Can Tücer 22203239 Section 1

Artun Berke Gül 22203316 Section 2

Orhun Güder 22202471 Section 2

1. **Name of The Programming Language**

We have decided to name our language as ç++.

1. **BNF Description**

<program> ::= <statement\_list> <program> | <comment\_list> <program> | <comment\_list> | <statement\_list>

<statement\_list> ::= <statement> <SEMICOLON> <statement\_list> | <statement> <SEMICOLON>

<statement> ::= <void\_statement> | <expression>

<comment\_list> ::= <COMMENT> <comment\_list> | <COMMENT>

<void\_statement> ::= <if\_statement> | <loop> | <function\_def> | <type\_def> | <BREAK> | <CONTINUE>

<if\_statement> ::= <IF> <expression> <DO> <LBRACE> <statement\_list> <RBRACE> | <IF> <conditional\_expression> <DO> <LBRACE> <statement\_list> <RBRACE> <ELSE> <DO> <LBRACE> <statement\_list> <RBRACE>

<loop> ::= <while\_loop> | <for\_loop>

<while\_loop> ::= <WHILE> <expression> <DO> <LBRACE> <statement\_list> <RBRACE>

<for\_loop> ::= <FOR> <LP> <assignment\_expression> <SEMICOLON> <expression> <SEMICOLON> <assignment\_expression> <RP> <DO> <LBRACE> <statement\_list> <RBRACE>

<function\_def> ::= <DEF> <VARIABLE> <LP> <variable\_list> <RP> <DO> <LBRACE> <statement\_list> <RETURN> <LP> <expression> <RP> <RBRACE>

<variable\_list> ::= <VARIABLE> <COMMA> <variable\_list> | <VARIABLE> | <EMPTY>

<type\_def> ::= <TYPE> <VARIABLE>

<expression> ::= <conditional\_expression> | <assignment\_expression>

<conditional\_expression> ::= <conditional\_expression> <condition\_operator> <low\_precedence\_arithmetic\_expression> | <LP> <conditional\_expression> <RP> | <BOOLEAN>

<condition\_operator> ::= <LESSER> | <LARGER> | <LESSER\_EQ> | <LARGER\_EQ> | <EQUALS> | <NOT\_EQUALS> | <AND> | <OR> | <XOR>

<assignment\_expression> ::= <type\_def> <ASSIGNMENT> <expression> | <VARIABLE> <ASSIGNMENT> <expression> | <low\_precedence\_arithmetic\_expression>

<low\_precedence\_arithmetic\_expression> ::= <low\_precedence\_arithmetic\_expression> <low\_precedence\_operator> <high\_precedence\_arithmetic\_expression> | <high\_precedence\_arithmetic\_expression>

<high\_precedence\_arithmetic\_expression> ::= <high\_precedence\_arithmetic\_expression> <high\_precedence\_operator> <low\_precedence\_arithmetic\_expression> | <highest\_precedence\_arithmetic\_expression>

<highest\_precedence\_arithmetic\_expression> ::= <highest\_precedence\_operator> <highest\_precedence\_arithmetic\_expression> | <LP> <low\_precedence\_arithmetic\_expression> <RP> | <value>

<low\_precedence\_operator> ::= <MINUS> | <PLUS>

<high\_precedence\_operator> ::= <DIVISION> | <MULTIPLICATION> | <MODULUS>

<highest\_precedence\_operator> ::= <NOT>

<value> ::= <VARIABLE> | <INTEGER> | <FLOAT> | <STRING> | <BOOLEAN> | <function\_call>

<function\_name> ::= <VARIABLE> | <PRIMITIVE\_FUNCTION>

<function\_call> ::= <function\_name> <LP> <expression\_list> <RP>

<expression\_list> ::= <expression> <COMMA> <expression\_list> | <expression> | <EMPTY>

<EMPTY> ::=

1. **General Description** 
   1. **Beginning of Execution**

Execution starts from top and moves down line by line for every program unit. The first function defined in the file is the function ran once the script is executed. Every other function is only executed with function calls.

* 1. **Operator Usage**

Operators with tokens +, -, \*, /, <, <=, >, >=, ==, and, or need to be separated from surrounding tokens with at least one white space character. Operator behaviors and value type compatibility depends on context. Special contexts are:

* 1. If only one side’s type is boolean, TRUE is converted into 1 and FALSE is converted into 0.
  2. If both side’s types are boolean, operations are turned into bitwise operations. + means or, \* means and, - and / means exclusive or.
  3. If only one side’s types is float, the other side’s value is converted into float after other possible conversions are made.
  4. If one side’s type is string and the other’s type is float, float is converted into int.
  5. If one side’s type is string and the other’s type is int, syntax is valid only when usage is like the following: string \* int (repeats the string), string + int (converts int to string, concatenates), string – int (removes characters from end). Otherwise, an error is thrown.
  6. If both side’s types are string, syntax is valid only when the usage is like the following: string + string (concatenates strings),

Operator with token not accepts only one

* 1. **Primitive Functions**

There are some predefined functions in this language. Those functions can be called from every program unit. Names of those functions are reserved and they can’t be used in new function or variable definitions.

* 1. getHeading, getAltitude, getTime, ascend, descend, verticalStop, moveForward, moveBackward, horizontalStop, turnLeft1Deg, turnRight1Deg, enableSpray, disableSpray functions are used to manage the physical activities of drone.
  2. input and output functions are used to manage the keyboard input and console output.
  3. connect, send, request functions are used to manage URL connections and data transactions through network.
  4. **Terminals**
* **program**

This is the starting token for BNF description. It represents the whole file and it is described as the combination of functional statements and documentation comments used within code. Uses left recursion for combinations.

* **statement\_list**

A list of functional statements inside the program.

* **statement**

A statement is the largest functional building element. They can consist of single lines or multiple lines. They are divided into two categories depending on their return value. They should always be closed using a semicolon.

* **void\_statement**

A void statement is a statement that has no return value. It is a set of if statements, for and while loops, function definitions, variable type definitions and some reserved keywords. Those statements can’t be used in places that require a value return.

* **if\_statement**

An if statement is a void statement that uses decision making to run or not run another statement. They take value statements as input to decide whether the given statement will be run or not. They are always initiated by the if keyword. The end of the condition and start of the conditioned statement is initiated by the do keyword. Optionally, an else statement can be used after the if statement. Conditional statements can be nested. The condition is accepted as positive (first given statement is run, else statement is not) when the value statement given as connection is anything but “false” (of Boolean type), 0 (of int type) or 0.0 (of float type).

* **else\_statement**

Used in cases user would like to run a statement when the given value statement is negative. Either can be used with an additional condition (else if) or without any additional condition (else). An else statement ends the else statements, another else if or else statement can’t be used when the associated if statement is already matched with an else statement.

* **loop**

Loops are used to run statements multiple times. There are two types of loops depending on their decision-making processes. Every kind of loop can be nested.

* **while\_loop**

This kind of loop is used when a statement should run continuously as long as a given condition is met. When the statement is done and condition is positive, it will be run again from the top. Similarly to conditional statements, they take value statements as input to decide whether the given statement will be run or not. They are always initiated by the while keyword. The end of the condition and start of the conditioned statement is initiated by the do keyword. The condition is accepted as positive (given statement is run) when the value statement given as connection is anything but “false” (of Boolean type), 0 (of int type) or 0.0 (of float type).

* **for\_loop**

This kind of loop is used when a more advanced control is needed upon the running conditions. Within for loop, a new variable can be defined, a running condition can be specified and a statement that will run after each iteration the inside statement finishes running can be implemented. Everything this statement does can be done using while loops, however for loops provide a more readable and organized structure for specific needs. They are always initiated by the for keyword. The end of the condition and start of the conditioned statement is initiated by the do keyword. The condition is accepted as positive (given statement is run) when the value statement given as connection is anything but “false” (of Boolean type), 0 (of int type) or 0.0 (of float type).

* **function\_def**

A function definition is a void statement that assigns a variable name to a block of code so that it can be used multiple times throughout the program without repeatedly writing the same thing over and over. A function definition is initiated by the keyword “def”, followed by the name of the function, variable\_list in parenthesis, the do keyword, followed by the function itself in curly braces. Within the function definition, the “return” keyword is used to return a value to the function caller.

* **variable\_list**

A variable list is the list of value names that should be passed to a function. It is required in order to declare a function.

* **variable\_list**

An expression list is the list of expressions that are passed to a function. It is required in order to use the previously declared function somewhere else in the program.

* **type\_def**

A type def is a definition of a variable and its type. It is written by a type in front of a variable name.

* **expression**

Expression is the counterpart and the reverse of void statements. It is any statement that has a return value.

* **conditional\_expression**

Conditional expressions are equations that result in boolean values. They are used to compare the values, to check their equity, and for binary operations.

* **assignment\_expression**

Assignment expressions are used to assign values to variables. A variable should either have a previously declared type, or it needs to have the type declaration in the assignment expression. Assignment expressions return the assignment value. Left hand side has to be a variable while right hand side can be any expression.

* **low\_precedence\_arithmetic\_expression**

A low\_precedence\_arithmetic\_expression is one of 2 things: either an arithmetic operation between another low precedence arithmetic expression and a high precedence arithmetic expression using a low precedence operator or it is a high precedence arithmetic expression. It is designed to result in left associativity following real life conventions.

* **high\_precedence\_arithmetic\_expression**

A high\_precedence\_arithmetic\_expression is one of 2 things: either an arithmetic operation between another high precedence arithmetic expression and a highest precedence arithmetic expression using a high precedence operator or it is a highest precedence arithmetic expression. It is designed to result in left associativity following real life conventions.

* **highest\_precedence\_arithmetic\_expression**

A highest precedence arithmetic expression can be a highest precedence arithmetic expression acted on by an operator, a low precedence arithmetic expression in paranthesis, or a value.

* **low\_precedence\_operaton**

A low precedence operator is a group of operators that have lower precedence in arithmetic, like addition and substraction operators.

* **high\_precedence\_operator**

A high precedence operator is a group of operators that have higher precedence than the low precedence operators, like multiplication and division operators.

* **highest\_precedence\_operator**

A highest precedence operator is a group of operators that have precedence over all other defined operators.

* **value**

A value is anything that is returnable. It can be a literal, variable or a function call.

* **function\_call**

A function call is the format to invoke an already defined function. It is the combination of, first a function name and then an expression list filled with expressions matching the pre-defined parameter list.

* **function\_name**

A function name is either a variable that was used in the definition of a function before or a primitive function which is a pre-defined function by the language. It has to be alphanumerical.

1. **Token Definitions**

* **COMMENT**

A comment is a terminal. They are non-functional and are meant to be used for documentation / explanation purposes of the code. Single line comments can be initiated with the // symbol and they can’t be longer than a line. Multiline comments can be initiated with the /\* symbol, and are ended whenever a \*/ symbol is found. Comments can consist of any character.

* **VARIABLE**

A variable is a name that stores data values. The information it holds can be changed or updated. They are declared with a type such as int, string, float. Type determines what type of data a variable hold. They must be alphanumerical. Reserved keywords or operator names can’t be used as variable names.

* **TYPE**

A type defines the data set a variable can hold, and how the operations on that variable are performed. Examples: int, string, float…

* **INTEGER**

Integer is a set of values that represents all integers, e.g. 5, -7, 2 , 0…

* **FLOAT**

Float is a set of values that represents all floating point numbers, e.g. 3.1, -9.7 , 1.12335664…

* **STRING**

String is a set that represents a combination of characters and are enclosed within “”, e.g. “egg”, “apple”, “happiness” … They can consist of any character.

* **BOOLEAN**

Boolean is a value representing either true or false.

* **Reserved Keywords (IF, ELSE, WHILE, FOR, DO, DEF, RETURN, BREAK, CONTINUE)**

These keywords are used special statements. Else can only be used after an If. Return can be only used inside a function. Break and Continue are only usable inside loops.

* **Operators (LESSER, LARGER, LESSER\_EQ, LARGER\_EQ, EQUALS, NOT\_EQUALS, AND, OR, XOR)**

Standard operators are defined as such.