Energirenovering

P4 Project
Group sw408f16
Software
Aalborg Universitet
Den 26. maj 2016

AALBORG UNIVERSITET
STUDENTERRAPPORT

Fourth semester at
DEPARTMENT OF COMPUTER SCIENCE
Software
Selma Lagerlöfs Vej 300
9220 Aalborg East, DK
http://www.cs.aau.dk/

Titel:

Not known yet.

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: X
Pagecount: 27
Appendix range: X
Published 26-05-2016

Synopsis:

Synopsis

Preface

Insert preface here..

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

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, 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 due to the fact that this requires one to know an object oriented language in advance.

This is a problem, since Robocode could be a potentially great way of introducing these students for programming languages. If there was a domain-specific language, with more intuitie type systems and good writeability, 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.

Analysis 2

The analysis chapter is of purpose to create a basis for the 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. Contributors 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]

For meget intro til Robocode?

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, Updated 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 robot, when the game is downloaded, in order to give the users/players 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. http://robowiki.net/wiki/Robocode/My_First_Robot

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, some of these sample robots will be examined, and the general setup and main events or methods will be presented.

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

Listing 2.1. Eksampel of the main loop in Robocode label

```
public void run() {
        while(true) {
                //Robots behaviour
        }
}
```

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. Eksampel of the onScannedRobot event from Robocode label

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

The robot's radar will spot enemies when they get within the vision 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. Eksampel of the onHitByBullet event from Robocode label

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

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. Eksampel of the onHitWall event from Robocode label

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

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 the robot class, 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 the system contains.

2.1.1 Robot

The robot is the core of the game. The robot can be coded to act differently in various of 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 matches the event that occurred. The robots also have a gun. The gun is used to damage antagonist robots, in the arena. The robot has a gun, and the possibility to choose different types of bullets. For example, one play could use small and fast bullets, they won't necessarily hurt very much, but it allows the player to shoot more frequent, compared to larger slower bullets.

The robot also has a radar. The radar allows the robot to scan for other robots scan 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 fight each other. It's also the visible field on screen when the game is running. When coding, the battlefield can be used for different reasons, for example, to get the number of enemies that are alive. A robot might be programmed to act differently if there areess than three robots left. All this depends on the way the player decided to program his robot. Some of the other examples that the battlefield can be queriedor could be the field size, time or the current round number.

2.1.3 Energy

Robots have energy, which is the spendable resource when shooting bullets, it is the 'health' of a tank, since being hit by a wall or another robots bullet causes a robot to lose energy. But if one robot hits another one, it regains energy. All robots start at 100 energy at the start of a fight, 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. By 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 maneuvering 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 following:

- 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 een 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 had a build-in lexical analyzer, and a plugin for Eclipse/IntelliJ, generating abstract syntax trees while writing the grammar. Other parser generators were discussed before making a final decision, such as CUP, but the lack of lexical analyzers 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 was a single plugin the user had 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 has 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 analyzer. 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, language correct (syntactically correct) sentences (source code) in a computer language. These trees have a top representing the program, and nodes representing terminals and non-terminals. The roots 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, where 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 interesting one 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 rightclick 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 rightclick and choose to test the grammar.

Language design 3

In this chapter the design decisions made during the process of creating the language 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 priortized. The MoSCoW analysis concerns what the project *Must have*, *Should have*, *Could have* and *Should 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 languages, but this would be needed to code robots in Java. This would be concealed from the user with this reports language, 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 this language being constructed for beginners, only a few primitive types and very few constructs will be implemented.

3.1.2 Writability

To have a higher level of writeability, the language will be a DSL for Robocode. The genereal purpose of a DSL language is to be able to make solutions for a specific problem. As mentioned in the section above, the language should have a high level of orthogonality, which will also help on the writability of the language.

3.1.3 Reliability

To make the programming language reliable to use, a lot of focus has been put into making the language as write- and readable as possible, and design the language in a way that helps the user create code without errors. By making programs written in the language 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, 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. The language also does not have 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 can 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 our language.

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 (assignment) While loop Reserved calls Robot naming If/Else/Elseif statements Arithmetic expressions and operators Logical expressions and operators
Should have	Events Void and type methods
Could have	Cos, Sin & Tan For loops Arrays Strings Print statements Comments Setup block
Won'tve	Random number generator Other robot types Other RoboCode gamemodes

Table 3.1. Outcome of the MoSCoW analysis

Language Description 4

In this chapter we will overview some features of our language by means of a simple example. Then we will provide formal description of its syntax.

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. Eksampel of the sample robot "Corners" in our language

```
Tankname corners;
Num others;
Num corner = 0;
Bool stopWhenSeeRobot = False;

Setup{
  others = Battlefield.enemies();
  run goCorner();
  Num gunIncrement = 3;
}

Repeat{
  Num i = 0;

  repeat while(i < 30){
    Gun.turn(gunIncrement * -1);
    i = i+1;
  }
  gunIncrement = gunIncrement * -1;
}

Action goCorner(){
  stopWhenSeeRobot = false;
  Tank.turn(Math.normalRelativeAngleDegrees(corner - Tank.heading()));
  stopWhenSeeRobot = true;
  Tank.forward(5000);
  Tank.turn(-90);
  Tank.forward(5000);
  Gun.turn(-90);
}

When scannedRobot{
  if(stopWhenSeeRobot){
    Tank.stop();
    run smartFire(Event.distance());
    Radar.scan();
    Tank.resume();
  }else{
    run smartFire(Event.distance());
  }
}
```

```
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 - getOthers()) / 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, CornersNelson [Unknown], will as find a specific corner where to 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 see if it did well or poorly it 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 there 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

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

To make the robot do repetitive actions during the battles, Robot, 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 Robot, 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 have been turned a total of 90 or -90 degrees.

Listing 4.3. Code listing of the Repeat block

```
Repeat{
  Num i = 0;

  repeat while(i < 30){
    Gun.turn(gunIncrement * -1);
```

```
    i = i+1;
  }
  gunIncrement = gunIncrement * -1;
}
```

In our language an Action is compared to the langauge C, a procedure in 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 robots heading. "Heading" is getting the robot's heading, which is the direction the robot 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/-90 degrees, which will be from one wall to the other it will turn its gun.

Listing 4.4. Code listing of the Action goCorner()

```
1 Action goCorner(){
2   stopWhenSeeRobot = false;
3   Tank.turn(Math.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 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 shoot will be determined by the distance between the robot and the scannedRobot. The robot will then use Radar.scan() and resume it's movement towards the corner. If the bool was false, the robot is already in the corner, and will shoot at 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 running the Repeat block.

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

```
When scannedRobot{
  if(stopWhenSeeRobot){
    Tank.stop();
    run smartFire(Event.distance());
    Radar.scan();
    Tank.resume();
  }else{
    run smartFire(Event.distance());
  }
}
```

Consider if the event Death should be explanied here for the example Corners.

4.2 Grammar

In this section we present the syntax of the language. The syntax is formally presented by means of a context-free grammar in extended BNF.

```
grammar Grammar;

prog : dcls EOF;

setupblock : 'Setup' block;

repeatblock : 'Repeat' block;

dcls : (actdcl | funcdcl | vardcl';' | setupblock | repeatblock | 'Tankname' ID
    ';' | event | print';')* ;

actdcl : 'Action' ID '(' params? ')'block;

funcdcl : 'Function' ID '(' params? ')' 'returns' TYPE block;

params : param (',' param)*;

param : TYPE ID;

event : 'When' ID block;

block : '{' stmts '}';

stmts : (assign';'|vardcl';'|ifstmt|whilestmt|returnstmt';'|call';'|print';')*;

assign : ID '=' expr;

vardcl : TYPE (ID|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')';

call : acall | fcall | rcall | ecall;

acall : 'run' ID'('args?')';

fcall : ID'('args?')';

rcall : 'Tank.'ID'('args?')' | 'Gun.'ID'('args?')'
      | 'Radar.'ID'('args?')' | 'Battlefield.'ID'('args?')'
      | 'Math.'ID'('args?')';
```

```

ecall : 'Event.'ID'('args?')';

args : expr (',' expr)*;

expr : orexpr ;

orexpr : andexpr (OR andexpr)*;

andexpr : eqexpr (AND eqexpr)*;

eqexpr : relexpr (EQ relexpr)*;

relexpr : addexpr (REL addexpr)*;

addexpr : mulexpr (ADD mulexpr)*;

mulexpr : unexpr (MUL unexpr)*;

unexpr : 'NOT'? atomic;

atomic : '(' expr ')' | ID | NUM | STRING | call | BOOL ;

ID : [_a-z] [_a-zA-Z]* ;
OR : 'OR';
AND : 'AND';
EQ : 'IS='|'NOT=';
REL : '>'|'<'|'>='|'<=';
ADD : '+'|'-';
MUL : '*'|'/';
NUM : '-'?[0-9]+('.'[0-9]+)?;
BOOL : 'false' | 'true';
STRING : '"'.*'"';
TYPE : 'Num'|'Bool'|'String';

Expr precedence:

HIGHEST

'NOT' atomic
* | /
+|-
> | < | >= | <=
IS= | NOT=
AND
OR

LOWEST



COMMENT : '/*'.*'*/' -> skip;
```

```
SPACE : [ \t\n] -> skip;
```

4.2.1 Lexicon

The definition of what input are allowed for each lexical in the grammar is defined by regular expressions. This will be described here with a table of terminals with matching regex. A stream of characters is read by the scanner of the compiler and then turned into a lexical 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.

<table>
<tr><th>Terminal</th><th>Regular expressions</th></tr>
<tr><td>ID</td><td>[a-z] ([a-z] | [A-Z])*</td></tr>
<tr><td>TYPE</td><td>Num | Bool | Text</td></tr>
<tr><td>Num</td><td>[0-9]+("."[0-9]*)?|"."[0-9]+</td></tr>
<tr><td>Bool</td><td>false | true</td></tr>
</table>

Table 4.1. Table with terminals and matching regular expressions.

Conclusion 5

Discussion 6

7 Future work

Bibliography

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

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

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

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.

Rettelser

Appendiks name A

Reserved calls	Our language	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)
	adjustGunForRobotTurn(boolean)	adjustGunForRobotTurn(boolean independant)
Radar.	heading()	getRadarHeading()
	scan()	scan()
	adjustRadarForGunTurn(boolean)	adjustRadarForGunTurn(boolean independant)
	adjustRadarForRobotTurn(boolean)	adjustRadarForRobotTurn(boolean independant)
	turn(num degrees)	turnRadarLeft(double degrees)
Battlefield.	height()	getBattleFieldHeight()
	width()	getBattleFieldWidth()
	numOfRounds()	getNumRounds()
	enemies()	getOthers()
	roundNume()	getRoundNum()
	time()	getTime()

OL / Robocode Event	OL Event Information	Robocode Event Information
BulletHit / onBulletHit	bullet()	getBullet()
	energy()	getEnergy()
BulletHitBullet / onBulletHitBullet	bullet()	getBullet()
HitByBullet / onHitByBullet	bearing()	getBearing()
	bearingDegrees()	getBearingDegrees
	bullet()	getBullet()
	heading()	getHeading()
	headingDegrees()	getHeadingDegrees()
	power()	getPower()
	velocity()	getVelocity()
HitRobot / onHitRobot	bearing()	getBearing()
	bearingDegrees()	getBearingDegrees
	energy()	getEnergy()
	myFault()	isMyFault()
Death / onDeath	time()	getTime()
HitWall / onHitWall	bearing()	getBearing()
	bearingDegrees()	getBearingDegrees()
EnemyDeath / onRobotDeath	time()	getTime()
RoundEnded / onRoundEnded	round()	getRound()
	totalTurns()	getTotalTurns()
	turns()	getTurns()
ScannedRobot / onScannedRobot	bearing()	getBearing()
	distance()	getDistance()
	energy()	getEnergy()
	heading()	getHeading()
	velocity()	getVelocity()
Status / onStatus	status()	getStatus()
Win / onWin	time()	getTime()