****

**CS 315**

**Programming Languages**

**Project 2**

**Language Name : ZGRobot**

**Zeynep Ayça Çam**

**21502269**

**Section 1**

**Gökçe Sefa**

**21400596**

**Section 3**

**Zeynep Sonkaya**

**21501981**

**Section 3**

# 

[**Introduction**](#_eejkj9wvc3zg) **3**

[**Explanation for Each Language Construct**](#_sqhge17srdgr) **6**

[Program <program>](#_ufqxm99zicsl) 6

[Block Statements <blockStatement>](#_449gk39yxxei) 6

[Statements <statements>](#_vp3ojqba91ec) 6

[Non-In Case Statement<nonIncaseStatement>](#_kh0jt268jkyc) 7

[Input Statement <input\_statement>](#_4kkuqzf9d6ji) 7

[Return Statement <return\_statement>](#_9kyuviuod6im) 7

[Assignment Statement <assign\_statement>](#_9avl7bi7q6yn) 8

[Declarations <declarations>](#_na6zg8nftnue) 8

[Unmatched Incase <unmatched>](#_evngo5e3h4yt) 8

[Matched Incase <matched>](#_n63dr7ac5fel) 8

[Iter Loop<iter\_statement>](#_plnl1cyx5gdr) 8

[Generic For Function Calls <otherFunctionCalls>](#_rni9k43opc1) 12

[Parameters of a Function <param>](#_4sfi4n4abot5) 12

[Helper Rule of Function Declaration Parameter<loo>](#_v0c34xbkqv1q) 12

[Helper Rule of Function Call Parameter <kal>](#_3rvbqphe6usa) 12

[Say Something Function Call<sayCall>](#_4jbirfaekgn5) 12

[Move Function Call<moveCall>](#_zdq4fndwdz9c) 13

[Turn Function Call<turnCall>](#_98cbop8q05xw) 13

[Grab Function Call <grabCall>](#_fathqq5qznka) 13

[Release Function Call<releaseCall>](#_npahtrr33d08) 13

[Read Data Function Call<readDataCall>](#_y463yzvtzbs6) 13

[Sensor Id of a Robot <sensorID>](#_7misn7mbp999) 13

[Send Data Function Call<sendDataCall>](#_7hxur3c7mpha) 13

[Receive Data Function Call <receiveDataCall>](#_mujuotxo6gwb) 14

[Constant Values <constant>](#_y0euloutzs49) 14

[**General Description of the Structure of Our Language**](#_4vgtc6j03uxp) **14**

[**Explanation of Conflicts**](#_xbq10x2h1jpx) **14**

[**Descriptions of Nontrivial Tokens**](#_2lu1f8z6w9ib) **15**

[Comments](#_q9hhcadk2qqk) 15

[Identifiers](#_cy83sfiztmnp) 15

[Literals](#_or2jhxvuf5pu) 15

[Reserved words](#_or2jhxvuf5pu) 15

# **Introduction**

In this report, we will introduce our programming language ZGRobot, a proper language to be used by Robots and Masters. Its full design in BNF is given. Furthermore, each language constructs and nontrivial tokens in ZGRobot is explained.

**Complete BNF Of Our Language,ZGRobot**

<program> ::= <statements>

| <empty>

<blockStatement> ::= start <statements> done

<statements> ::= <statement>

| <statements> <statement>

| <statements> <singleComment>

<statement> ::= <matched>

| <unmatched>

<nonIncaseStatement> ::= <iter\_statement>

| <while\_statement>

| <assign\_statement>

| <function\_call>

| <declarations>

| <return\_statement>

<return\_statement>::= return <expression>

<assign\_statement> ::= <variable> <assignment\_op> <expression>

<declarations> ::= <constant\_declaration>

| <variable\_declaration>

| <function\_declaration>

<unmatched> ::= inCase <LP> <logic\_express> <RP><blockStatement>

| inCase <LP> <logic\_express> <RP><matched>notTheCase <unmatched>

<matched> ::= inCase <LP> <logic\_express> <RP> <matched> notTheCase <matched>

| <nonIncaseStatement> <endStatement>

<iter\_statement> ::= iter <LP> <assign\_statement> <endStatement>

<logic\_express> <endStatement> <assign\_statement> <RP> <blockStatement>

<while\_statement> ::= while <LP> <logic\_express><RP><blockStatement>

<expression> ::= <nonLogicExpression>

| <logic\_express>

<nonLogicExpression> ::= <math\_expression>

| <function\_call>

<math\_expression>::=<math\_expression><math\_op\_post><math\_expressionHigher>

| <math\_expressionHigher>

<math\_expressionHigher>::=<math\_expressionHigher><math\_op\_pre><expressionable>

| <expressionable>

<expressionable> ::= <constant>

| <variable>

<math\_op\_pre> ::= <mult\_op>

| <div\_op>

| <mod\_op>

<math\_op\_post> ::= <add\_op>

| <sub\_op>

<function\_declaration>::= <main\_function\_declarations>

| <function\_declarations>

<function\_declarations>::= <funcHead> <funcBody>

<main\_function\_declarations>::= main <LP> <RP> <funcBody>

<funcHead> ::= fun <variable> <LP> <parameterList> <RP>

<parameterList> ::= <type> <variable>

| <type> <variable> , <loo>

| <empty>

<loo> ::= <type> <variable> , <loo>

| <type> <variable>

<funcBody> ::= <blockStatement>

<constant\_declaration> ::= const <type> <assign\_statement>

<variable\_declaration> ::= <type> <variable>

| <type> <assign\_statement>

<logic\_express> ::= <logic\_express><logical\_op><nonLogicExpression>

|<nonLogicExpression><logical\_op><nonLogicExpression>

<logical\_op> ::= <equal>

| <notEqual>

| <biggerOrEqual>

| <smallerOrEqual>

| <smaller>

| <bigger>

| <and>

| <or>

<function\_call> ::= <moveCall>

| <sayCall>

| <turnCall>

| <grabCall>

| <releaseCall>

| <readDataCall>

| <sendDataCall>

| <receiveDataCall>

| <otherFunctionCalls>

| <inputCall>

<inputCall> ::= input <LP> <RP>

<otherFunctionCalls>::= <variable> <LP> <param> <RP>

<param> ::= <expression>

| <expression> , <kal>

| <empty>

<kal> ::= <expression> , <kal>

| <expression>

<sayCall> ::= say <LP> <string> <RP>

<moveCall> ::= move <LP> <RP>

<turnCall> ::= turn <LP> <string> <RP>

<grabCall> ::= grab <LP> <RP>

<releaseCall> ::= release <LP> <RP>

<readDataCall> ::= readData <LP> <sensorID> <RP>

<sendDataCall> ::= sendData <LP> <RP>

<receiveDataCall> ::= receiveData <LP> <RP>

<constant> ::= <string>

| <integer>

| <floatingPointNumber>

| <logic>

# **Explanation for Each Language Construct**

## **Program <program>**

A Program is a non-terminal that contains statements. It can be formed as <statements> or <empty> that contains empty statements or filled statement. Thus by this, we enabled the user to have an empty program. This program is an abstraction of the main program flow. Its components will be taken in order. According to ZGRobot conventions, program can start with any statement without any indicated reserved word.

## **Block Statements <blockStatement>**

Block Statements are used when an Iter statement,While Statement or Incase Statement is used. They are generally groups of statements which perform different actions. However a block statement does not have to contain a statement. An empty Block Statement is also allowed with only “start” and “done” reserved words. According to the convention a block statements must begin with “start” reserved word and end with “done” reserved word. Number of statements inside the block statement is up to the user. As long as it contains necessary reserved word, statements between “start” and “end” can be various type of statements.

## **Statements <statements>**

Statements, which are included by Block Statements, are lined up set of <statement>‘s in the language. Also, Statements may be ended with the <singleComment> which is the explanation about the line or statements from the users. <singleComment> is a comment feature does not any influence in the language. It is used for understanding for another people who does not write the code in advanced. Statement used in the general flow of the program. They can be used inside Block Statements or in the Program. They do not need a beginning and ending keyword.

**Statement <statement>**

Statement is the main working mechanism of the program. They are used for simple but primitive functions of any programming language. They are separated to two kinds for solving problems related to incase Statements. So they can be <matched> or <unmatched>. In matched there existed a rule that they might be non-incase statements. So if a statement is not incase statement it goes to matched one and there to <nonIncaseStatement>. Incase means in case of an logical expression’s truth value is 1 ,perform a statement. This distinction is used for differentiate content of Block Statements in statements. According to the convention of ZGRobot, every statement must end with “!” other than incase statement itself thus at the end of an incaseStatement there are no “!”. However every nonincase statement inside a incaseStatement has end “!” at the end.

## **Non-In Case Statement<nonIncaseStatement>**

This non-terminal, non-in case statement is for performing primitive functions of a programing language other than “if”(incase).Those primitive functions are loops, function calls, declarations,return statements and assignment statements. An non-in case statement can be <iter\_statement>, <while\_statement>, <assign\_statement>, <function\_call> ,<declarations> or <return\_statement>. As any statements in ZGRobot those statements ends with “!”. This rule of the language is handled in different components of the language.

## **Input Statement <input\_statement>**

This non-terminal, input statement is for performing reception of input from the user. It is used to getting value from the user for any variable. According to the convention, variables which used in this statement should be declared before. After this statement is used in the program, the program waits for an input from the user. As any statements in ZGRobot those statements ends with “!”.

## **Return Statement <return\_statement>**

This non-terminal <return\_statement> is constructed of a “return” terminal and an non-terminal called <expression>. As the name indicates return statement is a statement therefore it is used with an <endStatement> (“!”). Return statements are used in functions for giving a value to the user after a function called. According to the convention if a function does not return a thing it should not write a return statement. “return!” is not allowed. Thus if a function does not return a valid type,it shouldn’t have a return statement.

## **Assignment Statement <assign\_statement>**

An assignment statement is used to assign a value to a variable (identifier). A variable holds values of different types. Holded value by a variable can be changed by assignment statement. Replaced value is given by an <expression>. Expression can differ but all has a value. In an <assign\_statement> assignment is performed by taking the <expression> from the right side of <assignment\_op> and give it to the <variable>. At the end of a statement there must be “!”.

## **Declarations <declarations>**

Declarations can be performed as constant, variable and function declaration in ZGRobot. Declarations are common name of them. All are used for declare a new type of themselves. Declarations are also <statement> that end with “!” in ZGRobot convention.

## **Unmatched Incase <unmatched>**

Unmatched Incase means there is a incase statement without a notTheCase or a incase statement with an incase that includes a <matched> and a notTheCase with a <unmatched>. Usage of this component let to proper matching of inCase and notTheCase reserved words.

<unmatched> non-terminal contains terminals with reserved words, inCase and notTheCase, terminals.

## **Matched Incase <matched>**

Matched Incase means there is a incase statement with only a <nonIncaseStatement> and <endStatement> or a incase statement with an incase that includes a <matched> and a notTheCase with a <matched>. Recursion lets to proper matching of inCase and notTheCase reserved words. <matched> non-terminal contains terminals with reserved words, inCase and notTheCase, terminals.

## **Iter Loop<iter\_statement>**

As name indicates iter loop performs iterations. It is used to loop and repeat performing a sequence of statements. It starts with terminal “iter” and parantehesis that holds <assign\_statement> <endStatement> <logic\_express> <endStatement> <assign\_statement>. First Assignment Statement is written to perform once at the beginning of the loop and followed by a <endStatement> .Logic expression is tested after every iteration to getting out of the loop. Second Assignment Statement is written to perform after every loop and followed by a <endStatement> according to the convention.

**While loop <while\_statement>**

This method is nonterminal illustrated loop statement in the program, method executes the <block\_statement> until reach logical expression which is specified as <logical\_expression>. In grammatically, logical expression needs parenthesis and block statement needs reserved word of start and done.

**Expression <expression>**

Expression is a combination of <nonLogicExpression> and <logic\_express> in the program. In this two parts have unique precedences in their inside. In this part especially logical expression <logic\_express> is divided because it has more complicated parts than other non-logical statements.

**Non-Logical Expression <nonLogicExpression>**

Non-Logical Expression is a the mathematical expressions <math\_expression> or calling functions <function\_call> in this program. In the expressions, there is determined rules between the precedences and expression values.

**Mathematic Expressions <math\_expression>**

This nonterminal term is used for arithmetic calculations for only addition and subtraction. In grammatically using two mathematical expressions, expressionables and higher mathematical expressions combined with the <math\_op\_post> due to priority of algebra.

**Mathematical Expressions for Higher <math\_expressionHigher>**

This nonterminal term is used for arithmetic calculations for multiplication, division and mode . In grammatically two expressionables or expressionable and higher mathematical expression are combined with the <math\_op\_pre> due to priority of the algebra.

**Expressionable <expressionable>**

Nonterminal term is a expressionable is combined of the constants <constant> or variables <variable>. This is used like a template for the mathematical expression for the abbreviation.

**Mathematical Operation for Precedence <math\_op\_pre>**

The nonterminal term of <math\_op\_pre> determined that there is a strict rule between the arithmetical operations. In the program, multiplication <mult\_op>, division<div\_op> and mode<mod\_op> has priority while calculating arithmetical states.

**Mathematical Operation for Post <math\_op\_post>**

A nonterminal term states that there is a priority before the < math\_op\_post>, In the unique language of math, there is an algebraic rule between the operations so, addition <add\_op> and subtraction <sub\_op> is calculated after the <math\_op\_pre> .

**Function Declaration <function\_declaration>**

<Function\_decleration> is enlarged from the <function\_decleration> with the using <main\_function\_decleration> because the discrimination of the only function and main function should be different in the program.

**Function Declaration <function\_declarations>**

This nonterminal term is combined by <funcHead> and <funcBody>. First part of function declaration is the function, it’s name and it’s parameter list to call function. Second part is a function body which is the block statements of the program.

**Main Function Declaration <main\_function\_declarations>**

The basis function in the program for working principle.The main function does not contain parameter list since it does not take any parameter. It can be declared by the user but cannot be called by the user. It is called by the program when execution starts. According to conventions, the user should declare just one main function. The main function also has their parenthesis as <LP> and <RP> to become separable in the program.

**Function Header <funcHead>**

The nonterminal <funcHead> symbolized the function methods’ names and parameters. There should be “fun” is written to declare that it is function method or not. This header must have parameters to getting parameter variables from input, user or constantly written. Function header occurs as a variable then with the parenthesis <LP> and <RP>, <parameterList> put at the last part of the <funcHead>.

**Function Body <funcBody>**

Function body nonterminal word in the program occurs directly block statements which is constituted by statements.

**Parameter List <parameterList>**

This nonterminal term is used for get and declare the parameters when the calling the functions. This list is constituted by the mainly <type> and <variable>. Also it may be empty or includes another <loo> which includes also variable and type in the program. Type is crucial to determine the variable features.

**Constant Declaration<constant\_declaration>**

Constant <constant\_declaration> is a special word to use in everywhere without no changing. If the type is specified by constant, these type value never change. Constant declaration is constituted by the <type> and <assign\_parameter>.

**Variable Declaration <variable\_declaration>**

This nonterminal is provide initialization for variables. Variables can be declared as only type also, with the assign statements.

**Logic Expression <logic\_express>**

This nonterminal is an another crucial piece of the program because it helps to determine decisions on the certain conditions. Combining the result of two conditions, or logical operators utilize the program in producing the expression . Logic express may be constituted by <logic\_express> <logical\_op> <nonLogicExpression> OR <nonLogicExpression> <logical\_op> <nonLogicExpression> to make more complicated procedures in the program .

**Logical Operation <logical\_op>**

<logical\_op> is an another nonterminal and named as check state and conditions. There are various logical operations could be used in the logic expression, for equality <equal> ==, for inequality <notEqual> =/=, to check bigger or equality and smaler or equality <biggerOrEqual> >=, <smallerOrEqual> <=, to check smaller or bigger<smaller> <, <bigger> >.

**Function Call <function\_call>**

This nonterminal is used for calling the functions that are defined by the program itself. These functions already declared and user have to call them by their name and appropriate parameters(<parameterList>) in the left and right parentheses. It also needs <endstatement> at the end. By this statement user can call functions such as: sayCall , moveCall , turnCall , grabCall , releaseCall , readDataCall , sendDataCall ,inputCall and receiveDataCall .

## **Generic For Function Calls <otherFunctionCalls>**

This nonterminal is used for calling functions that are defined by the user. These functions have to declared by the user before called, as stated in the <function\_declaration>. After declaration, the user can call these functions by their declared name (in our case variable) and their parameters(<param>) in the left and right parentheses. It also needs <endstatement> at the end.

## **Parameters of a Function <param>**

This nonterminal is used for creating a parameter list for other functions which are created by the user. It can be empty, have one or more expression. This expressions present parameters of functions. If there is more than one expression, they must be separated by a comma.

## **Helper Rule of Function Declaration Parameter<loo>**

This nonterminal term is used for the view and aesthetic, when taking parameters in the function declaration header. Before we used loo , parameter list could end with “,”. In order to prohibit this situation <loo> is declared. It is constituted of type and variables in the program.

## **Helper Rule of Function Call Parameter <kal>**

This nonterminal term is a helper to dispose of “,” at the end of the each parameter because this program also provide an empty space in <param>. In this <kal> is used for the layout view of the parameters between the expressions.

**Input Call <inputCall>**

Nonterminal term of the <inputCall> is composed of the reserved word of input and also parenthesis as <LP> and <RP> for easy separation. Input is getting from the users to use in the function calling as <function\_call> in the program.

## **Say Something Function Call<sayCall>**

This nonterminal is present one of the predefined functions. This function is used for printing any string on the screen. It gets this string from parameter. Then this function output a text. It must be called with “say” word. The parameter must be specified in the left and right parentheses. It also needs <endstatement> at the end. This is our output statement.

## **Move Function Call<moveCall>**

This nonterminal is present one of the predefined functions. This function is used for moving robot ahead in steps of 1 mm. It doesn’t get any parameter. It must be called with “move” word. It must need left and right parentheses. It also needs <endstatement> at the end.

## **Turn Function Call<turnCall>**

This nonterminal is present one of the predefined functions. This function is used for turning the robot. It gets a string as a parameter. This string must be “left” or “right”. Then this function turns the robot specified way to 1 degree. It must be called with “turn” word. The parameter must be specified in the left and right parentheses. It also needs <endstatement> at the end.

## **Grab Function Call <grabCall>**

This nonterminal is present one of the predefined functions. This function gives a command to the robot to grab an object. It doesn’t get any parameter. It must be called with “grab” word. It must need left and right parentheses. It also needs <endstatement> at the end.

## **Release Function Call<releaseCall>**

This nonterminal is present one of the predefined functions. This function gives a command to the robot to release an object. It doesn’t get any parameter. It must be called with “release” word. It must need left and right parentheses. It also needs <endstatement> at the end.

## **Read Data Function Call<readDataCall>**

This nonterminal is present one of the predefined functions. This function gives a command to the robot to read data from a sensor. It gets sensorID as a parameter. Then this function read data from a sensor given the sensor ID. It must be called with “readData” word. The parameter must be specified in the left and right parentheses. It also needs <endstatement> at the end.

## **Sensor Id of a Robot <sensorID>**

This nonterminal is used for readData function. By this variable, the robot can determine to use which sensor. It must start with words ID and follow by <unsignedInt>.

## **Send Data Function Call<sendDataCall>**

This nonterminal is present one of the predefined functions. This function gives a command to the robot to send data to the master. It doesn’t get any parameter. It must be called with “sendData” word. It must need left and right parentheses. It also needs <endStatement> at the end.

## **Receive Data Function Call <receiveDataCall>**

This nonterminal is represent one of the predefined functions. This function gives a command to the robot to receive data from the master. It doesn’t get any parameter. It must be called with “receiveData” word. It must need left and right parentheses. It also needs <endstatement> at the end.

## **Constant Values <constant>**

This nonterminal what is defines as constant. They are <string>, <integer>, <floatingPointNumber> and <logic>.

# **General Description of the Structure of Our Language**

In general we have used left recursion thus our language is left associative, it can be seen in statements. Also we have right recurtions for parameters. We have to create sub non-terminals to resolve the problem of parameters. We have decided to have parameter lists that separated with “,” and also parameter list can be empty too. For applying this language design, we have modified our grammar.

We have precedence rule in mathematical operations such as addition multiplication subtraction and mode according to algebraic logic. We have solved our conflicts and get rid of ambiguity by applying operator precedence. We have assigned different precedence levels to operators in two category (math\_op\_pre and math\_op\_post). Thus we do not have two or more possible parse trees for our mathematical calculations in our language.We have used left recursion thus our mathematical operations are left associative.

We have resolved ambiguity caused by division of statement that earlier were parted as incaseStatement and nonIncaseStatement is modified to solve ambiguity a nonIncaseStatement in an incaseStatement by embedding nonIncaseStatement to matched. Also we have used matched and unmatched for differenciate incaseStatement to prevent ambiguity.

# **Explanation of Conflicts**

We do not have any conflict left after the modifications in the ZGRobot’s BNF.

# **Descriptions of Nontrivial Tokens**

## **Comments**

In ZGRobot, we have one type of comment named as singleComment. It is handled by lex at the beginning of Lex Description File. SingleComment starts with // and ends with a new line. It can contain any character other than new line. Our approach to comment increase writability for making a single comment type in the program. Writing comments are very easy. Usage of single comment but not having multiline comment, decreases the writability but increses the readibility; however, an IDE can handle the need of multicomment with a shortcut like Python has with # and ctrl+/. There is no problem with reliability of comments while there is no unsuccessful attempt of multiline comments.

## **Identifiers**

Identifiers, in our program there are indicated as <variable> are user-defined names for integers, strings, booleans, floating point numbers, and functions. This names must be determined at the declaration of these variables. The user then can access later to these variables value by using these names. In our program, the user can’t use reserved words as an identifier. In addition, for readability purposes, in our program identifiers must start with a letter. Other than these two exceptions, the user can use any combination of letters and digits as an identifier.

## **Literals**

As literals, we have string literals which are combination of chars that can be anything other than new line that are wrapped with double quotations. These literals can be arguments to calls and can appear in right hand side of the assignment operations.

## **Reserved words**

**start:** indicate the start of the block statement

**done:** indicate the end of the block statement

**return:** indicate which expression is returned from the function, used as a last statement in the function

**inCase:** used at the beginning of the inCase statement

**notTheCase:** used after inCase reserved word, if there are any other options

**iter:** used at the beginning of the iter loop

**while:** used at the beginning of the while loop

**fun:** used at the beginning of the function declaration

**const:** used at the beginning of the constant variable declaration

**say:** indicate one of the predefined functions, say something function

**move:** indicate one of the predefined functions, move function

**turn:** indicate one of the predefined functions, turn function

**grab:** indicate one of the predefined functions, grab function

**release:** indicate one of the predefined functions, release function

**readData:** indicate one of the predefined functions, read data function

**ID:** used at the beginning of the sensor id

**sendData:** indicate one of the predefined functions, send data function

**receiveData:** indicate one of the predefined functions, receive data function

**int:** used in the declaration of an Integer type variable, scanned as <type>

**string:**used in the declaration of an String type variable,scanned as <type>

**logic:** used in of the declaration of an Logic(boolean) type variable, scanned as <type>

**float:** used in of the declaration of an Floating Point Number type variable, scanned as <type>

**true**: scanned as <logic> and used as a truth value of an logic type

**false:** scanned as <logic> and used as a truth value of an logic type

**main:** used when an main function declared

**input:** used at the beginning of input function call

**Note:** Below there are predefined symbols of our language and their scanned versions as tokens by the lex description file.

! : <endStatement>

( : <LP>

) : <RP>

: <empty>

("-"|"+")?[0-9]+ : <integer>

“(.\*)” : <string>

("-"|"+")?[0-9]+.[0-9]+ : <floatingPointNumber>

&& : <and>

|| : <or>

+@ : <add\_op>

-@ : <sub\_op>

\* : <mult\_op>

/ : <div\_op>

% : <mod\_op>

== : <equal>

=/= : <notEqual>

>= : <biggerOrEqual>

<= : <smallerOrEqual>

< : <smaller>

> : <bigger>

= : <assignment\_op>

([a-zA-Z])+([0-9a-zA-z])\* : <variable>

int | string | float | boolean : <type>

ID[0-9]+ : <sensorID>

"//".\*"\n" : <singleComment>