Name	Student ID	Section
Arda Güven Çiftçi	21801763	3
Erkin Aydın	22002956	3
Yağız Berk Uyar	21902318	1

BilScript Language

Complete BNF Description

I. PROGRAM

```
<program> ::= <BP><statement_list><EP>
<statement_list> ::=<statement> | <statement> <statement_list>
<statement> ::= <comment>
 | <if_else>
 | <assignment>
 | <loop>
 | <output>
 | <initialize_list>
 | <function_list>
 | <primitive_function_list>
 | <early_return>
<comment> ::=<comment_op> <word_list>
```

II. LOOP

```
<loop>::= <while_statement> | <for_statement>
<while_statement>::=<while> <LP> <logic_list> <RP> <LB> <statement_list> <RB>
<for_statement>::=<for> <LP> <initialize_list> <semicolon><logic_list> <semicolon>
<assignment_list> <RP>  <LB> <statement_list> <RB>
```

III. INPUT AND OUTPUT

```
<input_expression>::= cin<IOR><param>

<output_expressions_list> ::= <IOL><output_expressions>
                              | <IOL><output_expressions><output_expressions_list>
<output_expressions>::= <var>
                              |  <algebra_list>
                              | " <word_list> "
<output_expression>::=cout<output_expressions_list>
```

IV. TYPES

```
<var_dec>::=<variable_type><word><semicolon>

<variable_type> ::= <int_dec>
                  | <char_dec>
                  | <string_dec>
                  | <bool_dec>
                  | <float_dec>

<bool_dec>::= bool
<int_dec>::= int
<string_dec>::= string
<char_dec>::= char
<float_dec>::= float
<void_dec>::= void
<sign>::=  +| -
<var> ::=  <char> | <int> | <bool> | <float> | <string>
<bool> ::= true | false
<int> ::= 0 | <non_zero><digit_list>
<digit_list>::= <digit> | <digit><digit_list>
<signed_int> ::= <sign><int>
<digit> ::= 0 | <non_zero>
<early_return> ::= | <return> <semicolon>
<non_zero> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<param_int> ::= <int> | <word>
<float>::=<signed_int>.<digitlist> | <int>.<digit_list> | .<digit_list>
<char>::=‘<letter>’ | ’<digit>’
<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
<word>::= <char> | <char><word>
<word_list>::= <word> | <word><space_list><word_list>
<space_list>::= <space> | <space> <space_list>
```

V. CONDITIONAL STATEMENTS

```
<arith_op1> ::= <arithmetic_op2>
              | <arithmetic_op1> + <arithmetic_op2>
              | <arithmetic_op1> - <arithmetic_op2>
<arith_op2> ::= <fact>
              | <arith_op2> * <fact>
              |<arith_op2> / <fact>
<fact> ::= <var>
         | <arith_op1>
```

```
<algebra_list> :: = <arith_op1> | <arith_op2>
<alg_eq> ::= <arith_op><assignment_operator>
<and_or> ::= <and> | <or>
<logic_list> ::= <logic> | <logic>  <and_or> <logic_list> ( a>0 && b>1 || c < 1 && a==b)
<logic> ::= <not> <logic> | <bool> | <int> <logic_op><int>
<logic_op> ::= <gt> | <gt_eq> | <lt> | <lt_eq> | <eq> | <not_eq>


<if_else> ::= <matched> | <unmatched>
<matched> ::= if <LP> <logic_list> <RP><LB> <matched><RB> else<LB>
<matched><RB> | <statement_list>
<unmatched> ::= if<LP> <logic_list><RP> <LB> <statement_list><RB>
          | if <LP> <logic_list><RP> <LB> <matched> <RB> else <LB> <unmatched> <RB>
```

VI. INITIALIZE AND ASSIGN STATEMENTS

```
<initialize_list>::= <init> <semicolon> | <init> <comma> <initialize_list>

<init> ::= <int_init> | <char_init> | <string_init> | <boolean_init>

<int_init> ::= <int_dec> <word>  < assignment_operator> <int>
<char_init> ::=  <char_dec> <word>  < assignment_operator> <char>
<string_init> ::= <string_dec> <word>  < assignment_operator> <string>
<bool_init> ::= <bool_dec> <word>  < assignment_operator> <string>
<float_init> ::= <float_dec> <word> < assignment_operator> <float>

<assignment> ::= <var> <assignment_operator><algebra_list>
          | <var><alg_eq><algebra_list>
          | <var><add_op><add_op>
          | <var><substract_op><substract_op>

<assignment_list> ::= <assignment> <semicolon>
                    | <assignment> <semicolon> <assignment_list>
```

VII. PRIMITIVE FUNCTIONS AND FUNCTION DECLARATIONS

```
<primitive_function_list>::= <primitve_function>
                            |<primitve_function> <primitve_function_list>

<primitive_function>::=<read_temp>
                            |<read_hum>
                            |<read_airp>
```

```
                                |<read_airq>
                                |<read_light>
                                |<read_sound>
                                |<read_s7>
                                |<read_s8>
                                |<read_s9>
                                |<read_s10>
                                |<read_timestamp>
<read_temp>::= readTemperature()
<read_hum>::= readHumidity()
<read_airp>::= readAirPressure()
<read_airq>::= readAirQuality()
<read_light>::= readLight()
<read_sound>::= readSound()
<read_s7>::= readS7()
<read_s8>::= readS8()
<read_s9>::= readS9()
<read_s10>::= readS10()
<read_timestamp>::= readTimer()
<function_list>::= <function> | <function> <function_list>

<function>::=<variable_type> <word> <LP> <param_list> <RP> <LB> <statement_list>
<return_statement> <RB>
                | <void_dec> <word> <LP>  <param_list> <RP> <LB> <statement_list> <RB>

<param_list>::= <param> | <param> <param_list>
<param>::= <variable_type> <var>
<return_statement>::= <return> <var>
<url>::=<scheme_name><host_name><port_number><path_list><query_str>
                | <scheme_name><host_name><port_number><path_list><query_str><frag_ident>

<con_url>::= connectURL(<url>)
<send_int>::= sendInt <LP> <url> <comma> <int_dec> <param_int> <RP>
<receive_int>::= receiveInt<LP> <url> <RP>
<scheme_name>::= SCHEME_NAME
<host_name>::= HOST_NAME
<port_number>::= PORT_NUM

<path_list>::= <path> | <path><path_list>
<path>::= DIR_NAME<div_op>
<query_str>::=QUERY_STR
<frag_ident>::=FRAGMENT_IDENT

<sw1>::=SWITCH_1
```

<sw2>::=SWITCH_2
<sw3>::=SWITCH_3
<sw4>::=SWITCH_4
<sw5>::=SWITCH_5
<sw6>::=SWITCH_6
<sw7>::=SWITCH_7
<sw8>::=SWITCH_8
<sw9>::=SWITCH_9
<sw10>::=SWITCH_10

VIII. SYMBOLS

<BP> ::= begin_program
<EP>::= end_program
<LP>::= (
<RP>::=)
<LB>::={
<RB>::=}
<LSB>::=[
<RSB>::=]
<while>::= while
<return>::= return
<newline>::= /n
<semicolon>::= ;
<comma>::= ,
<space>::= “ ”
<and>::= &&
<or>::= ||
<assignment_operator>::= =
<equal>::= ==
<not_equal>::= !=
<not>::= !
<add_op> ::= +
<substract_op> ::= -
<div_op> ::= /
<mult_op> ::= *
<mod_op> ::= %
<gt> ::= >
<gt_eq> ::= >=
<lt> ::= <
<lt_eq> ::= <=
<comment_op> ::= //
<IOL> ::= <<
<IOR> ::= >>

EXPLANATIONS

I. Program

<program> ::= <BP><statement_list><EP>

In our language program has to start with begin_program and end with end_program. This syntax also decides what to write inside of a program.

<statement_list> ::=<statement> | <statement> <statement_list>

<statement> ::= <comment>
| <if_else>
| <assignment>
| <loop>
| <output>
| <init_list>
| <function_list>
| <primitive_function_list>
| <early_return>

In our program we allow users to create multiple statements with recursion while these statements can be if,else statements, assignment statements, loops, outputs, initialize, primitive functions ,functions and break option.

<comment> ::=<comment_op> <word_list>

We create our comments with a comment operator and word list.

II. Loop

<loop>::= <while_statement> | <for_statement>

This terminal allows the user to have while and for loops.

<while_statement>::=<while> <LP> <logic_list> <RP> <LB> <statement_list> <RB>
<for_statement>::=<for> <LP> <initialize_list> <semicolon><logic_list> <semicolon> <assignment_list> <RP> <LB> <statement_list> <RB>

These terminals show the syntaxes of while and for loops. In the while loop first, the user should write a “while” special word then inside parentheses there will be a logic list and finally there will be a statement list inside of the loop. In for loop, its syntax is first write “for” special case word then there will be initialization then logic list and assignment list which are separated by semicolons. After these there will be a statement list inside the Brackets.

III. Input and Output

<input_expression>::= cin<IOR><param>

This terminal allows the program to have user input which syntax is “cin” special case word and two parentheses. Inside of the parentheses there will be an algebra list which allows the user to enter any type that he/she wants.

<output_expressions_list> ::= <IOL><output_expressions>
| <IOL><output_expressions><output_expressions_list>

<output_expressions>::= <var>

| <algebra_list>
| " <word_list> "

Output_expression allows us to print 3 different statement types as output. These are the results of algebraic expressions, the values of the variables themselves, and an unspecified string and output_expression_list allows users to use these expressions repeatedly.

<output_expression>::=cout<output_expressions_list>

This terminal allows the program to have output for users. Which syntax is "cout" special case word for output case...

IV. Types

<var_dec>::=<variable_type><word><semicolon>

This terminal allows users to declare variables. It's syntax is first decide variable type then variable name and end with semicolon.

<variable_type> ::= <int_dec>
| <char_dec>
| <string_dec>
| <bool_dec>
| <float_dec>

Our language allows user to create integer, character, string, boolean and float variables.

<bool_dec>::= bool
<int_dec>::= int
<string_dec>::= string
<char_dec>::= char
<float_dec>::= float
<void_dec>::= void

Declarations of the variable types.

<sign>::= +| -

This terminal allows users to have signed integers and floats.

<var> ::= <char> | <int> | <bool> | <float> | <string>

This terminal shows the variable types.

<bool> ::= true | false

This terminal shows booleans can only have 2 types which are true or false.

<int> ::= 0 | <non_zero><digit_list>

This terminal allows us to create integers by recursively adding digits.

<signed_int> ::= <sign><int>

This terminal shows the syntax of signed integers.

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

These terminals show how the digits and digit lists are created.

<float>::=<signed_int>.<digitlist> | <int>.<digit_list> | .<digit_list>

This terminal shows the syntax of float variable type.

<char>::='<letter>' | '<digit>'

This terminal shows the syntax of character variable type.
<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
<word>::= <char> | <char><word>
<word_list>::= <word> | <word><space_list><word_list>
These terminals show how words and sentences have been created in our program.
<space_list>::= <space> | <space> <space_list>
This terminal allows users to create space or multiple spaces by recursion.

```
<arith_op1> ::= <arithmetic_op2>
              | <arithmetic_op1> + <arithmetic_op2>
              | <arithmetic_op1> - <arithmetic_op2>
<arith_op2> ::= <fact>
              | <arith_op2> * <fact>
              | <arith_op2> / <fact>
<fact> ::= <var>
         | <arith_op1>
<algebra_list> :: = <arith_op1> | <arith_op2>
```

V. Conditional Statements

```
<arith_op1> ::= <arithmetic_op2>
              | <arithmetic_op1> + <arithmetic_op2>
              | <arithmetic_op1> - <arithmetic_op2>
<arith_op2> ::= <fact>
              | <arith_op2> * <fact>
              | <arith_op2> / <fact>
<fact> ::= <var>
         | <arith_op1>
```

In our program there can be 5 different arithmetic operations made. These are addition, subtraction, multiplication, and division. arith_op1 , arith_op2 and <fact> allow to form a unique parse tree and setting the precedence..
<algebra_list> :: = <arith_op1> | <arith_op2>
Algebra list allows us to perform these operations with respect to arithmetic precedence.
<alg_eq> ::= <arith_op><assignment_operator>
This terminal allows us to assign arithmetic operations.
<and_or> ::= <and> | <or>
This terminal shows the “and” and “or” logical expressions.
<logic_list> ::= <logic> | <logic> <and_or> <logic_list>
This terminal shows the logic list with “and” or “or” operations.
<logic_op> ::= <gt> | <gt_eq> | <lt> | <lt_eq> | <eq> | <not_eq>
In our program there can be 6 different logical comparisons.
<logic> ::= <not> <logic> | <bool> | <int> <logic_op><int>
This terminal shows the not operation with boolean and logical operation between integers.
<if_else> ::= <matched> | <unmatched>

In our language if else are decided by matched and unmatched statements.

<matched> ::= if <LP> <logic_list> <RP><LB> <matched><RB> else<LB> <matched><RB> | <statement_list>

This terminal shows the matched if statement and syntax of it.

<unmatched> ::= if<LP> <logic_list><RP> <LB> <statement_list><RB>
| if <LP> <logic_list><RP> <LB> <matched> <RB> else <LB> <unmatched> <RB>

This terminal shows the unmatched if else statement and syntax of it. This also allows user to have multiple if statements.

VI. Initialize and Assign Statements

<initialize_list>::= <init> <semicolon> | <init> <comma> <initialize_list>
<init> ::= <int_init> | <char_init> | <string_init> | <boolean_init>

Initialize list and init allow recursive and flexible use of already defined variable initialize annotations.

<int_init> ::= <int_dec> <word> < assignment_operator> <int>
<char_init> ::= <char_dec> <word> < assignment_operator> <char>
<string_init> ::= <string_dec> <word> < assignment_operator> <string>
<bool_init> ::= <bool_dec> <word> < assignment_operator> <string>
<float_init> ::= <float_dec> <word> < assignment_operator> <float>

In order to avoid definition confusion and not create ambigiuty, we have provided separate initialize definition for all variable types. These definitions allow the user to assign a value to the variable while defining a variable, by controlling the variable type.

<assignment_list> ::= <assignment> <semicolon>
| <assignment> <semicolon> <assignment_list>

Assignment list allows us to use assignment operations in a recursive way.

<assignment> ::= <var> <assignment_operator><algebra_list>
| <var><alg_eq><algebra_list>
| <var><add_op><add_op>
| <var><substract_op><substract_op>

Assignment terminal allows us to perform simple mathematical operations with our variables.It also provides the ability to write variables by subtracting one from itself and adding one to itself at once(i++ , i--)

VII. Primitive Functions and Function Declarations

<primitive_function_list>::= <primitve_function>
|<primitve_function> <primitve_function_list>
<primitive_function>::=<read_temp>
|<read_hum>
|<read_airp>
|<read_airq>
|<read_light>
|<read_sound>
|<read_s7>
|<read_s8>

|<read_s9>
|<read_s10>
|<read_timestamp>
Primitive function list allows us to call our primitive functions (sensors) according to the user's request. Primitive function consists of 10 different sensors and timestamps
<read_temp>::= readTemperature()
It is a primitive sensor for detecting temperature
<read_hum>::= readHumidity()
It is a primitive sensor for detecting humidity
<read_airp>::= readAirPressure()
It is a primitive sensor for detecting air pressure
<read_airq>::= readAirQuality()
It is a primitive sensor for detecting air quality
<read_light>::= readLight()
It is a primitive sensor for detecting light
<read_sound>::= readSound()
It is a primitive sensor for detecting sound
<read_s7>::= readS7()
<read_s8>::= readS8()
<read_s9>::= readS9()
<read_s10>::= readS10()
read_s7 read_s8 read_s9 and read_s10 are some additional sensors we provide for users. Can be defined later by the user.
<read_timestamp>::= readTimer()
A timer that returns the current timestamp

<function_list>::= <function> | <function> <function_list>
<function>::=<variable_type> <word> <LP> <param_list> <RP> <LB> <statement_list> <return_statement> <RB>
| <void_dec> <word> <LP> <param_list> <RP> <LB> <statement_list> <RB>
Function list allows us to use the terminals in the function in a flexible and recursive way. The function terminal allows the user to specify the type of the function, to name the function, to write parameters to the function and to write statements in it.
<param_list>::= <param> | <param> <param_list>
<param>::= <variable_type> <var>
Paramlist and param allow us to define the parameters that we will use inside the functions.
<return_statement>::= <return> <var>
It allows us to return a variable at the end of the function.
<url>::=<scheme_name><host_name><port_number><path_list><query_str>
|<scheme_name><host_name><port_number><path_list><query_str><frag_id>
It allows the user to create the url
<con_url>::= connectURL(<url>)
Mechanism to define a connection to a given URL

<send_int>::= sendInt <LP> <url> <comma> <int_dec> <param_int> <RP>
<receive_int>::= receiveInt<LP> <url> <RP>
<scheme_name>::= SCHEME_NAME
<host_name>::= HOST_NAME
<port_number>::= PORT_NUM
<path_list>::= <path> | <path><path_list>
<path>::= DIR_NAME<div_op>
<query_str>::=QUERY_STR
<frag_ident>::=FRAGMENT_IDENT
These are the fragments of the Url that are defined in our language.
<sw1>::=SWITCH_1
<sw2>::=SWITCH_2
<sw3>::=SWITCH_3
<sw4>::=SWITCH_4
<sw5>::=SWITCH_5
<sw6>::=SWITCH_6
<sw7>::=SWITCH_7
<sw8>::=SWITCH_8
<sw9>::=SWITCH_9
<sw10>::=SWITCH_10
There are 10 switches that can be set as on/off to control some actuators.

WHAT HAS BEEN CHANGED IN SECOND PART?

In the first part of the assignment we created simple logical expressions which did not have any precedence between operators. In the second part of the project we changed that and now the program has precedence between arithmetic and logical operators. We also change our input and output syntax in order to fix the conflict in our yacc file. This also resulted in better and improved input and output expressions. We also added early_return to our program which allows users to have a break from our loops or other statements. These additions were made to resolve some conflicts that have been encountered in yacc. Even though we fixed some, there are still some remaining conflicts that we couldn’t understand or resolve in our language.

HOW DID WE DEFINE NONTRIVIAL TOKENS IN OUR LANGUAGE?

Most of our non-trivial tokens, the comment "//" is derived from Java. This is primarily because we are more accustomed to Java than any other programming language. But in addition to drawing inspiration from Java, C and Kotlin, we also created our own input/output syntax which is similar to C++. The other justification for using them is that we also believed they were easier to write and read, so we combined them into a single language.