CS 315 Programming Languages

PROJECT 2 - "DROLAN"

A Programming Language for Drones

TEAM 40

Emin Adem Buran - 21703279 - Section 1

Onur Oruç - 21702381 - Section 2

Ömer Yavuz Öztürk - 21803565 - Section 2

BNF OF DROLAN

```
<program>::= BEGIN <stmts> END

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

<stmt>::=  <if_stmt>
             | <loop_stmt>
             | <funcall_stmt>
             | <assign_stmt>
             | <decl_stmt>
             | <input_stmt>
             | <output_stmt>
             | <comment_stmt>

<if_stmt>::= IF LP <logical_exprs> RP  BEGIN <stmts> END
             | IF ( <logical_exprs>)  BEGIN <stmts> END else BEGIN <stmts> END

<comment_stmt>::= COMMENT

<input_stmt>::= INPUT LP IDENTIFIER RP | INPUT LP IDENTIFIER COMMA
<print_expr> RP

<output_stmt>::= OUTPUT LP <print_expr> RP | OUTPUT LP <print_expr> COMMA
<print_expr> RP

<print_expr>::= IDENTIFIER
             | <primitive_value>
             |  IDENTIFIER PLUS < print_expr>
             | <primitive_value>   PLUS <print_expr>
             | <primretfun_call> PLUS <print_expr>
             | <returnfun_call> PLUS <print_expr>
             | <primretfun_call>
```

```
                    | <returnfun_call>

<loop_stmt>::= <while_loop> | <for_loop>

<while_loop> ::= WHILE LP <logical_exprs> RP BEGIN <stmts> END

<for_loop> ::= FOR LP <decl_stmt> SC  <logical_exprs> SC <assign_stmt> RP BEGIN
<stmts> END

<funcall_stmt>::= <returnfun_call>
                    | <voidfun_call>
                    | <primretfun_call>
                    | <primvoidfun_call>

<returnfun_call>::= IDENTIFIER LP <call_param_list> RP
<voidfun_call>::= VOID IDENTIFIER LP <call_param_list> RP


<primretfun_call>::= <readincl_call>
                    | <readalt_call>
                    | <readtemp_call>
                    | <readacc_call>
                    | <readtime_call>

<primvoidfun_call>::= <togglecam_call>
                    | <takepic_call>
                    | <connectwifi_call>
                    | <takeof_call>
                    | <land_call>
                    |  <changealt_call>
                    |  <rotate_call>
                    |  <goforward_call>
                    |  <gobackward_call>
```

<readincl_call>::= READINC LP RP

<readalt_call>::= READALT LP RP

<readtemp_call>::= READTEMP LP RP

<readacc_call>::= READACC LP RP

<readtime_call>::= READTIME LP RP

<togglecam_call>::= TOGGLECAM LP RP

<takepic_call>::= TAKEPIC LP RP

<connectwifi_call>::= CONNECTWIFI LP RP

<takeof_call>::= TAKEOF LP RP

<land_call>::= LAND LP RP

<changealt_call>::= CHANGEALT LP <primitive_value> RP | CHANGEALT LP IDENTIFIER RP

<rotate_call>::= ROTATE LP <primitive_value> RP | ROTATE LP IDENTIFIER RP

<goforward_call>::= GOFORWARD LP <primitive_value> RP | GOFORWARD LP IDENTIFIER RP

<gobackward_call>::= GOBACKWARD LP <primitive_value> RP | GOBACKWARD LP IDENTIFIER RP

```
<assign_stmt>::= IDENTIFIER ASSIGN_OP <primretfun_call>
                 | IDENTIFIER ASSIGN_OP <returnfun_call>
                 | IDENTIFIER ASSIGN_OP <expression>
                 | IDENTIFIER ASSIGN_OP STRING
                 | IDENTIFIER ASSIGN_OP CHAR
```

```
<decl_stmt>::= <func_decl>
                 | <var_decl>
                 | <const_decl>
```

```
<func_decl>::= <returnfunc_decl>
                 | <voidfunc_decl>
```

<returnfunc_decl>::= FUNCTION IDENTIFIER LP <dec_param_list> RP DOES <func_stmts> END

```
<voidfunc_decl>::= FUNCTION VOID IDENTIFIER LP <dec_param_list> RP DOES
<stmts> END

<func_stmts> ::= <func_stmt>
                  | <func_stmt> <func_stmts>


<func_stmt>::=  <func_if_stmt>
                  | <func_loop_stmt>
                  | <funcall_stmt>
                  | <assign_stmt>
                  | <decl_stmt>
                  | <return_stmt>
                  | <output_stmt>
                  | <input_stmt>

<func_if_stmt>::= IF LP <logical_exprs> RP  BEGIN <func_stmts> END
                  | IF LP <logical_exprs>RP  BEGIN <func_stmts> END ELSE BEGIN
                  <func_stmts> END
<func_loop_stmt>::= <func_while_loop> | <func_for_loop>
<func_while_loop> ::= WHILE LP <logical_exprs> RP BEGIN <func_stmts> END
<func_for_loop> ::= FOR LP <decl_stmt> SC <logical_exprs> SC <assign_stmt> RP BEGIN
<func_stmts> END

<return_stmt>::= RETURN STRING
                  | RETURN CHAR
                  | RETURN <expression>
                  | RETURN <returnfunc_call>
                  | RETURN <primretfun_call>

<dec_param_list>::=
                  | IDENTIFIER
                  | IDENTIFIER COMMA <dec_param_list>
```

```
<call_param_list>::=
                    | <primitive_value>
                    | IDENTIFIER
                    | <returnfuc_call>
                    | <primretfun_call>
                    | IDENTIFIER COMMA <call_param_list>
                    | <returnfuc_call> COMMA <call_param_list>
                    | <primretfun_call> COMMA <call_param_list>
                    | <primitive_value> COMMA <call_param_list>

<expression>::= <logical_exprs>
                     | <arith_exprs>
<logical_exprs> ::= <logical_exprs> OR <logical_term>
                    |<logical_term>
<logical_term>::= <logical_term> AND <logical_expr>
                    | <logical_expr>
<logical_expr> ::= LP <logical_exprs> RP
                    | <comparison>
                    | FALSE
                    | TRUER
                    |  NOT LP <logical_exprs> RP
<comparison>::= IDENTIFIER <logic_op> IDENTIFIER
                    | <primitive_value> <logic_op> IDENTIFIER>
                    | IDENTIFIER <logic_op> <primitive_value>
                    | <returnfun_call> <logic_op> IDENTIFIER
                    | IDENTIFIER <logic_op> <returnfun_call>
                    | <returnfun_call> <logic_op> <returnfun_call>
                    | <primitive_value> <logic_op> <returnfuc_call>
                    | <returnfuc_call> <logic_op> <primitive_value>
                    | <primretfun_call> <logic_op> <returnfuc_call>
                    | <returnfuc_call> <logic_op> <primretfun_call>
                    | <primretfun_call> <logic_op> IDENTIFIER
                    | IDENTIFIER <logic_op> <primretfun_call>
                    | <primretfun_call> <logic_op> <primitive_value>
                    | <primitive_value> <logic_op> <primretfun_call>
                    | <primretfun_call> <logic_op> <primretfun_call>

<logic_op>::= EQ | GT | GTE | LT | LTE | NE

<arith_expr>::= <arith_expr> PLUS <term>
                    | <arith_expr> MINUS <term>
                    | <term>

<term>::= <term> MULT <base_term>
                    | <term> DIV <base_term>
                    | <base_term>

<base_term>::= LP <arith_expr> RP
                    | INT
                    | DOUBLE
                    | IDENTIFIER

<var_decl>::= VAR IDENTIFIER
                    | VAR IDENTIFIER ASSIGN_OP <returnfunc_call>
                    | VAR IDENTIFIER ASSIGN_OP <primretfun_call>
                    | VAR IDENTIFIER ASSIGN_OP STRING
                    | VAR IDENTIFIER ASSIGN_OP <arith_expr>
                    | VAR IDENTIFIER ASSIGN_OP CHAR
                    | VAR IDENTIFIER ASSIGN_OP TRUE
                    | VAR IDENTIFIER ASSIGN_OP FALSE

<const_decl>::= CONST IDENTIFIER ASSIGN_OP
<primitive_value>

<primitive_value> ::= INT | DOUBLE | STRING | CHAR | TRUE | FALSE
```

LANGUAGE EXPLANATION

DROLAN can be used for everyone. People who purchase a drone may not be computer engineers so DROLAN is designed as a programming language that is very close to daily language. Additionally, DROLAN is easy to use for computer engineers as its syntax is similar to existing programming languages such as Python and easy to understand for computer engineers. While we were designing DROLAN, we benefited from other languages' basic rules such as JavaScript and Python. A program in DROLAN starts with a “BEGIN” token and ends with an “END” token.

1. <program>::= BEGIN <stmts> END

The nonterminals above indicate that our program starts with a “BEGIN” token and ends with an “END” token. There are statements between these two tokens.

2. <stmts>::= <stmt>
| <stmt> <stmts>

The nonterminals above represent that the statements of our language can consist of one statement or multiple statements.

3. <stmt>::= <if_stmt>
| <loop_stmt>
| <funcall_stmt>
| <assign_stmt>
| <decl_stmt>
| <input_stmt>
| <output_stmt>
| <comment_stmt>

A statement can be an if statement, or a loop statement, or a function call, or an assignment statement, or a declaration statement, or an output statement, or an input statement, or a comment statement. Details of these statements are explained below.

4. <if_stmt>::= IF LP <logical_exprs> RP BEGIN <stmts> END
| IF (<logical_exprs>) BEGIN <stmts> END else BEGIN <stmts> END

An if statement can be in two forms, unmatched or matched with else. The first design was "IF (<logical_exprs>) BEGIN <stmts> END else BEGIN <stmts> END" for the matched with else and the second design was "IF LP <logical_exprs> RP BEGIN <stmts> END " for the unmatched with else.

5. <comment_stmt>::= COMMENT

A comment statement can be by a token "COMMENT" returned from lex file.

6. <input_stmt>::= INPUT LP IDENTIFIER RP | INPUT LP IDENTIFIER COMMA <print_expr> RP

An input statement consists of a token "INPUT" and an IDENTIFIER in parentheses. It is used to get input from the user. When the user enters an input, the input is assigned to the identifier. Also, we can get input from a file by using the second parse tree. In the second parse tree, we can enter the name of the file we want to take input from as a <print_expr>. These statements increase the writability of the language.

7. <output_stmt>::= OUTPUT LP <print_expr> RP | OUTPUT LP <print_expr> COMMA <print_expr> RP

An output statement consists of a token "OUTPUT" and print expression between parentheses. Also, we can write a text to a file by using the second parse tree. In the second parse tree, we can enter the name of the file we want to write as a <print_expr>. These statements increase the writability of the language.

```
8.<print_expr>::= IDENTIFIER
                | <primitive_value>
                | IDENTIFIER PLUS < print_expr>
                | <primitive_value>   PLUS <print_expr>
                | <primretfun_call> PLUS <print_expr>
                | <returnfun_call> PLUS <print_expr>
                | <primretfun_call>
                | <returnfun_call>
```

A print statement is used to give the user a message consisting of combinations of primitive values, identifiers, primitive return function calls and return function calls .

9. <loop_stmt>::= <while_loop> | <for_loop>

A loop statement can be a while loop statement or a for loop statement.

10. <while_loop> ::= WHILE LP <logical_exprs> RP BEGIN <stmts> END

While loops start with a "WHILE" token followed by a logical expression in parentheses. The code segment which will be inside the while loop starts with a "BEGIN" token. After writing the necessary statements inside the loop, the loop will be closed with an "END" token.

11. <for_loop> ::= FOR LP <decl_stmt> SC <logical_exprs> SC <assign_stmt> RP BEGIN <stmts> END

For loops start with a "FOR" token followed by a declaration statement, a logical expression and an assignment statement respectively in parentheses. The code segment which will be inside the for loop starts with a "BEGIN" token. After writing the necessary statements inside the loop, the loop will be closed with an "END" token.

12. <funcall_stmt>::= <returnfun_call>
| <voidfun_call>
| <primretfun_call>
| <primvoidfun_call>

A function call statement can call a function which has a return type, or a function that does not return anything (void), or primitive versions of these two.

13. <returnfun_call>::= IDENTIFIER LP <call_param_list> RP
<voidfun_call>::= VOID IDENTIFIER LP <call_param_list> RP

A function call statement consists of the name of the function and the parameters of the function. Functions without return are called with a "VOID" token.

14. <primretfun_call>::= <readincl_call>
| <readalt_call>
| <readtemp_call>
| <readacc_call>

| <readtime_call>

A primitive return function call statement is used to call a primitive function which has return value. For the language to be reliable, these statements can be used in the right hand side of the assignment statement or as a single statement.

15. <primvoidfun_call>::= <togglecam_call>
| <takepic_call>
| <connectwifi_call>
| <takeof_call>
| <land_call>
| <changealt_call>
| <rotate_call>
| <goforward_call>
| <gobackward_call>

A primitive void function call statement is used to call a primitive function which has no return value. For the language to be reliable, these statements can be used just as a single statement.

16. <readincl_call>::= READINC LP RP

This primitive function is used to read the inclination of the drone and it returns a primitive value.

17. <readalt_call>::= READALT LP RP

This primitive function is used to read the altitude of the drone and it returns a primitive value.

18. <readtemp_call>::= READTEMP LP RP

This primitive function is used to read the temperature and it returns a primitive value.

19. <readacc_call>::= READACC LP RP

This primitive function is used to read the acceleration of the drone and it returns a primitive value.

20. <readtime_call>::= READTIME LP RP

This primitive function is used to read the time and it returns a primitive value.

21. <togglecam_call>::= TOGGLECAM LP RP

This primitive function is used to turn on/off the camera of the drone.

22. <takepic_call>::= TAKEPIC LP RP

This primitive function is used to take pictures.

23. <connectwifi_call>::= CONNECTWIFI LP RP

This primitive function is used to connect to any available Wifi.

24. <takeof_call>::= TAKEOF LP RP

This primitive function is used to make the drone take of.

25. <land_call>::= LAND LP RP

This primitive function is used to make the drone land.

26. <changealt_call>::= CHANGEALT LP <primitive_value> RP | CHANGEALT LP IDENTIFIER RP

This primitive function is used to change the altitude of the drone and it takes primitive value or identifier which shows how many meters the height of the drone should change.

27. <rotate_call>::= ROTATE LP <primitive_value> RP | ROTATE LP IDENTIFIER RP

This primitive function is used to change the rotation of the drone and it takes primitive value or identifier which shows how many degrees the rotation of the drone should change.

28. <goforward_call>::= GOFORWARD LP <primitive_value> RP | GOFORWARD LP IDENTIFIER RP

This primitive function is used to change the position of the drone and it takes primitive value or identifier which shows how many meters the drone should go forward.

29. <gobackward_call>::= GOBACKWARD LP <primitive_value> RP | GOBACKWARD LP IDENTIFIER RP

This primitive function is used to change the position of the drone and it takes primitive value or identifier which shows how many meters the drone should go backward.

30. <assign_stmt>::= IDENTIFIER ASSIGN_OP <primretfun_call>
| IDENTIFIER ASSIGN_OP <returnfun_call>
| IDENTIFIER ASSIGN_OP <expression>
| IDENTIFIER ASSIGN_OP STRING
| IDENTIFIER ASSIGN_OP CHAR

An assignment statement is used to assign a value to an identifier. The value can be a function with return, or another identifier, or a primitive value, or an expression.

31. <decl_stmt>::= <func_decl>
| <var_decl>
| <const_decl>

A declaration statement can be a function declaration, or a variable declaration, or a constant declaration.

32. <func_decl>::= <returnfunc_decl>
| <voidfunc_decl>

A function declaration can be a declaration of a returning function or a void function.

33. <returnfunc_decl>::= FUNCTION IDENTIFIER LP <dec_param_list> RP DOES <func_stmts> END

This is the declaration of a return function. “FUNCTION” token is used with the name of the function at the start. Then parameters are put between the parentheses. “DOES” token is added for the readability of the function declaration. Inside the function, there are function statements. Although return statements can be included in the function statements (<func_stmts>), it is not forced. So the user must add return statements through the code (in <func_stmts>). The declaration ends with an “END” token.

34. <voidfunc_decl>::= FUNCTION VOID IDENTIFIER LP <dec_param_list> RP DOES <stmts> END

This is the declaration of a void function. Differently from <returnfunc_decl>, “VOID” token is added before the name of the function. Another difference is that there is no return statement and statements are indicated with <stmts>, which cannot include a return statement unlike <func_stmts>.

- Rules **35 through 40** are set to have a group of statements that can include return statements since <stmts> cannot and should not include a return statement (because <stmts> is used outside of functions too). So the difference between <func_stmts> and <stmts> is that <func_stmts> can include return statements. There are also different if and loop statements (<func_if_stmt> and <func_loop_stmt>) which can include return statements. <func_stmts> is used only in the declaration of a return function.

35. <func_stmts> ::= <func_stmt>
| <func_stmt> <func_stmts>

Function statements consist of one or more function statements.

36. <func_stmt>::= <func_if_stmt>
| <func_loop_stmt>
| <funcall_stmt>
| <assign_stmt>
| <decl_stmt>
| <return_stmt>
| <output_stmt>
| <input_stmt>

A function statement can be anything that a normal statement can be. The difference is that a function statement can also be a return statement. In addition, if it is an if statement or a loop statement, they are able to include a return statement in them.

37. <func_if_stmt>::= IF LP <logical_exprs> RP BEGIN <func_stmts> END
| IF LP <logical_exprs>RP BEGIN <func_stmts> END ELSE BEGIN <func_stmts> END

This is an if statement but it has <func_stmts> inside of it instead of <stmts>.

38. <func_loop_stmt>::= <func_while_loop>

| <func_for_loop>

A function loop statement can be a function while statement or a function loop statement.

39. <func_while_loop> ::= WHILE LP <logical_exprs> RP BEGIN <func_stmts> END

This is a while statement but it has <func_stmts> inside of it instead of <stmts>.

40. <func_for_loop> ::= FOR LP <decl_stmt> SC <logical_exprs> SC <assign_stmt> RP BEGIN <func_stmts> END

This is a while statement but it has <func_stmts> inside of it instead of <stmts>.

41. <return_stmt>::= RETURN STRING

| RETURN CHAR

| RETURN <expression>

| RETURN <returnfunc_call>

| RETURN <primretfun_call>

Return statements are written with a “RETURN” token and return value. Return value can be a primitive value, or an expression, or a return function call.

42. <dec_param_list>::=

| IDENTIFIER

| IDENTIFIER COMMA <dec_param_list>

Parameter lists that are used in function declarations can include identifiers only.

43. <call_param_list>::=

| <primitive_value>

| IDENTIFIER

| <returnfuc_call>

| <primretfun_call>

| IDENTIFIER COMMA <call_param_list>

| <returnfuc_call> COMMA <call_param_list>

| <primretfun_call> COMMA <call_param_list>

| <primitive_value> COMMA <call_param_list>

Parameter lists that are used in function calls can include combination of values which can be primitive values, identifiers, or returning function calls.

44. <expression>::= <logical_exprs>

| <arith_exprs>

An expression can be a logical or an arithmetic expression.

45. <logical_exprs> ::= <logical_exprs> OR <logical_term>

|<logical_term>

A group of logical expressions consists of logical expressions with “OR” tokens among them.

46. <logical_term>::= <logical_term> AND <logical_expr>

| <logical_expr>

A group of logical term consists of logical expressions with “AND” tokens among them. Thanks to this, “AND” token can get precedence over “OR”.

47. <logical_expr> ::= LP <logical_exprs> RP

| <comparison>

| FALSE

| TRUE

| NOT LP <logical_exprs> RP

A logical expression can be logical expressions between parentheses, or a comparison, or “FALSE” token, or “TRUE” token, or not logical operation with logical expressions between parentheses. Thanks to this, a logical expression between parentheses can get precedence over “AND”.

48. <comparison>::= IDENTIFIER <logic_op> IDENTIFIER

| <primitive_value> <logic_op> IDENTIFIER>

| IDENTIFIER <logic_op> <primitive_value>

| <returnfun_call> <logic_op> IDENTIFIER

| IDENTIFIER <logic_op> <returnfun_call>

| <returnfun_call> <logic_op> <returnfun_call>

| <primitive_value> <logic_op> <returnfuc_call>

| <returnfuc_call> <logic_op> <primitive_value>

| <primretfun_call> <logic_op> <returnfuc_call>

| <returnfuc_call> <logic_op> <primretfun_call>

| <primretfun_call> <logic_op> IDENTIFIER

| IDENTIFIER <logic_op> <primretfun_call>

| <primretfun_call> <logic_op> <primitive_value>

| <primitive_value> <logic_op> <primretfun_call>

| <primretfun_call> <logic_op> <primretfun_call>

A comparison can be used to compare two values which have a logic operator between them. A value can be a return function call, or an identifier, or a primitive value.

49. <logic_op>::= EQ | GT | GTE | LT | LTE | NE

A logic operator can be "==", ">", ">=", "<", "<=", "!=".

50.<arith_expr>::= <arith_expr> PLUS <term>

| <arith_expr> MINUS <term>

| <term>

An arithmetic expression consists of <term>s added and subtracted.

52. <term>::= <term> MULT <base_term>

| <term> DIV <base_term>

| <base_term>

A term consists of <base_term>s multiplied and divided. Thanks to this, MULT and DIV operations has precedence over MINUS and PLUS.

53. <base_term>::= LP <arith_expr> RP

| INT

| DOUBLE

| IDENTIFIER

A base term can be an arithmetic expression in parentheses, or a value, or an ıdentifier. These last three rules are made this way to provide a priority order. Parentheses have priority over multiplication and division. Multiplication and division have priority over addition and subtraction.

54. <var_decl>::= VAR IDENTIFIER

| VAR IDENTIFIER ASSIGN_OP <returnfunc_call>

| VAR IDENTIFIER ASSIGN_OP <primretfun_call>

| VAR IDENTIFIER ASSIGN_OP STRING

| VAR IDENTIFIER ASSIGN_OP <arith_expr>

| VAR IDENTIFIER ASSIGN_OP CHAR

| VAR IDENTIFIER ASSIGN_OP TRUE

| VAR IDENTIFIER ASSIGN_OP FALSE

A variable declaration consists of a “VAR” token and an identifier. Optionally it can be assigned a value at the moment of declaration.

55. <const_decl>::= CONST IDENTIFIER ASSIGN_OP <primitive_value>

A constant declaration consists of a “CONST” token, an identifier, an assignment operator and then a primitive value.

56. <primitive_value> ::= INT | DOUBLE | STRING | CHAR | TRUE | FALSE

A primitive value can be integer, or double, or string, or true, or false, or char.

In DROLAN, there is no conflict left unresolved.

DESCRIPTIONS OF TOKENS

BEGIN token is used to indicate the beginning of something. For example the whole program and the insides of if and loop statements start with this token.

END token has a similar usage to begin token except it is used to indicate the end of something.

IF token is used to indicate an if (conditional) statement. If the logical expression in parentheses that comes right after the token is true, the statements that are between “begin” and “end” tokens will be executed.

ELSE token can be used if combined with an “if” token. It comes after the statements that are normally between the “begin” and “end” tokens of the associated “if” token. It divides the statements into two groups. If the logical expression is true, only the group before the “else” token is executed but if it is false, only the group after the “else” token is executed. One “begin” and one “end” tokens are used per an if-else pair.

WHILE token is used to indicate a while loop statement.

FOR token is used to indicate a for loop statement.

VOID token is used to declare a function that does not return anything.

FUNCTION token is used when a function is declared.

DOES token indicates that a function does the following statements until the end token.
RETURN token is used when a variable will be returned.
VAR token is used when a variable is declared
CONST token is used when a constant is declared**.**
INPUT token is used to prompt users to enter inputs to the program.
OUTPUT token is used to display values on the console.
IDENTIFIER token is used to give name to functions and variables. It can be a non reserved word beginning without numbers.
STRING token is used to form a string. A string can be defined between “ characters.
CHAR token is used to form a char. A char can be defined between ‘ charactes.
INT token is used to form an integer. It consists of numbers.
DOUBLE token is used to form a double. It consists of one or two integers and there is . character between integers.