Robot Vacuum

Abdeljalil Nabaoui : stud-abdeljalil@ruc.dk : 79707
Kristian Vestbo : vestbo@ruc.dk : 75638
Thea Johansen : stud-theaj@ruc.dk : 77931
Klara Kringelbach : joverkring@ruc.dk : 74486
Andreas Højgaard Tofthold : andreast@ruc.dk : 76368

Supervisor: Morten Rhiger

May 2024

Abstract

This project documents the development of "Robot Vacuum", an educational game designed to teach fundamental programming concepts utilizing interactive gameplay. The game is implemented in Java with our custom programming language called ATAKK handling the main user functionality, which is allowing players to code actions to control their own robot using algorithms. Which will navigate it through various tasks like avoiding obstacles and cleaning tiles with the aim of the game being who's robot can clean the most dirt within the allotted time.

This report comprehensively overviews the design of the Robot Vacuum game. It includes a detailed program description, programming environment, and user manual. The report also explains the use of JavaCUP and JFlex for the parser and lexer generation. Additionally, it supplies user manuals and coding documentation, ensuring accessibility and comprehension for both beginners and more experienced programmers.

Contents

Chapter 1

Introduction

Incorporating gamification in education has become a compelling strategy to improve learning experiences across different fields. In computer programming, gamification bridges theoretical knowledge and practical application, making concepts more convenient and enjoyable for beginners. This report presents the development of a Robot Game. This game aims to introduce basic programming principles via interactive gameplay with the goal of creating an independent robot.

The Robot Game presents an environment in which the custom language ATAKK runs in. This language incorporates fundamental programming constructs such as logical operations, control structures, and data management, allowing players to script and control a robot within simulated scenarios. The goal of ATAKK is to provide a practical learning experience.

This report delves into the architecture and design of the Robot Game, offering a thorough analysis of the challenges faced during its development and the solutions implemented to overcome them. Further sections of the report will elaborate on the design considerations that affected the development process, describe the game's programming environment, and discuss the user interaction strategies employed. Additional documentation includes a user manual for the ATAKK language and clear guides for coding within the game, assuring that players of all levels can gain educational benefits from the experience.

1.1 Motivation

Using a domain-specific programming language, the 'Robot Game' introduces users to essential coding concepts and problem-solving techniques in an interactive environment which was inspired by games like [Codewars], Code Wizard and [Screeps].

ATAKK provides a coding environment in which programming principles can be introduced without the user needing previous programming experience. This tailored approach improves the learning experience by converting abstract coding tasks into tangible, interactive challenges within the game environment, equipping users with practical skills that can be applied.

Attention has been paid to the "Robot Game" design to ensure it aligns with its educational goals. Although the game does not feature traditional interfaces, it incorporates a visual and interactive setting where users can directly see the results of their coding choices through the robot's actions. This real-time feedback comes from watching the robot in-game, an essential gamification aspect. It enables players to comprehend the outcomes of their decisions visually, reinforcing the learning process through practical application and immediate results.

Chapter 2

Game introduction

2.1 Requirements

This following section details the requirements for developing a game to guide and demonstrate our programming language, ATAKK. It explains the interdependence between the game and the language, the simplicity and accessibility needed for users of different coding skill levels, and the time-based steps that emphasizes efficient algorithm creation.

We wanted to make a game that could serve as a guide for the scope of our programming language. The scope of the game would be the deciding factor in determining what functions we needed to include in the language itself, in conjunction with this function, the game also serves as the platform for the usage of ATAKK. Due to this relationship between the game and ATAKK, any change in either the game or ATAKK would necessitate a change in the other as well, since the game decides what is needed in the language, while any functionality added to the language needs a usage in the game in order to prove that it functions as intended.

We wanted to keep the game itself relatively simple as the ATAKK language and interpreter was the main focus of the project while the game would mainly be there to validate and demonstrate the usability of ATAKK.

We also wanted the game to be playable by users of varying levels of coding skills, since it is supposed to increase users interest and coding skill. It was therefore necessary to make the game simple enough that someone with little coding skill would be able to make a script for a robot, with significant feedback, so that users will be able to see tangible improvement to their coding skills as they continue to play the game.

To control the flow of the game, we wanted it to run in time-based steps. This meant that the user would have a limited amount of time to clean the room. So the goal will be to clean as much as possible of the room in an allotted amount of time, rather than cleaning the entire room eventually with unlimited time. The challenge is then in creating efficient cleaning algorithms, rather than fast algorithms.

2.2 Solution

In the following section, we will describe the solution developed based on the outlined requirements. It explains the creation of our game and covers obstacle generation, the challenges faced, the scoring system, the game statistics display, and the step-based game structure that highlights algorithm efficiency over speed. With these points in mind we came up with a simple game, in which the user would program a robot with the purpose of traversing a room with randomly placed obstacles while also cleaning dirty tiles in the room.

Obstacle generation was implemented with the intention of generating varied rooms and challenges for the player. Obstacles are created as abstract clusters, allowing for a wide variety of rooms to be generated, without any restrictions on specific obstacle shape or size. It makes it less predictable, forcing users to write more generalized algorithms.

Although we kept the game relatively simple, it still presented some interesting challenges, for example generating dirtiness levels using noise generation and making sure that the generated map did not have sections blocked off by obstacles. These challenges would make sure that we get a multitude of different maps even though the parameters of each instance are the same.

In order to make the game more engaging for the players we added in a scoring system using different levels of dirtiness, meaning that each dirty tile have a score which will be added to the robots score when it cleans the tile, this makes the game more engaging as players will strive to make better scripts that can get a higher score within the time-frame of the game, it also adds a competitive aspect to the game as players can compare their scores and, by extension, their scripts.

The game will display the statistics of the active room and the robot operating in it. Four numbers are shown:

The room dirtiness displays the current dirtiness of the room that the robot is operating inside. The room percentage displays what percentage of the dirt in the room is left. Robot points displays how many points the robot has gathered. Steps display how many steps have passed, and for how many steps the game will run in total.

The game runs in steps of 500 ms, each step the robot can move or clean, its action dependant on the command given from the script. This strict step-based structure was chosen to normalize the algorithms players input. Since the robot can only take one action per step, players will not be rewarded for writing scripts that run fast, but rather for the contents of their algorithm.

The normal game speed of 500 ms per step is intended to be used while the player is constructing the algorithm itself. The slow pace lends itself well to locating unintended behaviour, and correct it. The player also has the option to view the map from the robots point of view, this is also to help the construction of algorithms. Having access to a visual guide, signifying the information the robot has, can help players to use that information effectively.

Once the player has a finalized algorithm they are pleased with, they can use the build in speed up function to play the game at 10 times speed. This reduces the time spent per step to 50 ms, and lends itself well to testing the efficiency of an algorithm. The player can go through many rooms quickly, letting them easily collect data for their algorithm.

Chapter 3

Language introduction

The language used for writing algorithms for the robots within the game, is called ATAKK.

This part of the report provides an overview of the ATAKK programming language's features and functionality. It includes explanations and code examples for variables, subprograms (routines), loops, conditional statements, logic, math operators, and the print statement. The section also references the relevant context-free grammar used in ATAKK and highlights specific design decisions and their implications.

3.1 ATAKK runtime environment

The script the players write is intended to be run once for each step. The script should be able to independently analyze and decide the action for that particular step, and then run as a new instance the next step.

Once the script settles on a course of action, either moving or cleaning, it will send that information to the game. The script will then stop, and it will not be interpreted further than that command. This is important because the player should not be able to do more than one action per step, so terminating the script once a command is given is how we circumvent that.

Since the script is run in these instances of steps, variables are only stored as the script is being interpreted, and are cleared, going into the next step. This means that values of the variables will not remain between the steps of the game, once the robot has decided what to do, and has sent that information to the game, the script is terminated and all information lost.

One exception to this rule is the variable `PERM`, which will retain any given value between steps. This variable was included very late in the development process as the group was testing the game. It was discovered that without some communication between steps, the player would be extremely limited in their options. The `PERM` variable allows for commands to be sent between steps in the form of integers, meaning a series of top level if statements can control how the robot will act given the value of `PERM`. For an example of how `PERM` was used to create an algorithm impossible without its existence, view the zigzag algorithm presented in 6.2.

3.2 Requirements

Our main goal was to make ATAKK in a way that users could learn the basic principles of writing an algorithm. It can help those new to programming focus on understanding how algorithms work. For experienced programmers, ATAKK presents a different challenge due to ATAKK's unique syntax. It forces users to think less about the coding itself, and more about the solution they are implementing.

ATAKK is a domain-specific programming language we created for this project. It is meant to control a robot within the game, and not necessarily be used anywhere else. Its main purpose is to script the movement algorithm for the robot.

ATAKK should not include everything expected of modern coding languages, rather it should be as limited as it can be. The goal of the language is to write self contained algorithms, so standard support for object oriented programming, such as classes, will not be needed. The language should be kept limited in its scope, so users can have an easy overview of what is and isn't possible within the game, resulting in less syntax to be memorized.

3.3 Solution

ATAKK mirrors many conventional programming languages in regards to what it can do. This the following section will be an overview of ATAKK's functionality with code examples. Alongside some code examples, the relevant context-free grammar will be presented. For an overview of the context-free grammar of all expressions and statements implemented into ATAKK, consult section A.4 of the appendix.

Variables

```
store value as my_variable;
```

ATAKK only uses global variables, and all variables already exist with the value 0.

Variables are limited to integer values; there are no strings or booleans in ATAKK. Other variable types were not included since time constraints did not allow a proper focus on them; higher priority features were focused instead. It was determined to be a 'nice to have' instead of a 'need to have', as the current integer-only variables are sufficient. This was determined by the group writing several algorithms of varying complexity, to test the limits of what was possible within the confines of ATAKK.

The boolean was initially planned to be included as a minimum requirement alongside integers. Instead of a full implementation of this, where booleans within ATAKK would have a literal boolean value. It was decided that treating the booleans like integers 'under the hood' would have no impact on the experience of using booleans, while freeing up resources to work on other, more intricate, parts of the language.

Strings were never intended to be included in ATAKK; they were determined to be bloat since they wouldn't have any unique use cases within ATAKK. The value of strings would have been in our preset routines, where using words for intake variables, rather than numbers, would be more accessible for new programmers. Instead of implementing strings, we solved this by including constants so that users have easy, convenient, and intuitive access to essential values, such as the values for each direction of movement/sensing. We implemented booleans similarly, resulting in a purely visual implementation, letting all expressions still evaluate to an integer.

Subprograms

```
__ Declare method __
new routine args
    ...
as my_routine;

__ Context-free grammar __
    NEW ROUTINE var_list stm_list AS VAR SEMI
|   NEW ROUTINE stm_list AS VAR SEMI

__ Call method __
start my_routine args;
```

In ATAKK, subprograms are named routines, and is by far the most unorthodox syntax in ATAKK compared to other more common languages. It is a way to store a set of instructions, a subprogram, and call upon it later.

The syntax was created to match the syntax of declaring variables, where you first declare the innards (the value, or here, the subprogram) and then the name it should be saved under.

All variables created within the routine are global variables, similar to all other variables within ATAKK. This choice was made partially to save on time, as the support to implement a function similar to the 'return' function seen in other languages was not there. This meant that most of the routine implementation would need to be rewritten, which wasn't possible given the time constraints. Therefore, making all variables declared within a routine global, would still allow users to extradite information out of them. This was deemed essential functionality, as running repeated calculations and similar programs is the intended use of routines.

Routines are the way the script will send commands to the game world and receive information about it. There are four preset routines, three of which have some direct connection to the game itself. The `clean` routine takes no argument, it cleans the tile the robot is on, terminates the script, and moves the game one step forward. The `move` routine takes one argument specifying direction, it moves the robot, terminates the script, and moves the game one step forward. `radar` takes one argument specifying direction, it returns the dirtiness level of the direction. The last preset routine is `random`, which was implemented because some users may want to experiment with making a robot randomly move. It takes two arguments, an upper and lower limit for the random generation of numbers, both inclusive.

Loops

```
__ While loop __
think logic about
    ...
end

__ For loop __
count my_variable as value find logic else direction
    ...
end

__ Context-free grammar __
    COUNT VAR AS expr FIND expr ELSE INC stm_list END
|   COUNT VAR AS expr FIND expr ELSE DEC stm_list END
```

The loop is structured much like it is in other high-level languages, but with all symbols replaced by some keyword.

The for loop is restricted to counting one up or one down each iteration. users can still implement custom counting intervals by using the `SUB` and `ADD` operators. The word `direction` is not a variable; it can only take the form of one of two keywords, `inc` and `dec`. Writing `dec` will decrease the counting variable, `inc` will increase the counting variable. These loops allow the users to manipulate and traverse ranges with precision and simplicity. Such features make ATAKK particularly easy to use for beginners learning the fundamentals of loop control.

if, elif, else

```
if logic then
    ...
elif logic then
    ...
```

```
else
    ...
end

__ Context-free grammar if-construct__
    IF expr THEN stm_list END
|   IF expr THEN stm_list elif_list ELSE stm_list END
|   IF expr THEN stm_list ELSE stm_list END
|   IF expr THEN stm_list elif_list END

__ Context-free grammar elif_list__
    ELIF expr THEN stm_list
|   elif_list ELIF expr THEN stm_list
```

The if statements can be seen as parts of one much bigger construction, with else and elif, namely the if-construction. This is also why we only require the `end` keyword at the end of the if-construction since it is (to the parser) seen as one big statement instead of many small ones. This if-construction only works for chains like this: initial if, some number of elif, and one (or zero) else. If statements alone cannot be chained in the same way and would need their own `end` keywords.

The only mandatory part of this construction is the if part; all the others can be left out if they are not needed.

Logic and math

```
x GRT y    returns 1 if value of x is greater than y, 0 if not
x LRT y    returns 1 if value of x is less than y, 0 if not
x EQL y    returns 1 if value of x and y are the same
x NEQ y    returns 1 if value of x and y are not the same
NOT x      returns 1 if value of x is 0 or below, 1 if value is 1 or
    above
x AND y    returns 1 if value of x and y are both 1 or above, 0 if
    they are both 0 or below
x OR y     returns 1 if value of either x or y is 1 or above, 0 if
    they are both 0 or below
x ADD y    returns the value of x and y added together
x SUB y    returns the value of x and y subtracted from eachother
```

Words have replaced math and logic operators; here, max three letters, all caps. They are in all caps to set them apart from keywords; since they aren't, they are operators and return a value. While operators can be (and in this case, are) keywords, meaning something with a predefined meaning, operators specifically refer to some mathematical or logical action. For example, comparing two integers to check if they are equal. They are three letters since it would allow us to visually group the operators and still use intuitive acronyms, making them easier to remember (For example, saying E-Q-L sounds like the word equal, and

ADD is a common shortening for addition).

Print

```
    print value;
```

ATAKK has a 'hidden' statement called print. This statement will not be introduced to players in the user manual, as it is useless to them if they run the.jar file from the terminal. The print currently uses `System.out.println` to output the value given to it, so without a visible terminal, this output cannot be viewed.

Plans were made to make this print usable by users by supplying an asset within the game that can hold the outputs from this statement, but it was never implemented. It was set aside and never picked up again. It could have been valuable for the player when debugging.

3.4 Errors

Currently, errors in script writing depend on error handling native to an external library used in the language's implementation. This error handling could be better and often outputs unhelpful messages. Because the player will (presumably) not run the program from the command line, these messages will also be completely invisible to the player.

Future iterations of this project should improve this part of the program, as it is hard for users to understand what parts of their script go wrong. This is unfriendly to newcomers, as it relies on the user's own analysis of their script to locate any errors.

Chapter 4

Program description

The game integrates an interpreter to read the user made scripts and execute them in the game. Our implementation of the interpreter consists of 3 phases:

- Lexical phase - This phase utilizes a lexer to scan through a given text file. The lexer will translate this text file into a series of symbols based on given regular expression.
- Parsing phase - This phase uses a parser to section the symbols given by the lexer into recognized code. It does this by defining expressions and statement in a context-free grammar. It will translate the symbols into an Abstract Syntax Tree (AST).
- Interpreting phase - This phase uses an interpreter to execute the AST, moving from the 'bottom' of the tree to the top. This outputs the desired result of the script, in this case, it should output either a command to move the robot, or clean.

We will explain our approach to each of these 3 elements in detail in the following chapters, starting at the first phase, the lexer.

4.1 Lexer - Java Flex

In our project, we utilized a tool called JavaFlex, also known as JFlex, it simplified the process of generating scanners or lexers. A lexer in the context of programming languages is a tool that processes input text to identify and classify substrings (lexems) according to predetermined patterns, effectively laying the groundwork to help the parser.

JFlex reads specifications which describe the lexical structure of a language through regular expressions and actions associated with these expressions. These rules are usually kept in a .flex file which JFlex then reads to generate a Java program that can perform lexical analysis [JFLEX].

For our project's language, we started by defining a set of tokens/keywords, identifiers, literals, and operators that we wanted our language to support and specifying how to recognize them using regular expressions in the .flex file. JFlex then generated a lexer that could convert raw script text into tokens. These tokens were then given to the parser generated by JavaCUP, enabling our software to interpret and execute user-generated code.

In conclusion, integrating JavaFlex into our project allowed us to automate the tokenization process, which is crucial for any language processing project. Despite the initial issues in crafting precise regular expressions, the automated token generation facilitated by JFlex proved important in building an efficient and robust parsing system for our game.

4.2 Parser - Java CUP

We decided to employ JavaCUP for our project; the "CUP" in JavaCUP stands for "Constructor of Useful Parsers." It's part of the many tools and libraries provided by the Java programming environment designed to generate parsers from specified grammars. Similar to grammatical rules that help understand sentences in a language, a parser is a tool for analyzing text to identify logical structures.

JavaCUP works by taking user made rules that specify grammar (context-free grammar) for the programming language, similar to linguistic grammar but tailored for programming languages, and automatically generates a Java parser based on these rules. This functionality proved to be crucial for our project, where the alternative was for us to develop a parser from the ground up, which is outside of the scope of this project.

Context-free grammar means that the context of the text, or here, line of code, doesn't mean anything. A written line of code should mean and do the same thing regardless of where it is written. This is what a parser uses to generate executable code, by recognizing patterns created by the context-free grammar.[Foundations of Computation]

At first, we had to define the syntax of our scripting language using a specification file. This process, which was meticulous and somewhat tedious, involved outlining every rule and exception that the parser needed to handle. This ensured the parser could interpret scripts accurately and helped us filter our ideas for the project.

In this specification file, we defined the symbols of your grammar, including terminal symbols (for example T1, T2) and non-terminal symbols (for example N1, N2), along with the production rules (for example LHS ::= RHS1 — RHS2;). Each production rule was given action code (for example : RESULT = myFunction(); :) which the parser would call after performing a reduction with that particular production. We can use these callbacks to assemble an Abstract Syntax Tree (AST). [JAVACUP]

JavaCUP automatically generates a parser that can understand and process our scripts, enabling it to perform commands as dictated by the user scripts.

This parser was then integrated into our software.

In summary, JavaCUP allowed us to create a custom parser that was integral to the functionality of our software. Despite the complexity involved in defining the grammar, the effort was justified by the capabilities it granted for our project.

4.3 Interpreter

This section will discuss the parts of the interpreter. While other parts, such as the parser and lexer, were generated by helpful liberates, the interpreter had to be made from scratch to fit our language needs. This is because this is the part where we define the actual functionality of each part of the language.

The interpreter is given an AST from the parser, which, in our case, is a list of statements. The interpreter will then execute each statement, any statements within the statement, any statements within those, and so on. This is a rather abstract concept and is much better understood visually. We will, therefore, visualize the following code snippet as a syntax-tree.

```
new routine x,y
    think y GRT 0 about
        store x_times_y ADD x as x_times_y;
        store y SUB 1 as y;
    end
as multiply;

start multiply 5,2;
store x_times_y ADD 5 as result;

result -> 15
```

We built the classes that JavaCUP uses to generate the executable program. The classes created were formed from two abstract classes: Statement and Expression. Statements are objects that can be executed and may store variables and routines; they have no explicit value. Expressions are objects that are evaluated for the extraction of explicit values; they may not modify variable or routine storage.

ATAKK only accepts statements as ”top level” executable, this means, that unlike languages such as Python, the user is unable to execute a program where an expression is evaluated in isolation. This means that expressions must always be evaluated in the context of a statement, such as acting as a logic operator, or value of a variable.

JavaCUP parses the whole program into a list of Statements, executed in a while loop during the interpreting phase.

An essential part of our interpreting phase is the ability to terminate interpretation at will. To do this, every statement returns a boolean: true if the program continues after the statement has passed, false if it should terminate

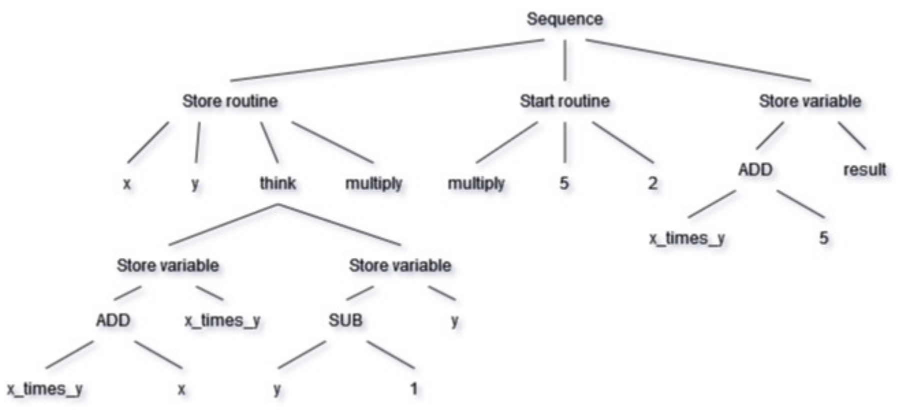

Figure 4.1: Visualization of the AST for the code snippet above. Execution happens start from the bottom-left, executing by moving upwards and to the right when one branch is fully executed.

after the statement. This boolean will be carried through the entire program tree, preventing it from executing further than it already has.

The interpreter is closely tied to the game itself, and almost all preset routines call for some function within the game world to modify or interact with it. We also followed common conventions when deciding how to execute statements from ATAKK. This is most obvious when using the ATAKK version of the for statement (the count statement). Like most other programming languages, this is defined functionally as a while loop.

4.4 Game

4.4.1 JavaFX

For the visual representation of the game, we used JavaFX. A circle represents the robot itself, and various colors represent the state of each tile, such as the dirtiness level and whether or not an obstacle is present.

At the same time, we created two different views of the game world: one that shows the whole world and one that is focused on the player's current location.

The robot-focused point of view (POV) is a 3 by 3 section centered around the robot's current location, blacking out all remaining tiles. This is done to give a perspective close to what the robot can see, being the four cardinal directions; all other tiles are blacked out. The dirtiness level is represented by various shades of blue; the darker the shade, the dirtier the floor, while obstacles are represented by black. We also added some statistics to the game screen so that players could track their robots' current performance. These statistics include the world's total dirtiness level, the percentage of dirt remaining, and the robot's current points.

JavaFX helped us design a highly interactive and visually appealing user interface. It enabled us to create responsive visual elements. JavaFX's robust event handling facilitated smooth interaction with the game state in real-time, allowing players to see immediate visual feedback based on the robot's actions.

The utility to produce multiple views such as the full world view and the robot focused POV, notably improved the user experience by offering both a strategic overview and a detailed close-up perspective. in addition JavaFX's set of UI components made it straightforward to integrate statistics, providing players with statistics to monitor and evaluate their robot's cleaning efficiency. This level of interactivity and visualization wouldn't be possible to achieve without the advanced features offered by JavaFX.

4.4.2 Game loop

We wanted to make a loop that used time as a factor rather than any game conditions. We utilized the Timeline function in JavaFX, which acts as an event handler with a delay between each call to the body of the function; this delay is measured in milliseconds, which, combined with the ability to set a count for the amount of calls, allows us to define how long each instance of the game will run for.

Once the game runs out of steps, it will freeze, letting the user read and evaluate the statistics of the script supplied to the robot.

4.5 Map generation

The game utilizes procedural generation to make maps. This was accomplished with varying techniques depending on what part of the map we generated. The map consists of two elements combined to construct the final map the player will interact with. The first part is the floor tiles' map, where dirt distribution is determined. The second part is the obstacle generation, which is done separately since it needs to be validated. We cannot produce maps where obstacles block off parts of it; that would be problematic since it will impact how much of the dirt a robot can reach and then modify the max score one can get on that map.

Obstacles and floor tiles were generated together in the first iteration of this procedural generation; the map as a whole would be regenerated if it was invalid. Once the decision was made to use more advanced methods for dirt distribution, it was obvious that it would be better to generate the two separately to avoid having to regenerate the floor map. Once an obstacle map is validated, the World class will iterate through the obstacle and floor maps, populating the map with their respective parts.

The different parts of the map generation will now be presented in detail, elaborating on how each part is generated. Finally, we will explain how obstacle maps are validated. A finalized map will look something like this:

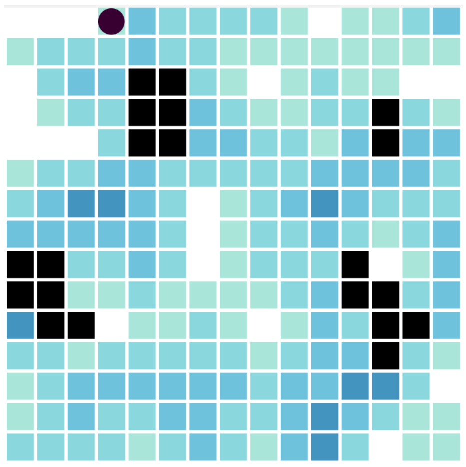

Figure 4.2: Picture of game map, with noise used to generate floors and cluster obstacle generation

4.5.1 Floor generation

For the Floor tiles, we use noise generation to determine each tile's dirtiness value; we generate a 2D array with values between 0 and 1 using Perlin noise generation. We then compare this array to our game map and give a dirtiness value based on the noise level; this effectively allows us to provide a percentage chance of each dirtiness level since the noise value is between 0 and 1. This way, we can easily adjust at any point, whether we want more dirty or cleaner tiles from the beginning of the game. We use a randomly generated seed for our noise generation; this way, we will get different values even though we apply the same variables. This seed, combined with the random generation of the obstacles, allows for many other maps. Even if one were to get the same seed twice in a row, the obstacles would likely differ as they are generated separately.

Perlin noise was chosen because it would generate gradient clusters of filthy tiles (tiles with a high dirtiness score). This would allow the player to create scripts using this fact, making the robot search for the center of a cluster to get more points since it will be dirtier the closer you get to the center of the cluster. Instead of making our script for generating this Perlin noise, we used a preexisting script from Rosetta Code [Rosetta Code Perlin Noise]; this method takes 5 parameters. The first two determine the map size the Perlin noise will be generated. The third (octave count) determines how "complex" the noise will be, many small clusters vs few large ones. The fourth (persistence) determines how faded the clusters will be. The fifth is the seed; if we use the same seed, octave count, and persistence, we will generate identical noise, so this is the component that is used to randomize the noise since the other two will modify the nature of the noise itself.

4.5.2 Obstacle generation

It was decided that we needed an algorithm to generate obstacles, which would result in more cohesive obstacles rather than the (often) scattered look of random generation.

The algorithm chosen would generate fewer larger obstacles, which would snake out from a source. It still has many random elements, generating many types of obstacles, thus allowing for many different kinds of challenges.

The generation works by finding four source positions in each of the four equally divided quadrants of the map. A random number will then be chosen as the length of this snake that will be generated. The snake will then randomly select any of the four cardinal directions to move in, generating an obstacle there, making the newly generated obstacle the new source, and the length will be decreased by one. The snake will stop if it tries to go outside the map or it runs out of length. This method will generate large cohesive structures that can be either large blobs or maze-like.

The obstacles are also generated as an integer matrix instead of a tile matrix. This is because integers are much faster to work with than most other types of objects since they are primitive types. This allows for fast generation and lets us easily discard an invalid map (meaning maps where not all floor tiles are reachable) to generate a new one without impacting performance.

4.5.3 Checking map validity

To ensure that a map is valid we need an implementation that verifies that every floor tile is within the reach of the robot at the start of the game.

To check map validity we create a sort of water effect that spreads to every single floor tile, then we check if the amount of tiles that the water reached is equal to the amount of free floor tiles. The way it is implemented in this program is by using a variant of a DFS (Depth First Search), which works by having a list of visited tiles/nodes that get updated with the current node for every call to the method. This works by giving the method a starting node, an obstacle map, and a visited list. The list and map is passed down every time the method is called, and the x,y of the node is different depending on which node we are currently visiting. When we visit a new node it is added to the `vistedList`, from that node we do recursive calls to the method. This continues until we are at a node where we cannot reach a new node that has not already been visited. It then returns to a previous node, where there are still connected nodes that have not been visited yet. This is the part of the method that is similar to a DFS, and while it is usually used on trees, it has here been modified to work on a matrix. Once the method no longer has any new nodes to visit, we check whether the amount of visited tiles are equal to the total amount of floor tiles. If that's true then the map is valid and we don't need to generate new obstacles.

As expected the runtime of the DFS algorithm is adequate, it will check every tile once and add the result of the check to an array, namely `visitedList`.

Since the array is defined before the method, and it has a static size, accessing or changing an element in the array has the complexity O(1). Since all other checks done within the method are also O(1), the overall complexity of checking the validity of a map is linear. This means that the total runtime of the method would be determined by the size of the map.

Chapter 5

User manual

5.1 Running the program

The program is a .jar file called RobotVaccum.jar. To run the program, simply run this file and the program will run with the preset script.

To replace the script with your own, replace the file inside Scripts/Active/ with your own .txt file containing the script. Ensure that the Active folder only has 1 script in it at a time. The name of the script doesn't matter. Any additional scripts can be stored in the Scripts/ folder.

5.2 Game rules

The game is inspired by a novel technology, the robot vacuum.
In the game, a user constructs an algorithm to control a robot vacuum in a 2D room of tiles. The room is a finite space, that does not change size during the course of the game. The floor of the room is dirty, and the robot should clean it to the best of its ability.

5.2.1 Rules

The robot starts its cleaning protocol in the top left corner of the room, and has a certain amount of steps to clean the room. Each step the robot will be allowed to either move or clean, you control which action the robot will take. Each game step will run the script from the beginning once again, so your algorithm should take this into account. The game is run in a room made up of tiles, the tiles can either be floor or an obstacle.
Here are the ground rules for the game to keep in mind while writing an algorithm:

- The robot may only move in the 4 cardinal directions.
- Moving the robot moves the game one step.

- The dirtier the floor is, 4 levels of dirty, the more points the robot receives for cleaning it.
- Cleaning the floor tile the robot occupies moves the game one step.
- The robot can only occupy floor tiles.
- The game runs for a total of 300 steps.

5.3 ATAKK Documentation

The coding language used for the game, called ATAKK has its own somewhat unique syntax and grammar, and a few quirks.

How to code in the language will be described here:

5.3.1 Types

This part is simple, there is only one type in this language, namely, integers. All variables, preset variables and logic are integers. This makes silly things such as:

```
TRUE GRT FALSE -> TRUE
Interpeted like: 1 > 0 -> 1
```

Completely possible and legal.

The values "TRUE" and "FALSE" are just constants that you cannot overwrite with the values 1 and 0, but any value above 0 will be accepted as true as well, meaning that:

```
if 10 then ... end

if TRUE then ... end
```

Will give the same result. False works much in the same way, all numbers from 0 and below are false.

This language has few symbols, and indentation doesn't matter. Instead it uses many keywords, to construct a command like language, which focuses on "sounding" like you are instructing a robot to perform some actions instead of writing abstract code.

5.3.2 Variables

Variables are quite important, as it is the only way to store numbers to be used at a later point. Every variable in this language is global, even iterators or routine variables, so be careful not to overwrite anything. On the flip side, you will never get an error for writing a variable you haven't defined, all variables "exist", but have the base value of 0.

You define variables like so:

```
store value as my_variable;
```

They are called by simply writing out their name, the above variable would be called by writing `my_variable`.

There are a number of constants for convenience, they are variables which have a predefined value that cannot be overwritten by a user.

Name	Description	Actual value
CURRENT	Refers to the tile the robot is on	0
UP	Refers to the tile north of the robot	1
RIGHT	Refers to the tile east of the robot	2
DOWN	Refers to the tile south of the robot	3
LEFT	Refers to the tile west of the robot	4
CLEAN	Value of a clean tile	0
MESSY	One possible value of a dirty tile	1
DIRTY	One possible value of a dirty tile	2
FILTHY	One possible value of a dirty tile	3
DISGUSTING	One possible value of a dirty tile	4
WALL	Value of a wall tile	-1
OBSTACLE	Value of a blocked tile	-2
STEP	Amount of steps passed since the game begun	0 to x
TRUE	Lowest possible value considered true	1
FALSE	Highest possible value considered false	0

The first five constants, CURRENT, UP, RIGHT, DOWN, LEFT are classified as DIRECTIONS. The next five, CLEAN, MESSY, DIRTY, FILTHY, DISGUSTING are classified under DIRTINESS. The next two WALL and OBSTACLE are classified under OBSTRUCTIONS. The last three STEP, TRUE and FALSE have no formal classification.

While these classifications have no use within the code itself, it doesn't matter where the numbers come from, it is useful in the documentation.

Some preset routines can only handle numbers within one of these classifications, but it doesn't matter if it gets that number in the form of one of these constants, or just as the number itself. This means that if you wish, you can just as well write 0, CURRENT or FALSE instead of CLEAN and everything would work just the same.

There is one more constant, which is a permanent storage solution under the name "PERM". This variable will retain its value between steps of the game, so you can store numbers in here for your robot to read later.

5.3.3 Routines

Routines is the way to store subroutines (a smaller program within the program itself) for later or repeated use. To declare and later call a routine do the following:

```
new routine argument_1, argument_2,...
    code_snippet
as routine_name;

start routine_name argument_1_value, argument_2_value,...;
```

Routine arguments act exactly like variables, meaning that an argument with the name "bob" will, once the routine is called, become a global variable named "bob". After the routine has run, all variables created by routine arguments will be voided. If you name a routine argument the same an a variable you are already using, this variable will be overwritten and voided, so be careful! You can declare and run routines with no arguments, in which case, just leave them out.

Unlike other languages, you cannot return any values from routines. However, since all variables are global, even those declared inside a routine, returning values is as simple as storing them.

Unlike variables, routines don't "exist" unless declared, so declaring routines within routines can result in errors, if the parent routine is not run before the child routine.

A simple example would be a routine that tests for cleanliness, and acts accordingly:

```
new routine clean_value
    if clean_value EQL MESSY then
        start move LEFT;
    elif clean_value EQL DIRTY then
        start move RIGHT;
    elif clean_value EQL FILTHY then
        start move DOWN;
    elif clean_value EQL DISGUSTING then
        start clean;
    end
as eval_cleanliness;

start radar LEFT;
start eval_cleanliness radar;
```

Here we also introduce a few preset routines, they interact with the game world or perform actions that would otherwise be impossible. Some of the preset routines return values, they will be returned in variables with the same name as the routine that created them. These routines cannot be overwritten by a user.

clean
Cleans the tile the robot is on, this will stop the script and send the command to move to the next step of the game.

Takes no arguments and has no return value.

move
Moves the robot in the specified direction, this will stop the script and send the command to move to the next step of the game.

Takes one argument of the DIRECTIONS classification, except CURRENT, and has no return value.

radar
Reads the dirtiness level of the tile in the specified direction.

Takes one argument of the DIRECTIONS classification.

Returns a value of either the CLEANLINESS or OBSTRUCTIONS classification. Return variable ”radar”.

random
Generates a random number.

Takes two arguments a minimum and a maximum, both inclusive.

Returns a random value between minimum and maximum. Return variable ”random”.

5.3.4 Logic and math

Logic and math is written out with simple two-three letter keywords. While all the usual logic operators are represented, there is only limited amount of math operators. Here is a list of logic and math keywords, and its java equivalent:

Keyword	Java equivilant
x EQL y	x == y
x NEQ y	x != y
x GRT y	x > y
x LRT y	x < y
NOT x	! x
x AND y	x && y
x OR y	x || y
x ADD y	x + y
x SUB y	x - y

You can wrap logic in parenthesis to prioritise it. The language will automatically prioritise EQL, NEQ, GRT, LRT over other logic, so:

```
    (x GRT y) AND (x GRT z) == x GRT y AND x GRT z
```

Furthermore it will prioritize ADD and SUB over all logic, so:

```
(x ADD y) LRT (x SUB z) == x ADD y LRT x SUB z
```

5.3.5 Statements

ATAKK has a number of statements which you can use. They are if, elif, else, count, and think.

if, elif, else
The `if` statement is constructed like this:

```
if logic then ... end
```

If the logic is true, the program (here represented by "...") will execute. It can be followed by any number of `elif` statements:

```
if logic then ...
elif logic_2 then ...
elif logic_3 then ... end
```

The `else if` will only test its logic if the original `if` or preceding `elif` was false, and execute its program if its own logic is true. Any `if` or string of `elif`'s may be followed by one `else` statement:

```
if logic then ...
else ... end

if logic then ...
elif logic_2 then ...
elif logic_3 then ...
else ... end
```

The `else` will execute if the original `if` and all `elif`'s following it is false.
You only need the `end` keyword at the very last statement in the if chain

count
The count statement is constructed like this:

```
count var_name as value find logic else inc/dec
    ...
end
```

The count creates a variable named "var_name" which we call the iterator and it has the value "start_val". It then tries to find the point where logic flips from true to false by iterating over the iterator. To inc or dec controls how the count iterates, inc increases the iterator by 1 each round, and dec decreases the iterator by 1 each round.

Usually the logic statement will include the iterator, so you can iterate from one number to another. Here are two examples which iterates over some ranges, the values of the iterator will be written out at the end:

```
count to_the_moon as 0 find to_the_moon LRT 4 else inc
   store to_the_moon as position;
end

result: position :
       0
       -> 1
       -> 2
       -> 3

count to_the_core as 0 find to_the_core GRT -4 else dec
   store to_the_core as position;
end

result: position :
       0
       -> -1
       -> -2
       -> -3
```

think

The think statement executes a code snippet over and over so long some logic is true, equal to a while loop in other programming languages. It is constructed like this:

```
think logic about ... end
```

Be careful with the logic here, as it should at some point reach the conclusion that it is false, otherwise it would run forever. You can also interrupt a think statement with any preset routine that terminates the script.

Here is an example where we randomly decide between moving and cleaning, and then sense randomly until we find somewhere to move:

```
start random 1, 20;

if random GRT 15 then start clean; end

start random 1, 4;
start radar random;
think (radar LRT 0) OR (radar EQL 0) about
   start random 1, 4;
   start radar random;
end
```

```
start move random;
```

Notice that this think statement will run forever if the robot stands amongst clean tiles...

5.3.6 Examples

Move and clean randomly
Program to make a robot move randomly and clean randomly.

```
store 11 as chance_clean;
store 21 as chance_up;
store 31 as chance_right;
store 41 as chance_down;

start random 1, 50;

if random LRT chance_clean then
    start clean;
elif random LRT chance_up then
    start move UP;
elif random LRT chance_right then
    start move RIGHT;
elif random LRT chance_down then
    start move DOWN;
else start move LEFT;
end
```

Move randomly, clean when dirtiness is above a threshold
Program to make a robot move randomly and clean once the dirtiness of the tile it is on is above a certain threshold.

```
store 2 as threshold;

start radar CURRENT;
if (radar GRT threshold) OR (radar EQL threshold) then
    start clean;
end

start random 1, 4;
start radar random;

think radar LRT CLEAN about
    start random 1, 4;
    start radar random;
end
start move random;
```

This will never loop forever since the robot will never be in a situation where it is completely surrounded by OBSTRUCTIONS.

Chapter 6

Testing

6.1 isValid and genObs

We decided to test two methods called genObs and isValid in the World class to check their robustness and correctness. We implemented a series of JUnit tests. We chose these methods because they're important central pieces of our system, as they're responsible for generating the obstacle maps and validating them. Below is an analysis of each test and an explanation of the results.

Firstly, the testGenObsMinSize test checks the genObs method when the map size is the smallest possible (1x1). This test failed with a StackOverflowError. Since a one-square map can't have any valid obstacle generation that allows movement, the genObs method continuously tries generating new maps recursively, leading to an infinite loop.

Afterwards, we have the testGenObsZeroDimensions that analyses what happens when the map's dimensions are set to zero. This test fails with an IndexOutOfBoundsException because a zero-dimensional map is invalid, and attempting to access an element leads to an error.

The testGenObsLargeMap examines the action of generating a very large map (1000x1000). This test fails with a StackOverflowError. The reason for this is the fact that the isValid method uses a static array and the recursive nature of the isValid method, which is used within genObs. The deep recursion required causes a stack overflow when validating a very large map.

Subsequently, the testGenObsBoundaryConditions explores whether the genObs method handles the map's boundary conditions correctly. This test passed as genObs successfully generated a map with all obstacles correctly placed within the map boundaries.

The testGenObsRecursion test guarantees that the genObs method can correctly handle a small map (5x5) with the recursion terminating. This test passed because the method correctly generated a small map and terminated the recursion as wanted.

Afterwards, the testIsValidWithValidMap tests the isValid method with a valid obstacle map. This test passes, signifying that the method correctly identified a valid map.

The testIsValidBlockedStart checks if the isValid method correctly recognizes an invalid map in which the robot is blocked in the starting position; this test passes, which confirms that it accurately detects invalid maps.

The testIsValidFullyBlocked tests whether the isValid method detects a fully blocked map. This test passed, affirming that the method recognises a map with no open paths.

Finally, the testIsValidLargeMap test checks that the isValid method works with a very large map (1000x1000). This test is a success, showing that the method can process very large maps without issues as the map is valid.

Overall, these tests opened our eyes into the limitations and performance of these two methods. The failures for testGenObsMinSize, testGenObsZeroDimensions and testGenObsLargeMap highlights the option for improving the handling of edge cases and recursion depths. The succesfull tests reaffirm that the methods function correctly when normal conditions are met.

6.2 ATAKK

To effectively test our language ATTAK, we implemented a zigzag cleaning algorithm. This algorithm was picked for its simplicity and the way it comprehensively covers the map, which makes it a perfect candidate to test our language's fundamental capabilities.

Our implementation of the zigzag algorithm focuses on making the robot clean the tiles by moving back and forth in horizontal lines. We initialized the algorithm by setting up variables. The robot's starting position is top left, and we used a global variable, ”PERM,” to track the robot's direction: 0 for moving right and 1 for moving left.

The robot commences by utilising the radar method to check the cleanliness of the tile it's on, if the tile is dirty the robot executes the clean method to clean that specific tile.

The robot proceeds in the direction selected by the ”PERM” variable. Before that specific move, the robot uses the radar method to check if the next tile is an obstacle or a wall. If such things are detected, the robot changes directions and moves down one tile to the next row, maintaining the zigzag pattern. In practical terms, when moving right (PERM EQL 0), detecting an obstacle or wall makes the robot set ”PERM” to 1 and move down.

Here is the ATTAK implementation of the zigzag algorithm:

```
start radar CURRENT;

if radar GRT CLEAN then
    start clean;

elif PERM EQL 0 then
    start radar RIGHT;
```

```
        if radar EQL OBSTACLE OR radar EQL WALL then
            store 1 as PERM;
            start move DOWN;
        else
            start move RIGHT;
        end

    elif PERM EQL 1 then
        start radar LEFT;
        if radar EQL OBSTACLE OR radar EQL WALL then
            store 0 as PERM;
            start move DOWN;
        else
            start move LEFT;
        end
    end
```

This test verifies that the robot correctly executed the clean command on dirty tiles. The robot's ability to move right and left, changing direction upon encountering obstacles or walls, and moving down a row to continue the zigzag pattern.

Chapter 7

Conclusion

The development of the "Robot Game" reaches its goal of demonstrating the integration of educational purposes with interactive gameplay to teach fundamental programming concepts. By utilizing our custom programming language, ATAKK, the game engages users in problem-solving tasks. It requires them to program a robot to navigate and clean a room with randomly generated obstacles and dirtiness levels. Integrating tools like JavaCUP and JFlex for parser and lexer generation enabled an organized system for processing user-generated scripts.

Throughout the project, we faced and overcame different challenges. These included designing a game that was both simple and complex enough to serve as a platform for demonstrating ATAKK's capacities, ensuring that the game was accessible to users of varying coding skill levels, and creating a dynamic and unpredictable environment through generation of obstacles and dirtiness levels.

Implementing a time-based step structure within the game highlights the importance of efficient algorithm creation over speed, encouraging users to focus on the quality and logic of their code. The scoring system and game statistics display provide feedback, improving user engagement and allowing players to track their progress and improve their coding skills.

Furthermore, the development of ATAKK itself required thorough consideration of language design principles. We prioritized simplicity and functionality, limiting the language to integer variables and a set of core commands while ensuring it remained strong enough to write valuable algorithms for the game.

In conclusion, the "Robot Vacuum" game provides an engaging and creative platform for beginners to practice and comprehend programming principles. As we look to the future, there is potential for developing the game's capabilities and refining ATAKK to include more advanced features.

Bibliography

[JAVACUP] CUP. Tum.edu. Published 2024. Accessed May 14, 2024. `https://www2.cs.tum.edu/projects/cup/`

[JFLEX] Klein G. JFlex - JFlex The Fast Scanner Generator for Java. Jflex.de. Published 2023. Accessed May 14, 2024. `https://www.jflex.de/`

[Rosetta Code Perlin Noise] Rosetta code. Acessed May 21 2024. `https://rosettacode.org/wiki/Perlin_noise`

[Foundations of Computation] (Critchlow & Eck). Chapter 4.1: Context-free Grammars. HTML-version. Accessed May 22, 2024. `https://math.hws.edu/FoundationsOfComputation/`

[Screeps] Screeps: MMO RTS sandbox for programmers. Screeps. Published 2014. Accessed May 28, 2024. `https://screeps.com/`

[Codewars] Codewars. Codewars. Published 2024. Accessed May 28, 2024. `https://www.codewars.com/`

Appendix A

interpreter

Chapters represent the java packages the files are sorted into, sections represent the name of the files. Generated code from jflex and java cup will be left out. Likewise, code snippets copied from the internet, such as the Perlin Noise code, will be left out.

A.1 Expression.java

```java
package robotgame.newmaven.interpreter;

public abstract class Expression {
    public abstract Integer evaluate(Interpreter interpreter);
}
class Variable extends Expression {
    String name;
    public Variable(String name){
        this.name = name;
    }
    public Integer evaluate(Interpreter interpreter){
        return interpreter.getVar(name);
    }
}

class Number extends Expression {
    Integer value;
    public Number(Integer value){
        this.value = value;
    }
    public Integer evaluate(Interpreter interpreter){
        return value;
    }
}
```

```
class Bool extends Expression {
    Integer value;
    public Bool(Boolean value){
        if(value){
            this.value = 1;
        }
        else {
            this.value = 0;
        }
    }
    public Integer evaluate(Interpreter interpreter){
        return value;
    }
}

class GRT extends Expression {
    Expression expr1;
    Expression expr2;
    public GRT(Expression expr1, Expression expr2){
        this.expr1 = expr1;
        this.expr2 = expr2;
    }
    public Integer evaluate(Interpreter interpreter){
        if(expr1.evaluate(interpreter) > expr2.evaluate(interpreter)){
            return 1;
        }
        return 0;
    }
}

class LRT extends Expression {
    Expression expr1;
    Expression expr2;
    public LRT(Expression expr1, Expression expr2){
        this.expr1 = expr1;
        this.expr2 = expr2;
    }
    public Integer evaluate(Interpreter interpreter){
        if(expr1.evaluate(interpreter) < expr2.evaluate(interpreter)){
            return 1;
        }
        return 0;
    }
}

class EQL extends Expression {
    Expression expr1;
    Expression expr2;
    public EQL(Expression expr1, Expression expr2){
        this.expr1 = expr1;
        this.expr2 = expr2;
    }
    public Integer evaluate(Interpreter interpreter){
        if(expr1.evaluate(interpreter) == expr2.evaluate(interpreter)){
            return 1;
        }
        return 0;
    }
}

class NEQ extends Expression {
    Expression expr1;
    Expression expr2;
    public NEQ(Expression expr1, Expression expr2){
        this.expr1 = expr1;
        this.expr2 = expr2;
    }
    public Integer evaluate(Interpreter interpreter){
        if(expr1.evaluate(interpreter) != expr2.evaluate(interpreter)){
            return 1;
        }
        return 0;
    }
}

class NOT extends Expression {
    Expression expr;
    public NOT(Expression expr){
        this.expr = expr;
    }
    public Integer evaluate(Interpreter interpreter){
        if(expr.evaluate(interpreter) > 0){
            return 0;
        }
        return 1;
    }
}

class AND extends Expression {
    Expression expr1;
    Expression expr2;
    public AND(Expression expr1, Expression expr2){
        this.expr1 = expr1;
        this.expr2 = expr2;
    }
    public Integer evaluate(Interpreter interpreter){
        if((expr1.evaluate(interpreter) >= 1) &&
             (expr2.evaluate(interpreter) >= 1)){
            return 1;
        }
        return 0;
    }
}

class OR extends Expression {
    Expression expr1;
    Expression expr2;
    public OR(Expression expr1, Expression expr2){
        this.expr1 = expr1;
        this.expr2 = expr2;
    }
    public Integer evaluate(Interpreter interpreter){
        if((expr1.evaluate(interpreter) >= 1) ||
             (expr2.evaluate(interpreter) >= 1)){
            return 1;
        }
        return 0;
    }
}

class ADD extends Expression {
    Expression expr1;
    Expression expr2;
    public ADD(Expression expr1, Expression expr2){
        this.expr1 = expr1;
        this.expr2 = expr2;
    }
    public Integer evaluate(Interpreter interpreter){
        return expr1.evaluate(interpreter) + expr2.evaluate(interpreter);
    }
}

class SUB extends Expression {
    Expression expr1;
    Expression expr2;
    public SUB(Expression expr1, Expression expr2){
        this.expr1 = expr1;
        this.expr2 = expr2;
    }
    public Integer evaluate(Interpreter interpreter){
        return expr1.evaluate(interpreter) - expr2.evaluate(interpreter);
    }
}
```

A.2 Interpreter.java

```
package robotgame.newmaven.interpreter;

import robotgame.newmaven.Util.Robot;

import java.io.FileReader;
import java.util.ArrayList;

public class Interpreter {
    Robot robot;
    Integer step;
    ArrayList<Statement> parsedScript;
    public Interpreter(Robot robot){
        this.robot = robot;
        step = 0;
        putPresets();
        try{
            parsedScript = parse();
        }
        catch (Exception e){
            System.err.println("Something went wrong during parse");
            System.err.println(e.getMessage());
        }
    }
    public void interpret(){
        try {
            for(Statement statement:parsedScript){
                if(!statement.execute(this)){
                    break;
                }
            }
            step++;
            voidAllMemory();
            putPresets();
        }
        catch (Exception e){
            System.err.println("Something went wrong during interpret");
            System.err.println(e.getMessage());
        }
    }
    void putPresets(){
        for(int i = 0; i< Presets.presetVars.size(); i++){
            putVar(Presets.presetVars.get(i),
                 Presets.presetVarsVal.get(i));
          }
        putVar("PERM",robot.permaStorage);
        if(step != 0){
            putVar("STEP",step);
        }
    }
    void voidAllMemory(){
        robot.variables.clear();
        robot.routines.clear();
    }
    public void putVar(String name, Integer value){
        if(name.equals("PERM")){
            robot.permaStorage = value;
        }
        robot.variables.put(name, value);
    }
    public void voidVar(String name){
        robot.variables.remove(name);
    }
    public void putRou(String name, RoutineStorage routine){
        robot.routines.put(name, routine);
    }
    public Integer getVar(String name){
        if(robot.variables.containsKey(name)){
            return robot.variables.get(name);
        }
        return 0;
    }
    public RoutineStorage getRou(String name){
        if(robot.routines.containsKey(name)){
            return robot.routines.get(name);
        }
        return null;
    }
    ArrayList<Statement> parse(){
        try {
            Lexer l = new Lexer(new FileReader(robot.getScript()));
            // SymbolFactory sf = new ComplexSymbolFactory();
            parser p = new parser(l /* , sf */);
            Object result = p.parse().value;
            if(result instanceof ArrayList){
              @SuppressWarnings("unchecked")
              ArrayList<Statement> res = (ArrayList<Statement>) result;
              return res;
            }

          } catch (Exception e) {
            /* do cleanup here -- possibly rethrow e */
            System.err.println(e.getMessage());
          }
          return null;
    }

}
```

A.3 lexer.flex

```
package robotgame.newmaven.interpreter;
import com.github.jhoenicke.javacup.runtime.*;

%%

%class Lexer

%line
%column

%cup

%{
    private Symbol symbol(int type) {
        return new Symbol(type, yyline, yycolumn);
    }

    private Symbol symbol(int type, Object value) {
        return new Symbol(type, yyline, yycolumn, value);
    }
%}

LineTerminator = \r|\n|\r\n
WhiteSpace    = {LineTerminator} | [ \t\f]
dec_int   = 0 | -*[1-9][0-9]*
dec_bool = "FALSE" | "TRUE"

variable_routine_name = [A-Za-z_][A-Za-z_0-9]*

%%

<YYINITIAL> {

/* Symbols */
   ";"          {return symbol(sym.SEMI);}
   "("          {return symbol(sym.LPARAN);}
   ")"          {return symbol(sym.RPARAN);}
   ","          {return symbol(sym.COMMA);}

   "end"      {return symbol(sym.END);}
   "about"      {return symbol(sym.ABOUT);}
   "print"      {return symbol(sym.PRINT);}

/* Variable declaration */
   "store"      {return symbol(sym.STORE);}
   "as"      {return symbol(sym.AS);}
/* Method declaration */
   "new"      {return symbol(sym.NEW);}
   "routine"     {return symbol(sym.ROUTINE);}
   "start"    {return symbol(sym.STARTROUTINE);}

/* Boolean and int */
   {dec_bool}    {return symbol(sym.BOOL, Boolean.valueOf(yytext()));}
   {dec_int}     {return symbol(sym.NUMBER, Integer.valueOf(yytext())); }

/* Logic parts */
   "GRT"      {return symbol(sym.GRT);}
   "LRT"      {return symbol(sym.LRT);}
   "EQL"      {return symbol(sym.EQL);}
   "NEQ"      {return symbol(sym.NEQ);}
   "NOT"      {return symbol(sym.NOT);}
   "AND"      {return symbol(sym.AND);}
   "OR"       {return symbol(sym.OR);}
   "ADD"      {return symbol(sym.ADD);}
   "SUB"      {return symbol(sym.SUB);}

/* If equivalent */
   "if"       {return symbol(sym.IF);}
   "elif"        {return symbol(sym.ELIF);}
   "else"       {return symbol(sym.ELSE);}
   "then"       {return symbol(sym.THEN);}

/* While loop equivalent */
   "think"      {return symbol(sym.THINK);}

/* For loop equivalent */
   "count"      {return symbol(sym.COUNT);}
   "find"       {return symbol(sym.FIND);}
   "inc"      {return symbol(sym.INC);}
   "dec"      {return symbol(sym.DEC);}

/* to not confuse any tokens as variables */
   {variable_routine_name} {return symbol(sym.VAR, yytext());}

/* Whitespace */
   {WhiteSpace}  {/* just skip what was found, do nothing */}
}
[^]                  {throw new Error("Illegal character <"+yytext()+">
    at column: " + yycolumn + " line: " + yyline);}
```

A.4 parser.cup

```
  package robotgame.newmaven.interpreter;

import java.util.ArrayList;

parser code {:
  public void report_error(String message, Object info) {
    System.err.println(message);
  }

  public void report_fatal_error(String message, Object info) {
    report_error(message, info);
    System.exit(1);
  }
:};

terminal          STORE, AS, SEMI, IF, THEN, ELSE, ELIF, END, THINK,
    ABOUT, LPARAN, RPARAN, GRT, LRT, EQL, NEQ, NOT, AND, OR,ADD,SUB,
    PRINT, COMMA, DEC, INC, NEW, FIND, COUNT, STARTROUTINE, ROUTINE;
terminal Integer  NUMBER;
terminal String   VAR;
terminal Boolean  BOOL;

non terminal Statement stm;
non terminal ArrayList<Statement> stm_list, elif_list;
non terminal Expression expr;
non terminal ArrayList<Expression> expr_list;
non terminal ArrayList<String> var_list;
non terminal Statement if_statement;

precedence left AND, OR;
precedence left NOT;
precedence left GRT, LRT, EQL, NEQ;
precedence left ADD, SUB;

stm_list ::= stm:s
              {:  ArrayList<Statement> lonely_statement = new
                  ArrayList<Statement>();
                  lonely_statement.add(s);
                  RESULT = lonely_statement;
                  :}
         |    stm_list:stms stm:s
              {:  stms.add(s);
                  RESULT = stms;
                  :}
         ;
```

```
expr_list ::= expr:e {: ArrayList<Expression> lonely_expr = new
    ArrayList<Expression>();
                    lonely_expr.add(e);
                    RESULT = lonely_expr; :}
          | expr_list:es COMMA expr:e {: es.add(e); RESULT = es; :}
          ;

var_list ::= VAR:i {: ArrayList<String> lonely_var = new
    ArrayList<String>();
                    lonely_var.add(i);
                    RESULT = lonely_var; :}
          | var_list:vars COMMA VAR:i {: vars.add(i); RESULT = vars; :}
          ;

stm ::=
    STORE expr:e AS VAR:var_name SEMI
    {: RESULT = new VarDeclaration(var_name, e); :}
    | THINK expr:e ABOUT stm_list:s END
    {: RESULT = new Think(e, s); :}
    | PRINT expr:e SEMI
    {: RESULT = new MyPrint(e); :}
    | COUNT VAR:i AS expr:e1 FIND expr:e2 ELSE INC stm_list:s END
    {: RESULT = new ForLoopInc(e2, s, i, e1); :}
    | COUNT VAR:i AS expr:e1 FIND expr:e2 ELSE DEC stm_list:s END
    {: RESULT = new ForLoopDec(e2, s, i, e1); :}
    | NEW ROUTINE var_list:args stm_list:s AS VAR:name SEMI
    {: RESULT = new RouDeclaration(name, args, s); :}
    | NEW ROUTINE stm_list:s AS VAR:name SEMI
    {: RESULT = new RouDeclaration(name, new ArrayList<String>(), s); :}
    | STARTROUTINE VAR:routine_name expr_list:args SEMI
    {: RESULT = new Routine(routine_name, args); :}
    | STARTROUTINE VAR:routine_name SEMI
       {: RESULT = new Routine(routine_name, new
           ArrayList<Expression>()); :}
    | if_statement:if_stm
    {: RESULT = if_stm; :}
    ;

if_statement ::= IF expr:logic THEN stm_list:statement_true END
                  {: RESULT = new If(logic, statement_true); :}
                | IF expr:logic THEN stm_list:statement_true
                    elif_list:elifs ELSE stm_list:else_stms END
                  {: RESULT = new IfConstruct(new If(logic,
                      statement_true), elifs, new Else(else_stms)); :}
                | IF expr:logic THEN stm_list:statement_true ELSE
                    stm_list:else_stms END
                  {: RESULT = new IfConstruct(new If(logic,
                      statement_true), new ArrayList<Statement>(), new
                      Else(else_stms)); :}
                | IF expr:logic THEN stm_list:statement_true
                    elif_list:elifs END
                  {: RESULT = new IfConstruct(new If(logic,
                      statement_true), elifs, null); :}
                ;

elif_list ::= ELIF expr:logic THEN stm_list:stms
                {: ArrayList<Statement> lonely_elif = new
                  ArrayList<Statement>();
                       lonely_elif.add(new If(logic,stms));
                       RESULT = lonely_elif; :}
            | elif_list:elifs ELIF expr:logic THEN stm_list:stms
                {:
                elifs.add(new If(logic,stms));
                RESULT = elifs;
                :};

expr     ::= LPARAN expr:e RPARAN
                  {: RESULT = e; :}
         |    NUMBER:n
                  {: RESULT = new Number(n); :}
         |    BOOL:b
                  {: RESULT = new Bool(b); :}
         |    VAR:i
                  {: RESULT = new Variable(i); :}
         |    expr:e1 GRT expr:e2
                  {: RESULT = new GRT(e1,e2); :}
         |    expr:e1 LRT expr:e2
                  {: RESULT = new LRT(e1,e2); :}
         |    expr:e1 EQL expr:e2
                  {: RESULT = new EQL(e1,e2); :}
         |    expr:e1 NEQ expr:e2
                  {: RESULT = new NEQ(e1,e2); :}
         |    NOT expr:e
                  {: RESULT = new NOT(e); :}
         |    expr:e1 AND expr:e2
                  {: RESULT = new AND(e1,e2); :}
         |    expr:e1 OR expr:e2
                  {: RESULT = new OR(e1,e2); :}
         |    expr:e1 ADD expr:e2
                  {: RESULT = new ADD(e1,e2); :}
         |    expr:e1 SUB expr:e2
                  {: RESULT = new SUB(e1,e2); :}
         ;
```

A.5 Presets.java

```
package robotgame.newmaven.interpreter;

import java.util.Arrays;
import java.util.List;

public class Presets {
    public static List<String> keywords = Arrays.asList(
            "end","about","print","store","as","new","routine",
            "start","GRT","LRT","EQL","NEQ","NOT","AND","OR",
            "ADD","SUB","if","elif","else","then","think",
            "count","find","inc","dec"
    );
    public static List<String> presetVars = Arrays.asList(
    "LEFT","RIGHT","UP","DOWN","CURRENT",
    "CLEAN","MESSY","DIRTY","FILTHY","DISGUSTING",
    "WALL","OBSTACLE",
    "STEP",
    "CURRENT_LOC","START_LOC",
    "TRUE","FALSE"
  );
  public static List<Integer> presetVarsVal = Arrays.asList(
    4,2,1,3,0,
    0,1,2,3,4,
    -1,-2,
    0,
    0,0,
    1,0
  );
  public static List<String> presetRoutines = Arrays.asList(
    "move","clean","radar","random"
  );
}
```

A.6 RoutineStorage.java

```java
package robotgame.newmaven.interpreter;

import java.util.ArrayList;

public class RoutineStorage {
    public String name;
    public ArrayList<String> args;
    public ArrayList<Statement> stms;
    public RoutineStorage(String name, ArrayList<String> args,
        ArrayList<Statement> stms) {
        this.name = name;
        this.args = args;
        this.stms = stms;
    }
}
```

A.7 Statement.java

```java
package robotgame.newmaven.interpreter;
import java.util.ArrayList;
import java.util.Random;

public abstract class Statement {
    public abstract boolean execute(Interpreter interpreter);
}
class VarDeclaration extends Statement {
    String name;
    Expression expr;

    public VarDeclaration(String name, Expression expr) {
        this.name = name;
        this.expr = expr;
    }
    public boolean execute(Interpreter interpreter) {
        if(Presets.presetVars.contains(name)){
            System.err.println("Attempt to overwrite preset variable
                blocked!\nProblem with: "+name);
            return false;
        }if (Presets.keywords.contains(name)){
            System.err.println("Invalid name for variable, name is a
                protected keyword!\nProblem with: "+name);
        }
        interpreter.putVar(name, expr.evaluate(interpreter));
        return true;
    }
}

class RouDeclaration extends Statement {
    public String name;
    public ArrayList<String> args;
    public ArrayList<Statement> stms;
    public RouDeclaration(String name, ArrayList<String> args,
        ArrayList<Statement> stms) {
        this.name = name;
        this.args = args;
        this.stms = stms;
    }
    public boolean execute(Interpreter interpreter) {
        if (Presets.presetRoutines.contains(name)) {
            System.err.println("Attempt to overwrite preset routine
                blocked!\nProblem with: " + name);
            return false;
        }
        if (Presets.keywords.contains(name)){
            System.err.println("Invalid name for routine, name is a
                protected keyword!\nProblem with: "+name);
            return false;
        }
        interpreter.putRou(name, new RoutineStorage(name,args,stms));
        return true;
    }
}

class Routine extends Statement {
    String name;
    ArrayList<Expression> args;
    RoutineStorage routinePattern;
    public Routine(String name, ArrayList<Expression> args) {
        this.name = name;
        this.args = args;
    }
    public boolean execute(Interpreter interpreter) {
        if(Presets.presetRoutines.contains(name)){
            return new PresetRoutines(name, args).execute(interpreter);
        }
        routinePattern = interpreter.getRou(name);
        if(routinePattern == null){
            System.err.println("Tried to run a routine which doesnt
                exist! \nProblem with: "+name);
            return false;
        }
        if(routinePattern.args.size() != this.args.size()){
            System.err.println("Tried to run a routine with insufficient
                arguments!! \nProblem with: "+name+
            " \nTarget args: "+routinePattern.args.size()+" supplied:
                "+this.args.size());
            return false;
        }
        for(int i = 0; i<this.args.size();i++){
            new
                VarDeclaration(routinePattern.args.get(i),this.args.get(i)).execute(interpreter);
        }
        for(Statement statement : routinePattern.stms){
            if(!statement.execute(interpreter)){
                return false;
            }
        }
        for(String arg:routinePattern.args){
            interpreter.voidVar(arg);
        }
        return true;
    }
}
```

```
class PresetRoutines extends Statement {
    String name;
    ArrayList<Expression> args;
    public PresetRoutines(String name, ArrayList<Expression> args) {
        this.name = name;
        this.args = args;
    }
    public boolean execute(Interpreter interpreter){
        if(name.equals("move") && args.size() == 1){
            interpreter.robot.moveRobot(args.get(0).evaluate(interpreter));
            return false;
        }
        else if(name.equals("clean") && args.size() == 0){
            interpreter.robot.clean();
            return false;
        }
        else if(name.equals("radar") && args.size() == 1){
            int val =
                 interpreter.robot.sense(args.get(0).evaluate(interpreter));
            new VarDeclaration("radar", new
                 Number(val)).execute(interpreter);
        }
        else if(name.equals("random") && args.size() == 2){
            int val = new
                 Random().nextInt(args.get(0).evaluate(interpreter),args.get(1).evaluate(interpreter)+1
            new VarDeclaration("random", new
                 Number(val)).execute(interpreter);
        }
        return true;
    }
}

class MyPrint extends Statement {
    Expression expr;
    public MyPrint(Expression expr) {
        this.expr = expr;
    }
    public boolean execute(Interpreter interpreter) {
        System.out.println(expr.evaluate(interpreter));
        return true;
    }
}

class Else extends Statement {
    Expression logic;
    ArrayList<Statement> statement_false;
    public Else(ArrayList<Statement> statement_false) {
        this.statement_false = statement_false;
    }
    public boolean execute(Interpreter interpreter){
        for(Statement statement : statement_false){
            if(!statement.execute(interpreter)){
                return false;
            }
        }
        return true;
    }
}

class If extends Statement {
    public Expression logic;
    ArrayList<Statement> statement_true;
    public If(Expression logic, ArrayList<Statement> statement_true) {
        this.logic=logic;
        this.statement_true = statement_true;
    }
    public boolean execute(Interpreter interpreter){

        if(logic.evaluate(interpreter) >= 1){
            for(Statement statement : statement_true){
                if(!statement.execute(interpreter)){
                    return false;
                }
            }
        }
        return true;
    }
}

class IfConstruct extends Statement {
    Statement if_stm;
    ArrayList<Statement> elifs;
    Statement else_stm;
    IfConstruct(Statement if_stm, ArrayList<Statement> elifs, Statement
         else_stm){
        this.if_stm = if_stm;
        this.elifs = elifs;
        this.else_stm = else_stm;
        elifs.addFirst(if_stm);
    }
    public boolean execute(Interpreter interpreter){
        boolean cont = true;
        Integer previous_logic = -1;

        for(Statement stm : elifs){
            If made_if = (If) stm;
            if(previous_logic < 1){
                cont = made_if.execute(interpreter);
                previous_logic = made_if.logic.evaluate(interpreter);
                if (!cont) {
```

```
                return cont;
            }
        }
    }
    if(previous_logic < 1 && else_stm != null){
        cont = else_stm.execute(interpreter);
    }
    return cont;
  }
}

class Think extends Statement {
    Expression logic;
    ArrayList<Statement> statements;
    public Think(Expression logic, ArrayList<Statement> statements){
        this.logic = logic;
        this.statements = statements;
    }
    public boolean execute(Interpreter interpreter){
        Integer check = logic.evaluate(interpreter);
        while(check >= 1){
            for(Statement statement : statements){
                if(!statement.execute(interpreter)){
                    return false;
                }
            }
            check = logic.evaluate(interpreter);
        }
        return true;
    }
}

class ForLoopInc extends Statement {
    Expression logic;
    ArrayList<Statement> statements;
    String iterator_name;
    Expression value;
    public ForLoopInc (Expression logic, ArrayList<Statement>
         statements, String iterator_name, Expression value) {
        this.logic = logic;
        this.statements = statements;
        this.iterator_name = iterator_name;
        this.value = value;
    }
    public boolean execute(Interpreter interpreter){
        new VarDeclaration(iterator_name,value).execute(interpreter);

        Integer check = logic.evaluate(interpreter);
        while(check >= 1){
            for(Statement statement : statements){
                if (!statement.execute(interpreter)){
                    return false;
                }
            }
            value = new Number(value.evaluate(interpreter) + 1);
            new VarDeclaration(iterator_name,value).execute(interpreter);
            check = logic.evaluate(interpreter);
        }
        return true;
    }
}

class ForLoopDec extends Statement {
    Expression logic;
    ArrayList<Statement> statements;
    String iterator_name;
    Expression value;
    public ForLoopDec(Expression logic, ArrayList<Statement> statements,
         String iterator_name, Expression value) {
        this.logic = logic;
        this.statements = statements;
        this.iterator_name = iterator_name;
        this.value = value;
    }
    public boolean execute(Interpreter interpreter) {
        new VarDeclaration(iterator_name, value).execute(interpreter);

        Integer check = logic.evaluate(interpreter);
        while (check >= 1) {
            for (Statement statement : statements) {
                if (!statement.execute(interpreter)) {
                    return false;
                }
            }
            value = new Number(value.evaluate(interpreter) - 1);
            new VarDeclaration(iterator_name, value).execute(interpreter);
            check = logic.evaluate(interpreter);
        }
        return true;
    }
}
```

Appendix B

Tiles

B.1 Floor.java

```
package robotgame.newmaven.Tiles;

import robotgame.newmaven.Util.Tile;
public class Floor extends Tile {

    public Floor() {
        this.dirtiness=0;
        this.type = 1;
    }
    @Override
    public boolean isClear() {
        return true;
    }
    @Override
    public void cleanTile() {
        this.dirtiness = 0;
    }
    public void makeDirty(float noise){
        if(noise<0.20){ this.dirtiness=0;}
        else if(noise<0.40){ this.dirtiness=1;}
        else if(noise<0.60){ this.dirtiness=2;}
        else if(noise<0.80){ this.dirtiness=3;}
         else if(noise<1){ this.dirtiness=4;}
    }

}
```

B.2 Obstacle.java

```
package robotgame.newmaven.Tiles;

import robotgame.newmaven.Util.Tile;

public class Obstacle extends Tile {
    public Obstacle() {
        this.dirtiness = -1;
        this.type = -1;
    }
    @Override
    public void cleanTile() {
    }
    @Override
    public boolean isClear() {
        return false;
    }
}
```

B.3 Wall.java

```java
package robotgame.newmaven.Tiles;

import robotgame.newmaven.Util.Tile;

public class Wall extends Tile {
    public Wall() {
        this.dirtiness = -1;
        this.type = -2;
    }
    @Override
    public void cleanTile() {
    }
    @Override
    public boolean isClear() {
        return false;
    }
}
```

Appendix C

UIElems

C.1 JavaFXApp.java

```
package robotgame.newmaven;

import javafx.animation.KeyFrame;
import javafx.animation.Timeline;
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.input.KeyCode;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.HBox;
import javafx.scene.layout.StackPane;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;
import javafx.util.Duration;
import robotgame.newmaven.Util.RobotsGame;
public class JavaFXApp extends Application {
    private RobotsGame game;
    private boolean spedUp = false;
    public void init() {
        this.game = new RobotsGame(15, 15);
    }
    public void start(Stage primaryStage){
        primaryStage.setTitle("Robot game");
        GridPane fullViewPane = game.fullPlayScreen.robotPane;
        GridPane POVViewPane = game.povPlayScreen.robotPane;
        VBox statDisplay = game.statsDisplay.textBox;
        game.statsDisplay.toggleSpeedUp(spedUp);
        POVViewPane.setVisible(false);
        StackPane worldScene = new StackPane();
        worldScene.getChildren().addAll(fullViewPane,POVViewPane);
        HBox compile = new HBox(worldScene,statDisplay);
        compile.setAlignment(Pos.CENTER);
        compile.setSpacing(30);
        Scene scene = new
             Scene(compile,(fullViewPane.getWidth()+statDisplay.getWidth()),
             (fullViewPane.getHeight()));

        final Timeline timeline = new Timeline();
        timeline.setCycleCount(300);
        timeline.getKeyFrames().add(
                new KeyFrame(new Duration(500), new
                     EventHandler<ActionEvent>() {
                    @Override
                    public void handle(ActionEvent actionEvent) {
                        game.step();
                    }
                }
                )
        );
        timeline.play();
        scene.setOnKeyPressed(keyEvent -> {
            if(keyEvent.getCode() == KeyCode.T){
                fullViewPane.setVisible(!fullViewPane.isVisible());
               POVViewPane.setVisible(!POVViewPane.isVisible());
            }
            if(keyEvent.getCode() == KeyCode.S){
                if(spedUp){
                    timeline.setRate(1);
                }
                else {
                    timeline.setRate(10);
                }
                spedUp = !spedUp;
                game.statsDisplay.toggleSpeedUp(spedUp);
            }

        });
        primaryStage.setScene(scene);
        primaryStage.show();
    }
    public static void main(String[] args) {
        launch(args);
        }
}
```

C.2 GameStatsDisplay

```
package robotgame.newmaven.UIElems;

import javafx.scene.layout.VBox;
import javafx.scene.text.Text;
import robotgame.newmaven.Util.Robot;
import robotgame.newmaven.Util.World;

public class GameStatsDisplay {

    public VBox textBox;
    World world;
    Robot robot;
    Text worldDirtiness;
    Text robotPoints;
    Text worldPercentage;
    Text steps;

    Text spedUp;
    int worldDirtinessVal;
    public GameStatsDisplay(World world, Robot robot){
        setUp();
        this.world = world;
        this.robot = robot;
        worldDirtinessVal =world.getWorldDirtyness();
    }
    public void updateValues(){
        worldDirtiness.setText("\t"+world.getWorldDirtyness()+"/"+worldDirtinessVal);
        worldPercentage.setText("\t"+((int)((double)(world.getWorldDirtyness()*100)/(double)worldDirti
            %");
        robotPoints.setText("\t"+robot.getPointsCollected());
        steps.setText("\t"+robot.variables.get("STEP")+"/"+300);
    }
    void setUp(){
        worldDirtiness = new Text();
        robotPoints = new Text();
        worldPercentage = new Text();
        steps = new Text();
        spedUp = makeHeader("Game is sped up!",24);
        textBox = new VBox(makeHeader("Statistics",24),
                makeHeader("Room dirtiness:",18),
                worldDirtiness,
                makeHeader("Room percentage:",18),
                worldPercentage,
                makeHeader("Robot points:",18),
                robotPoints,
                makeHeader("Steps:",18),
                steps,
                spedUp,
                makeHeader("Press \"t\" to toggle in and out of the robot
                    POV",18),
                makeHeader("Press \"s\" to speed up the game, press \"s\"
                    to slow it down again.",18)
                );
    }
    public void toggleSpeedUp(boolean toggle){
        spedUp.setVisible(toggle);
    }
    Text makeHeader(String content, int fontSize){
        Text text = new Text(content);
        text.setStyle("-fx-font-weight: bold;-fx-font-size:
            "+fontSize+"pt;");
        return text;
    }
}
```

C.3 FullPlayScreen.java

```
package robotgame.newmaven.UIElems;

import javafx.scene.layout.GridPane;
import javafx.scene.paint.Paint;
import javafx.scene.shape.Circle;
import robotgame.newmaven.Util.Robot;
import robotgame.newmaven.Util.World;

public class FullPlayScreen {

    int robotPaneSize = 500;
    int marginFrac = 10;
    int width;
    int height;
    public GridPane robotPane = new GridPane();
    World robotWorld;
    UITile[][] map;

    Circle robotMarker;
    Robot robot;

    public FullPlayScreen(World robotWorld, Robot robot, int width, int
         height) {
        this.width = width;
        this.height = height;
        this.robotWorld = robotWorld;
        this.robot = robot;
        map = new UITile[width][height];
        robotMarker = new Circle();
        robotMarker.setRadius((double) ((robotPaneSize / height) *
            (marginFrac-1)/marginFrac)/2);
        robotMarker.setFill(Paint.valueOf("#370031"));

        initPane();
    }
    private void initPane(){
        robotPane.setHgap((double) (robotPaneSize / height) /marginFrac);
        robotPane.setVgap((double) (robotPaneSize / width) /marginFrac);

        double cellHeight = (double) (robotPaneSize / height) *
            (marginFrac-1)/marginFrac;
        double cellWidth = (double) (robotPaneSize / width) *
            (marginFrac-1)/marginFrac;

        for (int row = 0; row < height; row++){
            for (int col = 0; col < width; col++){
                UITile cell = new UITile(cellWidth,cellHeight);
                fillColor(cell,
                    robotWorld.getTile(col,row).getDirtinessLevel());
                map[row][col] = cell;
                robotPane.add(cell.tile,col,row);
            }
        }
        robotPane.add(robotMarker,robot.getX(),robot.getY());
    }
    public void updateRobot(){
        robotPane.getChildren().remove(robotMarker);
        robotPane.add(robotMarker,robot.getX(),robot.getY());
    }
    public void changeCell(int col, int row){
        UITile cell = map[row][col];
        fillColor(cell, robotWorld.getTile(col,row).getDirtinessLevel());
    }
    public void fillColor(UITile cell, int dirtiness){
        cell.changeColor(dirtiness);
    }
}
```

C.4 POVPlayScreen.java

```java
package robotgame.newmaven.UIElems;

import javafx.scene.layout.GridPane;
import javafx.scene.paint.Paint;
import javafx.scene.shape.Circle;
import robotgame.newmaven.Util.Robot;
import robotgame.newmaven.Util.World;

public class POVPlayScreen {

    int robotPaneSize = 500;
    int marginFrac = 10;
    public GridPane robotPane = new GridPane();
    World robotWorld;
    UITile[][] map;

    Circle robotMarker;
    Robot robot;

    public POVPlayScreen(World robotWorld, Robot robot) {
        this.robotWorld = robotWorld;
        this.robot = robot;
        map = new UITile[3][3];
        robotMarker = new Circle();
        robotMarker.setRadius((double) ((robotPaneSize / 3) *
            (marginFrac-1)/marginFrac)/2);
        robotMarker.setFill(Paint.valueOf("black"));

        initPane();
        updateView();
    }
    private void initPane(){
        robotPane.setHgap((double) (robotPaneSize / 3) /marginFrac);
        robotPane.setVgap((double) (robotPaneSize / 3) /marginFrac);

        double cellHeight = (double) (robotPaneSize / 3) *
            (marginFrac-1)/marginFrac;
        double cellWidth = (double) (robotPaneSize / 3) *
            (marginFrac-1)/marginFrac;

        for (int row = 0; row < 3; row++){
            for (int col = 0; col < 3; col++){
                UITile cell = new UITile(cellWidth,cellHeight);
                map[row][col] = cell;
                robotPane.add(cell.tile,col,row);
            }
        }
        robotPane.add(robotMarker,1,1);
    }
    public void updateView(){
        int robotX = robot.getX();
        int robotY = robot.getY();
        int[][] modifiers = {{-1,0},{1,0},{0,-1},{0,1},{0,0}};
        for(int[] modifier : modifiers){
            UITile cell = map[1+modifier[0]][1+modifier[1]];
            fillColor(cell,
                 robotWorld.getTile(robotX+modifier[1],robotY+modifier[0]).getDirtinessLevel());
        }
    }
    public void fillColor(UITile cell, int dirtiness){
        cell.changeColor(dirtiness);
    }

}
```

C.5 UITile.java

```java
package robotgame.newmaven.UIElems;

import javafx.scene.paint.Paint;
import javafx.scene.shape.Rectangle;

public class UITile {
    Rectangle tile;

    public UITile(double width, double height){
        tile = new Rectangle();
        tile.setWidth(width);
        tile.setHeight(height);
    }
    public void changeColor(int dirtiness){
        String color = switch (dirtiness) {
            case 0 -> "white";
            case 1 -> "#a9e5d9";
            case 2 -> "#8ad8dd";
            case 3 -> "#6ec2db";
            case 4 -> "#4394bf";
            default -> "black";
        };
        tile.setFill(Paint.valueOf(color));
    }
}
```

Appendix D

Util

D.1 Robot.java

```java
package robotgame.newmaven.Util;

import robotgame.newmaven.interpreter.RoutineStorage;

import java.io.File;
import java.util.HashMap;

public class Robot {
    String name;
    //UUID
    String id;
    World world;
    int x;
    int y;

    public HashMap<String,Integer> variables;
    public HashMap<String, RoutineStorage> routines;

    public Integer permaStorage;
    String script;
    int pointsCollected;

    public Robot(String name, String id, World world, int x, int y) {
        this.name = name;
        this.id = id;
        this.world = world;
        this.x = x;
        this.y = y;

        this.variables = new HashMap<>();
        this.routines = new HashMap<>();
        this.permaStorage = 0;

        try {
            this.script = "Scripts/Active/"+findFile();
        }
        catch (Exception e){
            System.err.println(e.getMessage());
            System.exit(1);
        }
        this.pointsCollected = 0;
    }
    private String findFile() throws Exception{
        String scriptDir = "Scripts/Active/";
        File dir = new File(scriptDir);
        File[] scripts = dir.listFiles();
        assert scripts != null;
        return scripts[0].getName();
    }
    public void moveRobot(Integer direction){
        switch (direction) {
            case 1:
                goUp();
                break;
            case 2:
                goRight();
                break;
            case 3:
                goDown();
                break;
            case 4:
                goLeft();
                break;
            default:
                break;
        }
    }
    public void goLeft() {
        move(-1,0);
    }

    public void goRight() {
        move(1,0);
    }

    public void goUp() {
        move(0,-1);
    }

    public void goDown() {
        move(0,1);
    }

    private void move(int xMod, int yMod){
        int newX = x + xMod;
        int newY = y + yMod;
        Tile tile = world.getTile(newX,newY);
        if (tile.isClear()) {
            this.x += xMod;
            this.y += yMod;
        }
    }
    public int sense(Integer direction) {
        int newX = x, newY = y;
        switch (direction) {
            case 1: newY--; break;
            case 2: newX++; break;
            case 3: newY++; break;
            case 4: newX--; break;
        }
        Tile tile = world.getTile(newX, newY);
        int tile_info = tile.getDirtinessLevel(); //from 0 to 4
        if (!tile.isClear()) {
            tile_info = tile.getType(); // Wall -2 or Obstacle -1
        }
        return tile_info;
    }
    public void clean() {
        Tile tile = world.getTile(x, y);
        if (tile != null && tile.isClear()) { // Check if it's a clear
             tile
            //While it is nice to have fail-safes, the robot cannot ever
                 be on a not_clear or a null tile
            addPoints(tile);
            tile.cleanTile();
        }
    }
    void addPoints(Tile tile){
        pointsCollected += tile.getDirtinessLevel();
    }
    public int getX() { return x; }
    public int getY() { return y; }
    public int getPointsCollected(){
        return pointsCollected;
    }
    public String getScript() {
        return script;
    }
}
```

D.2 RobotsGame.java

```
package robotgame.newmaven.Util;

import robotgame.newmaven.UIElems.FullPlayScreen;
import robotgame.newmaven.UIElems.GameStatsDisplay;
import robotgame.newmaven.UIElems.POVPlayScreen;
import robotgame.newmaven.interpreter.Interpreter;

public class RobotsGame {
    private World world;
    private Robot robot;
    public FullPlayScreen fullPlayScreen;
    public POVPlayScreen povPlayScreen;
    public GameStatsDisplay statsDisplay;

    private Interpreter i;

    public RobotsGame(int width, int height) {
        this.world = new World(width, height);
        this.robot = new Robot("Robot1", "1", world, 0, 0);
        this.i = new Interpreter(robot);

        this.fullPlayScreen = new
            FullPlayScreen(world,robot,width,height);
        this.povPlayScreen = new POVPlayScreen(world,robot);
        this.statsDisplay = new GameStatsDisplay(world,robot);
    }
    public void step(){
        int robotPrevX = robot.getX();
        int robotPrevY = robot.getY();
        i.interpret();
        fullPlayScreen.changeCell(robotPrevX,robotPrevY);
        fullPlayScreen.updateRobot();
        povPlayScreen.updateView();
        statsDisplay.updateValues();
    }
}
```

D.3 Tile.java

```java
package robotgame.newmaven.Util;

public abstract class Tile {
    public int dirtiness;
    // Determines how dirty the tile is.
    public int type;
    //Tells what type of tile it is to player
    public abstract boolean isClear();

    public int getType(){
        return type;
    };

    // Returns the dirtiness level or indicates an obstacle/wall.
    public int getDirtinessLevel(){
        return dirtiness;
    };

    public abstract void cleanTile();
}
```

D.4 World.java

```
package robotgame.newmaven.Util;

import robotgame.newmaven.Tiles.Floor;
import robotgame.newmaven.Tiles.Obstacle;
import robotgame.newmaven.Tiles.Wall;

import java.util.Random;

import static robotgame.newmaven.Util.PerlinNoise.generatePerlinNoise;

public class World {
    private Tile[][] tiles;
    private int width;
    private int height;
    float[][] noise;
    private Random random = new Random();

    public World(int width, int height) {
        this.width = width;
        this.height = height;
        noise = generatePerlinNoise(width,height,2,0.01f, new
            Random().nextInt(1100));
        this.tiles = new Tile[width][height];
        initializeTiles();
    }
    // Initializes the floor and obstacles and merges them into one tile
        map
    // Separation of floor and obstacle generation makes it easier to
        write algorithms for thier individual generation
    private void initializeTiles() {
        Tile[][] floorMap = genFloor();
        int[][] obsMap = genObs();
        for (int i = 0; i < width; i++) {
            for (int j = 0; j < height; j++) {
                if(obsMap[i][j] == 1){
                    tiles[i][j] = new Obstacle();
                }
                else {
                    tiles[i][j] = floorMap[i][j];
                }
            }
        }
    }
    //  Method to generate all floor tiles
    //  Generates randomly, to generate non-randomly, changes needs to
        made in the floor class.
    private Tile[][] genFloor(){
        Floor[][] floorMap = new Floor[width][height];
        for (int i = 0; i < width; i++) {
            for (int j = 0; j < height; j++) {
                floorMap[i][j] = new Floor();
                floorMap[i][j].makeDirty(noise[i][j]);
            }
        }
        return floorMap;
    }
    // Generate all obstacles, uses ints because they are quick to
         generate
    // Since it is only here we can experience maps that are invalid, we
         only need to check and regenerate maps here
    private int[][] genObs(){
        int[][] obsMap = new int[width][height];
        // Find vertical and horizontal center of play field
        int vCenter = height/2;
        int hCenter = width/2;
        // Divide the map into 4 areas
        int[][] boxes = {
                {0,vCenter,0,hCenter},
                {vCenter,height,0,hCenter},
                {vCenter,height,hCenter,width},
                {0,vCenter,hCenter,width}
        };
        // Generate 1 obstacle in each area
        // Obstacles are generated like a snake moving randomly in any
             of the 4 cardinal directions 3 to 10 times
        for(int[] box : boxes){
            int xWalk = random.nextInt(box[0],box[1]+1);
            int yWalk = random.nextInt(box[2],box[3]+1);
            int walkAmount = random.nextInt(3,11);
            while (walkAmount > 0){
                if(xWalk >= 0 && xWalk < height && yWalk >= 0 && yWalk <
                     width){
                    obsMap[xWalk][yWalk] = 1;
                    if(random.nextInt(1,3) == 1){
                        xWalk += random.nextInt(-1,2);
                        continue;
                    }
                    yWalk += random.nextInt(-1,2);
                }
                else {
                    xWalk = random.nextInt(box[0],box[1]+1);
                    yWalk = random.nextInt(box[2],box[3]+1);
                }
                walkAmount--;
            }
        }
        if(isValid(obsMap,new boolean[height][width],0,0)){
```

```
        return obsMap;
    }
    return genObs();
}

public boolean isValid(int[][] obsMap, boolean[][] visitMap, int
     currentX, int currentY){
    if (obsMap[0][0] >0){
        return false;
    }
    boolean[][] visitedMap =
         mapVisitor(obsMap,visitMap,currentX,currentY);

    int floorCount = 0;
    for(boolean[] row : visitedMap){
        for(boolean tile : row){
            if (tile){
                floorCount++;
            }
        }
    }

    int target = countFloor(obsMap);
    return floorCount == target;
}

public boolean[][] mapVisitor(int[][] obsMap, boolean[][] visitMap,
     int currentX, int currentY){
    visitMap[currentX][currentY] = true;

    int[][] children = {
            {currentX-1,currentY},
            {currentX+1,currentY},
            {currentX,currentY-1},
            {currentX,currentY+1},
    };
    for(int[] child : children){
        int x = child[0];
        int y = child[1];
        if(isWithinBounds(x,y)){
            if(obsMap[x][y] == 0 && !visitMap[x][y]){
                isValid(obsMap,visitMap,x,y);
            }
        }
    }
    return visitMap;
}

int countFloor(int[][] obsMap) {
    int area = height * width ;
```

```
        int obsCount = 0;
        for (int[] row : obsMap) {
            for(int num : row) {

                obsCount += num;
            }
        }
        return area - obsCount;
    }
    // Added method to access a tile at (x, y)
    public Tile getTile(int x, int y) {
        if (isWithinBounds(x, y)) {
            return tiles[x][y];
        }
        return new Wall(); // Return a wall if the requested tile is out
             of bounds
    }
    public int getWorldDirtyness(){
        int worldDirtyness = 0;
        for(Tile[] tileRow : tiles){
            for(Tile tile : tileRow){
                worldDirtyness += tile.getDirtinessLevel();
            }
        }
        return worldDirtyness;
    }
    // Checks if a given position is within the world bounds
    //This method should return a tile type of wall and should be called
         by the robots sensor
    public boolean isWithinBounds(int x, int y) {
        return x >= 0 && y >= 0 && x < width && y < height;
    }
}
```