



Bilkent University

DEPARTMENT OF COMPUTER ENGINEERING

---

# CS 315 Programming Languages

## Project 2 Report

### Parser for a Robot PL

*Language Name: TooEZ*



*Authors: Group 33 – “ThePuzzlePieces”*

---

Naisila Puka	21600336	Section 1
Fatbardh Feta	21600334	Section 1
Kunduz Efronova	21600469	Section 1

*November 15, 2018*

## TABLE OF CONTENTS

<b>1</b>	<b>Introduction .....</b>	<b>1</b>
<b>2</b>	<b>General Description of TooEZ .....</b>	<b>1</b>
2.1	Variables and Types .....	2
2.2	Primitive Functions.....	2
2.3	Function Definitions.....	2
2.4	Assignment Statements .....	2
2.5	Arithmetic Expressions.....	2
2.6	Robot Sensors .....	3
2.7	Selection Statements .....	3
2.8	Loop Statements .....	3
2.9	IO Statements .....	4
<b>3</b>	<b>BNF Description and Nonterminals' Explanations .....</b>	<b>4</b>
<b>4</b>	<b>Motivations and Constraints for Chosen Tokens.....</b>	<b>11</b>
<b>5</b>	<b>Conclusion .....</b>	<b>12</b>



# TooEZ

## 1 Introduction

In this report we show the design of a new programming language created by us, called TooEZ. Our logo shows three puzzle pieces coming together, which represents us, three CS students collaborating on creating TooEZ.

Firstly, let us give an understanding of how we came up with this name. We had to design a Robot Programming Language for CS315 Programming Languages course, as part of the first assignment of the term project. The first assignment of the project specifically says: “People who purchase these robots may not be computer engineers; so it is important that the programming language is readable, writable and reliable.” So the first thought we had was that our language should be comfortable to understand and fully functional and reliable to make life EASY for people who will use it for the robots. So why not name it EZ programming language? Or furthermore, lets double it to 2EZ and make it TooEZ!

Our language grammar tries to stand stiff close to easiness to read and write and reliability. The report gives an introduction to our language by a brief general description of it, followed by language syntax in BNF form with nonterminal explanations, followed by descriptions of how nontrivial tokens of our language help the language achieve readability, writability and reliability goals, as well as a brief conclusion.

## 2 General Description of TooEZ

The language main structure is very easy to understand, even from someone who doesn't have much previous code experience. Everything that you want to get executed has to be denoted in TooEZ by specifically pointing out where it begins and where it ends. As we decided, **beginEZ** and **endEZ** reserved words will be used. If you want some shorthand functions to use in the execution (e.g. a predefined *walk* function that tells the robot to move a certain number of steps), then you specify these functions with **functEZ** command before writing the **beginEZ** command. In general, statements end with a full stop. You may keep comments anywhere you want to help you and other people better understand the code. Here's a brief code outline in TooEZ:

```
## This is a simple program outline in TooEZ for my robot.

## This is a function that makes my robot walk a specific number of steps
functEZ void walk<-(int steps)
    ## function statements
endfunct
## This is where execution of my program starts
beginEZ
    walk<-(10).
    ## Other statements to be executed
endEZ ## execution end
```

Following are some of our main language components with a brief description. You may read more about nonterminals' descriptions in section 3 of this report: "[BNF Description and Nonterminals' Explanations](#)" and more about token choice motivations in section 4 of this report: "[Motivations and Constraints for Chosen Tokens](#)".

## 2.1 Variables and Types

Our language supports variable declarations, since they are a crucial component of every other rule of TooEZ. Variables are easily manipulated. Our language provides four basic data types which we believe to be the most useful ones for a robot programming language: **int**, **float**, **string**, and **boolean**. A variable has to be one of these types. For a variable to be properly declared, the type follows the name, e.g. `int moveDistance`.

## 2.2 Primitive Functions

Being a robot programming language, TooEZ already has some primitive functions in hand for your robot, which may be used according to the programmer's desire. These functions are **move**, **turn**, **grab**, **release**, **read** data from a sensor given the sensor ID, **send** and **receive** data from/to another robot or master. As specified, **read** works by providing a sensor id and **send** works by providing a string message.

## 2.3 Function Definitions

In the process of implementing TooEZ we tried to make function definitions as simple as possible. All function definitions in our languages start with reserved word "**functEZ**" followed by return type: either **void**, **int**, **string**, **boolean** or **float**, followed by name of function which should start with a letter and can contain digits and underscore. If function has a parameter list, name will be followed by **ARGS\_ARROW <-** and the list of arguments is enclosed in parentheses. The function definition ends with reserved word **endfunct**. Example of function definitions is given below:

```
functEZ void moveCertainDistance<-(int number)
  ## function statements
endfunct
functEZ int getProduct<-(int num)
  ## function statements
endfunct
```

## 2.4 Assignment Statements

Naturally, when you want to assign a value to anything, you think of making it equal to that value. For this reason, TooEZ keeps it simple by using **=** character: `variable = RHS of assign statement`

The right hand side of the assignment statement can be anything that holds a value of one of our language's types, such as another variable, arithmetic expression etc.

## 2.5 Arithmetic Expressions

We want a robot to perform to evaluate an arithmetic expression in the same way as we as humans have learnt it in math. Therefore, TooEZ obeys the following rules:

1. Expression inside brackets like this (arithmetic expression) have to be calculated first, no matter the operator precedencies.
2. Multiplication and division have precedence over addition and subtraction.
3. Exponentiation has precedence over multiplication and division.
4. All operators except exponentiation operator are left associative. The exponentiation (power) operator is right associative.

An example arithmetic expression is:

```
term0 + term1 ^ 5 * term3 / 4 - term4 - term5 * (term6 + term7)
```

## 2.6 Robot Sensors

The language provides easy and immediate access to robot sensor values (there are 10 robot sensors). Here is an example:

```
float value = sensor7.
```

## 2.7 Selection Statements

Our selection statements are similar to those found in other languages except for the fact that we use the “endif” lexeme to indicate the end of an “if” statement. This easily eliminates the *dangling-else* problem. So we also support nested “if” statements.

Let’s see an example of a selection statement in TooEZ:

```
if (count < 5)
    count++.
else
    if(count == 2)
        release.
    elseif(count == 3)
        grab.
    endif
endif
```

In this example we can see that similar to other languages we have an “if” statement with a logic expression. When the user wants to declare consecutive “if” conditions they can use “elseif” statement. Every “if” statement should end with “endif” lexeme so the language knows when the statement ends. Furthermore, this makes it more readable for the programmer to know where the statement ends.

## 2.8 Loop Statements

Our language supports two types of loop statements: **for** and **while** statements. Both our loop statements are similar to those found in other languages. In order to define a loop statement the user has to write the name of the loop they want followed by a logical expression. Special to TooEZ (with the motivation of making it easier), in order to show the end of the loop the user has to use the **endfor/endwhile** lexeme.

For example:

```
int count = 0.
for(int i = 0; i < 10; i++)
    count++.
endif
while(count > 0)
    count--.
endwhile
```

## 2.9 IO Statements

We have implemented both input and output statements in TooEZ language. Corresponding to our language name, we have implemented method called “**ezin<-(variable\_input)**” to get user input, where **variable\_input** is the variable that will be assigned an input from the user. In order to output we use method called “**ezout<-(to\_be\_printed)**”, where **to\_be\_printed** can either be a variable or a constant (e.g. string constant for personalized messages). Example is provided below:

```
int input.  
ezin<-(input).  
ezout<-("The number you entered is: ").  
ezout<-(input).
```

We believe **ezin** and **ezout** are relatively easy to remember and use.

## 3 BNF Description and Nonterminals' Explanations

In order to design our language syntax more easily, we have made use of the following EBNF perks in cases where we saw it really suitable:

- [ ] OPTIONAL
- { } 0 OR MORE
- ( X | Y | Z ) one of X or Y or Z

In our language syntax, we have used our language tokens and lexemes as terminal symbols as explained below:

- In case the token is composed of only one lexeme, we have used the lexeme. For example, in TooEZ, **BEGIN\_EZ** token has only the **beginEZ** lexeme, so we use **beginEZ**. Another example, **COMMA** token has only **,** lexeme, so we used **,**.
- In case the token is composed of many lexemes, we have used the token. For example **VAR\_NAME**, **POS\_INTEGER**, **TYPE**.
- Tokens and lexemes are indicated by *THIS DARK ORANGE COLOR*.
- Tokens are easily distinguished from lexemes since they are shown in all CAPITAL letters.

Following is the grammar of our language in BNF form, along with nonterminal explanations. We have based our language rules on the structures of other languages such as C/C++ and Ruby, as well as the course textbook and our own creativity on how a robot programming language should be!

```
<PROGRAM> => {COMMENT} {{COMMENT}<FUNCTION_DEF>} beginEZ <STMTS>  
endEZ
```

This is our start symbol and it represents a whole program written in TooEZ. It may have comment(s) at the beginning, as well as before each function definition. After the function definitions, the part of the program that is to be executed starts with **beginEZ**, continues with statement(s) and ends with **endEZ** lexemes. This idea is similar to the main function in C++. Note: comments are strongly suggested.

```
<FUNCTION_DEF> => functEZ <RETURN_TYPE> VAR_NAME[<-(<ARGS_LIST>)]
                  <STMTS> endfunct
```

Each function definition has to start with `functEZ` lexeme, followed by the type of variable that the function will return, followed by the function name. The function name is a `VAR_NAME`, which is a token that represents strings starting with a letter, followed by 0 or more letters, digits or underscore characters. In case the function needs a list of arguments, they are directed into the function using `ARGS_ARROW <-`. The list of arguments is enclosed in parentheses. Then the function statements follow, and the function definition ends with `endfunct` lexeme.

```
<RETURN_TYPE> => TYPE
                | void
```

This nonterminal is defined to be used only in function returns. The return type of a function can either be a `TYPE` token, or `void` lexeme in case the function doesn't return anything. `TYPE` token is composed of `int`, `string`, `boolean` and `float` lexemes.

```
<ARGS_LIST> => TYPE VAR_NAME
              | TYPE VAR_NAME, <ARGS_LIST>
```

Each argument/parameter in the arguments' list is of the form `TYPE VAR_NAME`. (e.g. `string master_msg`) The list can have one or more arguments.

```
<STMTS> => <STMT><STMTS>
          | <STMT>
```

Statements can be one or more of `<STMT>` non-terminal.

```
<STMT> => <DECLARE_STMT>.
        | <ASSIGN_STMT>.
        | <FUNCTION_CALL>.
        | <IF_STMT>
        | <LOOP_STMT>
        | <JUMP_STMT>.
        | <RETURN_STMT>.
        | <INPUT_STMT>.
        | <OUTPUT_STMT>.
        | COMMENT
```

This is the syntactic unit of TooEZ and it expresses some action to be carried out, which can be a declarative, assignment, conditional, loop, jump or return statement, as well as a function call which will execute the statements in the function definition. We have also added `COMMENT` as a statement just for simplicity of adding comments anywhere in the program or function definitions.

**<DECLARE\_STMT> => TYPE VAR\_NAME**

This statement is used for declaring a variable, i.e, making the program aware that a specific variable exists. It is of the form **TYPE VAR\_NAME** (e.g. int step\_count). This variable will most probably be assigned a value later in the program by using assignment statement.

**<ASSIGN\_STMT> => [TYPE] VAR\_NAME <ASSIGN\_OP> <RHS\_ASSIGN\_STMT>**  
| **<INCREMENT\_STMT>**  
| **<DECREMENT\_STMT>**

This statement makes it possible to set or reset the value of program variables and is a fundamental construct for our language. If the variable is not declared before it should have one of the variable types (part of **TYPE** token) in front of its **VAR\_NAME**. After the **VAR\_NAME** there should always be the **<ASSIGN\_OP>** and a **<RHS\_ASSIGN\_STMT>** which will provide a value for this variable. In order for this statement to be legal the variable type in the left hand side of **<ASSIGN\_STMT>** and the type of the **<RHS\_ASSIGN\_STMT>** should be the same. **<ASSIGN\_STMT>** can be also just a **<INCREMENT\_STMT>** or **<DECREMENT\_STMT>** which will be explained in more detail below.

**<ASSIGN\_OP> => =**

This nonterminal shows that the **ASSIGN\_OP** (=) token is used as our language's assignment operator.

**<RHS\_ASSIGN\_STMT> => <ARITHMETIC\_EXPR>**  
| **BOOLEAN**  
| **STRING**  
| **<FUNCTION\_CALL>**

This nonterminal describes the right hand side of an assignment statement: it is used to give values to variables. It can give to a variable the value of a previously declared program variable or the value of an arithmetic expression, **<ARITHMETIC\_EXPR>**. It can also give the value of a **<CONSTANT>** or the return value of a **<FUNCTION\_CALL>**.

**<ARITHMETIC\_EXPR> => <ARITHMETIC\_EXPR> + <TERM>**  
| **<ARITHMETIC\_EXPR> - <TERM>**  
| **<TERM>**

Arithmetic expression can be a **<TERM>** or a **+/-** operation between another **<ARITHMETIC\_EXPR>** and a **<TERM>**. Here we are using the power of left recursion to make sure that our subtraction operation is left recursive, as we expect it to be.



```

<TERM> => <TERM> * <FACTOR>
        | <TERM> / <FACTOR>
        | <FACTOR>

```

We use the special <TERM> nonterminal in <ARITHMETIC\_EXPR> to give precedence to multiplication and division in case the arithmetic expression is mixed with +, -, \* and /, and there are no guiding parentheses to show precedence (these parentheses are further explained in <COMPONENT>). Similar to the idea for subtraction, we use left recursion to make sure that division operation is left associative, as we expect it to be.

```

<FACTOR> => <COMPONENT> ^ <FACTOR>
          | <COMPONENT>

```

We use the special <FACTOR> nonterminal in <TERM> to give precedence to power operation in case the arithmetic expression is mixed with +, -, \*, / and ^, and there are no guiding parentheses to show precedence. In this case, since we want exponentiation to be right associative, we use right recursion.

```

<COMPONENT> => (<ARITHMETIC_EXPR>)
              | <NUMBER>

```

The bottom <COMPONENT> nonterminal shows the components of the arithmetic expression up to parentheses. It can be an arithmetic expression in parentheses, which will be given precedence, or just a number. After all parentheses are handled, operations are given precedence as explained in the above nonterminals.

```

<NUMBER> => VAR_NAME
          | SENSOR_ID
          | <CONSTANT_NR>

```

Each <NUMBER> must have a numerical value, which can be kept in either a variable with an assigned numerical value or a <CONSTANT\_NR>.

```

<CONSTANT_NR> => POS_INTEGER
                | NEG_INTEGER
                | POS_FLOAT
                | NEG_FLOAT

```

This nonterminal shows all numerical forms with which the user can make arithmetical operations. POS\_INTEGER and POS\_FLOAT should be written without a sign before them while NEG\_INTEGER and NEG\_FLOAT must be written in brackets (e.g 5, (-5), 0.35, (-0.35)).

```

<CONSTANT> => BOOLEAN
              | STRING
              | <CONSTANT_NR>

```

In our language constants can be only **BOOLEAN**, **STRING** and **<CONSTANT\_NR>**.

```

<INCREMENT_STMT> => VAR_NAME ++

```

**VAR\_NAME ++** is the short version of **VAR\_NAME = VAR\_NAME + VAR\_NAME**. This feature is added to make the language more writable.

```

<DECREMENT_STMT> => VAR_NAME --

```

**VAR\_NAME--** is the short version of **VAR\_NAME = VAR\_NAME - VAR\_NAME**. This feature is added to make the language more writable.

```

<FUNCTION_CALL> => VAR_NAME [<-(<ARGS_VALUES_LIST>)]
                  | <PRIM_FUNCT>

```

There are two kinds of functions that can be called in our language: the primitive functions and the functions defined with **<FUNCTION\_DEF>**. To call a **<FUNCTION\_DEF>** function the user must write **VAR\_NAME** of the function. If the function requires one or more arguments to execute the user can specify them by using the

**ARGS\_ARROW <-:**

**VAR\_NAME ARGS\_ARROW (<ARGS\_VALUES\_LIST>)** e.g. **foo<-("robot", 3)**

```

<ARGS_VALUES_LIST> => <ARGS_VALUE>
                    | <ARGS_VALUE>, <ARGS_VALUES_LIST>

```

The functions can have any number of arguments. Commas will separate the arguments.

```

<ARGS_VALUE> => VAR_NAME
              | <CONSTANT>

```

A function argument can either be a predefined **VAR\_NAME** or **<CONSTANT>**.

```

<PRIM_FUNCT> => move
              | turn<-(<DIRECTION>)
              | grab
              | release
              | read<-(<SENSOR_ID>)
              | send<-((STRING|VAR_NAME))
              | receive

```

This nonterminal can be one of the primitive functions for this robot programming language: **move**, **turn<-(<DIRECTION>)**, **grab**, **release**, **read<-(<SENSOR\_ID>)**, **send<-((STRING|VAR\_NAME))** and **receive**. **turn<-(<DIRECTION>)** function must get a **<DIRECTION>** variable to specify the movement direction. **read<-(<SENSOR\_ID>)** needs a **SENSOR\_ID** variable which can be any of the sensors that are located in the robot. For this programming language, we assume that sensor float

values will always be available to the program (because they are part of the robot), so they are automatically saved in float variables called `sensorid` where `id` is a digit from 0-9. `send-<((STRING|VAR_NAME))` requires a `STRING` constant or a `string` variable that will then be sent to the master or other robots as a mean of communication between them. All the other primitive functions do not need variables and can be called anytime in the program. The primitive function names are reserved words for our program, as will be specified in the Lex file.

```
<DIRECTION> => left
                | right
```

For movement robot will use `left` or `right` to get movement direction. These will also be reserved words in our language.

```
<IF_STMT> => if(<LOGIC_EXPR>) <STMTS> {elseif(<LOGIC_EXPR>) <STMTS>}
              endif
              | if(<LOGIC_EXPR>) <STMTS> {elseif(<LOGIC_EXPR>) <STMTS>}
              else <STMTS> endif
```

This nonterminal offers a non-ambiguous way to handle `if` statements, i.e, the Dangling-Else problem. Every `if` statement will end with `endif` lexeme, which will indicate inside which `if` statement each `else` belongs. This idea is similar to always using curly brackets to indicate where an `if` statement ends. However, we made this choice because `endif` lexeme is so readable and writable at the same time for humans! If the `<LOGIC_EXPR>` inside `if(<LOGIC_EXPR>)` is `true` then the `<STMTS>` that follow are executed. The `if` statement can be followed by zero or more `elseif(<LOGIC_EXPR>)<STMTS>` expressions. If the first `if` condition is not meet then all the other `elseif(<LOGIC_EXPR>)` will be checked iteratively. `if` conditions can have an `else` statement at the end which will always be executed if the prior statements conditions are all `false`. Since `if` statements are `<STMTS>` our language can support nested `if` statements.

```
<LOGIC_EXPR> => BOOLEAN
                | VAR_NAME
                | <COMP_EXPR>
                | (<LOGIC_EXPR> /\ <LOGIC_EXPR>)
                | (<LOGIC_EXPR> \/ <LOGIC_EXPR>)
                | ~(<LOGIC_EXPR>)
```

We use `<LOGIC_EXPR>` to deal with `BOOLEAN` operations. A `<LOGIC_EXPR>` can either be a `BOOLEAN`, `VAR_NAME` or a logic operation. Our language supports `/\`(and), `\/`(or) and `~`(negation) operations on `<LOGIC_EXPR>`'s. We chose these symbols for logic operations since they are very commonly used in logic statements outside of computer science as well.

**<COMP\_EXPR>** => <NUMBER> <COMP\_OP> <NUMBER>

This nonterminal makes it possible to compare two <NUMBER>s.

**<COMP\_OP>** => <  
| >  
| >=  
| <=  
| ==  
| ~=

This is the list of all comparison operations that two <NUMBER>s can undergo in our language.

**<LOOP\_STMT>** => <WHILE\_STMT>  
| <FOR\_STMT>

There are two kind of loops in our language: the <WHILE\_STMT> and the <FOR\_STMT>. Our language uses loop statements for a sequence of statements that it wants to continually execute: this is especially important for a robot programming language since a common thing that a robot does is **move** until certain condition is met. So the language will execute **move** primitive function inside a loop.

**<WHILE\_STMT>** => **while**(<LOGIC\_EXPR>) <STMTS> **endwhile**

This loop should begin with **while**(<LOGIC\_EXPR>) and end with **endwhile** lexeme. The function of this loop is the same as in most languages: While the <LOGIC\_EXPR> is true then the <STMTS> will be executed, otherwise the program will exit the loop. Of course, we provide helpers for exception cases, and these are the jump statements (you can find below).

**<FOR\_STMT>** => **for**(<ASSIGN\_STMT>; <LOGIC\_EXPR>; <ASSIGN\_STMT>)  
<STMTS> **endfor**

**for** loops in our language start with **for**(<ASSIGN\_STMT>; <LOGIC\_EXPR>; <ASSIGN\_STMT>) and end with **endfor**. The loop will repeat while the <LOGIC\_EXPR> is true. The first <ASSIGN\_STMT> is for choosing the iteration variable, and it executes once. After each iteration the second <ASSIGN\_STMT> will execute. Similarly, jump statements can be used for exceptions.

**<JUMP\_STMT>** => **skip**  
| **break**

This nonterminal provides statements that can “interrupt” loops. They are unconditional jumps to a specific part of the code. The **skip** statement passes control to the end of the loop (**endwhile** or **endfor**). This may be used to skip an iteration and go directly to the next one. The **break** statement terminates execution of the closest enclosing loop.

**<RETURN\_STMT> => `return [(VAR_NAME|<CONSTANT>)]`**

This statement is used to exit from a function in TooEZ. It may or may not hold a value, depending on the **functEZ**. **functEZ**-s which are designed to return a value use a **return** statement followed by a **VAR\_NAME** or a **<CONSTANT>**. **functEZ**-s whose **return** type is **void** use only a **return** statement. **return** is another reserved word in our program.

**<INPUT\_STMT> => `ezin<-(VAR_NAME)`**


This statement is used to assign an input value to the variable in **VAR\_NAME**. The user can enter a value from the keyboard. The input statement has to start with the **ezin** lexeme. In fact, this is a function call, where the function is used for input.

**<OUTPUT\_STMT> => `ezout<-((VAR_NAME|<CONSTANT>))`**

This statement is used to print a constant or a variable value to the console. The output statement has to start with the **ezout** lexeme. In fact, this is a function call, where the function is used for output.

## 4 Motivations and Constraints for Chosen Tokens

**>COMMENT:** The syntax of comments is very **simple**: they are one-line sentences and they start with two hashes **##**. Thus it is easy to distinguish between code and comment, and the program is more **readable**. It is also **writable** because the syntax is easy. The idea of using hash sign came from Ruby language.

**>VAR\_NAME:** To keep the language **easy to read and write**, we don't allow all characters to be part of variable names (identifier). Only letters, digits and underscore are allowed. It is up to user to use the underscore or not. The purpose of allowing it comes from the need of concatenating separate words. For example, for calling a variable as first sum, there are two optimal choices: firstSum and first\_sum. By not mixing other complex characters in identifiers the code is also **easier to debug**. Also, we have no restrictions on the length (**readability** )

**>SENSOR\_ID:** This is a choice for giving the language **easy access to robot sensors**. We assume that the robot has 10 sensors with which it perceives the environment, and we give read and write access of these values to every TooEZ program. The variables with which these sensors are controlled are called sensorid, where id can be a digit 0-9. E.g. sensor0, sensor1, etc. These are at the same time reserved words for the program. Since in most of the functions written for the robot, the master will want to manipulate these sensor values and put the robot in action according to those values, we decided the language is **more readable and writable** for this purpose when the sensors are always at hand!

> *Reserved words:* beginEZ endEZ for endfor while endwhile if endif functEZ endfunct ezin ezout

We have made use of these reserved words to increase the **readability** of the program. It is very easy to understand where the “main” part of the program is by finding **beginEZ**. Also, it is very easy to understand where loops and conditional statements start and end by looking at **end...** reserved words. Especially for a robot programming language, we decided that these reserved words go well with the primitive functions **move grab turn** etc because it makes it very readable to understand what actions the robot will take and when. Also, our reserved words really **save cost of learning** the language because it is very natural to end an if statement with **endif** (and so on)!

> **functEZ**: This reserved word is special because it gives **great ease to writing** a program in TooEZ: we allow function definitions which can then easily be used without repeating the inside statements everytime in the “main” part of the program.

> **TYPE**: We give great importance to type checking in our language to increase **reliability** for a robot programming language. Especially for assignment statements and in logic expressions, the type of the left hand side has to be equal to the type of the right hand side. We provide 4 simple types (**int, boolean, string, float**) so it is not difficult to stick to the type rules of our language. After all, you want to make a hardware robot perform actions, so it is highly important that the code is reliable and doesn't have type errors, which may result in unexpected actions performed by the robot.

## 5 Conclusion

Ideas of three junior CS students were merged to design this simple robot programming language, TooEZ, trying to make it readable, writable, reliable and efficient. The language includes primitive robot functions, variable names, assignment, logic and arithmetic operators, arithmetic and logic expressions, conditional statements, loops, composed function definitions with arguments list and arguments arrow, jump statements, return statements, special variables for sensors and comments.