Bilkent University

Department of Computer Engineering

CS315 Project 2

Section 1

Team 13

Bee++

Project Report

Group Members:

Efe Beydoğan 21901548

Emir Melih Erdem 21903100

Bee++

BNF of the Language

<program> ::= START <stmts> END

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

<stmt> ::= <conditional stmt> | <other stmt> SEMICOLON | <loops> | <block> | <main function>
| <function definition> | <function call> SEMICOLON

<conditional stmt> ::= IF LPARENTHESIS <expr> RPARENTHESIS <block>
| IF LPARENTHESIS <expr> RPARENTHESIS <block> ELSE block
| IF LPARENTHESIS <expr> RPARENTHESIS <block> <conditional elseif stmts>
| IF LPARENTHESIS <expr> RPARENTHESIS <block> <conditional elseif stmts>
ELSE block

<conditional elseif stmts> ::= ELSEIF LPARENTHESIS <expr> RPARENTHESIS <block>
| ELSEIF LPARENTHESIS <expr> RPARENTHESIS <block> <conditional elseif
stmts>

<other stmt> ::= <assign stmt> | <return stmt> | <output> | UP_FUNCTION |
DOWN_FUNCTION | VERTICAL_STOP_FUNCTION | FORWARD_FUNCTION |
BACKWARD_FUNCTION | HORIZONTAL_STOP_FUNCTION |
RIGHT_FUNCTION | LEFT_FUNCTION | NOZZLE_ON_FUNCTION |
NOZZLE_OFF_FUNCTION | CONNECT_FUNCTION | <go distance function>

<main function> ::= MAIN LPARENTHESIS RPARENTHESIS <block>

<assign stmt> ::= IDENTIFIER ASSIGNMENT_OP <expr> | <declaration stmt>

<declaration stmt> ::= <type> IDENTIFIER | <type> IDENTIFIER ASSIGNMENT_OP <expr>

<expr> ::= <or expr>

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

<and expr> ::= <and expr> LOGICAL_AND <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> |
<parenthesized expr>

<parenthesized expr> ::= LPARENTHESIS <expr> RPARENTHESIS | <term>

<loops> ::= WHILE LPARENTHESIS <expr> RPARENTHESIS <block>
| FOR LPARENTHESIS <assign stmt> SEMICOLON <expr> SEMICOLON
<assign stmt> PARENTHESIS <block>
| FOR LPARENTHESIS <assign stmt> SEMICOLON <expr> SEMICOLON
RPARENTHESIS <block>
| FOR LPARENTHESIS <assign stmt> SEMICOLON SEMICOLON <assign
stmt> RPARENTHESIS <block>
| FOR LPARENTHESIS <assign stmt> SEMICOLON SEMICOLON
RPARENTHESIS <block>
| FOR LPARENTHESIS SEMICOLON <expr> SEMICOLON <assign stmt>
RPARENTHESIS <block>
| FOR LPARENTHESIS SEMICOLON <expr> SEMICOLON RPARENTHESIS
<block>
| FOR LPARENTHESIS SEMICOLON SEMICOLON <assign stmt>
RPARENTHESIS <block>
| FOR LPARENTHESIS SEMICOLON SEMICOLON RPARENTHESIS <block>

<block> ::= LBRACE <stmts> RBRACE | LBRACE RBRACE

<function definition> ::= <type> FUNCTION_IDENTIFIER LPARENTHESIS <parameters>
RPARENTHESIS <block> | <type> FUNCTION_IDENTIFIER LPARENTHESIS
RPARENTHESIS <block>

<parameters> ::= <parameter> | <parameters> COMMA <parameter>

<parameter> ::= <type> IDENTIFIER

<function call> ::= FUNCTION_IDENTIFIER LPARENTHESIS <arguments> RPARENTHESIS |
FUNCTION_IDENTIFIER LPARENTHESIS RPARENTHESIS

<arguments> ::= <expr> | <arguments> COMMA <expr>

<return stmt> ::= RETURN <expr> | RETURN

<term> ::= IDENTIFIER | <function call> | INTEGER_CONST | CHAR | DOUBLE_CONST |
BOOLEAN_CONST | STRING | INPUT | GET_HEADING | GET_ALTITUDE |
GET_TEMPERATURE | VERTICAL_SPEED_FUNCTION |
HORIZONTAL_SPEED_FUNCTION | SPEED_FUNCTION

<relational op> ::= LESS_THAN | LESS_OR_EQUAL | GREATER_THAN |
GREATER_OR_EQUAL

<additive op> ::= ADDITION_OP | SUBTRACTION_OP

<multiplicative op> ::= MULTIPLICATION_OP | DIVISION_OP | MODULO_OP

<equality op> ::= EQUALITY_OP | NOT_EQUAL_OP

<type> ::= VOID_TYPE | INT_TYPE | CHAR_TYPE | DOUBLE_TYPE | STRING_TYPE |
BOOLEAN_TYPE

<go distance function> ::= GO_DISTANCE_FUNCTION LPARENTHESIS <expr>
RPARENTHESIS

<output> ::= OUTPUT LPARENTHESIS <expr> RPARENTHESIS

Description of the Structure of the Language

Every program written in the Bee++ language must start with the **start** keyword and end with the **end** keyword. Any statements that are legal in the language must be written between these two keywords. Also, every statement must be followed by a semicolon (;). The first statement included in Bee++ is the traditional conditional (if-else) statement. This statement starts with a single **if** keyword followed by an expression between parentheses and a block. If necessary, it can be paired either with a set of **elseif** keywords each followed by expressions in parentheses and blocks, with an **else** keyword followed by another block, or with both of them. In the Bee++ language, in order to avoid conflicts in yacc, we have only allowed blocks, namely statements between curly braces, to follow any conditional statement or the main function etc. The language features **for** and **while** loops as well. These loops are the same as they are in either the Java or C++ language. The three statements in the for loop (… ; …. ; …) are all optional and both of these loops must be followed by a block. A **block** is any set of statements between curly brackets, or it could be left empty. There is a **main** function in the language. The main function is used as follows: main() {...}. The next statement is the **function definition**. Function definitions in the language must start with a type, followed by a **function identifier**, then any number of **parameters** between parentheses and then a block. Parameters are either a single parameter with a type followed by an identifier, or a comma separated list of the same structure. There may not be any parameters as well. The function identifiers in Bee+ must start with "f_" followed by any legal **identifier**. Legal identifiers are ones that strictly start with a letter and comprise only letters or digits afterward. **Comment lines** in the language are any expression between two question marks (? … ?).

The **assign statement** is any identifier followed by the **assignment operator (=)** and then a term. Alternatively, on the right hand side of the assignment operator an expression could be placed. A **term** is any identifier, function call, integer, char, double, boolean constant, string or any of the built-in functions that return a value (**in()**), heading (**<heading>**), altitude (**<altitude>**), temperature (**<temperature>**), vertical speed (**verticalSpeed()**), horizontal speed (**horizontalSpeed()**) and the overall speed (**speed()**). The **return statement** is the traditional return statement in popular programming languages. It is either the **return** keyword followed by any expression or just the return keyword. The **output** statement is a built-in function that programmers may use to output any expression including strings to the console (**out(...)**). The remaining statements in the language are the built-in functions for making the drone go up (**up()**), down (**down()**), stop vertically (**verticalStop()**), go forward (**forward()**), go backward (**backward()**), stop horizontally (**horizontalStop()**), turn right (**right()**), turn left (**left()**), turn the nozzle on (**nozzleOn()**), turn the nozzle off (**nozzleOff()**), connect to the main computer (**connect()**) and go a specified distance (**goDistance(...)**). The goDistance(...) method takes any expression as a parameter to make the drone go forward for a given distance.

The expressions in Bee++ start with the **logical or operation (||).** This expression is left recursive, thus left associative, and has the lowest precedence amongst expressions. It is followed by the **logical and operation (&&)**, which is also left associative. The next expression

is the **relational expression**. The relational operations are expressed by using any of the symbols <, <=, > or >=. Although relational operators are nonassociative, the rule for them is left recursive to maintain the precedence of operators and avoid conflicts and/or ambiguities. Relational expressions are followed by the **equality expression,** which has higher priority. The equality operators are == and !=, they are again left associative and are followed by the **additive expressions** in our language. The additive operators are + and -, and they are left associative. The following expression is the **multiplicative expression** which is also left associative and includes the operations that are carried out by the signs *, / and %. The expressions with the highest priority in our language are the **parenthesized expressions**. Parenthesized expressions are any expression inside parentheses **()**.

Explanations of Non-Terminals

<program>
This is the starting non-terminal. It includes statements that form the whole language. It evaluates to the <stmt> nonterminal between the keywords “start” and “end”. Any legal statement in a program written in our language must be between these two keywords.

<stmts>
This non-terminal is defined recursively and is right recursive. It is either a single statement or a cascade of statements.

<stmt>
This is a single statement and includes all the statements that make up the language. The conditional statements, as well as for/while loops, the main function and function definitions are all included in this nonterminal. It also includes the <other stmt> nonterminal.

<conditional stmt>
The regular if-else statement. Includes a single if statement, an if-else block, or an if followed by an elseif block and an optional else statement.

<conditional elseif stmts>
This nonterminal contains the statements written with the “elseif” keyword. It is either a single elseif block or a cascade of elseif statements.

<main function>
The regular main function, used as follows: main() {...}.

<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.

<other stmt>
This non-terminal is evaluated to either an assignment statement, return statement, or any of the built in functions for making the drone go up (**up()**), down (**down()**), stop vertically (**verticalStop()**), go forward (**forward()**), go backward (**backward()**), stop horizontally(**horizontalStop()**), turn right (**right()**), turn left (**left()**), turn the nozzle on

(**nozzleOn()**), turn the nozzle off (**nozzleOff()**), connect to the main computer (**connect()**) and go a specified distance (**goDistance(...)**). The goDistance(...) method takes any expression as a parameter to make the drone go forward for a given distance. Additionally, the **out(...)** function for outputting an expression or a string is also included in this nonterminal.

<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”. Both of these loops must be followed by a block.

<block>
A block in our language is a set of grouped statements inside curly brackets. It acts as a single statement. The inside of the braces can also be left empty.

<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 f_foo(double a) {a = a + 1; return a;}. The function names are supposed to start with “f_”.

<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 <parameter>s.

<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 a function 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 a single 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, char, boolean and string constants; function calls and the input function with returned values; as well as the built-in measurements of heading direction, altitude of the drone, air temperature, vertical speed, horizontal speed and the overall speed.

<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, string and boolean..

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

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

Descriptions of Nontrivial Tokens

Tokens used in the BNF:

START: This token is the token for the special word “start” in our language. This word precedes any statement written in any program.

END: This token is the token for the special word “end” in our language. Every program written in our language must be concluded with this word.

COMMENT_LINE: This token is to identify comments in Bee++. Comments are any expression between question marks (? … ?). Our lex specification identifies any expression between question marks as a comment line.

MAIN: The MAIN token stands for the “main” reserved word in the language. It is used to write a main function. It must be followed by parentheses and a block (main() {}).

GET_HEADING: This token is the built-in expression for getting the heading of the drone, called as <heading>.

GET_ALTITUDE: This token is the built-in expression for getting the altitude of the drone, called as <altitude>.

GET_TEMPERATURE: This token is the built-in expression for getting the temperature value, called as <temperature>.

UP_FUNCTION: This token is the built-in function for making the drone go up at a rate of 0.1 m/s, called as up().

DOWN_FUNCTION: This token is the built-in function for making the drone go down at a rate of 0.1 m/s, called as down().

VERTICAL_STOP_FUNCTION: This token is the built-in function for making the drone stop vertically, called as verticalStop().

RIGHT_FUNCTION: This token is the built-in function for making the drone turn right for 1 degree, called as right().

LEFT_FUNCTION: This token is the built-in function for making the drone turn left for 1 degree, called as left().

FORWARD_FUNCTION: This token is the built-in function for making the drone go forward at a rate of 1 m/s, called as forward().

BACKWARD_FUNCTION: This token is the built-in function for making the drone go backward at a rate of 1 m/s, called as backward().

HORIZONTAL_STOP_FUNCTION: This token is the built-in function for making the drone stop horizontally, called as horizontalStop().

NOZZLE_ON_FUNCTION: This token is the built-in function for turning the nozzle of the drone on, called as nozzleOn().

NOZZLE_OFF_FUNCTION: This token is the built-in function for turning the nozzle of the drone off, called as nozzleOff().

CONNECT_FUNCTION: This token is the built-in function for connecting to the main computer, called as connect().

SPEED_FUNCTION: This token is the built-in function for getting the speed of the drone, called as speed().

VERTICAL_SPEED_FUNCTION: This token is the built-in function for getting the vertical speed of the drone, called as verticalSpeed().

HORIZONTAL_SPEED_FUNCTION: This token is the built-in function for getting the horizontal speed of the drone, called as horizontalSpeed().

GO_DISTANCE_FUNCTION: This token is the built-in function for making the drone move a specified distance, the token identifies the word “goDistance”, since the remaining part of the method is a regular function call with parentheses and an expression in between.

INPUT: Special function for taking input from the user, called as in().

OUTPUT: This token identifies the “out” keyword, the remaining part of the method is a regular function call with parentheses and an expression in between, to be output to the console.

FUNCTION_IDENTIFIER: Any identifier preceded by “f_”. It is used to specify defined functions’ names, and can be seen in function definitions and function calls.

Comments:

In our language, we define 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. Additionally, **function identifiers** are any legal identifiers preceded by "f_". This allows for easily separating regular identifiers from function names in lex.

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.

Boolean constants:

Boolean constants are regular boolean values of “true” and “false”.

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:

start: Used to define the start of the program.
end: Used to define the end of the program.
main: Used to define a main function.
if: Used in regular if statements.
else: Used in regular if-else statements.
elseif: Used in regular else if 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.
boolean: Boolean type, used when declaring boolean values.
true: True boolean value.
false: False boolean value.
<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. It takes the distance the drone should move as a parameter inside the parentheses.
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