js` - module containing class `Session` that holds the loaded tutorial; it is responsible for instantiating the `Scheduler` (specific implementation depends on the selected scheduling algorithm)

- `agent.js, agentType.js` - modules containing classes `Agent` and `AgentType` which wrap the agent and agent type definitions provided by the loaded tutorial

[5] https://github.com/englercj/resource-loader

- `instructions.js` - classes extending class `Instruction` are used to represent the code executed by the `Executor`; each instruction has method `perform(executor)` invoked by the executor that defines its behavior.
- `executor.js` - module responsible for executing the agent code; it holds the data structures to keep the execution state, and it handles runtime errors.
- `scheduler.js` - abstract class `Scheduler` defines the interface; this class holds instances of `Executor` (one per each agent) and invokes them according to the algorithm defined in the specific implementations:
  - `RoundRobinScheduler`
  - `RandomScheduler`
  - `LockstepScheduler`

4.2 Packages and Tutorials

Packages and tutorials are extensions to the platform; they may be added without re-building the application. These extensions consists of JavaScript modules and other resources–typically images or templates. Packages contain reusable code to make the creation of tutorials easier, and to avoid repetition of code in the tutorial implementation.

4.2.1 Packages

The following section gives a brief overview of packages that were created as reusable modules to simplify the creation of new tutorials. The full documentation of the interface of the package providing features for visualization can be found in the developer documentation (Appendix A).

Packages are composed of the main JavaScript module and arbitrary number of resources. The main module can contain arbitrary JavaScript code; however, the convention is that the package code belongs to one of the following categories:

- **package functions** - functions reserved for invocation from Parascript program
- **agent API** - set of package functions and constant definitions
- **agent type definitions** - set of Agent API and other metadata forming a definition of an agent type
- **other functions** - auxiliary code invoked by the platform or other packages, typically during initialization, reset, or destroy phase

math

Package containing agent API module with package functions implementing essential math operations and numeric constants.

utils

Package containing the following two agent API modules:

- **utils** - utility functions for array and map operations
- **agents** - package functions for accessing information about agents; for example, retrieving the ID or name of the (calling) agent, and the list of all agent names and types.

console

Contains agent API providing access to the main Parapple console.

pixi

Package containing auxiliary code implementing basic support for visualization using library PixiJS (http://www.pixijs.com/). This package initializes the PixiJS canvas in the simulator.

pixi2d

Dependencies: **pixi**

Pixi2d uses PixiJS to implement support for tutorials based on a 2-dimensional grid. It provides an abstraction for initializing the grid to an arbitrary size, changing the color of the grid tiles, and drawing sprites from image resources. The package implements helper functions that are typically used by other packages to simplify the manipulation of the grid objects.

ladybugs

Dependencies: **pixi2d**

This package uses the general functionality provided by *pixi2d* to provide easy access to specific images, grid objects, and agent types.

The *ladybugs* package defines the following agent types, each having slightly different set of agent API to provide programming interface matching exactly the tutorial needs. All agents have the ability to move on the 2d grid (*walking* - movement only in the direction the agent is facing, *flying* - free movement in any direction).

- ladybug (master) - flying, ability to read the color of all tiles, message passing
- butterfly (worker) - flying, and to change color of tiles
- beetle - walking, tile coloring
- bee - flying, ability to pick and drop objects
- snail - walking, ability to detect whether there is another agent at a neighboring position

- bug - walking, broadcast (channel communication in lock-step), ability to change color of tiles

Screenshot showing the visual style of this package depicting the initial state of tutorial *Enter the Forrest* is shown on Figure 4.1.

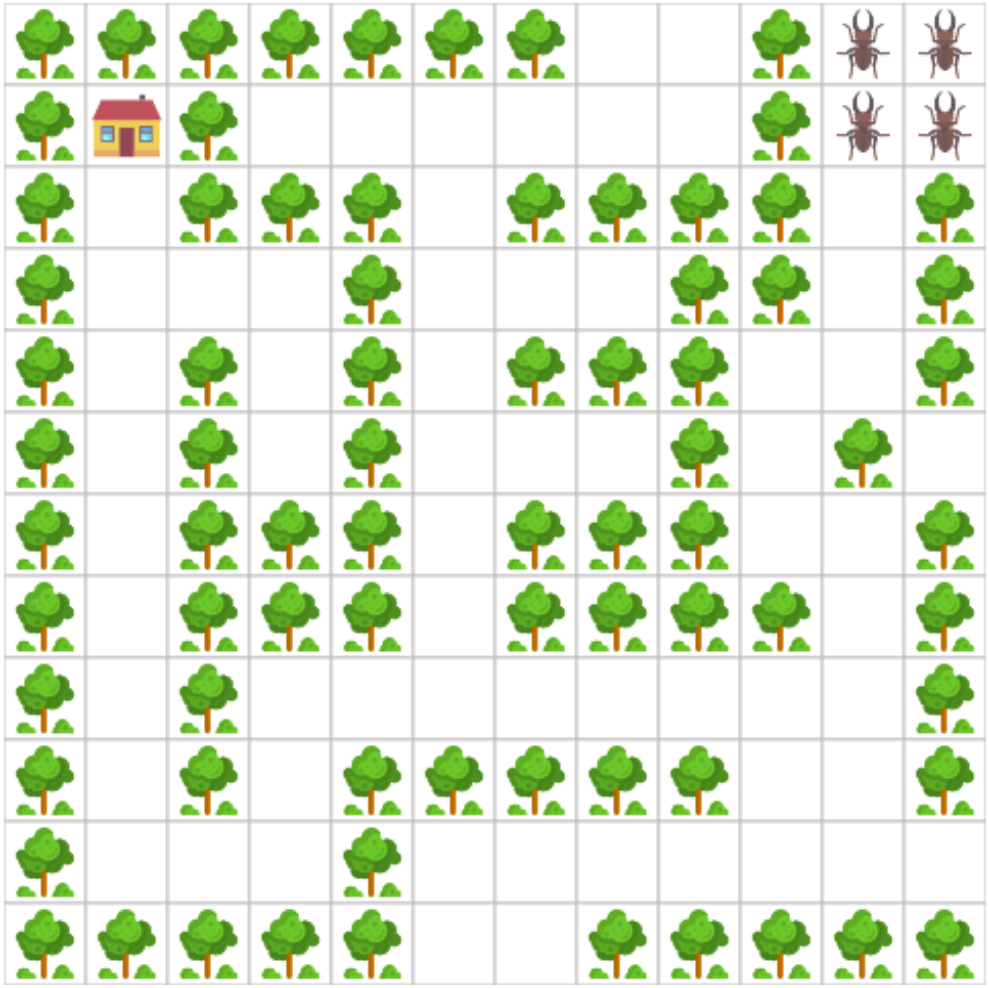


Figure 4.1: Visual style of the Ladybugs package[6]

broadcast

This package implements support for synchronous agent-to-agent communication in lock-step. It provides an important package function `broadcast` that allows the agents to send or receive arbitrary values based on the `mode` argument. This feature does not work in tutorials without the lock-step scheduler.

4.2.2 Introductory Tutorials

The introductory tutorials do not ask the student to solve a parallel programming task. Instead, they aim to gently introduce the user to the fundamental concepts of the platform and the programming language.

[6]Icons made by Freepik from www.flaticon.com licensed by CC 3.0 BY

- **Hello World** - The aim of this first tutorial is to just teach the student where to enter the agent code, how to launch the simulation, and where to look for console output.
- **Hello Parallel World** - Tutorial with multiple agents demonstrates execution of one program multiple times.
- **Everyday I am Schedulin'** - Introduces all three available scheduling strategies as well as the concept of variable instruction cost. The student is invited to attempt the same task with the use of various schedulers.
- **Enter the Forrest** - Agents (represented as beetles) must be programmed to find their way through a maze (screenshot captured on Figure 4.1). The agents can read and set the color of the grid tile they are currently occupying to help them navigate the maze.

4.2.3 Tutorials

The following tutorials are based on the tutorials presented in Chapter 1. The tutorials are listed here only for reference, full assignment of the programming task as well as explanation of the available functions (determined by the agent type) is provided in the attached application.

Counting

- **options**: number of agents (2-3)
- **agents**: all agents belong to a single agent type represented as butterfly; agents can freely move on the grid, read and set color of the occupied tile, and exchange messages
- **initial state**: randomly colored 10x10 grid and randomly positioned agents
- **assignment**: to program the agents count the number of orange tiles as quickly as possible; all agents must **know** the final result and submit it using `submit()` function

Synchronized Snail Race

- **agents**: 10 agents that can check whether there is another agent on a neighboring tile using function `hasNeighbor(orientation)`
- **initial state**: agents are positioned in the left-most column of the grid, one agent in each row
- **assignment**: to program the agents to cross to the right-most side of the grid while maintaining the horizontal distance between the left-most and right-most agent less than or equal to 1

Lock-step Coloring

- **options**: grid size (16x16 or 32x32) and number of broadcast channels (2, 4, 8, or 16); scheduling strategy limited to lock-step
- **agents**: agents (represented as grasshoppers) with the ability to communicate using `broadcast()` function implemented in package *broadcast*
- **initial state**: agents are positioned in the left-most column of the grid, one agent in each row; grid is colored randomly with blue and white
- **assignment**: similarly as in the previous tutorial, the agents must traverse to the other side of the grid (note that synchronizing their movement is trivial with lock-step); in every step they must decide which is the prevailing color in the particular grid column and color the column with that color (in the end, all tiles in any given column must have the same color)

Master-Worker

- **agents**:
  - **masters** (represented as ladybugs) that move quickly and may access the color of all tiles (however, they cannot set the color)
  - **workers** (represented as butterflies) that move, relatively to masters, slowly (higher instruction cost for movement); they see only the color of the tile they are occupying, but they can also change its color
- **options**: number of masters (2, 4, or 6), number of workers (4, 8, or 16) and grid size (12x12, 16x16, or 24x24)
- **initial state**: all grid tiles are white, and all agents are positioned randomly
- **assignment**: to write two programs, one for workers and another for master agents; the worker agents must color the entire grid; the point is to use the master agents to dynamically distribute the work between the workers

Mushroom Hunting

- **agents**: represented as bees, these agents can pick and drop objects (mushrooms); they can also read and set tile color
- **options**: number of agents (4, 6, or 8)
- **initial state**: the grid is initialized with trees and mushrooms according to a initialization matrix for the forrest module of the ladybugs package; the agents are placed randomly in the lower part of the grid
- **assignment**: program the agents to collect the mushrooms in the forrest and drop them at the tile marked with a house; the catch is that there are narrow pathways leading to the areas with mushrooms where only one agent can pass; therefore, they need to coordinate access to the narrow passages by coloring some grid tiles

4.3 Evaluation

This section provides a brief evaluation of the implemented application. The conclusions of this evaluation are based on observations from an informal user study.

4.3.1 User study

The participants of the user study were two programmers with several years of programming experience (and a good knowledge of JavaScript) and other two users with basic knowledge of programming (without experience with JavaScript). None of the participants considered themselves expert in parallelism and concurrency. The users were given only a brief introduction about the platform (5 minute explanation of the purpose of the platform, the concept of programmable agents, and the fact that Parascript is a language similar to JavaScript). Then they were asked to complete the introductory tutorials without additional instructions or help. It was assumed that the users would be able to start using the application just with the instructions included in the application itself.

Study Objectives

The main purpose of the study was to get general feedback, both from experienced and inexperienced programmers. The specific objectives of the user study were to evaluate the following qualities:

- **Usability** - Users should be able to use the platform intuitively without lengthy explanation or without the need to read the documentation.
- **Stability** - The application should not crash or behave erratically.
- **Approachability** - The users should be able to start programming in Parascript quickly; they should be able to easily necessary information in the help section.
- **Comprehensibility** - The users should have a clear sense of what they are trying to accomplish and would they could possibly learn out of it.

Propositions

The users were given approximately 45 minutes to work with the application. It was expected this will be enough time to complete the 4 introductory tutorials (the first 3 are short, but *Enter the Forrest* requires more time). The users could ask questions about the application and programming language.

4.3.2 Study Findings

All uses were able to open a tutorial and complete the first *Hello World* tutorial within minutes. This means that within few minutes the users successfully performed the following tasks which introduced them to the fundamental features of the platform:

- opening the tutorial picker modal window
- reading the tutorial assignment in the left panel
- entering the agent program into the coding area
- compiling the simple program (and reflecting possible compile errors, such as a missing semicolon)
- running the program and observing the output in the console

Introductory Tutorials

The following observations were made during the (attempted) completion of the remaining introductory tutorials:

- *Hello Parallel World* - All users were able to implement the solution, although the two inexperienced programmers required more time (approx. 15 minutes). Users raised question about the concept of one program being executed by multiple agents; in particular, one user was curious if they need to somehow *initialize* the agent context in the program code, i.e., instruct the program to *fork*. It was suggested that the first tutorial with multiple agents should already visualize the agents (as it is done in the next tutorials) to help emphasize the concept of agents.
- *Everyday I am Schedulin'* - The solution to this was easy for all users, but the objective of this tutorial is to explain different scheduling algorithms. This objective was not achieved; the tutorial would require more in-depth explanation of the various schedulers. Users were able to comprehend the scheduling algorithms with additional oral explanation.
- *Enter the Forrest* - This tutorial captivated the interest of the user the most, probably due to its simple yet challenging assignment. The inexperienced programmers gave up on the assignment (mostly due to time constraints) while the two experienced programmers were able to get a working solution (in approx. 45 minutes).

Usability of Parascript

The programming language proved to be easy for experienced JavaScript programmers. They quickly grasped the available language features. However, the two inexperienced users without prior exposure to JavaScript needed more time to make their programs to compile. They reported that they would appreciate quick language summary. The language syntax or behavior did not present a significant barrier for any of the users. All users were able to understand and reflect the messages from the compiler.

Suggestions for Improvements

Minor bug reports and feature requests were reported orally or using a built-in user feedback tool[7] which allows users take screenshots, highlight information, and send it together with comments.

An idea that came out of this user study is to implement a *Parascript cheat sheet*. This would be a quick informal language reference with the most basic Parascript features such as iteration loops and declaration of functions. This could be implemented as a floating panel so the user could read it while coding.

The two experienced programmers also attempted to use the built-in debugger. Although they managed to successfully place a breakpoint and step through the program, they missed several features. Most importantly, they reported the need for stepping through the execution of just one agent. At the moment the debugger switches to the next scheduled agent in every step. Second, they missed usual *step in*, *step over*, and *step out* functionality. These suggestions are opportunities for future development as mentioned in Section 4.4.

Summary

The platform proved to be intuitive and approachable. Users with various levels of programming experience were able to start using it quickly, and they understood the presented task. The application also demonstrated its stability; no unexpected errors or crashes occurred during the study.

[7] powered by Userback (www.userback.io)

4.4 Future Work

Fully functional prototype of the application is complete, and the project remains in development for future enhancements. Besides several minor bugs and feature requests (tracked as open issues on the bug-tracking system used for the project development), there are many opportunities for additional features. The following section presents the ideas for future work in three categories.

4.4.1 Potential Language Changes

- **Function values or namespacing** - User and package functions are defined as global identifiers. This leads to polluting the global namespace; names of package functions need to be either long (for example, prefixed with package name) or it is likely that the user will run into problems trying to define a user function with the same name. Supporting functions values in the language would solve this problem; functions could be defined as values in a map together with relevant constants; for example, there could be a *Math* map representing namespace for all math-related functions and constants.

- **Blocking package calls** - All calls to package functions return immediately; to simulate blocking calls, there could be an API on the scheduler that the package could use to pause and resume a given agent. From the perspective of the student, the agent program would stop while other programs would continue. Similar situation happens when the instruction cost of a package call is high (relative to the cost of instructions executed by other agents); however, the proposed feature would allow to keep the agent paused until some arbitrary event occurs (e.g., a message is received).

- **Conflicting operations** - Current implementation does not allow packages to handle conflicting operations other than allowing the first one to succeed. This could be improved by adding an optional handler (implemented as a function in the package module) that would be invoked by the scheduler at the end of each tick. The package could easily keep track of the calls performed within one tick, and optionally perform arbitrary post-processing (i.e. change return values) if conflicting operations were detected. Note that similar idea is used to implement the `broadcast()` operation for lock-step.

4.4.2 Enhancements

Some existing features of the application could be enhanced to improve the user experience, for example:

- **Debugger** should support stepping through the program of just one particular agent (the user could run the simulation until the next step of the currently paused agent). By the same token, it would be worth implementing break points that could be active only for certain agents.

- **Error messages** of syntax errors reported by the compiler are generated by the Jison parser generator; although these messages are correct and precise, they could be more succinct and user-friendly.
- **Runtime errors** result in termination of the entire simulation. Ruining a long simulation again because of a banal runtime error of just one agent program can be frustrating; instead, the user should have the option to continue the simulation, with or without the agent that caused the error.
- **Shared user functions** could be useful for writing reusable code for multiple agent types; at the moment, the user must copy the code to all agent tabs. Even more powerful feature would allow the user to write modules that could be uploaded and shared among users.

4.4.3 Major Features

There are also some major features that could be very useful for actual production deployment if the number of users would be expected to grow:

- **User accounts** - Although the application can remain freely accessible, establishing the user identity would be useful for other possible features, such as submitting solutions, leader boards, and tutorial uploading.
- **Persistence** - Currently there is no mechanism for saving the agent programs; it would be useful to implement such feature not just to allow the students to save their own work, but also to provide a way to submit (and possibly share) the solutions. Practical feature would be to allow the user to download all agent programs in an archive to local disk, and the possibility to load agent programs from local files.

The current state of the application demonstrates the working principles that were part of the objective of this thesis. Present version of the platform with the example tutorials can be used for demonstration and to validate the concept of the platform. The completion of the outlined future work would greatly increase its potential for use for education.

Conclusion

The main focus of this thesis was to create a platform for tutorials that would alleviate some of the difficulties that programmers face when starting to learn parallel programming. This objective was achieved by implementing an extensible online application for parallel programming tutorials. This application provides an intuitive coding environment that allows students to write programs for multiple agents to collaboratively solve a given task. The effort was divided between the following three areas: designing a suitable programming language, developing a web application, and implementing several model tutorials.

An essential part of the project was to find a programming language suitable for solving the tutorial assignments. The objective was achieved by designing a custom programming language called *Parascript* that strongly resembles JavaScript; therefore, it appears familiar to programmers with prior exposure to language with C-like syntax. The language is extensible with packages implemented in JavaScript.

It was necessary to build a compiler and a runtime environment for the custom programming language. The compiler leverages a parser which was created from a context-free grammar specification using the LALR(1) parser generator tool Jison. The runtime environment supports multiple scheduling algorithms for the parallel simulation. The scheduling takes into account variable instruction cost. The platform also supports running the parallel simulation in a mode simulating lock-step on GPUs.

The platform was implemented as a rich, single-page web application using state-of-the-art web frameworks and libraries. The platform provides an integrated development environment that allows the students to program the agents and to run the parallel simulations. The environment also provides adequate means for debugging the agent programs. The architecture of the platform puts a strong emphasis on extensibility; new tutorials and packages can be added without any modifications to the application source.

Several parallel programming tutorials were created to demonstrate that the concept and implementation of the online platform meet the goals of this thesis, and to verify the usability of Parascript. Additional introductory tutorials were designed to help new users become quickly familiar with the programming language and the platform environment. The tutorials were created with the help of reusable extension packages that provide a framework for visualizations.

The developed online platform is fully functional and satisfies the requirements for the envisioned parallel programming tutorials. Therefore, the objectives of this thesis were accomplished. As there are opportunities for future work on this project, the application will remain in development. With the enhancements proposed as future work the platform has the potential to become a valuable tool for helping programmers enter the world of parallel programming.

Bibliography

[1] Nir Shavit Maurice Herilhy. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2012.

[2] Wen-mei W. Hwu David B. Kirk. Programming massively parallel processors: A hands-on approach, 2012.

[3] Vipin Kumar Ananth Grama, George Karypis. *Introduction to Parallel Computing*. ADDISON WESLEY PUB CO INC, 2003.

[4] James Reinders Michael McCool, Arch D. Robison. *Structured parallel programming*. Elsevier, Morgan Kaufmann, 2012.

[5] Becky Owens et al. *Modern Instructor: Keys to Exceptional Online Teaching*. Modern Instructor, 2015.

[6] Brian Albers Peter Lubbers, Frank Salim. *Pro HTML5 Programming: Powerful APIs for Richer Internet Application Development*. Apress, 2011.

[7] Pierre Carbonnelle. Pypl popularity of programming language. `http://pypl.github.io/PYPL.html`, 2016. [Online; accessed 5-November-2016].

[8] GitHub. Language trends on github. `https://github.com/blog/2047-language-trends-on-github`, 2015. [Online; accessed 5-November-2016].

[9] Wikipedia.org. Server-side javascript implementations. `https://en.wikipedia.org/wiki/Server-side_JavaScript_implementations`, 2017. [Online; accessed 11-July-2017].

[10] Tom Occhino. React native: Bringing modern web techniques to mobile. `http://facebook.github.io/react-native/blog/2015/03/26/react-native-bringing-modern-web-techniques-to-mobile.html`, 2015. [Online; accessed 11-July-2017].

[11] Klint Finley. Javascript conquered the web. now it's taking over the desktop. `https://www.wired.com/2016/05/javascript-conquered-web-now-taking-desktop/`, 2016. [Online; accessed 11-July-2017].

[12] StackOverflow. Developer survey results 2016. `http://stackoverflow.com/research/developer-survey-2016#technology`, 2016. [Online; accessed 5-November-2016].

[13] Ecma. Ecmascript language specification. `https://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf`, 2016. [Online; accessed 14-June-2017].

[14] mozilla.org. Strict mode. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode`, 2017. [Online; accessed 14-June-2017].

[15] Zach Carter. Jison. https://zaa.ch/jison/about/, 2017. [Online; accessed 15-June-2017].

[16] whatwg.org. Html specification: Web workers. https://html.spec.whatwg.org/multipage/workers.html#workers, 2017. [Online; accessed 16-June-2017].

[17] Danu Pranantha et al. Designing contents for a serious game for learning computer programming with different target users. 2011.

[18] F. R. Graeml R. F. Maia. Playing and learning with gamification: An in-class concurrent and distributed programming activity. In *2015 IEEE Frontiers in Education Conference (FIE)*, pages 1–6, Oct 2015.

[19] NVIDIA. Learn gpu programming in your browser with nvidia hands-on labs. https://devblogs.nvidia.com/parallelforall/learn-gpu-programming-free-on-demand-gpu-training/, 2014. [Online; accessed 23-June-2017].

[20] Kyle Simpson. *You Dont Know JS: Types & Grammar*. 2015.

[21] Dimitri Mestdagh. You have to love javascript's type coercion. https://g00glen00b.be/javascript-coercion/. [Online; accessed 26-April-2017].

[22] Evan Hahn. *JavaScript Testing with Jasmine: JavaScript Behavior-Driven Development*. O'Reilly Media, 2013.

[23] Josh Powell Michael Mikowski. *Single Page Web Applications: JavaScript end-to-end*. Manning Publications, 2013.

[24] Adam Silver. The disadvantages of single page applications. https://adamsilver.io/articles/the-disadvantages-of-single-page-applications/. [Online; accessed 17-July-2017].

[25] Vue.js. Comparison with other frameworks. https://vuejs.org/v2/guide/comparison.html. [Online; accessed 17-July-2017].

[26] Roman Kuba. Consider vuejs for your next web project. https://blog.codeship.com/consider-vuejs-next-web-project/. [Online; accessed 17-July-2017].

List of Figures

List of Tables

A. Developer Documentation

Installation

The development setup assumes **Node.js** (nodejs.org) installed. Once you download the project source, run in the root folder the following command to download and install all application dependencies.

```
npm install
```

The project was tested with npm version 3.10.10.

The project uses Webpack (webpack.js.org) for processing all application modules and resources.

Running in Dev Mode

For development, you can run the application locally with Webpack and Node.js:

```
npm run dev
```

This should automatically open a browser window with the application. Alternatively, point your web browser to `localhost:8080`. The dev mode supports hot-reload; changes to any resources in the `src` folder trigger the build and reloading of the changed resources to browser without losing state.

Building for Production

To build the application for production deployment, run the following command in the project root directory:

```
npm run build
```

The build artifact is created in the `/dist` directory. The build consists of:

- `static` - folder containing all assets of the application, minified JavaScript, CSS for all components, and static resources (tutorials, packages)
- `index.html` - the index file of the application with proper URLs to the static resources

The a build can be uploaded to a web server. The application then works like a static website and it does not require any dependencies on the web server. Note that the URLs to the static resources are prefixed with `assetsPublicPath` specified in `/config/index.js`. This makes it possible to deploy the application to a subdirectory on the web server.

Project Structure

The application is created using `vue-webpack` template (Webpack boilerplate for Vue.js).

The following tree shows the structure of the application source. It is adoped from the documentation of the `vue-webpack` template documentation.[8]

[8] http://vuejs-templates.github.io/webpack/structure.html

```
.
├── build/                  # webpack config, dev-server
├── config/                 # vue-webpack prod/dev config
│   └── index.js            # main project config
├── src/                    # Parapple source code
│   ├── main.js             # app entry file
│   ├── constants.js        # shared project constants
│   ├── App.vue             # main app component
│   ├── components/         # UI Vue components
│   ├── parascript/         # grammar, parser, and compiler
│   ├── runtime/            # simulation runtime
│   ├── utils/              # shared auxiliary code
│   └── assets/             # assets processed by webpack
├── static/                 # directly copied assets
│   ├── index.json          # list of Parapple tutorials
│   ├── packages/           # Parapple packages
│   └── tutorials/          # Parapple tutorials
├── dist/                   # build artifact
├── .babelrc                # babel config
├── index.html              # main template
├── README.md               # installation instructions
└── package.json            # npm config and dependencies
```

Parascript Parser

The Parascript grammar specification is in file `parascript.jison`. The parser (file `parascript.js`) is generated using Jison (zaa.ch/jison/). To generate the Parascript parser from a modified grammar file, first install the Jison command line tool:

```
# Install Jison globally
npm install jison -g

# generate parser from parascript.jison
cd src/parascript/parser/
jison parascript.jison -o parascript.js
```

(Note: the above command overwrites `parascript.js`.)

Jison is a JavaScript port of the parser generator Bison. It also contains a lexer analyzer similar to Flex. The single `parascript.jison` file contains both lexer and grammar rules and it should look familiar to users knowing Bison and Flex.

Packages

This section provides an overview of package structure, description of the packages forming the visualization framework, and a guide for creation new packages.

Basic Concepts

Packages consist of the following:

- index.json - file with JSON containing package metadata
- resources - list of files contained in the package folder; these files are downloaded with `resource-loader` and passed to the constructor
- dependencies - list of package names that are instantiated prior to this package and passed to its constructor
- main module - main JavaScript file containing a function that is instantiated by the package loader. The instance of this function is injected to other packages or tutorials that reference this package as a dependency
- modules - JavaScript resources of the package can be specified as modules; in that case, they are evaluated and instantiated similarly like normal packages, just with the difference of being injected only to this package

Visualization Packages

The existing project implementation contains packages `pixi` and `pixi2d` that provide basic framework for visualization using PixiJS engine (www.pixijs.com).

pixi

Most importantly, this package initializes PixiJS stage on the simulation canvas element. The package contains the PixiJS library in resource `pixi.js`.

The package main module provides the following functions to be invoked by dependent packages or tutorials:

- `initialize(options)` - initializes PixiJS stage (canvas) with the dimensions specified in `options.width` and `options.height` (in pixels; default 480x480)
- `render(options)` - invokes `render()` on PixiJS stage
- `cleanup()` - destroys PixiJS stage and free all resources; called automatically

pixi2d

This package uses `pixi` as dependency. It contains the following useful modules:

- `grid.js` - abstraction for displaying 2-dimensional grid; the grid is represented as a collection of instances of class `Tile`
- `images.js` - support for displaying images (loaded as package resources) on the canvas

- `gridObjects.js` - framework of functions to work with images displayed on the 2D grid; class `GridObject` represents an image associated with `Tile`
- `gridAgents.js` - class `GridAgent` wraps `GridObject` and provides additional functionality, for example for moving the agent
- `shapes.js` - abstraction for simple PixiJS shapes
- `colors.js` - color constants and utilities
- `utils.js` - internal helper functions
- `apis.js` - objects with consolidated package functions that can be used to create agent types

Creating a Package

The following example demonstrates creation of a reusable package containing new agent type that provides programming interface with custom functions implemented in JavaScript. It uses `pixi2d` as dependency. The package, as well as the agent type, will be called *turtle*. The agent type will have access to special package function `jumpRandom()` that moves the agent to a random position on the grid.

First, create a folder in **`static`**`/packages` directory. It has to have the same name as the package (i.e., `turtle`):

```
.
├── index.json          # metadata
├── main.js             # main module
├── turtle.png          # icon for the agent
├── turtle.html         # API doc for turtle agent
└── jumping.html        # API doc for jumping API
```

The metafile `index.json` needs to specify the dependencies, resources, and the main module:

```
1 {
2     "main": "main.js",
3     "dependencies": [ "pixi2d", "console" ],
4     "resources": [ "turtle.png", "turtle.html" ],
5 }
```

The main (and only) module must evaluate to a function which takes dependencies and resources as arguments. It exposes a function `makeTurtle(name)` for tutorials using this package as a dependency; the function returns the agent definition with specified name (agent names must be unique).

```
(function (dependencies, resources) {
    var self = this;
    // Initialize turtule texture for PixiJS
    dependencies.pixi2d.images
        .makeTexture(resources["turtle.png"], "turtle");

    // Create agent API module with documentation
    // providing the jumpRandom function
    var jumpAPI = {
        name: "Jumping",
        doc: resources["jumping.html"],
        functions: {
            // @param agent pixi2d GridAgent instance
            jumpRandom: function(agent) {
                // Get next random tile
                var tile = dependencies.pixi2d.utils
                    .nextRandomTile();
                // Move agent to that tile
                agent.moveTo(tile.col, tile.row);
            }
        }
    }

    // Declare turtle agent type
    this.typeTurtle = {
            name: "Turtle",
            api: [
                {
                    name: "Turtle",
                    doc: resources["turtle.html"]
                },
                jumpAPI
            ]
        }

    // Turtle factory function
    this.makeTurtle = function(name) {
        // GridAgent with texture turtle
        var agent = dependencies.pixi2d.gridAgents
            .makeAgent(name, "turtle");
        // Set agent type
        agent.agentType = self.typeTurtle;
        return agent;
    }
})
```

Contents of `main.js` for package *turtle*.

Creating Tutorials

The following example demonstrates creation of a new tutorial that uses the `turtle` package created in the previous section. The assignment of this simplistic tutorial is to just invoke the `jumpRandom()` until the agent moves to position `[0, 0]`. The tutorial is called `jumping`.

The first step consists of creating a directory in `static/tutorials`. Again, it needs to have the same name as the tutorial.

```
.
├── index.json          # metadata
├── main.js             # tutorial main module
├── guide.html          # guide displayed in left panel
├── solution.ps         # solution (Parascript)
└── intro
    ├── index.html      # intro template
    └── index.js        # optional JS for intro template
```

The `intro` directory contains a template (with optional javascript) that is displayed in the tutorial picker modal window. The `index.json` will have the following contents:

```
{
    "name": "Jumping",
    "intro": {
        "template": "intro/index.html",
        "js": "intro/index.js"
    },
    "main": "main.js",
    "resources": [
        "solution.ps",
        "guide.html"
    ],
    "dependencies": [ "turtle" ],
    "schedulers": [ "roundrobin", "random", "lock-step" ],
    "guide": "guide.html"
}
```

The metadata file is quite self-explanatory, but it is worth mentioning that the option `scheduler` lists scheduler implementations that should be offered in the tutorial picker when choosing this tutorial.

The tutorial implementation is contained in `main.js`. This JavaScript module is loaded and instantiated like a package, but it must expose several functions that are mandatory for tutorials.

The tutorial module is mainly responsible for creating agent definitions (function `getConfig`) to initialize the workbench in the beginning, for delegating calls to `reset()` to reset the simulation state, and for keeping track of the tutorial state, that is implementing `getStatus()` returning an object indicating whether the tutorial has been solved (or failed).

```
(function (dependencies, resources) {

    var self = this;
    var status; // Tutorial status
    var agent;

    this.getConfig = function(options) {
        agent = dependencies.turtle.makeTurtle("turtle-1");
        // Set solution to turtle type
        var agentTypeTurtle = dependencies.typeTurtle;
        agentTypeTurtle.solution =
            resources["solution.ps"].data;
        // Return agent definition to the platform
        return {
            agentTypes: [ agentTypeTurtle ],
            agents: [ agent ],
            schedulerType: options.schedulerType
        }

    }

    this.initialize = function(session) {
        // Initialize 10x10 grid
        dependencies.turtle.pixi2d.initialize(session, {
            rows: 10,
            cols: 10
        });
    }

    this.reset = function(session) {
        dependencies.turtle.pixi2d.reset();
        // Position agent to 1,1
        agent.moveTo(1, 1);
        status = { done: false, solved: false };
    }

    this.getStatus = function() {
        // Check accomplishment of the objective
        var position = agent.getCoords();
        if (position.col == 0 && position.row == 0) {
            status = { solved: true, done: true }
        }
        return status;
    }

})
```

Contents of `main.js` for tutorial *jumping*.

Parapple Console

The console is located in the first tab of the *workbench*. It is a useful tool not just for the users, but also for the debugging of packages and tutorials. The console displays log messages in four levels–`DEBUG`, `INFO`, `WARNING`, `ERROR`. Each log is time-stamped and the *Module* column indicates which part of the application created the log.

Debugging Tips

System errors that occur within the platform are logged to the system console as `ERROR` messages (unless it is an uncaught exception, in which case you will only find the error only in the browser JavaScript console). The application also outputs `DEBUG` messages, for example, when loading packages and initializing a tutorial. Mistakes in package/tutorial definitions (such as wrong resource names or JavaScript errors resulting in failed main module instantiation) will show in the console as error logs.

Logging to Console

The console is implemented as a Vue component. There are actually more instances of this component in the GUI. Each agent has a separate console filtering only logs issued from its module.[9] All consoles, when *mounted*,[10] register to a singleton object *consoleProxy* (module `consoleProxy.js`). This object exposes API for logging and broadcast logs to the consoles. The following methods are designated for logging:

- `log(level, module, message)`
- `debug(level, module, message)`
- `info(module, message)`
- `warn(module, message)`
- `error(module, message)`

Logging in Parapple Core

To access the console proxy from the Parapple source (i.e., in Vue components or other JavaScript modules compiled with the application), the best way is to import the proxy singleton exported in `consoleProxy.js`:

```
1 import proxy from 'components/console/consoleProxy.js';
2 ...
3 proxy.error( "moduleXYZ", "This is an error." );
```

[9] The relevant package call passes the agent's name as the module

[10] mount() is an event of Vue components

Logging in Packages and Tutorials

Packages and tutorials can access the console as well. The global JavaScript object `PARAPPLE` contains a reference to `consoleProxy`. Therefore, anywhere in packages and tutorials, it is possible to use it like this:

```
1 PARAPPLE.info( "packageXYZ", "FYI, this is a log." );
```

This is how packages can implement Parascript functions that log to the console.

The PARAPPLE Object

The global `PARAPPLE` object contains references to more modules than just the console. It has references to the following modules:

- `instructionCost.js` - provides means for determining instruction cost for tuple `(agent, instruction)`
- `modalMessage.js` - proxy for displaying modal messages and queries
- `session.js` - the application runtime with reference to the tutorial instance, scheduler, and agent definitions