Bilkent University

Department of Computer Engineering

CS315 Project 1

Section 1

Team 13

Bee++

Project Report

Group Members:

Efe Beydoğan 21901548

Emir Melih Erdem 21903100

Bee++

BNF of the Language

<program> ::= <stmts>

<stmts> ::= <stmt>; <stmts> | <stmt>;

<stmt> ::= <matched> | <unmatched>

<matched> ::= if (<expr>) <matched> else <matched> | <other stmt>

<other stmt> ::= <assign stmt> | <loops> | <return stmt> | <block> | <output> | <up function> | <down function> | <vertical stop function> | <forward function> | <backward function> | <horizontal stop function> | <turn right function> | <turn left function> | <nozzle on function> | <nozzle off function> | <connect computer function>

<unmatched> ::= if (<expr>) <stmt> | if (<expr>) <matched> else <unmatched>

<assign stmt> ::= <ident> = <term> | <ident> = <expr> | <declaration stmt>

<declaration stmt> ::= <type> <ident> (= <expr>)?

<expr> ::= <or expr>

<or expr> ::= <or expr> <or op> <and expr> | <and expr>

<and expr> ::= <and expr> <and op> <relational expr> | <relational expr>

<relational expr> ::= <relational expr> <relational op> <equality expr> | <equality expr>

<equality expr> ::= <equality expr> <equality op> <additive expr> | <additive expr>

<additive expr> ::= <additive expr> <additive op> <multiplicative expr> | <multiplicative expr>

<multiplicative expr> ::= <multiplicative expr> <multiplicative op> <parenthesized expr> | <parenthesised expr>

<parenthesized expr> ::= (<expr>) | <term>

<loops> ::= while (<expr>) <stmt> | for (<assign>?; <expr>?; <assign>?) <stmt>

```
<block> ::= { <stmts>? }

<function definition> ::= <type> <ident> ( <parameters>? ) <stmt>

<parameters> ::= <parameter> | <parameters>, <parameter>

<parameter> ::= <type> <ident>

<function call> ::= <ident> ( <arguments>?)

<arguments> ::= <expr> | <arguments>, <expr>

<return stmt> ::= return <expr>?;

<term> ::= <ident> | <function call> | <int const> | <char const> | <double const> | <input> |
           <heading> | <altitude> | <temperature> | <vertical speed function> | <horizontal
           speed function> | <speed function> | <go distance function>

<or op> ::= ||

<and op> ::= &&

<relational op> ::= < | <= | > | >=

<additive op> ::= + | -

<multiplicative op> ::= * | / | %

<equality op> ::= == | !=

<type> ::= void | int | char | double | string

<int const> ::= <sign>? <unsigned int>

<unsigned int> ::= <unsigned int> <digit> | <digit>

<sign> ::= + | -

<string const> ::= “<string>”

<string> ::= <string> <character> | <character>

<double const> ::= <sign>? <unsigned int>?.<unsigned int>
```

```
<char const> ::= ‘<character>’

<ident> ::= <letter> | <ident> <alphanumeric character>

<character> ::= <symbol> | <alphanumeric>

<alphanumeric> ::= <letter> | <digit>

<symbol> ::=  ! | " | # | $ | % | & | ' | ( | ) | * | + | , | - | . | / | : | ; | < | = | > | ? | @ | [ | \ | ] | ^ | _ | ` | { |
              | | } | ~

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<letter> ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y |
             Z | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z

<heading> ::= \<heading\>

<altitude> ::= \<altitude\>

<temperature> ::= \<temperature\>

<up function> ::= up()

<down function> := down()

<vertical stop function> ::= verticalStop()

<forward function> ::= forward()

<backward function> ::= backward()

<horizontal stop function> ::= horizontalStop()

<turn right function> ::= right()

<turn left function> ::= left()

<nozzle on function> ::= nozzleOn()

<nozzle off function> ::= nozzleOff()

<connect computer function> ::= connect()
```

<vertical speed function> ::= verticalSpeed()

<horizontal speed function> ::= horizontalSpeed()

<speed function> ::= speed()

<go distance function> ::= goDistance(<expr>)

<input> ::= in()

<output> ::= out(<expr>) | out(<string>)

<comment> ::= \? <string> \?

Explanations of Non-Terminals

<program>
This is the starting non-terminal. It includes statements that form the whole language.

<stmts>
This non-terminal is defined recursively and is right recursive. It is either a single statement followed by a semicolon or a cascade of statements that each end with a semicolon.

<stmt>
This is a single statement and includes all the statements that make up the language. It is evaluated as either a matched or an unmatched statement. We employed this design pattern of either “matched” or “unmatched” statements to avoid the dangling else problem which makes a language ambiguous. By determining whether a statement is a matched if statement or not at the beginning, we intended to make our language clearer by eliminating ambiguity.

<matched>
This non-terminal contains the matched if statement (“if” with an “else”, both having statements that are matched) and all the other statements including unconditional statements and loops.

<unmatched>
This non-terminal includes unmatched if statements. An unmatched if statement is either an “if” without an “else” or an “if” with a matched statement followed by an “else” with an unmatched statement.

<other stmt>
This non-terminal includes all the statements that aren’t if statements, including assign statements, loops, return, built-in output statement, blocks, and the primitive functions required for the drones.

<assign stmt>
The assign statement is a very basic construct of our language. An identifier can be assigned either the value of another term (an identifier, int, double, etc.) or the result of an expression. The declaration statement is also included in this non-terminal.

<declaration stmt>
The declaration statement is similar to the declaration statements in other popular programming languages. Namely, first a type for the variable is needed, then the name of the variable, and then an assignment operator followed by the value that the user wants to assign to the variable is needed. However, an assignment is optional when declaring a variable and can be made later via an assignment statement as well.

<expr>
This is the non-terminal that includes all expressions in our language. In the first stage, it evaluates to the logical or operation (||), which hierarchically can be evaluated into other expressions. The or expression is at the top of the hierarchy since it is the operation with the least precedence in our language. We employed this hierarchical design pattern to ensure correct operator precedence so the expressions entered by the users are evaluated according to appropriate logical and mathematical rules.

<or expr>
The or expression is the logical or operation (||). To abide by the operator's left associativity, we made the rule left recursive. It evaluates to either more logical or operations followed by logical and operations or directly evaluates to a logical and operation, which has higher precedence.

<and expr>
The and expression is the logical and operation (&&). It is left-associative, so the rule in the BNF is left-recursive. It is followed in precedence by the relational expression so either evaluates to a relational expression or logical and operations followed by relational expressions.

<relational expr>
Relational expression encompasses the operations defined by <, >, <= or >=. It is not associative, however, the rule is still recursive and accommodates cascading several of these operators together. It is followed in precedence by the == operator.

<equality expr>
Equality expression (== or !=) is used to check if two terms are equal to each other or not. It takes terms on both sides. It is left recursive. It is evaluated as either a cascade of equality expressions followed by an additive expression or just an additive expression.

<additive expr>
The additive expression involves operations defined by either the addition (+) or subtraction (-) operators in mathematics. We implemented it as left-associative in our language, thus the rule is left recursive.

<multiplicative expr>
Multiplicative expression is the operation with the highest precedence in our language, it involves operations made by *, / or % operators. It is left-associative, so the rule for it is left recursive.

<parenthesized expr>
Parenthesized expression has the highest precedence among other expressions in our language. It is the last step for any mathematical or logical operations made in our language so it evaluates to an expression that is parenthesized or a single term.

<loops>
Loops in our language involve only the traditional while and for loops. “while” is a reserved word in our language and this word is followed by an expression in parentheses. It runs a statement until the expression inside the parentheses evaluates to “false”. The for loop takes three optional statements inside it, namely an assign statement followed by an expression followed by another assign statement and runs a statement until the expression inside it evaluates to “false”.

<block>
A block in our language is a set of grouped statements inside curly brackets. It acts as a single statement.

<function definition>
A function definition is the same as in Java or C++. A type is followed by a name of the function and then it takes optional parameters, with a statement following them. For example; int foo(double a) {a = a + 1; return a;}.

<parameters>
The parameters non-terminal is a recursive list defining the parameters taken by a function. It is either a single parameter or a set of parameters.

<parameter>
A parameter is a type name followed by an identifier. I.e “int myInt”.

<function call>
The function call non-terminal is defined for the users to be able to call a function they wrote. It requires an identifier which is the name of the function, followed by a set of optional arguments inside parentheses, as the called function requires them.

<arguments>
Arguments are either single expressions or a set of expressions to be passed into function calls.

<return stmt>
The return statement is the same as any return statement in Java or C++. It is the “return” keyword followed by an optional expression. It is used in function definitions. The expression is optional, allowing the statement “return;” to be valid, which can be used inside void functions to stop their execution.

<term>
A term is a set of the most fundamental data blocks in our language. It includes identifiers; integer, double, and char constants; function calls and the input function with returned values; as well as the built-in measurements of heading direction, altitude of the drone, and air temperature.

<or op>
This non-terminal defines the logical or operator ||.

<and op>
The logical and operator &&.

<relational op>
This non-terminal includes the operator <, >, <= and >=.

<additive op>
The additive operators are + and -.

<multiplicative op>
The multiplicative operators are *, / and %.

<equality op>
Equality operators are == and !=, which stand for equals and not equals, respectively.

<type>
The primitive types in our language are void, int, char, double, and string.

<int const>
Integer constants in our language are unsigned integers that optionally have signs in front of them.

<unsigned int>
Unsigned integers are defined recursively as a set of <digit> nonterminals.

<sign>
This nonterminal defines the sign symbols for integer constants. It is evaluated as either + or -.

<string const>
The string constant nonterminal is defined as any <string> nonterminal inside quotes (“”).

<string>
This nonterminal defines a string recursively as a set of <character> nonterminals. The <character> nonterminal includes all the symbols, digits, and letters.

<double const>
This nonterminal defines a double constant in our language. It is an optional sign, an optional unsigned integer followed by a “.” and again an unsigned integer So examples of a double constant in our language include “1.7”, “.2”, and “-18.141”.

<char constant>
The char constant nonterminal is any symbol, digit, or letter inside apostrophes (‘ ‘).

<ident>
This nonterminal defines an identifier in our language. An identifier is required to start with a letter and can optionally be followed by a set of alphanumeric characters, meaning an identifier can contain any number of letters and digits, as long as it starts with a letter. As there are no symbols included in the formation of an identifier e.g “_”, the associated convention for identifiers is the camel case.

<character>
A character in our language is either a symbol or a digit or a letter.

<alphanumeric>
This nonterminal defines a single alphanumeric character as a letter or a digit.

<symbol>
This nonterminal includes all the ASCII symbols.

<digit>
This nonterminal defines all the digits 0-9.

<letter>
All the letters in the English alphabet, upper and lowercase.

<heading>
This non-terminal defines the special keyword “\<heading\>” in our language. This keyword stands for the heading direction of the drone and is used to read the heading.

<altitude>
The altitude non-terminal defines the special keyword “ \<altitude\>” which is used in our language to read the altitude of the drone.

<temperature>
The temperature non-terminal defines the special keyword “\<temperature\>” which is used in our language to read the temperature.

<up function>
This non-terminal is the rule for the special “up()” function in our language. Upon calling, this function moves the drone upward at a rate of 0.1 m/s.

<down function>
This non-terminal is the rule for the special “down()” function in our language. Upon calling, this function moves the drone downward at a rate of 0.1 m/s.

<vertical stop function>
This non-terminal is the rule for the special “verticalStop()” function in our language. Upon calling, this function stops the movement of the drone in the vertical direction.

<forward function>
This non-terminal defines the special function “forward()”. As a result of calling this primitive function, the drone starts moving forward at a rate of 1 m/s.

<backward function>
This non-terminal defines the special function “backward()”. As a result of calling this primitive function, the drone starts moving backward at a rate of 1 m/s.

<horizontal stop function>
This non-terminal is the rule for the special “horizontalStop()” function in our language. Upon calling, this function stops the movement of the drone in the horizontal direction.

<turn right function>
This non-terminal defines the “right()” function in our language. This function is called to turn the direction of the drone to right by an angle of 1 degree.

<turn left function>
This non-terminal defines the “left()” function in our language. This function is called to turn the direction of the drone to left by an angle of 1 degree.

<nozzle on function>
This non-terminal defines the special “nozzleOn()” function, which is used to turn on the nozzle of the drone for spraying the chemical in its tank.

<nozzle off function>
This non-terminal defines the special “nozzleOff()” function, which is used to turn off the nozzle of the drone for stopping the spraying of the chemical in its tank.

<connect computer function>
This non-terminal defines the special function “connect()” which is called by the user to connect to the base computer via wifi.

<vertical speed function>
This non-terminal defines the special function “verticalSpeed()” which is called by the user to get the current speed in the vertical direction, positive or negative depending on whether the drone is going up or down.

<horizontal speed function>
This non-terminal defines the special function “horizontalSpeed()” which is called by the user to get the current speed in the horizontal direction, positive or negative depending on whether the drone is going forward or backward.

<speed function>
This non-terminal defines the special function “speed()” which is called by the user to get the current speed of the drone. This built-in function computes speed by bringing into account the movements in both horizontal and vertical directions.

<go distance function>
This non-terminal defines the special function “goDistance()” which takes an <expr> as a parameter and moves forward by that amount.

<input>
This nonterminal defines the in() function, which is used for taking user input.

<output>
This nonterminal defines the out() function, which takes either an <expr> or <string> as a parameter and outputs it to the console.

<comment>
This nonterminal defines how comments are written in our language. A comment statement in our language is any <string> nonterminal between two question marks (? … ?).

Descriptions of Nontrivial Tokens

Comments:

In our language, we defined comments as any expression between two question marks (? … ?). Our motivation in doing so was to ensure that users could write explanatory statements for their code whenever they needed it. The presence of comments contributes to the simplicity of our language since there is only one way to create a comment. As a result of the added simplicity, the readability of the language increases because complicated code segments can be easily explained with the addition of a comment block. Also, there is only one way to write comments so feature multiplicity isn't high. The writability of the language also becomes better due to the ease of writing comments.

Identifiers:

Identifiers in our language follow the same rules as in Java. Namely, as we defined it, an identifier is required to start with a letter and then can be followed by any number of alphanumeric characters. Our motivation in doing so was to make our language instantly familiar to programmers and provide easy usage. We tried to keep the rules for using identifiers as simple as possible to maximize the writability of our language. The only forced rule when creating identifiers in our language is to start with a letter, making this construct simple enough to use, thus boosting the readability as well.

Literals:

Integer constants:

Integer constants in our language are defined recursively as any combination of digits that optionally starts with a sign. This is simple for the sake of readability, an integer constant can be defined as in any other language. There are different data types for integers and doubles in our language to eliminate reliability issues that could have been caused by a common type for both data types.

Double constants:

Double constants are defined in our BNF as "<sign>? <unsigned int>?.<unsigned int>". They require an optional sign, an optional integer constant before the dot, and another integer constant after. This way, both "3.7" and ".7" are regarded as double constants in our language. By having both integer and double constants, we aimed to increase writability for the users with the addition of fundamental data types. Also, since there are only two ways to define a double constant, the readability doesn't get complicated either and a double constant can easily be identified when reading the code.

Char constants:

Char constants are any single letter, digit, or symbol between two apostrophes (' '). This follows the same convention as in C. It is easy to identify char constants in a code segment written in our language, so the way we define char constants positively affects the readability and writability of our language. Since as in C, the char constants in our language also can be interpreted as integers, making operations on characters (e.g comparison, addition) possible and thus, increasing the writability of our language. However, at the same time, this might cause reliability issues - negatively affecting overall readability.

String constants:

Our motivation for adding the string data type was mainly to make sentential outputs to the console possible. This way, the outputs of the programs can be easier to read and understand. String constants in our language are any combination of the <character> nonterminal between quotes (“ “). The <character> nonterminal includes symbols, letters, and digits in ASCII. This way, users can include any character they may need in a string constant, and the writability of our language increases as a result.

Reserved words:

Despite the special reserved words added as part of the drone requirements, the reserved words in the Bee++ language are mostly similar to the ones in Java or C++. The reserved words are defined as descriptive and helpful to users as possible to increase writability and readability by keeping the language simple. The list of reserved words in our language is as follows:

if: Used in regular if statements.
else: Used in regular if-else statements.
while: Used when writing a while loop, same as in Java.
for: Used when writing a for loop, the for loop structure in our language is the traditional one with definition for(…;...;...).
return: Used inside functions to return a value.
void: The void type to be used when a function has no return value.
int: Integer data type, used when declaring integers.
double: Double data type, used when declaring doubles.
char: Char data type, used when declaring chars.
string: String type, used when declaring strings.
<heading>: The <heading> reserved word is used to read the heading of the drone. The name of the variable as well as the convention of angled brackets were chosen for better readability. The properties which give information about the drone are enclosed in angled brackets for users to easily remember drone-related attributes and write code smoothly.
<altitude>: The <altitude> reserved word is used to read the altitude of the drone.
<temperature>: The <temperature> reserved word is used to read the temperature of the air around the drone.
up(): The up() reserved word is used to make the drone climb up with a speed of 0.1 m/s. The functions that change the movement of the drone are defined in our language as reserved words with descriptive yet simple names that have parentheses (e.g. funcName()) at the end to reflect that they can be used to alter the movement of the drone.
down(): The down() reserved word is used to make the drone climb down with a speed of 0.1 m/s.

verticalStop(): The verticalStop() reserved word is a special function in our language that can be used to stop the movement of the drone in the vertical direction.
forward(): This reserved word is used to move the drone forward at a speed of 1 m/s.
backward(): This reserved word is used to move the drone backward at a speed of 1 m/s.
horizontalStop(): This reserved word is used to stop the movement of the drone in the horizontal direction.
right(): Special function to turn the heading of the drone 1 degree to the right.
left(): Special function to turn the heading of the drone 1 degree to the left.
nozzleOn(): This reserved word is used to turn the nozzle of the drone on.
nozzleOff(): This reserved word is used to turn the nozzle of the drone off.
connect(): This reserved word is used to connect to the base computer via wifi.
verticalSpeed(): This reserved word is used to get the current speed value of the drone in the vertical direction.
horizontalSpeed(): This reserved word is used to get the current speed value of the drone in the horizontal direction.
speed(): This reserved word is used to get the current speed value of the drone.
goDistance(): This reserved word is used to move the drone forward by the given amount.
in(): Special word to take input from the user
out(): Special function to output something to the console, takes either an expression or a string as a parameter

Evaluation of the Language

Readability:
We’ve added comments in our language which can be used by programmers to explain complex code and avoid confusion. Feature multiplicity for comments isn’t high since there is only one way to write comments in the language. Feature multiplicities of other constructs in our language also aren't so high either, since the language doesn’t support too many different ways of doing operations, such as “x++” or “x += 1” etc., contributing to higher readability. Following this, the identifiers in our language also abide by a simple rule of starting with a letter and they can only consist of alphanumeric characters, hence increasing readability. The control statements in our language, namely the for and while loops, are not different from the traditional ones and they are simple enough to use, with familiar reserved words. There are different data types for integer and double constants, so they aren’t represented with the same type. However, there is no boolean data type and as in the C language, non-zero numbers are interpreted as true in our language. This is a potential threat to readability since it could be hard to perceive logical statements. We have defined reserved words for if/else statements, loops, function return statements, data types, and built-in drone properties and functions. The addition of these reserve words increases readability by clarifying the important functionalities that are used in the code to manipulate the drone’s movement.

Writability:
In our language, we have a number of reserved words and built-in functions for controlling the drone as well as receiving input/giving output. None of the internal workings of these functions are reflected to the user and the users can freely use them without being familiar with their structure. This kind of abstraction is useful for better writability since programmers are able to write code with readily defined functions for their fundamental needs to control the drone. The statements in our language end with a semicolon, just like in C or Java, and the logical/arithmetic operators are used in the same ways as their traditional usages in mathematics and other common programming languages. As a result, we have tried our best to keep the writability of our language at a maximum by employing design patterns that we thought would best fit programmers’ needs and expertise when writing code.

Reliability:
To make the language as reliable as possible, we put special attention into describing mathematical and logical operators so the precedence and associativity of these operators are in the right order. We used recursive rules to prevent ambiguity as much as we can. However, when describing relational operations (<, >, <=, >=) and equality operations (==, !=), we still used recursive rules in the BNF, although these operators are not associative. As a result, the relational and equality operators can be wrongfully cascaded (1 < 2 < 3 or 1 == 2 == 3). This poses the risk of leading to reliability issues since the users have to be aware that cascading these operators is not logical, nonetheless this is allowed in the Bee++ language. Also, as in the C language, chars are also integers in our language, so they can be used interchangeably since there is no type checking, which is yet another threat to reliability. The lack of type checking and

exception handling reduces reliability, however, we tried our best to minimize the number of ambiguities and implemented logical/mathematical operations in their correct ways to make the language as reliable as possible.