Bilkent University

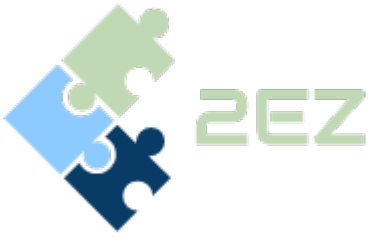
DEPARTMENT OF COMPUTER ENGINEERING

CS 315 Programming Languages

Project 1 Report

Lexical Analyzer for a Robot PL

Language Name: TooEZ



Authors

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

October 29, 2018

Table of Contents

TooEZ

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 assignment 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 our 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.

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>
| 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> => VAR_NAME
                   | <ARITHMETIC_EXPR>
                   | <CONSTANT>
                   | <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
          | <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> => VAR_NAME
| <CONSTANT>
| <ARGS_VALUES_LIST>, <ARGS_VALUES_LIST>

The functions can have any number of arguments, which can either be a predefined VAR_NAME or <CONSTANT>. Commas will separate the arguments.

<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> {elsif(<LOGIC_EXPR>) <STMTS>}
             endif
           | if(<LOGIC_EXPR>) <STMTS> {elsif(<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 elsif(<LOGIC_EXPR>)<STMTS> expressions. If the first if condition is not meet then all the other elsif(<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.

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
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.

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.