2020-2021 FALL SEMESTER

CS315 – PROGRAMMING LANGUAGES

TEAM: 34

SECTION 1

PROJECT REPORT

TURNA

GROUP MEMBERS:

1) Burak Yetiştiren, 21802608
2) Akın Parkan, 21703249
3) Mustafa Hakan Kara, 21703317

Table of Contents

1. BNF Description of Turna

<program> → <statements> | comment | <empty>

<statements> → <statement>SEMICOLON
| <statement>SEMICOLON <statements>
| comment <statements> | <conditional_or_loop_statement>
| <return_statement>

<statement> → <function_call> | <decleration_statement> | <assignment_statement>

<conditional_or_loop_statement> → <loop_statement> | <conditional_statement>

<loop_statement> → <while_statement> | <for_statement>

<conditional_statement> → **if** LP <expression> RP START <statements> END
| **if** LP <expression> RP START <statements>END
else START <statements> END

<declaration_statement> → <function_declaration> | <const_declaration>
| <variable_declaration>

<term> → <term> MULTIPLICATION <factor> | <term> DIVISION <factor>
| <term> % <factor> | <factor>

<factor> → LP <math_expression> RP | <int_literal> | <float_literal> | <identifier>
| <function_call>

<while_statement> → **while** LP <expression> RP START<statements>END

<for_statement> → **for** LP <variable_decleration> <expression>
<assignment_statement> RP START<statements> END

<type> → TYPE_BOOL | TYPE_CHAR | TYPE_INT | TYPE_FLOAT | TYPE_STRING

<const_decleration> → CONST <type> <identifier> <assignment_statement>

<variable_declaration> → <type> <decleration_list>

<decleration_list> → <identifier> ASSIGN_OP <expression> COMMA <decleration_list>
| <identifier> COMMA <decleration_list>
| <identifier> ASSIGN_OP <expression> | <identifier>

<assignment_statement> → <identifier> ASSIGN_OP <expression> | INCREMENT_OP
| <identifier> DECREMENT_OP

<expression>→ STRING | <or_expression>

<or_expression> → <or_expression> OR <and_expression> | <and_expression>

<and_expression> → <and_expression> AND <relational_expression>
| <relational_expression>

<relational_expression> → <relational_expression> <relational_operator>
<arithmetic_expression> | <arithmetic_expression>

<arithmetic_expression> → <arithmetic_expression> PLUS <term>
| <arithmetic _expression> MINUS <term> | <term>

<term> → <term> MULTIPLICATION <factor> | <term> DIVISION <factor>

| <term> MODULO <factor> | <factor>

<factor> → LP <or_expression> RP | <function_call> | IDENTIFIER | INTEGER | FLOAT

| BOOL | CHAR

<relational_operator> → EQUAL_OP | NOT_EQUAL_OP | LESS_THAN_OP
| LESS_THAN_OR_EQUAL_OP | GREATER_THAN_OP
| GREATER_THAN_OR_EQUAL_OP

<function_call> → <custom_function_call> | <primitive_function_call>

<custom_function_call> → <identifier> LP <expression> RP

<primitive_function_call> → <drone_function_call>
| <print_function_call>
| <scan_function_call>

<drone_function_call> → GET_INCL LP RP
| GET_ALTITUDE LP RP
| GET_SPEED LP RP
| GET_TEMP LP RP
| GET_ACCELERATION LP <parameter> RP
| SET_ALTITUDE LP <parameter> RP
| SET_SPEED LP <parameter> RP
| SET_TEMP LP <parameter> RP
| SET_ACCELERATION LP <parameter> RP
| SET_INCL LP <parameter> RP
| TURN_ON_CAM LP RP
| TURN_OFF_CAM LP RP
| TAKE_PICT LP RP
| GET_CURRENT_TIME_STAMP LP RP
| CONNECT_WIFI LP<parameter> COMMA <parameter> RP
| TAKE_OFF LP RP
| LAND LP RP
| FLY_UP_WITH_DISTANCE LP <parameter> RP
| FLY_DOWN_WITH_DISTANCE LP <parameter> RP
| FLY_RIGHT_WITH_DISTANCE LP <parameter> RP

| FLY_FORWARD_WITH_DISTANCE LP <parameter> RP
| FLY_BACK_WITH_DISTANCE LP <parameter> RP
| ROTATE_CW_WITH_ANGLE LP <parameter> RP
| ROTATE_CCW_WITH_ANGLE LP <parameter> RP
| FLIP_FORWARD LP RP
| FLIP_BACKWARD LP RP
| FLIP_LEFT LP RP
| FLIP_RIGHT LP RP
| SET_SPEED LP <parameter> RP
| HOVER_FOR_SECONDS LP <parameter> RP
| YAW_RIGHT_WITH_DEGREES LP <parameter> RP
| YAW_LEFT_WITH_DEGREES LP <parameter> RP
| HOVER LP RP

<scan_function_call> → SCAN_INT | SCAN_STR | SCAN_FLOAT | SCAN_BOOL

| SCAN_CHAR

<print_function_call> → PRINT | PRINT_NL

<function_decleration> → <function_type> <identifier>LP<parameters>RP
START<statements>END

<return_statement> → return <expression>

<parameters> → <parameters> COMMA <parameter> | <parameter>

<parameter> → <type> <identifier>

<literal> → <int_literal>
| <float_literal>
| <bool_literal>
| <string_literal>
| <char_literal>

<bool_literal> → TRUE | FALSE

<string_literal> → “<string>”

<return_type> → <type> | void

<char_literal> → ‘<character>’

<string> → <character><string> | <empty>

<identifier>→ <initial>
| <identifier><initial>
| <identifier><digit>

<initial> → <letter> | $ | _

<character> → < | > | ! | @ | # | $ | % | ^ | & | * | (|) | _ | + | } | { |]
| [| ” | : | \ | ’ | ; | , | . | / | <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

<int_literal> → <sign> ? <digits>
| <digits>

<float_literal> → <sign>?<digits>.<digits>
| <digits>.<digits>
| <sign>?.<digits>
| .<digits>

<digits> → <digits> <digit>
| <digit>

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

<sign> → + | -

<empty> → ε

2. Language Constructs

<program> The program consists of statements, comments or it can be empty.

<statements> Statements can consist of one statement or a sequence of statements following a statement or preceding a comment. Each statement has to end with a semicolon.

<statement> Statement is the construct to define the various operations fundamental for the purpose of the language. They can be function calls, loop statements, conditional statements, declaration statements, and assignment statements.

<conditional_or_loop_statement> Merges conditional and loop statements

<loop_statement> Collects the three types of loop statements present in the language. They are for statements and while statement. These statements are basically loops, those the programmers will use for executing a certain part of their code one or more times.

<conditional_statement> This construct embraces certain statements with start, and end terminals. Execution of the code is controlled by <expression> embraced in parenthesis. If the logic expression is true, then the embraced statements are executed. If not either the conditional statement is abandoned, or if the conditional statement has the else part that part is executed. Hence, in the language conditional statements can be written in two ways:

```
if(getAltitude() lt 10)
start
  setAltitude(10);
end

OR

if(getSpeed() lt 2)
start
  setSpeed(3);
end
else
start
  setSpeed(getSpeed()--);
end
```

<declaration_statement> Groups declarations of functions, constants, and variables.

<return_statement> Statement used to return required value or identifier from a function.

<math_expression> Consists of addition of math expressions and terms or subtraction of math expressions and terms or only terms. Because of the operator precedence only the addition and subtraction are yet defined.

<while_statement> While statement is controlled by the language construct <expression>. As long as the logic expression is true statements embraced with the start, and end terminals will be executed. For example:

```
while( getHeight() gt 3)
start
  setHeight( getHeight()--);
end
```

<for_statement> For statement is controlled by the language construct <expression> and driven by the assignment statement. The programmer has to declare a variable first, define a logic expression and drive the loop with the assignment statement. Then the executed statement must be written between the start, and end terminals. For example:

```
for( type_int = 0; i lte 15; i++)
start
 flipForward();
end
```

<type> Type construct accumulates the integer, float, string, and boolean variable type names.

<const_decleration> Structures the declaration of constants. For example:

```
const type_int PI = 3.14;
```

<variable_declaration> Structures the declaration of the variables. For example:

```
type_int numberOfTakeoffs;
type_int numberOfLandings  = 0;
```

<assignment_statement> Used to assign variables with desired values. For example:

```
type_float degree = PI * 0.75;
```

<expression> An expression is either a string or an or expression. It encompasses relational and arithmetic expressions, function calls, identifiers, and combination of these.

For example, the following are valid expressions in Turna:

```
"Turna"

a * 5 + (6 < 7) & foo() | true  - 'b'
```

<or_expression> An or expression is either an And expression or an expression that is obtained by combining an or expression and an and expression with an OR operation.

<and_expression> An and expression is either a relational expression or an expression that is obtained by combining an and expression and a relational expression with an AND operation.

<relational_expression> A relational expression is either an arithmetic expression or an expression that is obtained by combining a relational and an arithmetic expression with a relational operator.

<arithmetic_expression> An arithmetic expression is either a term or an expression that is obtained by combining an and expression and a term with an ADDITION or SUBTRACTION operation.

<term> A term is either a factor or an expression that is obtained by combining a term and a factor with a MULTIPLICATION, DIVISION or MODULO operation.

<factor> A factor is either an or expression inside parenthesis or one of the following:

- An integer, float, boolean or character literal
- An identifier
- A function call

<relational_operator> Cumulates all the logical operators in the language: equal, not equal, less than, less than or equal, greater than, greater than or equal operators

<function_call> This language construct merges two types of function calls: user function calls and primitive function calls.

<user_function_call> Defines the structure of the user function calls. An identifier is followed by a <expression> in parenthesis. For example:

```
name = getPilotName();
```

<primitive_function_call> → Used to group four types of primitive function calls. They are drone, write, scan, and print function calls.

<drone_function_call> Cumulates the primitive function calls regarding the drone such as, getAltitude(), getIncl(), turnOnCam(), takePict().

<input_statement> Defines the structure of an input statement. One example can be the following:

```
numberOfLoops = scanInt();
```

<scan_function_call> Cumulates the primitive scan functions. They are: scanInt(), scanStr, scanFloat() ,scanBool(), scanChar()

<print_statements> Used to gather two primitive print functions print() and print_nl().

<print> Defines the structure of the primitive print() function. To exemplify:

```
print("Flight completed.");
```

<print_nl> Defines the structure of the primitive print_nl() function. print_nl() function leaves an empty line after execution. Examples can be:

```
print_nl("Flight initialized.");
print_nl();
```

<function_decleration> Defines the structure of the user defined function declarations. First the function type is written, then the identifier is provided, and the parameters are provided in parenthesis. The statements of the function are embraced in start and end terminals. An example is:

```
type_bool getVerification( type_int altitude, type_int speed)
start
  if( speed lte 10 and altitude eq 5)
  start
     return True;
  end
  else
  start
      return False;
  end;
end;
```

<parameters> Delineates the parameters of functions. Parameters are followed by a parameter. Either one or multiple parameters can be present. Parameters are separated by commas. For example:

```
type_int numberOfLoops, type_float degrees
```

<parameter> Represents a single parameter. A parameter simply is a type followed by an identifier. For example:

```
type_int durationOfFlight
```

<literal> Cumulates the literals of floats, integers, booleans, strings, and characters.

<bool_literal> Represents the two types of boolean literals: True and False.

<string_literal> Represents the string literal. Puts the <string> construct in braces.

<return_type> Defines two categories of return types. One category is the non-void return types like integer, float, etc. The other type is void.

<char_literal> This language construct is used for categorizing the other language construct <character> as char literal, providing the proper structure.

<string> Defines the structure of a string

<identifier> Defines the rules of an identifier

<initial> Defines the set of initials for writing an identifier

<character> Represents the characters valid in Turna. All letters, digits and some symbols like “!”, “@” are accepted

<letter> Represents every letter in the domain [A-Za-z]

<int_literal> Describes the structure of the integer literals

<float_literal> Describes the structure of the float literals

<digits> Represents multiple digits or a single digit, this definition is used in structuring some of the literals

<digit> Represents every letter in the domain [0-9]

<sign> Represents the minus and plus signs

<empty> → ε

3. Nontrivial Tokens

3.1. Comments

Comments in Turna have two implementations. Programmers can use single and multi-line comments. Comments that are enclosed in "/#" and "#/" are multi-line comments. On the other hand, comments those initialize with "#" and end with "\n" are single line comments. We wanted programmers to be able to have multi-line comments as well as single line comments so that the writability of Turna would increase. Also, both the single- and multi-line comments are using the hash (#) symbol, so that the programmers or the readers of the programmers will not have to memorize two different symbols for comments. In this manner Turna would be more readable, as well as writable.

3.2. Identifiers

Identifiers in Turna must start with a set of characters. They are any letter in English alphabet (A-Za-z), or the dollar sign ("$"), or the underscore character ("_"). For the following characters, programmers can choose any initials or digits they wish to use. We have introduced initials; our ambition was to make the language more reliable. On the negative side, the language became less writable, because the programmers have to remember the rules for writing initializers.

3.3. Literals

The rules we have determined to represent the literals in Turna followed the conventions used in most of the C group languages. We believe that the rules are readable because they follow similar rules, when compared to the mathematical terminology mathematics, also the conventions that are being used in natural languages too. On the other hand, when writability is considered, because we have followed the natural ways to write the literals and there is approximately one way to write them, Turna is writable as well.

- **True** and **False** literals are boolean literals that are representing the true and false statements in logic.
- **Character** literals are enclosed between two apostrophe (‘) characters.
- **String** literals are enclosed between two quotation marks, similar to C group languages.
- **Integer** literals are similar to the most programming languages. They can have signs before them, if they are negative integers, they must have a minus sign (-) before them.

- **Float** literals can be positive and negative or neutral. If they are between 0 and one the programmer can put the decimal point without writing 0 in the beginning: .9 for 0.9.

3.4. Reserved words

While the selection of the reserved are being selected, our main ambition was to make them readable. Because of this ambition, we have chosen to use the abbreviation of some boolean operators. This has decreased the writability for some accounts, that the programmer would have learned about both ways to write such operators. We want to emphasize the primitive functions we have defined for Turna. We believe that the primitive functions implemented help the drone programmer to program their drones more conveniently with Turna.

Types

- **type_int** - used to precede the identifier for integer type
- **type_float** - used to precede the identifier for float type
- **type_string** - used to precede the identifier for string type
- **type_char** - used to precede the identifier for character type

Return

- **return** - used to return variables from the functions

Output

- **print** - used to print the inputs in one terminal line
- **print_nl** - used to print the inputs in one terminal line and skip to the next terminal line

Conditional operators

- **if** - used to initialize the conditional statements, precedes the logic expression in parenthesis
- **else** - precedes the reserved word start, that precedes statements for the conditional operator

Loops

- **for** - reserved word for for loops. After using this word programmer must continue with providing the required parts for the for loop
- **while** - reserved word for while loops, which precedes the logical expression that must be satisfied to enter the loop

Boundary statements

- **start** - reserved word for specifying the beginning of the code block of conditional and loop statements
- **end** - reserved word for specifying the ending of the code block of conditional and loop statements

Logical operators

- **eq -** equivalent of the terminal "==" checks if the given elements are the same
- **neq -** equivalent of the terminal "!=" checks if the given elements are different
- **and -** equivalent of the terminal "&&" checks if the given logical statements are both true
- **or -** equivalent of the terminal "||" checks if one of the given elements is true
- **lt -** equivalent of the terminal "<" checks if the given element is less than the other element
- **lte -** equivalent of the terminal "<=" checks if the given element is less than or equal to the other element
- **gt -** equivalent of the terminal ">" checks if the given element is greater than the other element
- **gte -** equivalent of the terminal ">=" checks if the given element is greater than or equal to the other element

Primitive functions

- **getIncl** - reserved word for specifying the primitive function for getting the inclination of the drone.
- **getAltitude** - reserved word for specifying the primitive function for getting the altitude of the drone
- **getSpeed** - reserved word for specifying the primitive function for getting the speed of the drone
- **getTemp** - reserved word for specifying the primitive function for getting the temperature of the drone
- **getAcceleration** - reserved word for specifying the primitive function for getting the acceleration of the drone
- **setIncl** - reserved word for specifying the primitive function for setting the inclination of the drone.
- **setAltitude** - reserved word for specifying the primitive function for setting the altitude of the drone.

- **setSpeed** - reserved word for specifying the primitive function for setting the speed of the drone.
- **setTemp** - reserved word for specifying the primitive function for setting the temperature of the drone.
- **setAcceleration** - reserved word for specifying the primitive function for setting the acceleration of the drone.
- **turnOnCam** - reserved word for specifying the primitive function for turning on the camera of the drone
- **turnOffCam** - reserved word for specifying the primitive function for turning off the drone
- **takePict** - reserved word for specifying the primitive function for taking a picture with camera of the drone
- **getCurrentTimeStamp** - reserved word for specifying the primitive function for getting the current timestamp of the drone
- **connectWifi** - reserved word for specifying the primitive function for connecting the drone to a Wi-Fi network.
- **takeOff** - reserved word for specifying the primitive function for taking off for the drone
- **land** - reserved word for specifying the primitive function for landing for the drone
- **flyUpWithDistance** - reserved word for specifying the primitive function for flying up with a distance of the drone
- **flyDownWithDistance** - reserved word for specifying the primitive function for flying down with a distance of the drone
- **flyLeftWithDistance** - reserved word for specifying the primitive function for flying left with a distance of the drone
- **flyRightWithDistance** - reserved word for specifying the primitive function for flying right with a distance of the drone
- **flyForwardWithDistance** - reserved word for specifying the primitive function for flying forward with a distance of the drone
- **flyBackWithDistance** - reserved word for specifying the primitive function for flying back with a distance of the drone
- **rotateCwWithAngle** - reserved word for specifying the primitive function for rotating the drone clockwise with given angle

- **rotateCcwWithAngle** - reserved word for specifying the primitive function for rotating the drone counterclockwise with given angle

- **flipForward** - reserved word for specifying the primitive function for flipping the drone forward

- **flipBackward** - reserved word for specifying the primitive function for flipping the drone backward

- **flipLeft** - reserved word for specifying the primitive function for flipping the drone left

- **flipRight** - reserved word for specifying the primitive function for flipping the drone right

- **hoverForSeconds** - reserved word for specifying the primitive function for making the drone stay in the hover for user defined duration

- **yawRightWithDegrees** - reserved word for specifying the primitive function for yawing the drone right with user defined degrees

- **yawLeftWithDegrees** - reserved word for specifying the primitive function for yawing the drone left with user defined degrees

- **hover** - reserved word for specifying the primitive function for the drone to stay in hover

- **scanInt** - reserved word for specifying the primitive function for scanning the integer the programmer will provide

- **scanStr** - reserved word for specifying the primitive function for scanning the string the programmer will provide

- **scanFloat** - reserved word for specifying the primitive function for scanning the float the programmer will provide

- **scanBool** - reserved word for specifying the primitive function for scanning the boolean the programmer will provide

- **scanChar** - reserved word for specifying the primitive function for scanning the character the programmer will provide

4. Language Evaluation

4.1. Readability

Readability of a language is important because a readable language provides programmers of a language to be familiarized easily, with that learning cost of the language would be decreased. For that purpose, we try to increase our language's readability as much as we could, while not compromising writability of our language as much as possible. Reserved words of our language are not much, and they cannot be used as variable names or user defined function names. These reserved words include primitive function names related to Drone programming since we are designing a language for drones. So having less reserved words and not letting programmers use them for defining variables or functions increase the readability of our language. Our language supports output functions similar to Python language as print() and print_nl(), and input functions similar to Java which are scanInt(), scanBool() etc. with that, it makes our language easier for the programmers to familiarize. Moreover our language includes single line and multi-line commands, it also increases our language's readability. Since our language requires start and end reserved words for loops' and conditional statements' for specifying the code blocks of the loop statement or conditional statement; our language's readability is increased since it is easier to follow the statements with specified code blocks. Our language will not allow programmers to overload operators, with that readability of the language increases. In contrast to python and JavaScript, our language has data types, with that also the readability increases. Lastly, in contrast to c, our language includes boolean type. With that ambiguity related to understanding if the boolean type holds 0/1 or True/False we solve this problem by specifying bool data type to hold true\false.

4.2. Writability

Writability of a language is a measurement which can be specified as how easy programmers can code with using a language. So we also aim to have a language with high writability as much as we can by supporting expressivity and simplicity. Our language supports expressivity by providing programmers to use different sets of loops which can be specified as for, while loops and also providing ++ and -- operators for integer types of the language in addition to classic increment operator (count = count +1), another thing that supports expressivity can be supported by multiple declaration on a single line, which is supported by our language. Since our language has a few primitive types it is a simple language. Since there is a trade-off between readability and writability and while forcing the usage of start - end reserved words for loops and conditionals and not supporting operator overloading increase the readability of the language, due to readability - writability trade-off language's writability decreases.

4.3. Reliability

In order to make our programming language reliable as possible we have done the following: Since variable types have different sizes in programming languages we provide the variable types as primitive types and require programmers to use variable types while defining the variables, and through the primitive input functions we specify the input type with scanInt(), scanBool(), scanFloat() etc. and also we force programmers to obey parameter types of functions(type checking) otherwise lexical analyzer will not supply the expected tokens. With all of them our aim is to lower errors that are related to Turna to minimum so that errors are mostly related to syntax rather than language. In addition, with operator precedence rules which are first the statements inside parentheses then multiplication and division and then subtraction and addition, with that we aim to have a reliable language on arithmetic operations.

5. Lex of the Language

```
%option main
digit   [0-9]
sign    [+-]
integer {sign}?{digit}+
float {sign}?{digit}*(\.)?{digit}+
letter  [a-zA-Z]
char    \'[^']\'
string    \"[^"]*\"
initial ({letter}|$|_)
identifier {initial}({initial}|{digit})*
literal ({integer}|{float}|{char}|{string})

get_incl "getIncl"
get_altitude "getAltitude"
get_speed "getSpeed"
get_temp "getTemp"
get_acceleration "getAcceleration"
set_incl "setIncl"
set_altitude "setAltitude"
set_speed "setSpeed"
set_temp "setTemp"
set_acceleration "setAcceleration"
turn_on_cam "turnOnCam"
turn_off_cam "turnOffCam"
take_pict "takePict"
get_current_time_stamp "getCurrentTimeStamp"
connect_wifi "connectWifi"
take_off "takeOff"
land "land"
```

```
fly_up_with_distance "flyUpWithDistance"
fly_down_with_distance "flyDownWithDistance"
fly_left_with_distance "flyLeftWithDistance"
fly_right_with_distance "flyRightWithDistance"
fly_forward_with_distance "flyForwardWithDistance"
fly_back_with_distance "flyBackWithDistance"
rotate_cw_with_angle "rotateCwWithAngle"
rotate_ccw_with_angle "rotateCcwWithAngle"
flip_forward "flipForward"
flip_backward "flipBackward"
flip_left "flipLeft"
flip_right "flipRight"
hover_for_seconds "hoverForSeconds"
yaw_right_with_degrees "yawRightWithDegrees"
yaw_left_with_degrees "yawLeftWithDegrees"
hover "hover"
scan_int "scanInt"
scan_str "scanStr"
scan_float "scanFloat"
scan_bool "scanBool"
scan_char "scanChar"
print "print"
print_nl "print_nl"

comment (#\/[.\n]*\/#|#.*\n)

assign_op "="
semicolon ";"
incrmnt_op "++"
decrmnt_op "--"
true "true"
false "false"
if "if"
else "else"
while "while"
for "for"
return "return"
start "start"
end "end"

lp "("
rp ")"
comma ","
```

```
{comment} printf("COMMENT ");
{get_incl} printf("GET_INCL ");
{get_altitude} printf("GET_ALTITUDE ");
{get_temp} printf("GET_TEMP ");
{get_speed} printf("GET_SPEED ");
{get_acceleration} printf("GET_ACCELERATION ");
{set_incl} printf("SET_INCL ");
{set_altitude} printf("SET_ALTITUDE ");
{set_temp} printf("SET_TEMP ");
{set_speed} printf("SET_SPEED ");
{set_acceleration} printf("SET_ACCELERATION ");
{turn_on_cam} printf("TURN_ON_CAM ");
{turn_off_cam} printf("TURN_OFF_CAM ");
{take_pict} printf("TAKE_PICT ");
{get_current_time_stamp} printf("GET_CURRENT_TIME_STAMP ");
{connect_wifi} printf("CONNECT_WIFI ");
{take_off} printf("TAKE_OFF ");
{land} printf("LAND ");
{fly_up_with_distance} printf("FLY_UP_WITH_DISTANCE ");
{fly_down_with_distance} printf("FLY_DOWN_WITH_DISTANCE ");
{fly_left_with_distance} printf("FLY_LEFT_WITH_DISTANCE ");
{fly_right_with_distance} printf("FLY_RIGHT_WITH_DISTANCE ");
{fly_forward_with_distance} printf("FLY_FORWARD_WITH_DISTANCE ");
{fly_back_with_distance} printf("FLY_BACK_WITH_DISTANCE ");
{rotate_cw_with_angle} printf("ROTATE_CW_WITH_ANGLE ");
{rotate_ccw_with_angle} printf("ROTATE_CCW_WITH_ANGLE ");
{flip_forward} printf("FLIP_FORWARD ");
{flip_backward} printf("FLIP_BACKWARD ");
{flip_left} printf("FLIP_LEFT ");
{flip_right} printf("FLIP_RIGHT ");
```

```
{hover_for_seconds} printf("HOVER_FOR_SECONDS ");
{yaw_right_with_degrees} printf("YAW_RIGHT_FOR_DEGREES ");
{yaw_left_with_degrees} printf("YAW_LEFT_FOR_DEGREES ");
{hover} printf("HOVER ");
{scan_int} printf("SCAN_INT ");
{scan_str} printf("SCAN_STR ");
{scan_float} printf("SCAN_FLOAT ");
{scan_bool} printf("SCAN_BOOL ");
{scan_char} printf("SCAN_CHAR ");
{print} printf("PRINT ");
{print_nl} printf("PRINT_NL ");
{semicolon} printf("SEMICOLON ");
{comma} printf("COMMA ");
{assign_op} printf("ASSIGN_OP ");
{incrmnt_op} printf("INCREMENT_OP ");
{decrmnt_op} printf("DECREMENT_OP ");
{eq_op} printf("EQUAL_OP ");
{neq_op} printf("NOT_EQUAL_OP ");
{lt_op} printf("LESS_THAN_OP ");
{lte_op} printf("LESS_THAN_OR_EQUAL_OP ");
{gt_op} printf("GREATER_THAN_OP ");
{gte_op} printf("GREATER_THAN_OR_EQUAL_OP ");
{and} printf("AND ");
{or} printf("OR ");
{const} printf("CONST ");
{type_bool} printf("TYPE_BOOL ");
{type_char} printf("TYPE_CHAR ");
{type_int} printf("TYPE_INT ");
{type_float} printf("TYPE_FLOAT ");
{type_string} printf("TYPE_STRING ");
```

```
{void} printf("VOID ");
{true} printf("TRUE ");
{false} printf("FALSE ");
{while} printf("WHILE ");
{for} printf("FOR ");
{start} printf("START ");
{end} printf("END ");
{if} printf("IF ");
{else} printf("ELSE ");
{return} printf("RETURN ");
{integer} printf("INTEGER ");
{float} printf("FLOAT ");
{identifier} printf("IDENTIFIER ");
{char} printf("CHAR ");
{string} printf("STRING ");
{lp} printf("LP ");
{rp} printf("RP ");
{plus} printf("PLUS ");
{minus} printf("MINUS ");
{mul} printf("MULTIPLICATION ");
{div} printf("DIVISION ");
{mod} printf("MODULO ");
```

6. Example Programs

6.1. Loops

```
#/
        An example program written in Turna to test loops.
/#
type_int x = 0, y = 0;
while(x < 100)
start
    for(y =0; y < 20; y++)
    start
                print_nl("Hello Turna!");
    end
    x++;
end
```

6.2. Operators

```
#/
        An example program written in Turna to test operators.
/#


type_int a = 4, b = 3, c = 2, d = 1, e;
type_bool f = true, g = false, h, i, j;
e = (a + b) * c / d;
e = a - e;
h = f && g
h = a < 4 || b <= 3 or c > 1 || c >= 2
i = a lt 4 or b lte 3 or c gt 1 or c gte 2;
j = h == i and h eq i;
j = h != i and h neq i;
```

6.3. Declaration

```
#/
        An example program written in Turna to test declerations.
/#
type_int thats_an_integer;
thats_an_integer=4;
type_int thats_an_other_integer = 3;
const type_int thats_an_constant_integer = 7;


type_string thats_an_str;
thats_an_str = "example1"
type_string thats_an_other_str = "example2";
const type_string thats_an_constant_str = "I Love PL";
```

6.4. Nested loops

```
#/
                An example program written in Turna to test nested loops.
/#
type_int x =0;
type_int y =0;
while(x < 100)
start
    for(y =0; y < 20; y++)
    start
        print_nl();
    end
    x++;
end
```

6.5. Nested conditional

```
#/
        An example program written in Turna to test primitive nested if statements
/#
type_bool b1 = True;
type_bool b2= False;
if(b1 == True)
start
  if( b2 == False)
    start
        land();
     end
  else
    start
        takeOff();
    end
```

6.6. Primitive drone functions

```
#/
        An example program written in Turna to test primitive drone functions.
/#
takeOff();
type_int dist;
dist = 10;
#fly Up 10 cm
flyUpWithDistance(dist);
type_int forward_dist = 20;
flyForwardWithDistance(forward_dist);
turnOnCam();
takePict();
turnOffCam();
flyDownWithDistance(dist);
land();
```

6.7. Input/Output

```
#/
        An example program written in Turna to test primitive drone input/output.
/#
type_int input_int = scanInt();
type_string input_str = scanStr();
type_float input_float = scanFloat();
type_bool input_bool =scanBool();
type_char input_char =scanChar();



print("thats an print statement without new line");
print_nl("thats an print statement with new line");
type_float thats_an_float;
thats_an_float = 1.0;
type_float thats_an_other_float = 2.0;
```

6.8. Test of All Primitive Functions

```
#/
        An example program written in Turna to test all primitive functions.
/#
getIncl();
getAltitude();
getSpeed();
getTemp();
getAcceleration();
setIncl();
setAltitude();
setSpeed();
setTemp();
setAcceleration();
turnOnCam();
turnOffCam();
takePict();
getCurrentTimeStamp();
connectWifi();
takeOff();
land();
flyUpWithDistance();
flyDownWithDistance();
flyLeftWithDistance();
flyRightWithDistance();
flyForwardWithDistance();
flyBackWithDistance();
rotateCwWithAngle();
rotateCcwWithAngle();
flipForward();
flipBackward();
```

```
flipLeft();
flipRight();
hoverForSeconds();
yawRightWithDegrees();
yawLeftWithDegrees();
hover();
scanInt();
scanStr();
scanFloat();
scanBool();
scanChar();
```

6.9. Comments

```
#This is a one-line comment


#/


    This is a multi-line comment


/#
```

6.10. Example Flight Program

```
#/
        An example program written in Turna to show an example flight program.
/#
takeOff();
flyUpWithDistance(100);
setSpeed(20);
type_int e  = 0;
while( hover())
start
  flyUpWithDistance(5);
  rotateCcwWithAngle(180);
  if( e < 100)
    start
      e++;
    end
```