BILKENT UNIVERSITY
COMPUTER ENGINEERING DEPARTMENT

CS 315
PROGRAMMING LANGUAGES

PROJECT 1

Group 52

Emre Açıkgöz	21801914	Section 2
Burak Yiğit Uslu	21801745	Section 2

17/10/2020

Contents

1. Introduction

Name of our language is “doplang”, which stands for **d**rone **o**riented **p**rogramming **lang**uage.

This programming language aims to be easy to write and read, while covering all of the basic functionalities required for programming a drone. Because of these aspects, doplang is a great language for someone new to programming or a technology hobbyist to learn.

To briefly discuss the submitted files, the lexical analyzer file for this language is “doplang_group52.l”. 5 example programs can be found in the “examples_group52.txt” file. In those 5 example programs, every language construct and operator is used at least once.

An important note is about the BNF file. The BNF description in this report makes use of tokens defined in the lex file, therefore, for example, instead of “(“, “LP” is used. An alternative BNF description written without using the tokens from the lexical analyzer is also included in the submitted files, with the name “alternative_form_of_BNF_group52.txt”.

2. The Complete BNF Description of doplang

```
<program> ::= <statement-list>

<identifier-list> :== IDENTIFIER
                    | IDENTIFIER COMMA <identifier-list>

<statement-list> ::= <statement>
                   | NEWLINE <statement-list>
                   | <statement> NEWLINE <statement-list>

<statement> ::= <assignment-statement>
              | <expression>
              | <structured-block-declaration>
              | <return-statement>
              | <break-statement>

<structured-block-declaration> ::= <while-loop>
                                 | <repeat-loop>
                                 | <conditional-statement>
                                 | <function-declaration>

<assignment-statement> ::=
<identifier-list> ASSIGNMENT_OP <expression-list>

<expression> ::= <logical-or-expression>
```

```
<expression-list> ::= <expression>
                      | <expression> COMMA <expression-list>


<logical-or-expression> ::= <logical-and-expression>
                      | <logical-or-expression> LOGICAL_OR_OP
                <logical-and-expression>

<logical-and-expression> ::= <logical-equality-expression>
      | <logical-and-expression> LOGICAL_AND_OP
      <logical-equality-expression>

<logical-equality-expression>::= <logical-ltgt-expression>
           | <logical-equality-expression> EQUALITY_OP
      <logical-ltgt-expression>
           | <logical-equality-expression> INEQUALITY_OP
      <logical-ltgt-expression>

<logical-ltgt-expression> ::= <unary-not>
      | <logical-ltgt-expression> LESS_THAN_OP <unary-not>
      | <logical-ltgt-expression> LESS_EQUAL_THAN_OP <unary-not>
      | <logical-ltgt-expression> GREATER_THAN_OP <unary-not>
      | <logical-ltgt-expression> GREATER_EQUAL_THAN_OP <unary-not>

<unary-not>::= UNARY_NOT <unary-not>
      | <addition>

<addition>::= <multiplication>
           | <addition> ADDITION_OP <multiplication>
           | <addition> SUBTRACTION_OP <multiplication>

<multiplication> ::= <group>
                 | <multiplication> MULTIPLICATION_OP <group>
                 | <multiplication> DIVISION_OP <group>

<group> ::= <primary>
           | LP <expression> RP
```

<primary>::= INTEGER_LITERAL
| BOOLEAN_LITERAL
| STRING_LITERAL
| IDENTIFIER
| <function-call>

<while-loop> ::= WHILE <expression> COLON LCBRACE <statement-list> RCBRACE

<repeat-loop> ::= REPEAT <expression> TIMES COLON LCBRACE <statement-list> RCBRACE

<conditional-statement> ::= IF <expression> COLON LCBRACE <statement-list> RCBRACE
| IF <expression> COLON {<statement-list>} ELSE COLON LCBRACE <statement-list> RCBRACE

<function-declaration> ::=
FUNCTION_DEFINITION IDENTIFIER LP <identifier-list> RP LBRACE <statement-list> RBRACE
| FUNCTION_DEFINITION IDENTIFIER LP RP LBRACE <statement-list> RBRACE

<return-statement> ::= RETURN <expression-list> | RETURN

<break-statement> ::= BREAK

<function-call> ::= <identifier> LP RP
| <identifier> LP <identifier-list> RP
| <built-in-function-with-variable>

<built-in-function-with-variable> ::=
PRINT LP <expression> RP
| <built-in-function> LP RP

```
<built-in-function> ::= READ_INCLINATION
                      | READ_ALTITUDE
                      | READ_TEMPERATURE
                      | READ_TIMER
                      | READ_ACCELERALION
                      | TURN_CAMERA_ON
                      | TURN_CAMERA_OFF
                      | TAKE_PICTURE
                      | CONNECT_TO_DRONE
                      | INPUT
                      | EXIT
                      | TAKE_OFF
                      | LAND
```

3. Explanations of Language Constructs

Identifier

Identifiers in doplang are used for identifying variables and functions. Identifiers in doplang may not start with digits but may include digits in them. An identifier must only consist of alphanumeric characters and the underscore "_". As a result of this, any identifiers for variables and functions in doplang cannot contain ".", ":" and such. The only exception to this rule are the built-in functions, all of which start with the "doplang." prefix, and therefore, all of the built in functions contain the ".".

Identifier List

```
<identifier-list> :==  IDENTIFIER
                    |  IDENTIFIER COMMA <identifier-list>
```

An identifier list is a list of identifiers separated by commas.

Statements

```
<statement> ::= <assignment-statement>
| <expression>
| <structured-block-declaration>
| <return-statement>
| <break-statement>
```

Statements in doplang consist of statement types written above. Each statement type is explained in detail in their respective subsections.

Statement Lists

```
<statement-list> ::= <statement>
                   | NEWLINE <statement-list>
                   | <statement> NEWLINE <statement-list>
```

In doplang, statement lists are a set of statements separated by new lines. Each individual statement in a statement list must be in a new line as new lines are the only constructs used for separating statements.

Assignment Statement

```
<assignment-statement> ::=
<identifier-list> ASSIGNMENT_OP <expression-list>
```

In doplang, assignment statements are statements that have the variable at the left hand side, the assignment operator "=" and the value to be assigned to the variable on the right hand side. The assignment statements do not necessarily have to contain one identifier and one expression. For assignment statements with multiple identifiers and multiple expressions, the number and the order of the respective identifiers and expressions must match.

An example assignment statement is as follows:

```
varA, varB, varC = 5 + 5, 22 * 15, “string1”
```

Structured Block Declaration

```
<structured-block-declaration> ::= <while-loop>
| <repeat-loop>
| <conditional-statement>
| <function-declaration>
```

In doplang, structured block declarations are while loops, repeat loops, conditional statements, and function declarations.

Conditional Statement

```
<conditional-statement> ::=
       IF <expression> COLON LCBRACE  <statement-list> RCBRACE
     | IF <expression> COLON  {<statement-list>} ELSE COLON
LCBRACE <statement-list> RCBRACE
```

Conditional statements in doplang are defined by the if and else keywords. An if keyword is followed by an expression, followed by a colon indicating the end of expression. If the expression inside if statements is evaluated to true, statement lists in the curly braces are executed. If it is evaluated to false, the statement list is not executed. If there is an else following the if, and if the if statement is evaluated to false, statements inside the curly braces following the else are executed. By convention, the users are expected to use conditional statements with boolean expressions for evaluation, however, if the expression inside the if statement is evaluated into anything other than false or 0, the expression inside the conditional statement is considered as true, and statement list in the curly braces following the if are executed.

There are no “else if” statements in the doplang. To achieve the same functionality, new conditional statements can be declared inside the curly braces of the super conditional statement.

Example conditional statements are as follows:

```
if a < b : {
      doplang.print( “a < b”)
}
else : {
      if a > b : {
            doplang.print( “a > b”)
      }
      else : {
            doplang.print( “a = b”)
      }
}
```

Break Statement

`<break-statement>` `::= BREAK`

Break statement breaks out of the loop with the smallest scope it was executed in, ignoring whether it was invoked in another statement block. For example, the following break statement breaks into the line indicated in the code fragment:

```
// ...
while a < b : {
      a = a * 2
      while true : {
            c = c + a
            if c < a : {
                  break
            }
      }
      // Next line is executed after the break statement
      doplang.print( “Out of the nested while loop”)
}
```

Return Statement

<return-statement> ::= RETURN <expression-list> | RETURN

Return statements are found in function definitions. A function may or may not have return statement(s). An empty return statement returns to the caller false, however, by convention, empty return statements should not be used to provide boolean functionality. Return statements may return more than one value, where values are an expression list. In that case, as tuples are not defined in doplang, the expected number of values must be equal to the number of values in the return statement.

For example, an example function defined as this:

```
func example_func( ) {
      // …
      return varA, varB, varC
}
```

must be called with 3 variables, in the following format:

```
var1, var2, var3 = example_func()
```

Function Declarations

<function-declaration> ::=
FUNCTION_DEFINITION IDENTIFIER LP <identifier-list> RP LBRACE <statement-list> RBRACE
| FUNCTION_DEFINITION IDENTIFIER LP RP LBRACE <statement-list> RBRACE

In doplang, new functions are declared using the keyword *func*, followed by the identifier of the function, a left and a right parentheses, and optionally, an identifier list inside the parentheses. The statement list the function executes when called must be declared inside curly braces following the parenthesis. By convention, the left curly brace is put on the same line as the beginning of the function declaration, while the right the right curly brace is put on the line following the last statement to be executed by the function. By convention, newly declared functions should be named in all lower case letters and digits (except the first character in the function name) and individual words making up the function identifier should be separated using "_". The function declarations should be done before the function is called.

An example function definition following the conventions of the language is as follows:

```
func example_function1( var1, var2) {
      doplang.print( var1)
      doplang.print( var2)
      return var1 + var2
}
```

Function Calls

```
<function-call> ::= <identifier> LP RP
                | <identifier> LP <identifier-list> RP
                | <built-in-function-with-variable>
```

Previously declared functions can be called in the code using the identifiers of the functions, followed by a matching set of parentheses. If by definition the function requires a set of parameters to be passed, the parameters to be passed into the function must be written inside the parentheses in the order the function parameters were defined in the function declaration. The number of parameters in the function declaration must match the number of user-defined arguments in the function call. All of the parameters passed to the function are passed by value.

An example function call for the function defined above is as follows:

```
anotherVariable = example_function( variableA, variableB)
```

Built-In Functions

```
<built-in-function-with-variable> ::=
                                 PRINT LP <expression> RP
                                 | <built-in-function> LP RP
<built-in-function> ::= READ_INCLINATION
                      | READ_ALTITUDE
                      | READ_TEMPERATURE
                      | READ_TIMER
                      | READ_ACCELERALION
                      | TURN_CAMERA_ON
                      | TURN_CAMERA_OFF
                      | TAKE_PICTURE
                      | CONNECT_TO_DRONE
                      | INPUT
                      | EXIT
                      | TAKE_OFF
                      | LAND
```

Functions that come as built-in functions in doplang follow the general rules about function calls and declarations with one exception: All of the built-in functions are called with the prefix “doplang.” For example, a statement including a function call for reading the inclination is as follows:

```
exampleVariable = doplang.read_inclination()
```

This is mainly due to 2 reasons: Firstly, due to the production rules defined in the BNF of doplang, defining built-in functions as example_built_in_func() leads to ambiguous grammar rules, because then the function calls fits into multiple production rules. By the addition of "doplang." to the beginning of these functions' names, as identifiers for user defined functions cannot include "." in their names, the ambiguity in grammar is eliminated. Secondly, as the doplang programs are to interact with drones, reliability is very important, and minor wrong things can cause drones to behave unexpectedly, causing damage. Addition of "doplang." shows which functions are reliable because they are built in, and which functions users should use on their own risk.

While Loop

```
<while-loop> ::= WHILE <expression> COLON LCBRACE  <statement-list> 
RCBRACE
```

While loops consist of the *while* keyword, followed by an expression, followed by a colon indicating the expression is finished, and a statement list inside curly braces. Statement list is executed until the expression is evaluated into *false* at the beginning of a new iteration. Statement list must be put inside curly braces. By convention, expression for evaluation should be evaluated to a boolean.

An example while loop is as follows:

```
while x <  5 : {
      x = x + 1
}
```

.

Repeat Loop

```
<repeat-loop> ::= REPEAT <expression> TIMES COLON LCBRACE 
<statement-list> RCBRACE
```

Repeat loop executes the statement list stated inside the curly braces repeatedly for the number of times that the expression evaluates to. Expression must evaluate to an integer literal. Otherwise, it is a runtime error.

Repeat loops are indicated by the *repeat* keyword, followed by an expression followed by the *times* keyword followed by a colon and a statement list inside curly braces.

If after the execution of one of the iterations the value expression evaluates to changes, the number of iterations will not change. How many times the repeat loop will be executed is determined before the first iteration, and this number does not change throughout iterations.

Expressions

```
<expression> ::= <logical-or-expression>
<expression-list> ::= <expression>
                        | <expression> COMMA <expression-list>
<logical-or-expression> ::= <logical-and-expression>
      | <logical-or-expression> LOGICAL_OR_OP 
<logical-and-expression>
```

```
<logical-and-expression> ::= <logical-equality-expression>
      | <logical-and-expression> LOGICAL_AND_OP
<logical-equality-expression>

<logical-equality-expression>::= <logical-ltgt-expression>
      | <logical-equality-expression> EQUALITY_OP
<logical-ltgt-expression>
      | <logical-equality-expression> INEQUALITY_OP
<logical-ltgt-expression>

<logical-ltgt-expression> ::= <unary-not>
      | <logical-ltgt-expression> LESS_THAN_OP <unary-not>
      | <logical-ltgt-expression> LESS_EQUAL_THAN_OP <unary-not>
      | <logical-ltgt-expression> GREATER_THAN_OP <unary-not>
      | <logical-ltgt-expression> GREATER_EQUAL_THAN_OP <unary-not>

<unary-not>::= UNARY_NOT <unary-not>
      | <addition>

<addition>::= <multiplication>
           | <addition> ADDITION_OP <multiplication>
           | <addition> SUBTRACTION_OP <multiplication>

<multiplication> ::= <group>
                | <multiplication> MULTIPLICATION_OP <group>
                | <multiplication> DIVISION_OP <group>

<group> ::= <primary>
           | LP <expression> RP
```

Expressions can be derived to a *boolean* literal, a *string* literal, an *integer* literal, a function call, or an *identifier*. The derived construct behaves according to the rules defined above. Following are the operators in doplang with respect to their order of precedence, from the operator with the highest precedence to the lowest. Operators that are in the same row have the same order of precedence. If in an expression, there are multiple operators with the same order of precedence, they are evaluated according to their associativity rule.

An expression consists of many nonterminals with different depths. The depth of the nonterminal determines the order of precedence of the operator among others. Higher the depth of the nonterminal, higher the precedence of the *operator*.

The nonterminal in which the operator is	operator terminal	operator name	Associativity
<function_call>	()	function call	Right to left
<group>	( )	grouping parentheses	Left to Right
<multiplication>	*, /	multiplication and division	Left to Right
<addition>	+, -	addition and subtraction	Left to Right
<not-unary>	not	logical not	Right to left
<logical-ltgt-expression>	<=, <, >=, >	less than or equal to, less than, greater than or equal to, greater than	Left to Right
<logical-equality-expression>	==, !=	equality, inequality	Left to Right
<logical-and-expression>	and	logical and	Left to Right
<logical-or-expression>	or	logical or	Left to Right

Operators

or operator: Evaluates to true if one of the operands evaluates to true.
and operator: Evaluates to true if both of the operands evaluates to true.
== operator: Evaluates to true if both of the operands evaluates to the same value.
!= operator: Evaluates to true if both of the operands evaluates to different values.
>= operator: Evaluates to true if the left operand is greater than or equal to the right operand.
> operator: Evaluates to true if the left operand is greater than right operand.
<= operator: Evaluates to true if the left operand is less than or equal to the right operand.
< operator: Evaluates to true if the left operand is less than right operand.
not operator: Evaluates to true if the operand evaluates to false.
+ operator: Evaluates to sum of its operands.
- operator: Evaluates to the subtraction of the left operand from the right operand.
* operator: Evaluates to multiplication of its operands.
/ operator: Evaluates to the division of the left operand by the right operand.

Logical operands expect boolean literals, identifiers or function calls that evaluate to boolean values. However, anything except 0 or false evaluates to true.

Relational operands expect constructs that evaluate to integers.

```
<primary>::= INTEGER_LITERAL
           | BOOLEAN_LITERAL
           | STRING_LITERAL
           | IDENTIFIER
           | <function-call>
```

Each primary nonterminal can be derived from an expression. Then, each primary construct evaluates to a literal value. Identifiers evaluate to the value of the literal they represent. Functions with return statements evaluate to the value of the expression they return. Functions without return statements evaluate to false. By induction, each expression evaluates to a literal value. However, it should be noted that, by convention, function without return statements should not be used for logical operations. This design is only for completeness of the language, because by production rules for the language, statements such as the following are theoretically valid, even if function foo does not have a return statement:

```
if true and foo() : {
     //…
}
```

However, by convention, such uses should be avoided.

4. Descriptions of Built-In Functions

doplang.read_inclination: Returns a set of 3 numerical values, between -180 and 180. These numbers are the drones inclination on the x-axis, y-axis and z-axis respectively. The returned values have the unit of degrees.
doplang.read_altitude: Returns the altitude of the drone in centimeters.
doplang.read_temperature: Returns the numerical value of the temperature of the drone in Celcius degrees.
doplang.read_timer: Returns the numerical value corresponding to the time drone has been turned on in seconds. The timer resets when the drone is restarted.
doplang.read_acceleration: Returns a set of 3 numerical values. Numbers are the acceleration of the drone in m/s on the x-axis, y-axis, z-axis respectively.
doplang.turn_camera_on: Turns the camera of the drone on. Returns true if successful. False, otherwise.
doplang.turn_camera_off: Turns the camera of the drone off. Returns true if successful. False, otherwise.
doplang.take_picture: Takes a picture. Returns true if successful.

doplang.connect_to_drone: Connects to the drone. Returns true if connection was successful. False, otherwise.
doplang.input: Returns the value user inputs to the terminal.
doplang.exit: Stops the execution of statements and exits the program after landing the drone if it was in the air. Returns true if successful, false otherwise. Has a higher likelihood of returning false, because if the drone has not safe conditions for landing, the exit process will not take place and return false and the drone will not land.
doplang.take_off: Initiates the drone to take off from the surface, in a vertical direction. Returns true if successful, false otherwise.
doplang.land: Lands the drone to the ground. Returns true if successful, false otherwise. Has a higher likelihood of returning false, because if the drone has not safe conditions for landing, it will fail to land.

5. Descriptions of Nontrivial Tokens in doplang

These tokens are defined in lexical analyzer and used in bnf grammar.

NEWLINE : `\n`
A regular expression that matches with a new line to separate statements.

IDENTIFIER : `[A-Za-z][_A-Za-z0-9]*`
Identifiers are used in variable names, user-defined function names, function parameter names. Every identifier must start with an alphabet character and can have "_" or alphanumeric characters in it.

The reason behind this design is directly related to lexical analysis. That is, if identifiers that started with numerical values were allowed, efficiency of lexical analysis would decrease, because the lexical analyzer would have to do backtracking for identifier names that started with numerical values.

INTEGER_LITERAL : `[+-]?[0-9]+`
Integer literals can have an optional sign and a magnitude.

COMMENT : `\/\/.*\n`
In doplang, comments start with "//" and continue up to a new line. Comments are ignored.

STRING_LITERAL : `\".*\"`
String literals can be created by a pair of double quotes.

BOOLEAN_LITERAL : `true | false`
Boolean literals are represented with two reserved words.

LP: `(`
RP: `)`
LBRACE : `{`

RBRACE : }
COLON : \:
ADDITION_OP : \+
SUBTRACTION_OP : \-
MULTIPLICATION_OP : *
DIVISION_OP : \/
ASSIGNMENT_OP : \=
EQUALITY_OP : \=\=
INEQUALITY_OP : \!\=
GREATER_THAN_OP : \>
LESS_THAN_OP : \<
GREATER_EQUAL_THAN_OP : \>\=
LESS_EQUAL_THAN_OP : \<\=
COMMA : \,
LOGICAL_OR_OP : or
LOGICAL_AND_OP : and
UNARY_NOT : not
WHILE : while
REPEAT : repeat
TIMES : times
FUNCTION_DEFINITION : func
IF : if
ELSE : else
RETURN : return
READ_ALTITUDE : doplang\.read_altitude
READ_INCLINATION : doplang\.read_inclination
READ_TEMPERATURE : doplang\.read_temperature
READ_TIMER : doplang\.read_timer
READ_ACCELERATION : doplang\.read_acceleration
TURN_CAMERA_ON : doplang\.turn_camera_on
TURN_CAMERA_OFF : doplang\.turn_camera_off
TAKE_PICTURE : doplang\.take_picture
CONNECT_TO_DRONE : doplang\.connect_to_drone
EXIT : doplang\.exit
INPUT : doplang\.input
LAND : doplang\.land
TAKE_OFF : doplang\.take_off

6. Conventions of doplang

- Variable names should start lower case and each new word in the variable name should have their first letter uppercase. Examples: photoCount, name, variable2
- Function names should consist of all lower case letters, each word separated by a "_". Examples: take_off(), land()
- The expressions to be evaluated in conditional statements, while loops should be expressions that are evaluated to boolean literals.
- The expressions to be evaluated in repeat loops should be evaluated to integer literals.
- When using operators, expressions that are evaluated into boolean literals should be compared between themselves, expressions that are evaluated into string literals should be compared between themselves and expressions that are evaluated into integer literals should be compared between themselves.

7. Evaluation of doplang in Terms of Readability, Writability and Reliability

As doplang does not necessitate parentheses, but parentheses still can be used if desired, the writability of the language is greatly improved. Furthermore, this also decreases the clutter in the code and improves readability. In addition to these, loop structures, conditional statements and such follow similar designs, all of them necessitate curly brackets containing the statement lists, and the loop declaration / expressions are separated from the statement list by colons. This kind of streamlining increases the writability, and it also removes the ambiguity, improving the readability.

To add to these, because the constructs and the conventions of doplang are so similar to the natural languages, it is intuitive for the programmer, greatly improving its readability and writability compared to non-intuitive languages such as the C-group languages.

In terms of reliability, doplang has several features that compliments it in that aspect. Firstly, as all the built-in functions in the language start with the "doplang." prefix, novice programmer can easily distinguish between "reliable" (built-in) functions and functions he/she should use at her own risk, namely user-defined functions.

Another important aspect of the language in terms of reliability is the fact that each expression in doplang evaluates to a value. Even though the way these expressions are used may be against conventions, still, by this design, we avoid problems such as null pointer exceptions. While it could also be argued that this reduces the "expectedness" of the behaviour of the language, one must consider this in the context the language was designed. As doplang is to be used with drones, it is of utmost importance that the programs do not run into run-time errors, because then, the user loses the control of the drone. This design eliminates a big chunk of run-time errors, errors that are caused by comparisons of incompatible values. From this perspective, this design choice makes doplang reliable for the area it was designed for.