DatLanguage

P4 Project
Group SW408F16
Software
Aalborg University
26th May 2016

AALBORG UNIVERSITY
STUDENT REPORT

Fourth semester at
Department of Computer Science
Software
Selma Lagerlöfs Vej 300
9220 Aalborg East, DK
http://www.cs.aau.dk/

Title:

DatLanguage

Project:

P4-project

Project period:

February 2016 - May 2016

Project group:

SW408F16

Participants:

Christian Dannesboe
Frederik Børsting Lund
Karrar Al-Sami
Mark Kloch Haurum
Emil Bønnerup
Søren Lyng

Supervisor:

Giovanni Bacci

Number of prints: Unknown
Pagecount: 62
Appendix range: 6
Published 26-05-2016

Synopsis:

This project approaches the fourth semester project regarding creating a domain specific language for programming Robocode, that is aimed at high school students, without knowledge of programming, in order to introduce programming to students, at an early stage.

The group approached the project by creating a domain specific language that is somehow similar to the programming language Java, but with some meaningful changes.

This domain specific language compiles the robots that has been created to a java file, which then compiles it yet another time, this time into the executable Robocode file.

Preface

This project has been developed as part of the fourth semester project by project group SW408F16 from Aalborg University, Software Engineering, from the period 1st February to 26th May 2016 .
The project is based on the *Aalborg-model*, study method, where problem and project based learning is the focus. The theme of this semester was to create a compiler for a new language. To do so, some subjects were introduced and the subject the group chosen, was *Domain Specific Language for Robocoders* .

The group would like to thank supervisor, Giovanni Bacci for his very much appreciated advice and guidance during the whole project.

Signatures

Christian Dannesboe	Frederik Børsting Lund	Karrar Al-Sami
Mark Kloch Haurum	Emil Bønnerup	Søren Lyng

Reading guide

This project has followed the courses Syntax and Semantics & Languages and Compiler. The context of this project has been written according to the order the course materials was taught and learned.

The sources in the report are being referred to by the Harvard citation method. This includes a last name and a publication year in the report, and in the ***Bibliography*** chapter all the used sources are listed in alphabetical order.
An example of a source in the text could be: ***[Sebesta, 2009]****.*
If the source is on the left side of a dot, then that source refers only to that sentence and if the source is on the right side of a dot, then it refers to the whole section.

Figures and tables are referred to as a number. The number is determined by the chapter and the number of figure it appears as.
For example: The first figure in a chapter will have the number ***x.1****, where x is the number of the chapter. The next figure, will have the number* ***x.2****, etc.*
The listings of source code are also referred to as the tables and figures.

Source code in the report are listed as code snippets, and they're not necessarily the same as the source code, meaning that code snippets may be shorter than the actual source code or missing comments from the source code. In order to show that, the use of the following three dots are used: "...", which means that some of the source code isn't listed in the code snippet, as it may be long and irrelevant.

Contents

Introduction 1

In Danish high schools all kinds of different languages are available for the students, and the programming languages has found their place as well. For beginners, the syntactic rules, type systems and the nature of the language can be hard to comprehend at first, which is why this report will focus on constructing a language for the high school students. The interest for computer games in the 21st century is bigger than ever [Wankel og Blessinger, 2012], and computer games is a fun and educating approach to learning programming languages. Robocode is a game where the player has to code their own robot, giving every player the opportunity to battle each other's robots, making it a competition of coding the 'best' robot. This is mainly coded in Java, which is an object oriented programming language, and this nature of the language can be hard to understand without having programming experience. There is no popular domain-specific language for Robocode, and therefore the high school students, who is not experienced programmers, may not be likely to code a working robot. This is due to the fact that it requires one to know an object oriented language in advance.

This is a problem, since Robocode could potentially be a great way of introducing these students for programming languages. If there was a domain-specific language, with more intuitive type systems and good writability, students could easily be introduced for a programming language, and afterwards expand their knowledge gradually on a general purpose programming language. In this report, Robocode will be studied, and the final product should be a domain-specific language for Robocode, compiled to Java.

Based on the above mentioned introduction, the project will try to answer the following problem statement:

How can a domain-specific language for Robocode simplify the creation of a robot for high school students, with little or no experience to programming?

- How can the Java type system be simplified?
- Which constructs are necessary for programming standard robots in Robocode?
- How can an easy to use interface be made for the users?

Analysis 2

The analysis chapter's purpose is to create a basis for further work with developing a programming language to make the use of Robocode easier for new programmers. This chapter contains a description of Robocode and cover the basics of how to use it.

Further in the analysis the choice of a parser generator for the project will be discussed. The work of the analysis has the purpose of leading to the next chapter, 3.

2.1 Robocode

Robocode is an open source game project on SourceForge originally started by Mathew A. Nelson in late 2000, who was inspired by RobotBattle from the 1990s. Contributions for the open source lead to two new projects, RobocodeNG and Robocode 2006, by Flemming N. Larsen. These two new versions had bug fixes, and new features by the community of Robocode, and in 2006 Flemming merged one of the projects, the Robocode 2006, into an official version 1.1. The Robocode client was introduced in May 2007, which can be used to create the robots for the game. These robots are usually coded in Java, but in the recent years, C# and Scala are popular as well. [RoboWiki, 2013]

In schools and universities, Robocode is introduced for education and research purposes, as it is intended to be fun and easy to understand the core principles: One robot each, with abilities to drive forward, backwards, turn to the sides, and shoot a gun. These core principles can be vastly expanded to more complicated demands, as the robots universe is bigger than it looks at a first glance. [Larsen, 2013]

The way this game works, is by writing code in one of their supported programming languages and then setting it into battle with other people's robots. There are some sample robots, when the game is downloaded, in order to give the users a chance to see how it's supposed to be written and from there, it's up to the single individual to make the best robot. [RoboWiki, 2014]

There are held tournaments around the world, where people from around the globe compete. It varies in size, some tournaments are only country based, while others are worldwide, some have leagues and the options are more or less limitless. [Nelson, 2015]

As mentioned before, the Robocode is usually coded in Java, which leads to this report only examining Java samples. This is to prevent any misleading keywords or misinterpretations. The Robocode client comes with a text editor, and the sample robots. In this chapter the general setup and main events or methods of Robocode will be presented.

When creating a new robot in the Robocode text editor, the following methods and events

are present:

Listing 2.1. Example of the main loop in Robocode label

```
1 public void run() {
2         while(true) {
3                 //Robots behaviour
4         }
5 }
```

This method is the loop for the robot, this loop will determine what the robot does constantly, unless interrupted by an event, which the user can define. The robot behaviour is what the user will code as the AI, along with the robot behaviour in the following code snippets.

Listing 2.2. Example of the onScannedRobot event from Robocode label

```
1 public void onScannedRobot(ScannedRobotEvent e) {
2         //Robots behaviour
3 }
```

The robot's radar will spot enemies when they get within the range of the radar, which will raise the onScannedRobot event. This event is used to determine how the robot reacts to spotting an enemy, where the ScannedRobotEvent e is the source for information about the enemy robot spotted.

Listing 2.3. Example of the onHitByBullet event from Robocode label

```
1 public void onHitByBullet(HitByBulletEvent e) {
2         //Robots behaviour
3 }
```

When the robot gets hit by another robot's bullet, the event onHitByBullet will be raised. The robot can then be programmed to act a specific way, change behaviour or carry out a task when the event is raised.

Listing 2.4. Example of the onHitWall event from Robocode label

```
1 public void onHitWall(HitWallEvent e) {
2         //Robots behaviour
3 }
```

When the robot drives into the wall, the event onHitWall will be raised, and the robot can be programmed to do a specific task when this occurs.

The above mentioned is only a few of the events that can occur in Robocode. In the while loop, events and functions the user can use many build-in methods from Robocode, which is moving and controlling the robot, controlling the radar and the gun and getting information about the battlefield, the user's robot, other robots and many other things.

In this section the Robocode concept will be described. It will include the different functions Robocode contains.

2.1.1 Robot

The robot is the core of the game. The robot can be coded to act differently in various encounters. There are some events in Robocode that the users can use to program their

strategies. One of the events could be onHitWall which basically tells the user that if the robot hits the wall, then the robot will execute the code defined by the event that occurred.

The abstraction "robot" will be the entirety of the robot, including the tank, gun and radar. The tank is the body of the robot.

The gun is used to damage antagonist robots, in the battlefield. The robot has the possibility to choose differently powered bullets, which is defined by the use of energy. The energy will be explained in further detail later on. For example, one user could use low energy cost fast bullets, they won't necessarily hurt very much, but it allows the user to shoot more frequent, compared to high energy cost slow bullets.

The radar allows the robot to scan for other robots and walls. This could once again affect the strategies of the robots.

2.1.2 Battlefield

The battlefield is the arena where the robots will battle each other. It's also the visible field on the screen when the game is running. When coding, the battlefield can be used for different purposes, for example, to get the number of enemies that are alive through the method call *getOthers()*. A robot might be programmed to act differently accordingly to the number enemies left. All this depends on the way the user decides to program the robot. Some of the other examples that the battlefield can be queried or could be the field size, time or the current round number.

2.1.3 Energy

Energy is a spendable resource when shooting bullets, it is the "health" of a robot, and being hit by a wall or another robots bullet causes a robot to lose energy. If a robot hits an enemy robot with a bullet, it regains energy. All robots have 100 energy at the start of a battle, but can exceed this amount, by hitting other robots to regain energy, without losing it. The amount of energy gained when hitting a robot is (3 * bulletpower), which is three times the power you spend shooting it. Being hit by a bullet, the robot lose (4 * bulletpower), and hitting a wall with an AdvancedRobot extended robot will cause the robot to lose energy as well.

If a robot shoots a bullet which uses the last energy that particular robot has, it will be disabled. A disabled robot will not be able to move or shoot. The last shot that robot took, has a chance to restore the robot, if it hits an enemy and thereby regaining energy.

Energy for the robots are both the health and the spendable resource for attacking, which makes every decision of manoeuvring and shooting count.

2.1.4 Scoring

Winning in Robocode is not about being the sole survivor, not even in the RoboRumble gamemode, which is the "every man for himself" type of gamemode. It is all about scoring, and there are different methods for scoring points. The various types of scoring are as follows:

- Survival score, every time a robot dies, all remaining robots get 50 points.
- Last survivor bonus, the last robot alive scores 10 points for every other robot that died before it.
- Bullet damage, robots scores 1 point for every point of damage that robot deals to other robots.
- Bullet damage bonus, if a robot kills another robot with a shot, it will gain 20% of all the damage it did to that robot as points.
- Ram damage, any robot that rams another robot gains 2 points for each damage they cause through ramming.
- Ram damage bonus, every time a robot kills another robot by ramming, it scores an additional 30% of all the damage it did to that robot as points.

When all the above scoring points for all robots in a battle has been added up, the robot with the most points wins the game.

2.2 Choice of parser generator

When choosing a parser generator, one also has to choose a lexer generator, for the lexical analysis. The choice of not building a parser for this project without a generator tool, was due to the fact that the ANTLR4 tool has a build-in lexical analyser, and a plugin for Eclipse/IntelliJ, generating abstract syntax trees while writing the program with a corresponding grammar. Other parser generators were discussed before making a final decision, such as CUP, but the lack of lexical analysers and abstract syntax tree builders, and at the same time the ease of installing the ANTLR4 parser generator, stated that the ANTLR4 tool was the choice of generator for this project. For the IntelliJ IDE there is a single plugin the user has to install, but for Eclipse and the abstract syntax tree builder for Eclipse, it required a few plugins and a bit of experience with the tool, to fully understand how to operate with the tree builder window. Both Eclipse and IntelliJ have been considered as the IDE to use. IntelliJ is preferred because of the ease of installation and use of the ANTLR4 plugin.

2.3 ANTLR4 parser generator

In this project, the ANTLR4 parser generator was chosen, as the tool generated both the parser and the lexical analyser. This tool, as a plugin for both IntelliJ and Eclipse, could also build abstract syntax trees (AST), which are the trees representing abstract syntactic structures, syntactically correct sentences (source code) in a computer language. These trees have a root representing the program, and nodes representing terminals and non-terminals. The leaves of the tree are terminals, which is the syntactically correct sentence.

The AST tree builder for the ANTLR4 Eclipse plugin would show the trees in an Eclipse window. The user would be able to write a sentence, and the window would show the AST for that particular sentence, if it was syntactically correct corresponding to the grammar described in a .g4 file in the Eclipse project. ANTLR4 parses through an LL(*) algorithm, which means it can process any LL(x) grammar, where 'x' is the amount of lookahead needed for parsing, and the LL means it parses from left to right, with leftmost

derivation. This makes it a top-down parser. The grammar input to this tool should be a CFG (Context-Free Grammar) in EBNF (Extended Backus-Naur form), which is a formal description of a formal language, including programming languages. This parser generator can parse to four different languages, where the language of interest for this project would be Java. This tool can also generate a C# output parser, but as this project narrows Robocode to only be written in Java, this wasn't in consideration for choosing the parser generator. Robocode can, as earlier stated, also be written in C#, which would enforce the choice made, if this project also included the C# source code for Robocode.

The ANTLR4 tool for Eclipse required a few other plugins to make the AST window work correctly, and it is a little bugged. When the user defines a grammar, the user then generates an ANTLR4 recognizer, if the .g4 file is then saved, the user then has to edit the document to make it a not-saved file to operate in the AST window. If the user has saved the document, without editing afterwards, the window would be unresponsive.

In IntelliJ the ANTLR4 plugin requires very little work to get started. To generate the ANTLR4 files the user has to right click and select the generate options of the plugin and the IDE does all the work. Similarly to generate the AST all one has to do is right click and choose to test the grammar.

Language design 3

In this chapter the design decisions made during the process of creating DatLanguage will be described. There are three criteria for the development of the language *readability, writability* and *reliability*. The decisions made to accommodate these will be described in detail in the first section of the chapter. In the second part of this chapter, a list of items for the MoSCoW analysis are prioritized. The MoSCoW analysis concerns what the project *must have*, *should have*, *could have* and *won't have*. Last in the chapter each of the items in the MoSCoW analysis will be shortly described why it have been considered and why it is placed where it is.

3.1 Language criteria

In this section the three main criteria for designing the language will be discussed with focus on the implementation of these in the language. The criteria are based on theory from the book *Concepts of Programming Languages* [Sebesta, 2009].

3.1.1 Readability

Readability refers to the ease of reading and understanding a programming language. The language in this report should be very simple, since the programming language is, as earlier mentioned, targeted for high school students with little or no programming experience. Beginners would not necessarily know the concepts of object oriented programming languages, but this would be needed to code robots in Java. This would be concealed from the user with DatLanguage, and simplifications to the type system would give the user a lesser flexibility, but it would highly increase the readability of the language. Readability will be the main priority for this report, as readability is key to understand and learn a new language.

Readability is directly affected by orthogonality, and with DatLanguage being constructed for beginners, only a few primitive types and few constructs will be implemented.

3.1.2 Writability

Writability is a contradiction to readability. If a language has advanced constructs that will help a programmer code big systems in little time, it will probably not be as easy for a beginner to understand. Take structs from C, if a non-experienced high school student had to implement a struct, that person wouldn't know what to do, or how it works. This is why writability will not be considered greatly in this report.

3.1.3 Reliability

To make the programming language reliable to use, a lot of focus has been put into making DatLanguage as readable as possible, and design the language in a way that helps the user create code without errors. By making programs written in DatLanguage less prone to errors, the more reliable it will also be.

Another thing being implemented that will have an effect on the reliability of the programming language, is type checking. This is to prevent the user from assigning values to a variable that is not of the same type. If there is no type checking, then bugs and errors can be hard to track down and fix, so it is better to be absolutely sure that all variables contains information of a certain type, rather than unexpected behaviour and output. DatLanguage has no pointers, so aliasing, having two or more variables pointing to the same memory cell, will not be a problem that could have a negative effect on reliability. It can't be ensured that there is no aliasing in our language, since even without pointers one variable cannot reference an object, which is aliasing.

By taking these precautions, the programming language should be reliable enough for a beginner to use, without feeling frustrated and losing interest because of language difficulties, but reliability will not be the of biggest concern in DatLanguage.

3.2 MoSCoW analysis

The *MoSCoW* method is used in this study with the purpose of specifying the importance of different requirements in the language. It is a good method for prioritizing work with the language. The criteria that has been designed can be seen in this section.

MoSCoW analysis	
Must have	Primitive types and variables While loop Reserved calls Robot naming If/Else/Elseif statements Arithmetic expressions and operators Logical expressions and operators
Should have	Events Comments Void and type methods
Could have	Cos, Sin & Tan For loops Print statements Strings Setup block
Won't have	Random number generator Other robot types

Table 3.1. Outcome of the MoSCoW analysis

3.2.1 Must have

Primitive types and variables are needed for any calculation and comparisons in the language.

While loop is needed to make the robots logic, since the Repeat block is a while loop. The Repeat block will be described in the following chapter.

Reserved calls are methods used to make the robot's behaviour, this is needed to make the robot shoot, scan for other robots and move around the battlefield.

Robot naming is necessary because some of the methods in Robocode returns the robot's name.

If/Else/Elseif statements is a very essential part of Robocode since it is a conditional statement, and since decision making is a very big role in Robocode.

Arithmetic expressions and operators are used to do calculations on primitive types and variables.

Logical expressions and operators are needed in the conditional statements.

3.2.2 Should have

Events are a big part of Robocode, it gives the robot the option to act on the environment and actions around it. Events are placed in should have because it is not necessary to code a moving robot, but to make the robot more intelligent.

Comments would help on the readability of the program, if the user was able to put comments in the code.

Void and type methods are another way to inject intelligence to the robot. Instead of using the Robocode events, the user can make their own functions with or without return types.

3.2.3 Could have

Cos, Sin & Tan could be useful when programming robots in Robocode, the user is often working with degrees in the reserved calls, so if the user wants to make their own functions, this could become necessary.

For loops would be another way for the user to iterate over data, but the for loop doesn't seem as necessary as the while loop for this.

Print statements would be a smart statement for debugging purposes.

Strings would be necessary if the print statement would be implemented.

Setup block would be necessary if there are any code that only has to be run once every round.

3.2.4 Won't have

Random number generator A random number generator would be a great tool to implement to the language. If some user would like to make their robots movements more unpredictable, a random number generator would be very useful.

Other robot types Overall there are six different robot types in Robocode which are: Robot, AdvancedRobot, JuniorRobot, TeamRobot, Droid and BorderSentry. This project will focus on the Robot type. JuniorRobot is very similar, where some of the other robots are allowing the user to make more advanced robots.

Language description 4

In this chapter, features of DatLanguage will be described by means of a simple example, and formal descriptions of its syntax in form of a grammar for the language, language constructs and type systems.

4.1 Syntax walkthrough

In this section an example of a sample robot from Robocode has been introduced and can be found in listing 4.1.

Listing 4.1. Example of the sample robot "Corners" in our language

```
1  Tankname corners;
2  Num others;
3  Num corner = 0;
4  Bool stopWhenSeeRobot = False;
5
6  Setup{
7    others = Battlefield.enemies();
8    run goCorner();
9    Num gunIncrement = 3;
10 }
11
12 Repeat{
13   Num i = 0;
14
15   repeat while(i < 30){
16     Gun.turn(gunIncrement * -1);
17     i = i+1;
18   }
19   gunIncrement = gunIncrement * -1;
20 }
21
22 Action goCorner(){
23   stopWhenSeeRobot = False;
24   Tank.turn(Utils.normalRelativeAngleDegrees(corner - Tank.heading()));
25   stopWhenSeeRobot = True;
26   Tank.forward(5000);
27   Tank.turn(-90);
28   Tank.forward(5000);
29   Gun.turn(-90);
30 }
31
32 When scannedRobot{
33   if(stopWhenSeeRobot){
34     Tank.stop();
35     run smartFire(Event.distance());
36     Radar.scan();
37     Tank.resume();
38   }else{
39     run smartFire(Event.distance());
```

```
40    }
41  }
42
43  Action smartFire(Num robotDistance){
44    if(robotDistance > 200 OR Tank.energy() < 15){
45      Gun.fire(1);
46    } else if (robotDistance > 50) {
47      Gun.fire(2);
48    } else {
49      Gun.fire(3);
50    }
51  }
52
53  When death{
54    if(others IS= 0){
55      return;
56    }
57
58    if((others - Battlefield.enemies()) / others < 0.75){
59      corner = corner + 90;
60      if(corner IS= 270) {
61        corner = -90;
62      }
63      print("I died and did poorly... switching corner to " + corner);
64    } else {
65      print("I died but did well. I will still use corner " + corner);
66    }
67  }
```

Robocodes sample robot, Corners [Nelson, 2008], will find a specific corner where the robot will stay for the entire round, and shoot at other robots whenever scanned. Initially *Corners* goes to the top left corner. When *Corners* dies, it will check whether it performed well or poorly, and will either use the same corner or move clockwise to the next corner on the battlefield.

In line 1-5 the Tankname is set and some global variables are declared and initialized. The Setup block in listing 4.2 is a block which is run once at the beginning of each round. In this example the Setup block is used to store in the global variable "other" the initial number of enemies. This value is obtained by calling the build-in function Battlefield.enemies(). It will run the action goCorner, which will be explained later. As the last thing, the Setup block declares a variable, gunIncrement, of type num and initializes its value to 3.

Listing 4.2. Code listing of the Setup block

```
1  Setup{
2    others = Battlefield.enemies();
3    run goCorner();
4    Num gunIncrement = 3;
5  }
```

To make the robot do repetitive actions during the battles, Tank, Gun and Radar behaviour can be placed in the Repeat block. The Repeat block is basically a loop, that will iterate through the robot's Tank, Gun and Radar behaviour. The Repeat block of the sample robot *Corners*, in listing 4.3, consists of a repeat while loop. At the end of the repeat while loop, the gun has been turned a total of 90 or -90 degrees.

Listing 4.3. Code listing of the Repeat block

```
1  Repeat{
2    Num i = 0;
```

```
3
4     repeat while(i < 30){
5       Gun.turn(gunIncrement * -1);
6       i = i+1;
7     }
8     gunIncrement = gunIncrement * -1;
9   }
```

In DatLanguage an Action is compared to a procedure in the language C. *Corners* uses an Action goCorner(), to move to the desired corner. In the first round of the battle, it will turn the robot the amount of degrees so it is facing the top wall, which is done at line 3 in listing 4.4. The degrees are calculated by subtracting the corner variable from the robot's heading. "Heading" is getting the robot's heading, which is the direction the tank is facing. It will then move forward until it hits the wall, turn -90 degrees, again move forward until it hits the wall and turn the gun -90 degrees. At the end of the execution of this action, the robot should be sitting in a corner, ready to turn its gun 90 degrees clockwise or counter-clockwise.

Listing 4.4. Code listing of the Action goCorner()

```
1  Action goCorner(){
2    stopWhenSeeRobot = False;
3    Tank.turn(Utils.normalRelativeAngleDegrees(corner - Tank.heading()));
4    stopWhenSeeRobot = True;
5    Tank.forward(5000);
6    Tank.turn(-90);
7    Tank.forward(5000);
8    Gun.turn(-90);
9  }
```

One of the most peculiar things about Robocode is the use of events. The event handlers are indicated by the reserved word "When" in DatLanguage. In the Action goCorner(), a variable of type Bool is set to false on line 2 and set to true in line 4 in listing 4.4. This variable is used in the event scannedRobot() found in listing 4.5. If the before mentioned Bool is true, a build-in function Tank.stop() is used to stop the robots movement, then it uses the action "smartfire", where the power of the shot will be determined by the distance between the robot and the scannedRobot. The robot will then use Radar.scan() and resume its movement towards the corner. If the Bool was false, the robot is already in the corner, and will shoot at the scanned robots.

If an event listener is listening to the event, it can detect when the event is triggered and make sure the event handler is executed. When the behaviour in the event has been run, the robot will continue to run the Repeat block.

Listing 4.5. Code listing of the event scannedRobot() label

```
1  When scannedRobot{
2    if(stopWhenSeeRobot){
3      Tank.stop();
4      run smartFire(Event.distance());
5      Radar.scan();
6      Tank.resume();
7    }else{
8      run smartFire(Event.distance());
9    }
10 }
```

4.2 Grammar

In this section the syntax of our language will be formally presented by means of a context-free grammar in EBNF.

```
grammar Grammar;

prog : dcls EOF;
tankname : 'Tankname' ID SEMI;
setupblock : 'Setup' block;
repeatblock : 'Repeat' block;

dcls : (actdcl
     | funcdcl
     | vardcl
     | setupblock
     | repeatblock
     | tankname
     | eventdcl
     | print)*
     ;

actdcl : 'Action' ID '(' params? ')'block;
funcdcl : 'Function' ID '(' params? ')' 'returns' TYPE functionBlock;
functionBlock : '{' stmts returnstmt SEMI '}';
params : param (',' param)*;
param : TYPE ID;
eventdcl : 'When' ID block;
block : '{' stmts '}';

stmts : (assign
      |vardcl
      |ifstmt
      |whilestmt
      |stmtcall
      |print)*
      ;

assign : ID '=' expr SEMI;
vardcl : TYPE (ID SEMI|assign);
ifstmt : 'if''('expr')' block elseif* ('else' block)?;
elseif : 'else''if''('expr')' block;
whilestmt : 'repeat' ('while''('expr')' block | block 'while''('expr')');
returnstmt : 'return' expr;
print : 'print('expr')' SEMI;
stmtcall : call SEMI;

call : acall
     | fcall
     | rcall
     | ecall
     ;

acall : 'run' ID'('args?')';
fcall : ID'('args?')';
rcall  : 'Tank.'ID'('args?')'       #tankcall
       | 'Gun.'ID'('args?')'        #guncall
       | 'Radar.'ID'('args?')'      #radarcall
       | 'Battlefield.'ID'('args?')' #battlefieldcall
       | 'Utils.'ID'('args?')'      #utilscall
       ;

ecall : 'Event.'ID'('args?')';
args : expr (',' expr)*;

```

```
61  expr   : MINUS expr                          #minusexpr
62         | NOT expr                            #notexpr
63         | expr op=(MULT | DIV | MOD) expr     #mulexpr
64         | expr op=(PLUS | MINUS) expr         #addexpr
65         | expr op=(LTEQ | GTEQ | LT | GT) expr #relexpr
66         | expr op=(EQ | NEQ) expr             #eqexpr
67         | expr AND expr                       #andexpr
68         | expr OR expr                        #orexpr
69         | atomic                              #atomicexpr
70         ;
71
72  atomic : '(' expr ')'
73      | call
74      | literal
75      ;
76
77  literal : ID      #id
78         | NUM      #num
79         | STRING   #string
80         | BOOL     #bool
81         ;
82
83  COMMENT : '/*'~['*/']*'*/' -> skip;
84  SPACE : [ \r\t\n] -> skip;
```

4.2.1 Lexicon

The definition of what input are allowed for each lexeme in the grammar is defined by means of regular expressions. This will be described here with a table of terminals with matching regex, found in table 4.1. A stream of characters is read by the scanner of the compiler and then turned into a token name defined by the regex. Due to the way the context-free grammar is implemented there are not a lot of terminals. This is because that the terminal *ID* is used widely through the CFG. The general usage of this term creates consistency for the user giving that they quickly can get a feel for what input is allowed.

4.3 Language constructs

In this section the language constructs for this project will be defined and explained, in order to give a clearer view of DatLanguage.

Token	Regular expressions
ID	[a-z] ([a-z] | [A-Z])*
OR	'OR'
AND	'AND'
EQ	'IS=' | 'NOT='
REL	'>'|'<'|'>='|'<='
ADD	'+'|'-'
MUL	'*'|'/'|'%'
NUM	[0-9]+("."[0-9]*)?|"."[0-9]+
BOOL	'False' | 'True'
STRING	'"'.*'"'
TYPE	'Num' | 'Bool' | 'String'

Table 4.1. Table with terminals and matching regular expressions.

4.3.1 Setup block

The Setup block is a block which is only executed once at the beginning of each round. In this block the user can define colors of different parts of the robot, call user defined methods and reserved calls. The Setup block can only be declared once and has to be in the code, else the user will not be able to run the code.

Definition 4.3.1
The Setup block consists of the reserved word ***Setup*** *followed by { **body** }, where the body is the user defined code.*

4.3.2 Repeat block

The Repeat block will iterate through the user's code, unless an event has occurred which then has first priority. The Repeat block is the general behaviour and logic of the robot, and will often contain the main movements of the robot. Just as the Setup block, the Repeat block can also only be declared once and has to be in the code.

Definition 4.3.2
The Repeat block is defined with the reserved word ***Repeat*** *followed by { **body** }, where the body is the user defined code.*

4.3.3 When block

The When block is handling an event whenever it occurs during the battle. The When block is performing the behaviour and logic when the events occur, and this will be defined by the user. Only a predefined number of events can be declared, which can be found in the Appendix A.2.

Definition 4.3.3
*The When block is defined as: When eventName { **body** }.* ***When*** *is the reserved word, the name of a Robocode event followed by the body which is the user defined code.*

4.3.4 Variables

In DatLanguage there are three different kinds of variables, **Num**, **Bool** and **String**. All numeric types will be interpreted as **Num**. **Bool** can either be True or False. **String** is an array of characters.

Definition 4.3.4
The variables can be declared using the reserved words ***Num***, ***Bool*** *and* ***String*** *followed by the desired ID of the variable. Should the variable also be initialized, the assignment operator "=" must be used followed by the value or expression for the variable.*

4.3.5 Action block

An Action block is an abstraction for statements with no return types. Actions can be called throughout the code. The parameters in the Action block is always call by value, just as in the programming language Java, this will be the same in the Function block.

Definition 4.3.5
An Action block is defined with the reserved word ***Action*** *followed by the ID of the Action, a desired amount of parameters in parentheses and the body of the Action block within {}.*
An Action block will look like: ***Action*** *myAction (parameters) { **body** }.*
To use an Action, the keyword run has to be before the Action's ID:
run *anAction(parameters)*

4.3.6 Function block

The Function block is very similar to the Action block. The difference of the two blocks is that the Function block has a return type.

Definition 4.3.6
The Function block is defined with the reserved word ***Function*** *followed by the ID of the Function, what the Function will be returning, a desired amount of parameters in parentheses and the body of the Function block within {}.*
An example of the Function block would be: ***Function*** *myFunction returns returnType (parameters) { **body** }*

4.3.7 Reserved calls

In DatLanguage, there are a few reserved calls, meaning that there are some predefined calls that can't be used for other purposes other than what they're intended to be used for. The reserved calls for DatLanguage are: **Tank.**, **Gun.**, **Radar.**, **Battlefield.** and **Utils**.

Definition 4.3.7
A reserved call is defined by the use of one of the five reserved calls: ***Tank.****,* ***Gun.****,* ***Radar.****,****Battlefield.*** *and* ***Utils*** *followed by a method. The list containing each method the reserved calls can call, can be found in Appendix* **??**.
A call with a reserved call could look like: ***Tank.****energy()*

4.3.8 Conditional block

The conditional block is the language's conditional statements, which will execute the user defined code in it's body depending on a boolean expression. Conditional blocks helps making the robots logic for its behaviour.

Definition 4.3.8
The conditional block contains a minimum of one if-construct, an optional number of else if-constructs and can be ended with an optional else-construct.

The if-construct is noted with the reserved word ***if*** *followed by the boolean expression in parentheses and the body within {}: If(true) body.*

The else if-construct's structure is the same as the if-construct, except that the reserved word ***else if*** *should be used instead of the* ***if****.*

*The else-construct is noted by the reserved word **else** followed by the the body in {}.*

4.3.9 Operators

The included operators in our language can be seen in table 4.2. In the same table there's also an example of how each operator can be used.

Operator	Description	Example
=	Assignment	assign = example
+, -	Addition and Subtraction	a + b - c = example
*, /, %	Multiplication, Division & Modulo	a * b / c % d = example
>, <, >=, <=	Comparison; Greater than, lesser than, etc.	a > b, c =< d
IS= , NOT=	Comparison; S equal and IS NOT equal	b IS= b, a NOT= c
OR	Logical OR	a > b OR c < d
AND	Logical AND	a > b AND c < d

Table 4.2. Table with operators in our language.

4.4 Type systems

In this section the type and scope rules will be explained. To do so, [Sebesta, 2009] has been used to understand and explain how the type and scope rules impacts our project.

4.4.1 Type rules

The following subsection will describe the type rules of the language. These rules will express which operators can be applied to each type, and how each expression will have a type, determined by the operands and operators in the expression. Operators have been divided into groups, and each group of the operators have semantically equivalent rules. The operators [+, -, *, /, %] will be used as the shorthand 'op'. Relational operators [<, >, <=, >=, NOT=, IS=, AND, OR] will be used as the shorthand 'relop'. There're two different types of negations, one for numbers and one for bools: -(num) for numbers, and NOT(bool) for bools.

$$\frac{\Gamma \vdash e_1 : Num \Gamma \vdash e_2 : Num}{\Gamma \vdash e_1 op e_2 : Num}$$

$$\frac{\Gamma \vdash e_1 : Num \Gamma \vdash e_2 : Num}{\Gamma \vdash e_1 relop e_2 : Bool}$$

$$\frac{\Gamma \vdash e_1 : Num}{\Gamma \vdash -e_1 : Num}$$

$$\frac{\Gamma \vdash e_1 : String \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 + e_2 : String}$$

$$\frac{\Gamma \vdash e_1 : Bool \Gamma \vdash e_2 : Bool}{\Gamma \vdash e_1 relop e_2 : Bool}$$

$$\frac{\Gamma \vdash e_1 : Bool}{\Gamma \vdash NOT e_1 : Bool}$$

4.4.2 Scope rules

Scope rules are where the scope of a name is part of the program, where the name binding is associated with an entity. Entities are variables, function names, action names, etc. The scoping mechanism used in this project, is the lexical scoping, as the scope of a name binding refers to a portion of the source code. In DatLanguage there are three kinds of scopes: local scope, global scope and function scope. A variable is visible in the specific scope, if it can be referenced to. Two variables of the same name, can't be in the same scope. The local variables are declared inside blocks of code. The scope of a name binding is delimited to the deepest block where it has been declared. The blocks are marked with { }. In the example in listing 4.6 the variable degrees is declared locally in the code block Setup, and it can't be used in other code blocks unless it is nested inside of the Setup block, it could e.g. be used in a if-loop in the Setup block.

Listing 4.6. Example of local scoping rules

```
1  Setup{
2          num degrees;
3          degrees = 90;
4
5          Tank.forward(degrees);
6  }
```

The parameters declared in a function, are visible in the function scope and can be used just as local variables for the function. It is possible to shadow parameters by creating an inner block and declare a variable with the same name. However it isn't possible to create two variables with the same name in the same block.

When a variable is declared in the global scope, it can be used in all of the other code blocks. When a variable is used, it will first search for it in the local scope, if it is not found in the local scope, it will then search for it in the global scope. In listing 4.7 Codeblock1 uses the global variable, where Codeblock2 uses a local variable with the same name as the global variable, but the local variable will be used.

Listing 4.7. Example of global scoping rules

```
1  Num number = 2; /* Declared globally */
2
3  Codeblock1{
4          5 + number; /* Will use the global variable number */
5  }
6
7  Codeblock2{
8          Num number = 5; /* Locally declared variable */
9          5 + number; /* Will use the local variable number */
10 }
```

Semantics 5

This chapter provides a formal description of the language semantics. The chapter uses techniques from the book *Transitions and Trees* [Hüttel, 2010].

5.1 Syntactic categories

To simplify the presentation of the semantics of our language, syntactic categories have been used. The syntactic categories are based on the grammar found in section 4.2. A collection of meta variables are presented in the following paragraphs which will be used throughout the chapter to present the operational semantics.

$e \in$ **Expr** – *Expressions*
$S \in$ **Stmt** – *Statements*
$n \in$ **Num** – *Numerals*
$b \in$ **Bool** – *Boolean literals*
$B \in$ **Block** – *Blocks*
$tx \in$ **String** – *String literals*
$t \in$ **Type** – Num, *Bool and String types*
$x \in$ **var** – *Variable name*
$f \in$ **Func** – *Function name*
$a \in$ **Act** – *Action name*
$D_a \in$ **ActDcl** – *Action declaration*
$D_f \in$ **FuncDcl** – *Function declaration*
$D_v \in$ **VarDcl** – *Gobal variable declaration*
$F \in$ **Battlefield** – $Battlefield\ configuration$

5.2 Formation rules

These rules have the purpose of defining how to format code in the language. It doesn't define how things works, this will be described later in the semantics.

$B ::= \{\ D_v\ S\ \}$
$S ::=\ skip \mid x\ =\ a \mid B \mid call\ a(e_1...,e_n) \mid return\ e \mid S_1;\ S_2 \mid if\ e\ B_1\ else\ B_2$
$\mid repeat\ while\ e\ B \mid active\ S\ end \mid atom\ S\ end$
$a ::=\ action\ a(x_1...x_n)\ is\ B\ D_a \mid \epsilon$
$f ::=\ function\ f(x_1...x_n)\ is\ B\ D_f \mid \epsilon$
$D_v ::=\ \tau\ x\ D_v \mid \epsilon$
$e ::=\ (\ e\ ) \mid call\ f(x_1...x_n) \mid e_1 ope_2 \mid\ !e \mid \epsilon$

$Game ::= S \mid\mid F$
$op \in +, -, *, /, \%, =, \neq, >, <, \not\leq, \not\geq, \not>, \not<, \geq, \leq, AND, OR$

5.3 Semantic functions

The purpose of the semantics functions to interpret syntactic elements to semantic elements. The semantic functions used in our language will be described in this section.

5.3.1 Numeral literals

The numbers literal in our language will be Numerals, which will be interpreted to real numbers by means of the semantic function:

$\mathcal{N} : \mathbf{Num} \to \mathbb{R}$

Using this function, numerals as $\mathcal{N}[\underline{5}]$ and $\mathcal{N}[\underline{5.36}]$ will be mapped to the corresponding values 5 and 5.36.

5.3.2 String literals

A String literal is a sequence of symbols and characters in UTF-8(Unicode Transformation Format 8-bit) except the delimiter ("). The sequence of symbols and characters must be within the delimiter, for example "Hello world!".
String literal are interpreted as string elements Text, by means of the following: τ : **String** $\to$ **StringL**
As an example : $\tau(\text{"}Insert\ text\ here\text{"}) \to Insert\ text\ here$

5.3.3 Boolean literals

The Boolean literals depict whether the expression is evaluated to True or False. The semantic value for true is $\top$, and false is $\bot$.

5.4 Environment store model

The environment store model describes how variables are bound to a storage during the execution of programs. The variables are bound to a storage-call, where the storage-cell contains the value of the variable. Storage-cells are also called *locations*, the set of locations is denoted **Loc**.

The variable environment is a function that will tell what storage location a variable is bound to. Variable environments is the set of partial functions from the variable to their location, as shown in the following equation:

$E_v \in \boldsymbol{EnvV} = \boldsymbol{Var} \cup \{next\} \rightharpoonup \boldsymbol{Loc}$

We assume the *next* element does not belong to *var* and will be used to point to the next available location in memory.

Let l be an arbitrary element of **Loc** and assume that **Loc** = $\mathbb{N}$. The function **new** found below, will return the successor for every location by defining **new** as: **new** l = l + 1.

$\boldsymbol{new}: \boldsymbol{Loc} \rightarrow \boldsymbol{Loc}$

Let E_v be an arbitrary element in **EnvV**, we denote by $E_v[x \mapsto l]$ the variable environment $E_v^{'}$ defined as follows:

$$E_v^{'}(y) = \begin{cases} E_v(y) & \text{if } y \neq x \\ l & \text{if } y = x \end{cases}$$

$Env_v[x \mapsto v]$ represents the update of $x \in var$ *to* $l \in Loc$.

A partial function in **St** is defined below with an arbitrary element st.

$st \in \boldsymbol{St} = \boldsymbol{Loc} \rightharpoonup \boldsymbol{Value}$

For updating the store st $\in$ **St** the update notation $sto[l \mapsto v]$ will be used. $sto[l \mapsto v]$ will denote the store function $sto^{'}$ defined as follows:

$$sto^{'}(l^{'}) = \begin{cases} sto(l) & \text{if } l^{'} \neq l \\ v & \text{if } l^{'} = l \end{cases}$$

5.5 Operational semantic

This section has the purpose of formally describing the language. It consists of semantic rules that give context to a program.
The semantics is described with big-step semantics due to the intuitiveness of this kind of semantics.
The environments during the operational semantics will be denoted with E. For simplicity the environments E_v, E_a, E_f will now be denoted as E_{vaf} and E_a, E_f will be denoted as E_{af}.

5.5.1 Blocks

$$[BLOCK-ENTER]\frac{\langle D_v,\ E_v\rangle \rightarrow_{Dv} E_v^{'}}{E_{eaf} \vdash \langle active\ S\ end,\ st,\ E_l\rangle \Rightarrow \langle active\ S\ end,\ st,\ E_v^{'} : E_l}$$

$$[BLOCK-CONTINUE]\frac{E_{eaf} \vdash \langle S,\ st,\ E_l\rangle \Rightarrow \langle S^{'},\ st^{'},\ E_l^{'}}{E_{eaf} \vdash \langle active\ S\ end,\ st,\ E_l\rangle \Rightarrow \langle active\ S^{'}\ end,\ st^{'},\ E_l^{'}\rangle}$$

$$[BLOCK-EXIT]\frac{E_{eaf}\ vdash\ \langle S,\ st,\ E_l\rangle \Rightarrow \langle st^{'},\ E_v : E_l^{'}\rangle}{E_{eaf} \vdash \langle active\ S\ end,\ st,\ E_l\ \rangle \Rightarrow \langle st^{'},\ E_l^{'}\rangle}$$

Declarations

$$[EMPTY-VARDCL]\quad \frac{}{\langle \epsilon,\ E_v\rangle \rightarrow_{Dv} E_v}$$

$$[VARDCL]\quad \frac{\langle D_v,\ E_v[x \mapsto l][next \mapsto new(l)]\rangle \rightarrow_{Dv} E_v^{'}}{\langle\ \tau\ x;\ D_v,\ E_v\rangle \rightarrow_{Dv} E_v^{'}} \qquad l = E_v(next)$$

$$[EMPTY-ACTDCL]\quad \frac{}{E_v \vdash \langle \epsilon, E_a\rangle \rightarrow_{Da} E_a}$$

$$[ACTDCL] \quad \frac{E_v \vdash \langle D_a,\ E_a[a \mapsto (B,\ x_1..x_n\ E_v)]\rangle \rightarrow_{Da}\ E_a'}{E_v \vdash \langle Action\ a(x_1..x_n)\ is\ B;\ D_a,\ E_a\rangle \rightarrow_{Da}\ E_a'}$$

$$[EMPTY - FUNCDCL] \quad \frac{}{E_v \vdash \langle \epsilon,\ D_f\rangle \rightarrow_{Df}\ E_f}$$

$$[FUNCDCL] \quad \frac{E_v \vdash \langle D_f,\ E_f[f \mapsto (B,\ x_1..x_n,\ E_v)]\rangle \rightarrow_{Df} E_f'}{E_v \vdash \langle Function\ f(x_1..x_n)\ is\ B;\ D_f,\ E_f\rangle \rightarrow_{Df}\ E_f'}$$

5.5.2 Statements

The interaction with the battlefield will not be explained since it is out of the scope of the project. The interaction that a battlefield configuration may have with a statement running in parallel will be explained.

The configuration of the semantic statements are:

- $\langle S,\ st,\ E_l\rangle$, which are intermediate configurations where S is a statement that may contain atomic-encapsulations or parallels.
- $\langle st,\ E_l\rangle$ is terminal configurations.

The transition system for statements is: $((\boldsymbol{Stmt} \times \boldsymbol{St} \times \boldsymbol{E_l}) \cup (\boldsymbol{St} \times \boldsymbol{E_l}), \Rightarrow, (\boldsymbol{St} \times \boldsymbol{E_l}))$

The set of run time stacks is: $\mathbf{Envl} = \mathbf{EnvV}^*$

We will denote by E_l a generic element in EvCxt For simplicity we will now denote E_l, E_a, E_f as E_{laf} and E_e, E_a, E_f as E_{eaf} and E_v, E_f as E_{vf}, and lastly the environments E_v, E_e, E_a, E_f as E_{veaf}.

$$[ATOM - CONTINUE] \frac{E_{eaf} \vdash \langle S,\ st,\ E_l\rangle \Rightarrow \langle S',\ st',\ E_l'\rangle}{E_{eaf} \vdash \langle atom\ S\ end,\ st,\ E_l\rangle \Rightarrow \langle atom\ S'\ end,\ st',\ E_l'\rangle}$$

$$[ATOM - EXIT] \frac{E_{eaf} \vdash \langle S,\ st,\ E_l\rangle \Rightarrow \langle st',\ E_v\ :\ E_l'\rangle}{E_{eaf} \vdash \langle atom\ S\ end,\ st,\ E_l\rangle \Rightarrow \langle st',\ E_l'\rangle}$$

In the formation rules section the formation rule "Game", describes the interaction between the battlefield and the program. The "F" in Game is the battlefield configuration.

F have to states in our language, either an event is raised or not. If no events are raised, F stays the same. If an event has been raise $F \rightarrow_{event(p)} F'$, which is an transition from the battlefield configuration. It is possible for the events to be raised with an parameter p.

$$[PARA - PROG] \frac{E_{eaf} \vdash \langle S,\ st,\ E_l\rangle \Rightarrow_S \langle S',\ st',\ E_l'\rangle}{E_{eaf} \vdash \langle S \parallel F,\ st,\ E_l\rangle \Rightarrow \langle S' \parallel F,\ st',\ E_l'\rangle}$$

$$[PARA{-}EVENT]\frac{F \ \rightarrow_{event(p)} \ F' \quad \langle D_v,\ E_v[x \mapsto l][next \mapsto new(l)]\rangle \ \rightarrow_{Dv} \ E_v'}{E_{eaf} \ \vdash \ \langle S \ || \ F,\ st,\ E_l\rangle \ \Rightarrow \ \langle atom\ S'\ end;\ S \ || \ F',\ st[l \mapsto p],\ E_v' \ :\ E_l\rangle}$$

$Where\ S\ is\ not\ of\ the\ form\ S_1\ end;\ S_2$
$E_v \ = \ E_v \ : \ E_l',\ and\ l \ = \ E_v(next)$
$E_e(event) \ = \ (begin\ D_v\ S'\ end,\ x,\ E_v)$

$$[PARA-QUIT]\frac{F \ \rightarrow_{event(p)} \ F' \quad \langle D_v,\ E_v[x \mapsto l][next \mapsto new(l)]\rangle \ \rightarrow_{Dv} \ E_v'}{E_{eaf} \ \vdash \ \langle S \ || \ F,\ st,\ E_l\rangle \ \Rightarrow \ \langle active\ S'\ end,\ st[l \mapsto p],\ E_v' \ :\ E_l\rangle}$$

$Where\ S\ is\ not\ of\ the\ form\ S_1\ end;\ S_2$
$E_v \ = \ E_v \ : \ E_l',\ and\ l \ = \ E_v(next)$
$E_e(quit) \ = \ (begin\ D_v\ S'\ end,\ x,\ E_v)$

$$[COMP-PARTLY]\frac{E_{eaf} \ \vdash \ \langle S_1,\ st,\ E_l\rangle \ \Rightarrow \ \langle S_1',\ st',\ E_l'\rangle}{E_{eaf} \ \vdash \ \langle S_1;\ S_2,\ st,\ E_l\rangle \ \Rightarrow \ \langle S_1';\ S_2,\ st',\ E_l\prime\rangle}$$

$$[COMP-COMPLETE]\frac{E_{eaf} \ \vdash \ \langle S_1,\ st,\ E_l\rangle \ \Rightarrow \ \langle\ st',\ E_l'\rangle}{E_{eaf} \ \vdash \ \langle S_2,\ st,\ E_l\rangle \ \Rightarrow \ \langle S_1';\ S_2,\ st',\ E_l\prime\rangle}$$

$$[ASSIGN]\frac{E_{veaf} \ \vdash \ \langle e,\ st\rangle \ \rightarrow_e \ (v,\ st')}{E_{eaf} \ \vdash \ \langle x \ = \ e,\ st,\ E_l\rangle \ \Rightarrow \ (st'[l \mapsto v],\ E_l)}$$

$Where \qquad E_l = E_v : E_l' \qquad l = E_v(x)$

$$[IF{-}_e^{\top}]\frac{E_{vf} \ \vdash \ \langle e,\ st\rangle \ \rightarrow_e \ (\top,\ st')}{E_{eaf} \ \vdash \ \langle if\ e\ B_1\ else\ B_2,\ st,\ E_l\rangle \ \Rightarrow \ \langle B_1,\ st',\ E_l\rangle}$$

$Where \quad E_l = E_v : E_l\text{‘}$

$$[IF{-}_e^{\bot}]\frac{E_{vf} \ \vdash \ \langle e,\ st\rangle \ \rightarrow_e \ (\bot,\ st')}{E_{eaf} \ \vdash \ \langle if\ e\ B_1\ else\ B_2,\ st,\ E_l\rangle \ \Rightarrow \ \langle B_2,\ st',\ E_l\rangle}$$

$Where \quad E_l = E_v : E_l\text{‘}$

$$[CALL-ACT]\frac{E_{veaf} \ \vdash \ \langle e_i,\ st_i\rangle \ \rightarrow_e \ (v_i,\ st_{i+1}') \quad for\ i \ \in \ 1...n}{E_{eaf} \ \vdash \ \langle call\ a(e_1 \ldots e_n),\ st,\ E_l\rangle \ \Rightarrow \ \langle active\ B\ end,\ st',\ E_v'' \ :\ E_l}$$

$Where \qquad st_1 = st, E_l = E_v : E_l, E_a(a) = (B, (x_1...x_n), E_v')$
$l_1 = E_v(next) and, l_{j+1} = new(l_j) for j \in 1...n$
$E_v'' = E_v'[x_1 \mapsto l_1]...[x_n...l_n][next \mapsto l_{n+1}]$
$st' = st_{n+1}[l_1 \mapsto v_1]...[l_n \mapsto v_n]$

$$[REPEAT]\frac{}{E_{eaf} \vdash \langle repeat\ while(e)\ B,\ st,\ E_l\rangle \Rightarrow \langle if\ e\ (B;\ repeat\ while(e)\ B)\ else\ skip,\ st,\ E_l\rangle}$$

$$[SKIP]\frac{}{\langle skip,\ st,\ E_l\rangle \Rightarrow (st,\ E_l)}$$

$$[RETURN]\frac{E_{veaf} \vdash \langle e,\ st\rangle \rightarrow_e (v,\ st')}{E_{eaf} \vdash \langle return\ e,\ st,\ E_l\rangle \Rightarrow (st'[r \mapsto v],\ E_l)} \quad Where \qquad E_l = E_v : E_l'$$

5.5.3 Expressions

$$[PAR-EXPR]\quad \frac{E_{veaf} \vdash \langle e,\ st\rangle \rightarrow_e (v,\ st'}{E_{veaf} \vdash \langle (e),\ st\rangle \rightarrow_e (v,\ st')}$$

$$[CALL{-}FUNC]\frac{E_{eaf} \vdash \langle active\ B\ end,\ st'',\ E_v''\rangle \Rightarrow^* (st', E_l) \qquad E_{veaf} \vdash \langle e_i,\ st_i\rangle \rightarrow_e (v_i,\ st_{i+1})}{E_{veaf} \vdash \langle call\ f(x_1...x_n),\ st\rangle \rightarrow_e (v,\ st')}$$

$Where \quad E_f(f) = (B, (x_1...x_n), E_v')$
$st_1 = st$
$l_1 = E_v(next), l_{j+1} = new(l_j) for j \in 1...n$
$E_v'' = E_v'[x_1 \mapsto l_1]...[x_n \mapsto l_n][next \mapsto l_{n+1}]$
$st'' = St_{n+1}[l_1 \mapsto v_1]...[l_n \mapsto v_n]$
$v = st'(r)$

5.5.4 Arithmetic expressions

$$[ASM-EXPR]\quad \frac{E_{veaf} \vdash \langle a_1,\ st\rangle \rightarrow_e (v_1, st'') \qquad E_{veaf} \vdash \langle a_2,\ st''\rangle \rightarrow_e (v_2,\ st')}{E_{veaf} \vdash \langle a_1\ op\ a_2, st\rangle \rightarrow_e (v, st')}$$

$op \in \{ +, -, * \}$
$v = v_1\ op\ v_2$

$$[DM-EXPR]\quad \frac{E_{veaf} \vdash \langle a_1,\ st\rangle \rightarrow_e (v_1, st'') \qquad E_{veaf} \vdash \langle a_2,\ st''\rangle \rightarrow_e (v_2,\ st')}{E_{veaf} \vdash \langle a_1\ op\ a_2, st\rangle \rightarrow_e (v, st')}$$

$op \in \{ /, \% \} | v \neq 0$
$v = v_1\ op\ v_2$

5.5.5 Boolean expressions

Most of the boolean expressions look alike even with the value of true/false, except the AND/OR expressions. The constructs of AND/OR uses short-circuiting, since in the OR expression if the first value is evaluated to true, there is no need to evaluate the second part of the expression.

$$[EXPR^{IS=}_{\top}] \quad \frac{E_{veaf} \vdash \langle e_1,\ st\rangle \rightarrow_e (v_1, st'') \qquad E_{veaf} \vdash \langle e_2,\ st''\rangle \rightarrow_e (v_2,\ st')}{E_{veaf} \vdash \langle e_1\ IS = e_2,\ st\rangle \rightarrow_e (\top,\ st')}$$

$v_1 = v_2$

$$[EXPR^{IS=}_{\bot}] \quad \frac{E_{veaf} \vdash \langle e_1,\ st\ \rangle \rightarrow_e (v_1, st'') \qquad E_{veaf} \vdash \langle e_2,\ st''\rangle \rightarrow_e (v_2,\ st')}{E_{veaf} \vdash \langle e_1\ IS = e_2,\ st\rangle \rightarrow_e (\bot,\ st')}$$

$v_1 \neq v_2$

$$[EXPR^{NOT=}_{\top}] \quad \frac{E_{veaf} \vdash \langle e_1,\ st\rangle \rightarrow_e (v_1, st'') \qquad E_{veaf} \vdash \langle e_2,\ st''\rangle \rightarrow_e (v_2,\ st')}{E_{veaf} \vdash \langle e_1\ NOT = e_2,\ st\rangle \rightarrow_e (\top,\ st')}$$

$v_1 \neq v_2$

$$[EXPR^{NOT=}_{\bot}] \quad \frac{E_{veaf} \vdash \langle e_1,\ st\rangle \rightarrow_e (v_1, st'') \qquad E_{veaf} \vdash \langle e_2,\ st''\rangle \rightarrow_e (v_2,\ st')}{E_{veaf} \vdash \langle e_1\ NOT = e_2,\ st\rangle \rightarrow_e (\bot,\ st')}$$

$v_1 = v_2$

$$[EXPR^{\geq}_{\top}] \quad \frac{E_{veaf} \vdash < e_1,\ st > \rightarrow_e (v_1, st'') \qquad E_{veaf} \vdash < e_2,\ st'' \rightarrow_e (v_2,\ st')}{E_{veaf} \vdash < e_1\ > e_2,\ st > \rightarrow_e (\top,\ st')}$$

$v_1 > v_2$

$$[EXPR^{>}_{\bot}] \quad \frac{E_{veaf} \vdash \langle e_1,\ st\rangle \rightarrow_e (v_1, st'') \qquad E_{veaf} \vdash \langle e_2,\ st''\rangle \rightarrow_e (v_2,\ st')}{E_{veaf} \vdash \langle e_1\ > e_2,\ st\rangle \rightarrow_e (\bot,\ st')}$$

$v_1 \ngtr v_2$

$$[EXPR^{\leq}_{\top}] \quad \frac{E_{veaf} \vdash \langle e_1,\ st\rangle \rightarrow_e (v_1, st'') \qquad E_{veaf} \vdash \langle e_2,\ st''\rangle \rightarrow_e (v_2,\ st')}{E_{veaf} \vdash \langle e_1\ < e_2,\ st\rangle \rightarrow_e (\top,\ st')}$$

$v_1 < v_2$

$$[EXPR^{<}_{\bot}] \quad \frac{E_{veaf} \vdash \langle e_1,\ st\rangle \rightarrow_e (v_1, st'') \qquad E_{veaf} \vdash \langle e_2,\ st''\rangle \rightarrow_e (v_2,\ st')}{E_{veaf} \vdash \langle e_1\ < e_2,\ st\rangle \rightarrow_e (\bot,\ st')}$$

$v_1 \nleq v_2$

$$[EXPR^{>=}_{\top}] \quad \frac{E_{veaf} \vdash \langle e_1,\ st\rangle \rightarrow_e (v_1, st'') \qquad E_{veaf} \vdash \langle e_2,\ st''\rangle \rightarrow_e (v_2,\ st')}{E_{veaf} \vdash \langle e_1\ >= e_2,\ st\rangle \rightarrow_e (\top,\ st')}$$

$v_1 \geq v_2$

$$[EXPR^{>=}_{\bot}] \quad \frac{E_{veaf} \vdash \langle e_1,\ st\rangle \rightarrow_e (v_1, st'') \qquad E_{veaf} \vdash \langle e_2,\ st''\rangle \rightarrow_e (v_2,\ st')}{E_{veaf} \vdash \langle e_1\ >= e_2,\ st\rangle \rightarrow_e (\bot,\ st')}$$

$v_1 \ngeq v_2$

$$[EXPR^{<=}_{\top}] \quad \frac{E_{veaf} \vdash \langle e_1,\ st\rangle \rightarrow_e (v_1, st'') \qquad E_{veaf} \vdash \langle e_2,\ st''\rangle \rightarrow_e (v_2,\ st')}{E_{veaf} \vdash \langle e_1\ <= e_2,\ st\rangle \rightarrow_e (\top,\ st')}$$

$v_1 \leq v_2$

$$[EXPR\ {}^{<=}_{\perp}] \quad \frac{E_{veaf} \vdash \langle e_1,\ st\rangle \rightarrow_e (v_1, st'') \qquad E_{veaf} \vdash \langle e_2,\ st''\rangle \rightarrow_e (v_2,\ st')}{E_{veaf} \vdash \langle e_1\ <= e_2,\ st\rangle \rightarrow_e (\perp,\ st')}$$

$v_1 \not\leq v_2$

$$[EXPR\ {}^{!}_{\top}] \quad \frac{E_{veaf} \vdash \langle e,\ st\rangle \rightarrow_e (\top, st')}{E_{veaf} \vdash \langle !e,\ st\rangle \rightarrow_e (\top,\ st')}$$

$$[EXPR\ {}^{!}_{\perp}] \quad \frac{E_{veaf} \vdash \langle e,\ st\rangle \rightarrow_e (\perp, st')}{E_{veaf} \vdash \langle !e,\ st\rangle \rightarrow_e (\perp,\ st')}$$

$$[EXPR\ {}^{AND}_{\top}] \quad \frac{E_{veaf} \vdash \langle e_1,\ st\rangle \rightarrow_e (\top,\ st'') \qquad E_{veaf} \vdash \langle e_2,\ st'' \rightarrow_e (v, st')}{E_{veaf} \vdash < e_1\ AND\ e_2,\ st > \rightarrow_e (v,\ st')}$$

$$[EXPR\ {}^{AND}_{\perp}] \quad \frac{E_{veaf} \vdash < e_1,\ st > \rightarrow_e (\perp,\ st')}{E_{veaf} \vdash < e_1\ AND\ e_2,\ st > \rightarrow_e (\perp,\ st')}$$

$$[EXPR\ {}^{OR}_{\top}] \quad \frac{E_{veaf} \vdash < e_1,\ st > \rightarrow_e (\top,\ st')}{E_{veaf} \vdash < e_1\ OR\ e_2,\ st > \rightarrow_e (\top,\ st')}$$

$$[EXPR\ {}^{OR}_{\perp}] \quad \frac{E_{veaf} \vdash \langle e_1,\ st\rangle \rightarrow_e (\perp,\ st'') \qquad E_{veaf} \vdash \langle e_2,\ st'' \rightarrow_e (v, st')}{E_{veaf} \vdash < e_1\ OR\ e_2,\ st > \rightarrow_e (v,\ st')}$$

Implementation 6

The phases the compiler goes through are shown in figure 6.1. Before this, the ANTLR tool has generated a scanner and a parser using the grammar showed in section 4.2. The ANTLR scanner is used to transform the source code into tokens. These tokens are then used by the ANTLR parser to generate the parse tree. After this the compiler does three traversals of this parse tree. First traversal is the funcDef phase, where every function the user has written is added to a funcTable for use in later phases. Second traversal is the Type checking phase. This phase uses a symbol table to check that the type and scope rules are followed by the code. The third traversal is the code generation phase, where the compiler goes through the parse tree and translates it to Java code. Lastly the Java code is sent to the Java compiler and compiled to the finished Java robot class usable by Robocode.

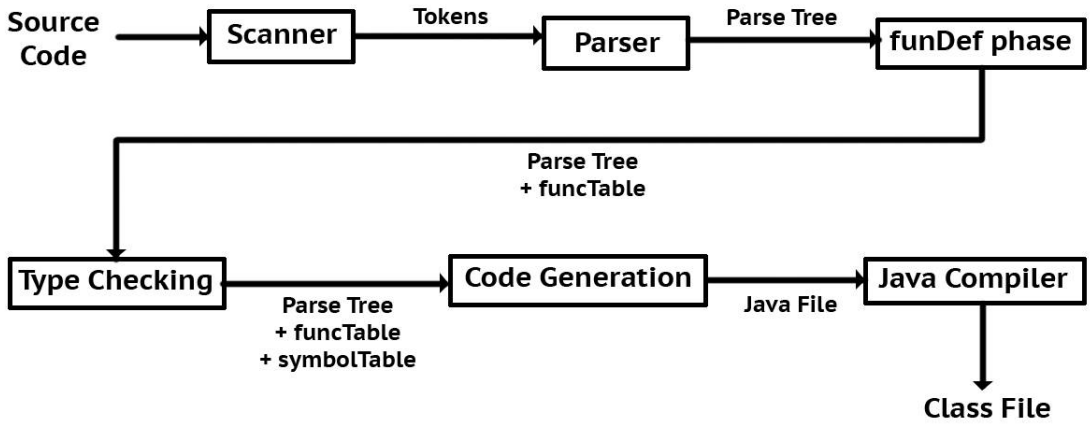

Figure 6.1. Compiler structure

6.1 ANTLR

The ANTLR tool takes as input the source code the user wants to parse along with the grammar for the language. It then performs a lexical analysis to scan the source code for compliance with the grammar to create the tokens. It then parses the tokenized source code to create a tree. On top of creating the parse tree, the ANTLR tool generates two visitor patterns, which can be extended and used. The first and simplest is the listener. The listener has two functions for each node in the parse tree, one called enter and one called exit. With the listener, a walker is needed to traverse the parse tree. As the walker

visits nodes in the parse tree, it calls the right function in the listener. The enter function is called as soon as the walker visits the node. After this, the walker visits all child nodes. When it returns from these child nodes the exit function is called. The second visitor pattern is the visitor. The visitor has one function per node. Each node function is responsible for calling all child nodes functions it has. This gives the developer of the compiler more freedom when to call the child nodes and in which order. Also the visitor supports return types on the functions, which makes it possible to pass data from one node to the caller node.

6.2 Function definition

After ANTLR has produced the parse tree, base listener and base visitor, the function definition phase starts. In this phase every function that already exists in Robocode, will be mapped to the names defined in the appendix A.1, and added to the function table. Also every user defined function or actions will be added to this function table.

6.2.1 FuncSymbol & FuncSymbolTable

To keep track of the functions and their parameters, return values and Robocode names, a FuncSymbol class is used as representation for a single function and a FuncSymbolTable class is used to keep track of every FuncSymbol in the program. The FuncSymbol has string properties for Name, Type, ReturnType, original Robocode names and a string array for the parameters.

The FuncSymbolTable is a key-value pair HashMap, where the key is the type of the function followed by the name of the function and the value is the FuncSymbol. The type in this case, can be either Function or Action for user defined functions, Tank, Gun, Battlefield, Radar or Utils from Robocode functions or the name of the event from Robocode events. This class has two functions, GetFuncSymbol and EnterFuncSymbol which can be found in listing 6.1.

The GetFuncSymbol takes a type and a name from a function, combines them, looks for them in the HashMap and returns the result.

EnterFuncSymbol takes a FuncSymbol and tries to add it to the HashMap. First, it checks if a function with the same type and name was already declared. If such a function exists, an error is thrown, otherwise the function is added to the HashMap.

Listing 6.1. FuncSymbolTable

```
1  public class FuncSymbolTable {
2
3          public LinkedHashMap<String, FuncSymbol> Map = new LinkedHashMap<String, FuncSymbol>();
4
5          public FuncSymbol GetFuncSymbol(String type, String name){
6          FuncSymbol sym = Map.get(type + name);
7          return sym;
8          }
9
10         public void EnterFuncSymbol(FuncSymbol fs){
11         FuncSymbol oldSym = GetFuncSymbol(fs.Type, fs.Name);
12         if (oldSym != null) {
13                 Error e = new Error("Function already declared");
```

```
14                  throw e;
15          }
16          Map.put(fs.Type + fs.Name, fs);
17          }
18  }
```

6.2.2 Robocode functions & events

To allow the type checker and code generation phase to do their jobs, every Robocode function and event needs to be added to the FuncSymbolTable. To do this, a simple text file was created which line for line have each function written out in a specific manner.

6.2.3 User defined functions & actions

To load the user defined functions from the program, a traverse of the parse tree is done using the ANTLR generated walker and listener. For every Function and Action the listener visits, a new FuncSymbol is created and added to the FuncSymbolTable. This is done with the FuncListener class, which extends the BaseListener ANTLR created.

The FuncListener class overrides the Enter/Exit Action declaration functions, the Enter/Exit Function declaration functions and the Enter parameter function. In the Enter Action and Function declaration functions, a new FuncSymbol is created called CurrentFunc. To CurrentFunc a name and a type is added, and in the Function case a return type is added as well. Name, type and return type are found using the parse tree. In the Enter parameter function adds a new parameter to the CurrentFunc's array of parameters. Now in the Exit Action and Function declaration functions the CurrentFunc is added to the FuncSymbolTable. The reason why the CurrentFunc is added in the Exit functions, is that the walker has to visit all the parameters.

6.3 Type checking

After the first traversal of the parse tree using the FuncListener, every function is now defined in the function table and we can begin type checking. To do type checking a Symbol class and a SymbolTable class are used to keep track of every variable and their scope, using these two classes the SymbolTypeVisitor class checks for type compatibility.

6.3.1 Symbol & symbol table

Every variable in the program is represented by the Symbol class. The Symbol class has the properties Name, Type, Var and Depth. The Name and Type represents the name and type of the variable, the interesting parts are the Var and Depth properties. Var is a reference to a Symbol of the same name but in an outer scope, this would work as a stack for all symbols of the same name, where the top of the stack will be the innermost encountered scope and therefore the one used. The Depth property indicates the depth of the scope the symbol is in, with 0 being the global scope.

The SymbolTable is mainly two things, the Map associates a name of a variable with a symbol as key value pair, these symbols are the ones on top of the stack described in the

paragraph before. The scope is an array of arrays of symbols. Each index of the outer array is a scope and each inner array contains the symbols of that scope.

The SymbolTable class has four functions: OpenScope, CloseScope, GetSymbol and EnterSymbol.

OpenScope in listing 6.2 is used to open a new scope. Depth is a global variable, which keeps track of the current scope's depth. When OpenScope is run, the depth is incremented. If the Scope size is less than the depth + 1, a new array is added to the scope, otherwise the old array is cleared.

Listing 6.2. OpenScope function

```
1 public void OpenScope(){
2     depth++;
3     if (Scope.size() < depth+1){
4         Scope.add(new ArrayList<Symbol>());
5     }else {
6         Scope.get(depth).clear();
7     }
8 }
```

The EnterSymbol function in listing 6.3, is used to enter a new symbol into the SymbolTable. When EnterSymbol is run it firstly checks if there already exists a symbol with the same name, if this is the case it checks whether the depth of this symbol is the same depth as the current scope. If the depths are the same, an error is thrown, since there can't exist two variables with the same name in the same scope. If no symbol exists in the Map or it is at another depth, a new symbol is created. This symbol gets the current depth in its depth property and it sets its var property to the old symbol. It then either replaces the new symbol with the old symbol or just adds it if the old symbol was null.

The GetSymbol function looks up a symbol in the Map and will return the name of that symbol.

Listing 6.3. EnterSymbol function

```
1  public void EnterSymbol(String name, String type){
2      Symbol oldsym = Map.get(name);
3      if (oldsym != null && oldsym.Depth == depth){
4          Error e = new Error("Duplicate declaration of " + name);
5          throw e;
6      }else{
7          Symbol newSym = new Symbol();
8          newSym.Name = name;
9          newSym.Type = type;
10         newSym.Depth = depth;
11         newSym.Var = oldsym;
12         Scope.get(depth).add(newSym);
13         Map.put(newSym.Name, newSym);
14     }
15 }
```

CloseScope in listing 6.4, is used to close the current scope. This is done by going through each symbol **s** at the current depth, and replacing **s** in the Map with **s.Var** which is a symbol with the same name but in an outer scope. If no such symbol exists, **s** is replaced with a null value.

Listing 6.4. CloseScope function

```
1 public void CloseScope(){
```

```
    Scope.get(depth).forEach(s -> {
        Symbol prevSym = s.Var;
        Map.replace(s.Name, s, prevSym);
    });
    depth--;
}
```

6.3.2 The SymbolTypeVisitor class

The SymbolTypeVisitor class extends the BaseVisitor class from ANTLR and is the second traversal through the parse tree. It visits every single node in the parse tree and each node returns its type to the caller. At each node it is checked whether the types returned match with the expected types, errors are thrown if they don't match.

At the basic level the types are returned as they are, which are the functions visitNum, visitString and visitBool, that can be found in listing 6.5.

Listing 6.5. SymbolTypeVisitor - visitNum, visitString and visitBool functions

```
@Override
public String visitNum(GrammarParser.NumContext ctx) {
    return "Num";
}

@Override
public String visitString(GrammarParser.StringContext ctx) {
    return "String";
}

@Override
public String visitBool(GrammarParser.BoolContext ctx) {
    return "Bool";
}
```

Also at the basic level variable identifiers are looked up in the SymbolTable and their type is returned. An error is thrown if the variable isn't found in the SymbolTable, as shown in listing 6.6.

Listing 6.6. SymbolTypeVisitor - visitId function

```
@Override
public String visitId(GrammarParser.IdContext ctx) {
    String test = ctx.getText();
    Symbol sym = ST.GetSymbol(ctx.getText());
    if (sym == null){
        Error e = new Error("Error at line: " +
               ctx.start.getLine() + ": Variable not found");
        throw e;
    }
    return sym.Type;
}
```

When visiting and, or, minus, not, mul, rel and eq expressions each expression type is calculated and compared to what is expected. Afterwards either an error is thrown or the right type is returned. An example of such function is the or-expression, which can be found in listing 6.7.

Listing 6.7. SymbolTypeVisitor - visitOrexpr function

```
@Override
```

```
2  public String visitOrexpr(GrammarParser.OrexprContext ctx) {
3      String left = visit(ctx.expr(0));
4      String right = visit(ctx.expr(1));
5      if (left.equals("Bool") && right.equals("Bool")){
6          return "Bool";
7      }else{
8          Error e = new Error("Error at line: " +
9                  ctx.start.getLine() + ": Expression did not evaluate to Bool.");
10         throw e;
11     }
12 }
```

The add-expression in listing 6.8, is treated slightly differently as the plus operator is overloaded and can be used on two numbers or a string and any other type. Therefore the function checks if both expressions evaluate to a number, or one of them is a string and the operator is a plus. It throws an error if none of the above is true else it returns the appropriate type.

Listing 6.8. SymbolTypeVisitor - visitAddexpr function

```
1  @Override
2  public String visitAddexpr(GrammarParser.AddexprContext ctx) {
3      String left = visit(ctx.expr(0));
4      String right = visit(ctx.expr(1));
5      if (left.equals("Num") && right.equals("Num")) {
6          return "Num";
7      }else if((left.equals("String") || right.equals("String")) && ctx.op.getText().equals("+") ) {
8          return "String";
9      }else{
10         Error e = new Error("Error at line: " +
11                 ctx.start.getLine() + ": Expression types did not match.");
12         throw e;
13     }
14 }
```

The function visitAssign in listing 6.9, checks whether the type of the symbol, found from the ID, equals the expression's type. It throws errors if the ID is not found in the SymbolTable or the types don't match.

Listing 6.9. SymbolTypeVisitor - visitAssign function

```
1  @Override
2  public String visitAssign(GrammarParser.AssignContext ctx) {
3      String id = ctx.ID().getText();
4      Symbol sym = ST.GetSymbol(id);
5      if (sym == null){
6          Error e = new Error("variable not found");
7          throw e;
8      }
9      if (!sym.Type.equals(visit(ctx.expr()))){
10         Error e = new Error("Error at line: " +
11                 ctx.expr().start.getLine() + ": Expression and variable are not type compatible");
12         throw e;
13     }
14     return sym.Type;
15 }
```

The function visitVardcl in listing 6.10, adds a new symbol in the SymbolTable. The type is found at the variable declaration node or at the assign node of the parse tree. visitVardcl determines where to find the ID and adds the new symbol to the SymbolTable.

Listing 6.10. SymbolTypeVisitor - visitVardcl function

```
1  @Override
2  public String visitVardcl(GrammarParser.VardclContext ctx) {
3      if (ctx.getChild(1) instanceof GrammarParser.AssignContext){
4          ST.EnterSymbol(((GrammarParser.AssignContext) ctx.getChild(1)).ID().getText(),
               ctx.TYPE().getText());
5          visit(ctx.assign());
6      }else {
7          ST.EnterSymbol(ctx.ID().getText(), ctx.TYPE().getText());
8      }
9      return ctx.TYPE().getText();
10 }
```

The visitBlock function in listing 6.11, opens a new scope and if its parent is the Action declaration, any formal parameter of the Action declaration is added to the scope. The function then visits all the statements in the block and then closes the scope.

Listing 6.11. SymbolTypeVisitor - visitBlock function

```
1  @Override
2  public String visitBlock(GrammarParser.BlockContext ctx) {
3      ST.OpenScope();
4      if (ctx.parent instanceof GrammarParser.ActdclContext){
5          FuncSymbol fs = FST.GetFuncSymbol("Action", ((GrammarParser.ActdclContext)
               ctx.parent).ID().getText());
6          fs.Params.forEach(tuple -> {
7              ST.EnterSymbol(tuple.x, tuple.y);
8          });
9      }
10     visit(ctx.stmts());
11     ST.CloseScope();
12     return "null";
13 }
```

The function visitArgs in listing 6.12 at line 1-8, visits each expression and concatenates with a comma in between and returns the whole thing as a string.

visitFcall which is also found in listing 6.12 at line 10-37, gets a function symbol from the FuncSymbolTable and then matches each argument type from the visitArgs with each parameter type in the function symbol. Errors are thrown if these don't match or the function ID doesn't match any function in the FuncSymbolTable. visitAcall, visitUtilscall, visitBattlefieldcall, visitRadarcall, visitGuncall and visitTankcall functions works similarly to the visitFcall function.

Listing 6.12. SymbolTypeVisitor - visitArgs and visitFcall functions

```
1  @Override
2  public String visitArgs(GrammarParser.ArgsContext ctx) {
3      String result = visit(ctx.expr(0));
4      for (int i = 1; i < ctx.expr().size(); i++){
5          result = result + ", " + visit(ctx.expr(i));
6      }
7      return result;
8  }
9
10 @Override
11 public String visitFcall(GrammarParser.FcallContext ctx) {
12     FuncSymbol fsym = FST.GetFuncSymbol("Function", ctx.ID().getText());
13     if (fsym == null){
14         Error e = new Error("Error at line: " +
15                 ctx.start.getLine() + ": Function not found");
16         throw e;
```

```
17      }
18      if (!fsym.Type.equals("Function")){
19          Error e = new Error("Error at line: " +
20                  ctx.start.getLine() + ": Incorrect function type.");
21          throw e;
22      }
23      if(ctx.getChildCount() == 4) {
24          String[] args = visit(ctx.args()).split(", ");
25          for (int i = 0; i < args.length; i++) {
26              String paramType = fsym.Params.get(i).y;
27              String arg = args[i];
28              if (!paramType.equals(arg)) {
29                  Error e = new Error("Error at line: " +
30                          ctx.args().expr(i).start.getLine() +
31                          ": Parameter number " + i + " not matched. Expected " + paramType);
32                  throw e;
33              }
34          }
35      }
36      return fsym.ReturnType;
37  }
```

visitEcall in listing 6.13, is a bit special compared to the other call functions, as it needs to check whether the function called is comparable with the event that it is called within. To do this, it checks the parent to see if that is an event declaration, if it is not, it checks the parent of the parent and so on, until it either reaches the event declaration or the root of the tree. If the root is reached an error is then thrown. If an event declaration is reached, the event declaration ID and the event call ID is used to look up the function in the FuncSymbolTable. If none is returned an error is thrown, otherwise the arguments are matched with the parameters as before. If successful the function's return type is returned.

Listing 6.13. SymbolTypeVisitor - visitEcall function

```
1   @Override
2   public String visitEcall(GrammarParser.EcallContext ctx) {
3       RuleContext parent = ctx.parent;
4       while(parent != null){
5           if(parent instanceof GrammarParser.EventdclContext){
6               FuncSymbol fsym = RoboFST.GetFuncSymbol(((GrammarParser.EventdclContext)
                     parent).ID().getText(), ctx.ID().getText());
7               if (fsym == null){
8                   Error e = new Error("Error at line: " +
9                           ctx.start.getLine() + ": Function not found");
10                  throw e;
11              }
12              if(ctx.getChildCount() == 5) {
13                  String[] args = visit(ctx.args()).split(", ");
14                  for (int i = 0; i < args.length; i++) {
15                      String paramType = fsym.Params.get(i).y;
16                      String arg = args[i];
17                      if (!paramType.equals(arg)) {
18                          Error e = new Error("Error at line: " +
19                                  ctx.args().expr(i).start.getLine() +
20                                  ": Parameter number " + i + " not matched. Expected " + paramType);
21                          throw e;
22                      }
23                  }
24              }
25              return fsym.ReturnType;
26          }
27          parent = parent.parent;
28      }
29      Error e = new Error("Event calls must be inside an event/when function");
```

```
30      throw e;
31  }
```

visitDcls makes sure that the program the user wrote has one and only one Tankname, Repeatblock and Setupblock declared.

Listing 6.14. SymbolTypeVisitor - visitDcls function

```
1   @Override
2   public String visitDcls(GrammarParser.DclsContext ctx) {
3       if (ctx.tankname().size() < 1){
4           Error e = new Error("A Tankname needs to be declared");
5           throw e;
6       }else if (ctx.tankname().size() > 1){
7           Error e = new Error("Error at line: " +
8                   ctx.tankname(1).start.getLine() + ": Too many tank names.");
9           throw e;
10      }
11      if (ctx.repeatblock().size() < 1){
12          Error e = new Error("A Repeat block needs to be declared");
13          throw e;
14      }else if (ctx.repeatblock().size() > 1){
15          Error e = new Error("Error at line: " +
16                  ctx.repeatblock(1).start.getLine() + ": Too many repeat blocks.");
17          throw e;
18      }
19      if (ctx.setupblock().size() < 1){
20          Error e = new Error("A Setup block needs to be declared");
21          throw e;
22      }else if (ctx.setupblock().size() > 1){
23          Error e = new Error("Error at line: " +
24                  ctx.setupblock(1).start.getLine() + ": Too many setup blocks.");
25          throw e;
26      }
27      super.visitDcls(ctx);
28      return "null";
29  }
```

6.4 Code generation

The CodeGen class extends the BaseListener class and is the third and last traversal of the parse tree. At this point the program is type checked and ready to get translated into Java code, the CodeGen class generates a string containing the translated code.

When visiting the literals bool, num and string nothing is changed and the text is returned. When visiting ID an "_" is added in front of the ID, to make sure the ID name doesn't clash with Java keywords. The literals ID and num can be seen in listing 6.15.

Listing 6.15. CodeGen - visitId & visitNum functions

```
1   @Override
2   public String visitId(GrammarParser.IdContext ctx) {
3       return "_" + ctx.ID().getText();
4   }
5
6   @Override
7   public String visitNum(GrammarParser.NumContext ctx) {
8       return ctx.NUM().getText();
9   }
```

When visiting expressions the only thing needed is to change AND, OR, IS=, NOT= and NOT to their equivalent operator in Java.

In the visitEcall function in listing 6.16 the appropriate FuncSymbol is found in the FuncSymbolTable using the same method as the SymbolTypeVisitor class. visitEcall adds "e." in front of the function name, before it returns. The "e" will be the parameter of the event which this call is within.

Listing 6.16. CodeGen - visitEcall function

```
1  @Override
2  public String visitEcall(GrammarParser.EcallContext ctx) {
3      FuncSymbol fsym;
4      RuleContext parent = ctx.parent;
5      while(parent != null){
6          if(parent instanceof GrammarParser.EventdclContext){
7              fsym = RoboFST.GetFuncSymbol(((GrammarParser.EventdclContext) parent).ID().getText(),
                   ctx.ID().getText());
8              if(ctx.getChildCount() == 5) {
9                  return "e." + fsym.RoboCodeName + "(" + visit(ctx.args()) + ")";
10             }else {
11                 return "e." + fsym.RoboCodeName + "()";
12             }
13         }
14         parent = parent.parent;
15     }
16     Error e = new Error("codeGen Error");
17     throw e;
18 }
```

In all other Robocode calls, like the visitTankcall function found in listing 6.17, the appropriate FuncSymbol is found in the FuncSymbolTable and the Robocode name is returned with potential arguments.

Listing 6.17. CodeGen - visitTankcall function

```
1  @Override
2  public String visitTankcall(GrammarParser.TankcallContext ctx) {
3      FuncSymbol fs = RoboFST.GetFuncSymbol("Tank", ctx.ID().getText());
4      if(ctx.getChildCount() == 5) {
5          return fs.RoboCodeName + "(" + visit(ctx.args()) + ")";
6      }else {
7          return fs.RoboCodeName + "()";
8      }
9  }
```

In Function and Action calls an "_" are added in front of the ID and potential parameters before returning. This is done like with the num, string and bool literals, so the ID's don't clash with any Java keywords. Listing 6.18 shows the visitFcall function.

Listing 6.18. CodeGen - visitFcall function

```
1  @Override
2  public String visitFcall(GrammarParser.FcallContext ctx) {
3      if (ctx.getChildCount() == 4){
4          return "_" + ctx.ID().getText() + "(" + visit(ctx.args()) + ")";
5      }else{
6          return "_" + ctx.ID().getText() + "()";
7      }
8  }
```

The visitIfstmt function uses a StringBuilder to add the **if** followed by any **elseif** needed, at the end the StringBuilder returns as a string. The visitIfstmt function is shown in listing 6.19

Listing 6.19. CodeGen - visitIfstmt function

```
@Override
public String visitIfstmt(GrammarParser.IfstmtContext ctx) {
    StringBuilder buf = new StringBuilder();
    buf.append("if(");
    buf.append(visit(ctx.expr()));
    buf.append(")");
    buf.append(visit(ctx.block(0)));
    for(int i = 0; i< ctx.elseif().size(); i++){
        buf.append(visit(ctx.elseif(i)));
    }
    if (ctx.block().size() > 1){
        buf.append("else");
        buf.append(visit(ctx.block(1)));
    }
    return buf.toString();
}
```

The visitVardcl function shown in listing 6.20, changes the type declared in the program to the equivalent type in Java. String and Bool will be changed to string and boolean, where the type Num will be changed to a double. Another thing worth mentioning is the "_" in front of the IDs, for the same reasons as mentioned before.

Listing 6.20. CodeGen - visitVardcl function

```
@Override
public String visitVardcl(GrammarParser.VardclContext ctx) {
    String result;
    if (ctx.TYPE().getText().equals("Num")){
        result = "double ";
    }else if(ctx.TYPE().getText().equals("Bool")){
        result = "boolean ";
    }else{
        result = "String ";
    }
    if (ctx.getChild(1) instanceof GrammarParser.AssignContext){
        result += visit(ctx.assign());
    }else {
        result += "_" + ctx.ID().getText() + ctx.SEMI().getText() + "\n";
    }
    return result;
}
```

The visitEventdcl function in listing 6.21 is worth noting the added "on" and the added "Event", these are used to create the Robocode eventhandler and the Robocode event. The "e" added after "Event" corresponds to the “e” in visitEcall.

Listing 6.21. CodeGen - visitEventdcl function

```
@Override
public String visitEventdcl(GrammarParser.EventdclContext ctx) {
    String id = StringUtils.capitalize(ctx.ID().getText());
    return "public void on" + id + "( " + id + "Event e )" + visit(ctx.block());
}
```

In the visitFuncdcl function a new public Java function is created with the appropriate return type, name, block and possible parameters. Once again an "_" is added in front of the ID, for the same reasons as before.

Listing 6.22. CodeGen - visitFuncdcl function

```
@Override
public String visitFuncdcl(GrammarParser.FuncdclContext ctx) {
    String type;
    if (ctx.TYPE().getText().equals("Num")){
        type = "double ";
    }else if(ctx.TYPE().getText().equals("Bool")){
        type = "boolean ";
    }else{
        type = "String ";
    }
    if (ctx.getChildCount() == 8) {
        return "public " + type + " _" + ctx.ID().getText() + "(" + visit(ctx.params()) + ")" +
            visit(ctx.functionBlock());
    }else{
        return "public " + type + " _" + ctx.ID().getText() + "()" + visit(ctx.functionBlock());
    }
}
```

The visitSetupblock and visitRepeatblock functions shown in listing 6.23 are a bit special as the Repeat block (is the while(true) statement in the Run function in Robocode) is inside the Setup block (Run function) in Robocode. To do this, Setup block creates the void function and adds the block, then cuts of the last curly bracket adds the Repeat block, which is renamed to while(true) and puts back the curly bracket.

Listing 6.23. CodeGen - visitSetupblock & visitRepeatblock functions

```
@Override
public String visitSetupblock(GrammarParser.SetupblockContext ctx) {
    String result = "public void run()" + visit(ctx.block());
    result = result.substring(0, result.length()-2);
    return result + visit(((GrammarParser.DclsContext) ctx.parent).repeatblock(0)) + "}";
}

@Override
public String visitRepeatblock(GrammarParser.RepeatblockContext ctx) {
    return "while (true)" + visit(ctx.block());
}
```

In the visitProg function a StringBuilder is used to add the import statements needed for Java to be able to compile the program. After that, the class is declared using the Tankname and then extends Robot to the class declaration. Then, the entire program is added followed by a closing curly bracket. The code generation is now complete and a string with the entire program in Java code is returned.

Listing 6.24. CodeGen - visitProg function

```
@Override
public String visitProg(GrammarParser.ProgContext ctx) {
    StringBuilder buf = new StringBuilder();
    buf.append("import robocode.*;\n");
    buf.append("import static robocode.util.Utils.*;\n");
    buf.append("public class ");
    buf.append(StringUtils.capitalize(visit(ctx.dcls().tankname(0))));
    buf.append(" extends Robot {\n ");
    buf.append(visit(ctx.dcls()));
    buf.append("}");
    System.out.print(buf.toString());
    return buf.toString();
}
```

6.5 Compiling source code

To make it easy to compile and test code created in DatLanguage, a simple script has been created. The script works, by taking the file as an argument and passing it to the compiler. The Java code the compiler returns, is then passed to the Java compiler, to create the final working Robocode robot. The robot is then put in a robots folder in the Robocode directory.

When the robot is created, a standard .battle file is created. A battle file contains the information required by Robocode to setup a battlefield, including width, height, number of rounds and participating robots. The Robot created by the compiler, is added to the list of robots that should play on the field, and the battle file is put in the Robocode directory.

Since Robocode can be run from a terminal with arguments specifying which battle to play, the script is able to start up Robocode and play the specified battle.

To run the script on Mac, open Terminal and find the root directory of the compiler, that should also contain the source file to be compiled. Type "./compile_mac" followed by the name of the source file, eg "./compile_mac RamFire", and hit enter.

On windows, the same thing has to be done in PowerShell, just replace "./compile_mac" with "./compile_windows.bat".

By automating the process of compiling and running, it is easier and faster to test the robots.

7 Tests

This chapter is covering the tests that have been done to confirm the functionality of DatLanguage. Two kinds of tests will be used, conversion tests where the language will be used in practice and tests using the JUnit Java library to ensure functionality.

7.1 Conversion tests

In this section tests will be run where sample robots from Robocode will be written in DatLanguage, compiled back to Java and compared with the original robot. The focus of this test is to make sure that the language works in practice. Doing work and documenting any errors found while using the language quickly helped discover some problems which then could be fixed.

7.1.1 Tracker

When writing the robot Tracker in DatLanguage a few errors were found. First an error was received that the Expression types didn't match for an assignment. This was quickly fixed as it turned out the Utils.normalRelativeAngleDegrees had the wrong return type in the ReservedFunctions list.

Another error occurred were the compiler returned the error Variable not found. This was fixed by changing "trackName = null" to "trackName = "0"" throughout the code when a target has not been found. The compiler also gives an error if a robot has been given another name in the code than in the file name.

To test if the converted version of Tracker works the same as the original version both robots has been tested versus the sample robot Crazy. The output log of Robocode shows that the converted version has a problem with an if-statement checking whether the robot hits the target it's looking for. Looking at the scoreboard of the match the converted robot earns 20% less points than the original sample robot. This shows that there has been a flaw in the conversion and that the robots does not share the exact same functionality. There are two causes for this problem. The first cause is that the Robocode command *setAdjustGunForRobotTurn(true)* is not implemented in our language. The second cause is that it's not possible to use a return statement inside an if-statement in our language. The code of the robot can be seen in appendix A.1.

7.1.2 Fire

When converting the Fire robot to DatLanguage there was an error in the compiler saying that the variable was not found. This error occurred because the variable Num dist was

declared in the setup block, it was fixed by moving the declaration outside the block.

When testing the converted Fire robot versus the sample Fire robot by matching both robots against the Crazy sample robot, both versions of the Fire robot behaved in the same way and earned scores close to each other. This shows that the robot has been compiled to have the same functionality after being written in DatLanguage as the original sample robot. The code of the robot can be seen in appendix A.2.

7.1.3 Corners

The same kind of test has also been performed on the robot Corners. The behavior of the converted Corners robot is not exactly the same as the original sample. This is due to the fact that the original robot uses a static variable to decide which corner to move to, DatLanguage doesn't support static variables so it moves to the same corner every time. Considering that static variables could be difficult to understand for a new user, implementing this functionality wouldn't be the best way to solve it. We would rather solve it by creating a functionality that makes it possible to have a variable stay the same through the rounds of a battle.

These tests have gathered some errors with DatLanguage that now can be worked on removing. The next section focuses on the tests made to ensure functionality of the FuncListener and SymbolTypeVisitor classes.

7.2 Functionality tests

To test that the classes FuncListener and SymbolTypeVisitor are working as expected, a series of programs has been written, to test specific things. To help doing this, a unit testing library for Java is used, called JUnit. By using JUnit, the program can be run multiple times with different source codes, to get an overview of which ones are successful and which have failed.

Condition	Expected result	Actual result	Status
Two functions with the same name	Throws error	Throws error	-
Action and function with same name	Compiles	Compiles	-
Addition of number and string	Compiles	Compiles	-
Subtract number and string	Throws error	Throws error	-
Bool AND Bool	Compiles	Compiles	-
Bool AND String	Throws error	Throws error	-
Assign to not declared variable	Throws error	Throws error	-
Assign String to Num	Throws error	Throws error	-
Declare same variable name in different scopes	Compiles	Compiles	
Declare same variable name in global scope and in setup	Compiles	Compiles	-
Declare same variable name in same scope	Throws error	Throws error	-
Call function with wrong parameter types	Throws error	Throws error	-
Call a function with too few parameters given	Throws error	Compiles	Fixed
Call a function with too many parameters	Throws error	Throws error	-
Call a function that hasn't been declared	Throws error	Throws error	-
Wrong event on event handler	Throws error	Throws error	-
If(3)	Throws error	Throws error	-

Table 7.1. Fulfilment of the MoSCoW analysis

What was discovered during the testing, was that supplying too few parameters to a function, would still end up creating the Java file. To fix this, the compiler now always checks if the amount of formal parameters matches the number of actual parameters, and throws an error if there is a mismatch.

Discussion 8

Error handling

In purpose of error handling DatLanguage throws errors whenever an error occurs. Errors will terminate the program and print out the assigned message for the specific error. An other options for error handling would be exceptions, which would make it possible for multiple errors to be reported at the same time. Also the group could have written their own exception handler, which would give more control for error reporting. The ANTLR tool throws exceptions, so a combined exception handler would have been possible.

Dot notation

DatLanguage is an imperative language, but with an object oriented feeling because of the dot notation on Tank, Gun, Radar, Battlefield and Utils. The dot notation is intended to give familiarity to the dot notation in object oriented languages. This might lead to misinterpretations since DatLanguage doesn't implement classes as object oriented languages do.

MoSCoW

In section 3.2 the MoSCoW analysis was created and described. The method was used to prioritize the requirements for the project's language. Table 8.1 shows which of the requirements that have been implemented and which have not been implemented.
All *Must have* requirements has been implemented as they should, since these were the most important and essential requirements for the language. These were needed to make the logic, movement etc., for the robot.
All *Should have* requirements has also been implemented. The Events and Void and type methods helps the users in order to make more intelligent robots, where Comments are useful for the readability of the program, made by the user.
Most of the requirements in *Could have*, has been implemented, which is Print statements, Strings and Setup block. Print statements and Strings have been implemented for the user to be able to debug their code. The Setup block is used for code, that should only be run once each round. Here the user could write code to make some initializing movements e.g. make the robot move to the wall. Cos, Sin & Tan and For loops were not implemented in DatLanguage.
None of the requirements from *Won't have* were implemented in this project.

MoSCoW analysis		
	MoSCoW items	Status
Must have	Primitive types and variables	Implemented
	While loop	Implemented
	Reserved calls	Implemented
	Robot naming	Implemented
	If/Else/Elseif statements	Implemented
	Arithmetic expressions and operators	Implemented
	Logical expressions and operators	Implemented
Should have	Events	Implemented
	Comments	Implemented
	Void and type methods	Implemented
Could have	Cos, Sin & Tan	Not Implemented
	For loops	Not Implemented
	Print statements	Implemented
	Strings	Implemented
	Setup block	Implemented
Won't have	Random number generator	Not Implemented
	Other robot types	Not Implemented

Table 8.1. Fulfilment of the MoSCoW analysis

Conclusion 9

This chapter will make a final conclusion of the project. First the Problem statement will evaluated and then the design criteria.

Problem statement

How can a domain-specific language for Robocode simplify the creation of a robot for high school students, with little or no experience to programming?
To simplify the creation of a robot in Robocode a domain-specific language has been designed, called DatLanguage. DatLanguage simplifies the Java type system and cuts down on language constructs, which lowers the learning curve for new programmers. Along the simplified type system and language constructs, a CLI has been made to facilitate easier compilation and start-up of the Robocode client for the user.

How can the Java type system be simplified?
Only one numeric type is used in DatLanguage, as new programmers might not grasp the concept of memory allocation of different types. As only one numeric type is used, there is no need to convert between types in arithmetic operations. DatLanguage does not support classes, as this is from the object oriented paradigm, which is complex. The imperative paradigm is enforced in DatLanguage, as this is most fitting for learning purposes of new programmers. The type conversion in DatLanguage is very restrictive, as new programmers may not fathom how types would behave after casts. Any type can be converted to a String type, any other conversions are not allowed.

Which constructs are necessary for programming standard robots in Robocode?
DatLanguage implements the constructs described in the discussion and throughout the report, this is deemed enough to program standard robots in Robocode.

How can an easy to use interface be made for the users?
A simple interface for compiling DatLanguage is the two scripts created for Mac and Windows. By typing a simple command in Terminal or PowerShell, all the steps of translating to Java, compiling the Java code and running Robocode, are performed. Without the script, these steps would have to be performed manually every time. This is to make it easy for the programmer to get started on actual coding and to not discourage the user because of an otherwise complex process of going from DatLanguage code, to actually seeing it in action in Robocode.

Design criteria

The main priority of the design criteria in this project was the readability of the language, where the language should be easy to read and understand, which means that the language should be as simple as possible. This is done by having a few sets of constructs, few sets of primitive types and static scoping. To conclude on this criterion, a usability test would be necessary. Without this test, we cannot conclude whether DatLanguage is more readable than Java, as this would be a subjective conclusion on our behalf. We know how the language was constructed and meant to be programmed, but a new, non-experienced user, would give us a better understanding on this topic.
The syntax of DatLanguage has been tried to be simplified by making the keywords look more like a spoken language e.g. the “When” block for the events.

Future work 10

This chapter will describe the thoughts for further development for DatLanguage.

MoSCoW

As earlier stated not all the requirements from the MoSCoW analysis were implemented, in this section these will be described why they would be a good addition to implement for future versions of DatLanguage.

Cos, Sin & Tan
Implementing Cos, Sin & Tan would allow the user to make more advanced calculations in the user's functions and actions, which again will allow the user to make more intelligent robots.

For this to be implemented to DatLanguage, the group believes that the language should extend AdvancedRobots, which it doesn't.

For loops
The for loop is an other way of iterating over data. Implementing the for loop will improve the writability of the language, especially if the users have a little experience with programming in general.

The for loop hasn't been implemented in this project, since the group didn't see any use for it with the standard "Robot" type, where the repeat while loop would be sufficient enough.

Random number generator
A random number generator could e.g. be used to make the robots movement patterns less predicable, which e.g. could be done as in example 10.1:

Listing 10.1. Simple example of a random number generator

```
1        Num MovementNumber = 20;
2        Random = "Random Number generator between 2 - 10"
3
4        Tank.forward(MovementNumber * Random);
```

The random number generator would've been a good implementation for the standard "Robot" type. The random number generator wasn't highly prioritized, because of it's minor functionality for a new programmer.

Other robot types
Implementing the five other robot types, would make it possible for the user to make

different and more advanced robots, but also battle in other game modes with the TeamRobot.

For this project it wasn't necessary to implement other robot types other than the standard "Robot" type. This is due to the fact that the scope of this project is high school students with little or no programming experience. Though it would be a great addition to further develop their programming skills with Robocode, by coding more advanced and intelligent robots.

Wiki-page & IDE

For making the user experience of DatLanguage better and make the language easier to learn, a wiki-page and a IDE could be made. The wiki-page would be used as a site to look up the different functions, actions, learn the syntax and other things about DatLanguage. Having an IDE would help the user write faster code with the intellisense.

Usertests

Conducting usertests of the language could further improve the usability of DatLanguage and lead to discovery of errors that has not been found yet. When doing a test with test-persons who haven't been involved in the creation process you get a new point of view, which gives completely different problems compared to when the creators of the language use it. A view of conducting a test with external users is to do an usability evaluation where a set of test-persons receive tasks to perform using DatLanguage. While solving the tasks, test-persons will tell what they do giving a new insight in the functionality and usability of the language. After the evaluation a prioritized list of problems will be made and this gives a base for continuing development of the language.

Bibliography

Hüttel, **2010**. Hans Hüttel. *Transitions and Trees*. ISBN 978-0-521-14709-5 and 978-0-521-19746-5. Cambridge University Press, 2010.

Larsen, **2013**. Flemming N. Larsen. *ReadMe for Robocode*. `http://robocode.sourceforge.net/docs/ReadMe.html`, 2013. Accessed: 03/03-2016.

Nelson, **2008**. Mathew Nelson. *Corners*. `http://robocode.sourcearchive.com/documentation/1.6.2/Corners_8java-source.html`, 2008. Accessed: 07/04-2016.

Nelson, **2015**. Mathew A. Nelson. *Robocode*. `http://robocode.sourceforge.net`, 2015. Accessed: 03/03-2016.

RoboWiki, **2014**. RoboWiki. *My First Robot*. `http://robowiki.net/wiki/Robocode/My_First_Robot`, 2014. Accessed: 25/02-2016.

RoboWiki, **2013**. RoboWiki. *Robocode*. `http://robowiki.net/wiki/Robocode`, 2013. Accessed: 03/03-2016.

Sebesta, **2009**. Robert W. Sebesta. *Concepts of Programming Languages*. ISBN 978-0-13-139531-2. Pearson, 2009.

Wankel og Blessinger, **2012**. Charles Wankel og Patrick Blessinger. *Increasing Student Engagement and Retention Using Immersive Interfaces: Virtual Worlds, Gaming, and Simulation*. ISBN 978-1-78190-240-0. Emerald, 2012.

Appendix A

A.1 Table of reserved calls

A table of all the reserved calls for DatLanguage/Robocode. The first column is the prefix of the reserved call in the second column. The third column is the equivalent Robocode call.

Reserved calls	DatLanguage	Robocode
Tank.	forward(num distance)	ahead(double distance)
	backward(num distance)	back(double distance)
	doNothing()	doNothing()
	energy()	getEnergy()
	heading()	getHeading()
	height()	getHeight()
	width()	getWidth()
	velocity()	getVelocity()
	xCoord()	getX()
	yCoord	getY()
	stop()	stop()
	resume()	resume()
	turn(num degrees)	turnLeft(double degrees)
Gun.	shoot(num power)	fire(double power)
	coolingRate()	getGunCoolingRate()
	heading()	getGunHeading()
	heat()	getGunHeat()
	turn(num ddegrees)	turnGunLeft(double degrees)
Radar.	heading()	getRadarHeading()
	scan()	scan()
	turn(num degrees)	turnRadarLeft(double degrees)

Battlefield.	height()	getBattleFieldHeight()
	width()	getBattleFieldWidth()
	numOfRounds()	getNumRounds()
	enemies()	getOthers()
	roundNum()	getRoundNum()
	time()	getTime()
Utils.	normalAbsoluteAngle	normalAbsoluteAngle
	normalAbsoluteAngleDegrees	normalAbsoluteAngleDegrees
	normalNearAbsoluteAngle	normalNearAbsoluteAngle
	normalNearAbsoluteAngleDegrees	normalNearAbsoluteAngleDegrees
	normalRelativeAngle	normalRelativeAngle
	normalRelativeAngleDegrees	normalRelativeAngleDegrees

A.2 Table of events

This table contains all the possible events for DatLanguage. The first column is the name of the event, the second column is the event's methods and the third column is the equivalent Robocode methods.

DatLanguage / Robocode Event	DatLanguage Event Information	Robocode Event Information
BulletHit / onBulletHit	bulletHeading()	getBullet().getHeading()
	bulletOwner()	getBullet().getName()
	bulletPower()	getBullet().getPower()
	bulletVelocity()	getBullet().getVelocity()
	bulletVictim()	getBullet().getVictim()
	bulletXCoord()	getBullet().getX()
	bulletYCoord()	getBullet().getY()
	bulletIsActive()	getBullet().isActive()
	energy()	getEnergy()
BulletHitBullet / onBulletHitBullet	bulletHeading()	getBullet().getHeading()
	bulletOwner()	getBullet().getName()
	bulletPower()	getBullet().getPower()
	bulletVelocity()	getBullet().getVelocity()
	bulletVictim()	getBullet().getVictim()
	bulletXCoord()	getBullet().getX()
	bulletYCoord()	getBullet().getY()

	bulletIsActive()	getBullet().isActive()
	enemyBulletHeading()	getHitBullet().getHeading()
	enemyBulletOwner()	getHitBullet().getName()
	enemyBulletPower()	getHitBullet().getPower()
	enemyBulletVelocity()	getHitBullet().getVelocity()
	enemyBulletVictim()	getHitBullet().getVictim()
	enemyBulletXCoord()	getHitBullet().getX()
	enemyBulletYCoord()	getHitBullet().getY()
	enemyBulletIsActive()	getHitBullet().isActive()
HitByBullet / onHitByBullet	bearing()	getBearing()
	bearingDegrees()	getBearingDegrees
	bulletHeading()	getBullet().getHeading()
	bulletOwner()	getBullet().getName()
	bulletPower()	getBullet().getPower()
	bulletVelocity()	getBullet().getVelocity()
	bulletVictim()	getBullet().getVictim()
	bulletXCoord()	getBullet().getX()
	bulletYCoord()	getBullet().getY()
	bulletIsActive()	getBullet().isActive()
	heading()	getHeading()
	headingDegrees()	getHeadingDegrees()
	power()	getPower()
	velocity()	getVelocity()
HitRobot / onHitRobot	bearing()	getBearing()
	bearingDegrees()	getBearingDegrees
	energy()	getEnergy()
	myFault()	isMyFault()
	enemyName()	getName()
Death / onDeath		
HitWall / onHitWall	bearing()	getBearing()
EnemyDeath / onRobotDeath	enemyName()	getName()
RoundEnded / onRoundEnded	round()	getRound()
	totalTurns()	getTotalTurns()
	turns()	getTurns()

ScannedRobot / onScannedRobot	bearing()	getBearing()
	distance()	getDistance()
	energy()	getEnergy()
	heading()	getHeading()
	enemyName()	getName()
	velocity()	getVelocity()
Status / onStatus	distanceRemaining()	getStatus.getDistanceRemaining()
	gunTurnRemaining()	getStatus.getGunTurnRemaining()
	radarTurnRemaining()	getStatus.getRadarTurnRemaining()
Win / onWin		

A.3 Robots

A.3.1 Tracker

Listing A.1. The robot Tracker written in our language

```
1  Tankname tracker;
2
3      Num gunTurnAmt;
4      String trackName;
5      Num count = 0;
6
7  Setup
8  {
9  trackName = "0";
10 gunTurnAmt = 10;
11 }
12
13 Repeat
14 {
15     Gun.turn(gunTurnAmt);
16
17     count = count +1;
18
19     if (count > 2)
20     {
21       gunTurnAmt = -10;
22     }
23
24     if (count > 5)
25     {
26       gunTurnAmt = 10;
27     }
28
29     if (count > 11)
30     {
31       trackName = "0";
32     }
33
34 }
35
36 When scannedRobot {
37     if ((trackName NOT= "0") AND (Event.enemyName() NOT= trackName)) {
38
39     }
40
41     if (trackName IS= "0") {
42     trackName = Event.enemyName();
```

```
43      print("Tracking" + trackName);
44      }
45
46      count = 0;
47
48      if (Event.distance() > 150)
49      {
50        gunTurnAmt = Utils.normalRelativeAngleDegrees(Event.bearing() + (Tank.heading() -
              Radar.heading()));
51        Gun.turn(gunTurnAmt);
52        Tank.turn(Event.bearing());
53        Tank.forward(Event.distance() - 140);
54      }
55
56      gunTurnAmt = Utils.normalRelativeAngleDegrees(Event.bearing() + (Tank.heading() -
            Radar.heading()));
57      Gun.turn(gunTurnAmt);
58      Gun.shoot(3);
59
60      if (Event.distance() < 100) {
61          if ((Event.bearing() > -90) AND (Event.bearing() <= 90))
62          {
63            Tank.backward(40);
64          }
65            else {Tank.forward(40);
66          }
67      }
68      Radar.scan();
69  }
70
71  When hitRobot {
72      if ((trackName NOT= "0") AND (trackName NOT= Event.enemyName()))
73      {
74        print("Tracking" + Event.enemyName() + " due to collision");
75      }
76
77      trackName = Event.enemyName();
78
79      gunTurnAmt = Utils.normalRelativeAngleDegrees(Event.bearing() + (Tank.heading() -
            Radar.heading()));
80      Gun.turn(gunTurnAmt);
81      Gun.shoot(3);
82      Tank.backward(50);
83  }
```

A.3.2 Fire

Listing A.2. The robot Fire written in our language.

```
1   Tankname fire;
2
3   Num dist = 50;
4
5   Setup {
6
7   }
8
9   Repeat {
10      Gun.turn(5);
11  }
12
13  When scannedRobot {
14      if (Event.distance() < 50 AND Tank.energy() > 50) {
15          Gun.shoot(3);
16      }
17      else {
```

```
18          Gun.shoot(1);
19      }
20      Radar.scan();
21  }
22
23  When hitByBullet {
24      Tank.turn(Utils.normalRelativeAngleDegrees(90 - (Tank.heading() - Event.heading())));
25                  Tank.forward(dist);
26                  dist = dist * (-1);
27                  Radar.scan();
28  }
29
30  When hitRobot {
31      Num turnGunAmt = Utils.normalRelativeAngleDegrees(Event.bearing() + Tank.heading() -
            Gun.heading());
32      Gun.turn(turnGunAmt);
33      Gun.shoot(3);
34  }
```