CS 315 Programming Languages

PROJECT 1 - "DROLAN"

A Programming Language for Drones and its Lexical Analyzer

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 (<logical_exprs>) begin <stmts> end
| if (<logical_exprs>) begin <stmts> else <stmts> end

<comment_stmt>::= ## <word>

<input_stmt>::= input(<identifier>)

<output_stmt>::= output(<print_expr>)

<print_expr>::= <identifier>
| <primitive_value>
| <identifier> + < print_expr>
| <primitive_value> + <print_expr>

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

<while_loop> ::= while (<logical_exprs>) begin <stmts> end

<for_loop> ::= for (<decl_stmt> ; <logical_exprs> ; <assign_stmt>) begin <stmts> end

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

<returnfun_call>::= <identifier> (<call_param_list>?)
<voidfun_call>::= void <identifier> (<call_param_list>?)

<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()
<readalt_call>::= readAlt()
<readtemp_call>::= readTemp()
<readacc_call>::=readAcc()
<readtime_call>::=readTime()
<togglecam_call>::=toggleCam()
<takepic_call>::=takePic()
<connectwifi_call>::=connectWifi()

<takeof_call>::=takeOf()

<land_call>::=land()

<changealt_call>::= changeAlt (<primitive_value>)

<rotate_call>::= rotate (<primitive_value>)

<goforward_call>::= goForward (<primitive_value>)

<gobackward_call>::= goBackward (<primitive_value>)

<assign_stmt>::= <identifier> = <primretfun_call>

| <identifier> = <returnfun_call>

| <identifier> = <identifier>

| <identifier> = <primitive_value>

| <identifier> = <expression>

| <identifier> = input

<decl_stmt>::= <func_decl>

| <var_decl>

| <const_decl>

<func_decl>::= <returnfunc_decl>

| <voidfunc_decl>

<returnfunc_decl>::= function <identifier> (<dec_param_list>?) does <func_stmts>

<return_stmt> end

<voidfunc_decl>::= function void <identifier> (<dec_param_list>?) 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>

<func_if_stmt>::= if (<logical_exprs>) begin <func_stmts> end
| if (<logical_exprs>) begin <func_stmts> else <func_stmts> end
<func_loop_stmt>::= <func_while_loop> | <func_for_loop>
<func_while_loop> ::= while (<logical_exprs>) begin <func_stmts> end
<func_for_loop> ::= for (<decl_stmt> ; <logical_exprs> ; <assign_stmt>) begin
<func_stmts> end

<return_stmt>::= return <value>

<value>::= <primitive_value>
| <identifier>
| <expression>
| <primretfun_call>
| <returnfun_call>

<dec_param_list>::= <identifier>
| <identifier> , <dec_param_list>

<call_param_list>::= <value>
| <value> , <call_param_list>

<identifier>::= <word>
| <word> <identifier>

<word>::= <letter>
| <word><digit>
| <word><letter>

<expression>::= <logical_exprs> |
<arith_exprs>

<logical_exprs> ::= <logical_exprs> and <logical_expr>

| <logical_exprs> or <logical_expr>
| <logical_expr>

<logical_expr> ::= (<logical_exprs>)
| <value>
| <comparison>

<comparison>::= <value> <logic_op> <value>

<arith_expr>::= <arith_expr> + <term>
| <arith_expr> - <term>
| <term>

<term>::= <term> * <base_term>
| <term> / <base_term>
| <base_term>

<base_term>::= (<arith_expr>)
| <value>

<var_decl>::= var <identifier>
| var <identifier> = <value>
| input(var <identifier>)

<const_decl>::= const <identifier> = <value>
| input(const <identifier>)

<primitive_value> ::= <integer> | <double> | <string> | <boolean> | <char>

DEFINITIONS

<integer>::= <unsigned_integer>

| <signed_integer>

<unsigned_integer>::= <digit>

| <unsigned_integer><digit>

<signed_integer>::= <sign> <unsigned_integer>

<sign>::= + | -

<string>::= <string_identifier> <string_element> <string_identifier>

<string_element><empty>

| <letter>

| <letter> <digit> <string>

| <letter> <string>

| <digit> <string>

<string_identifier>::= "

<boolean>::= true | false

<double>::= <unsigned_double>

| <signed_double>

<unsigned_double>::= <integer>.<decimal>

<signed_double>::= <sign><integer>.<decimal>

<decimal>:::=<digit>

|<digit><decimal>

<char>::= <char_identifier><letter><char_identifier>

<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

<char_identifier>::=’

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

<logical_op>:: => | < | == | <= | >= | !=

LANGUAGE EXPLANATION

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. Details of these statements are explained below.

4. <if_stmt>::= if (<logical_exprs>) begin <stmts> end

| if (<logical_exprs>) begin <stmts> else <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 part but “end” and “begin” tokens that were next to “else” were unnecessary.

5. <comment_stmt>::= ## <word>

A comment statement can be written using “##” token.

6. <input_stmt>::= input <identifier>

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. These statements increase the writability of the language.

7. <output_stmt>::= output(<print_expr>)

An output statement consists of a token “output” and print expression between parentheses.

8. <print_expr>::= <identifier>

| <primitive_value>

| <identifier> + < print_expr>

| <primitive_value> + <print_expr>

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

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 (<logical_exprs>) 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 (<decl_stmt> ; <logical_exprs> ; <assign_stmt>) 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> (<call_param_list>?)

<voidfun_call>::= void <identifier> (<call_param_list>?)

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()

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

17. <readalt_call>::= readAlt()

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

18. <readtemp_call>::= readTemp()

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

19. <readacc_call>::=readAcc()

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

20. <readtime_call>::=readTime()

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

21. <togglecam_call>::=toggleCam()

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

22. <takepic_call>::=takePic()

This primitive function is used to take pictures.

23. <connectwifi_call>::=connectWifi()

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

24. <takeof_call>::=takeOf()

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

25. <land_call>::=land()

This primitive function is used to make the drone land.

26. <changealt_call>::= changeAlt (<primitive_value>)

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

27. <rotate_call>::= rotate(<primitive_value>)

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

28. <goforward_call>::= goForward (<primitive_value>)

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

29. <gobackward_call>::= goBackward (<primitive_value>)

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

30. <assign_stmt>::= <identifier> = <primretfun_call>

| <identifier> = <returnfun_call>

| <identifier> = <identifier>

| <identifier> = <primitive_value>

| <identifier> = <expression>

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> (<dec_param_list>?) does <func_stmts> <return_stmt> end

This is the declaration of a returning 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 and at the end a return statement. Although return statements can be included in the function statements (<func_stmts>), it is not forced. So if there was no mandatory return statement (<return_stmt>) at the end, the function might not have any which is a problem since it is a returning function. So the user can add return statements through the code (in <func_stmts>) but the user must also add one at the end. The declaration ends with an “end” token.

34. <voidfunc_decl>::= function void <identifier> (<dec_param_list>?) 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 returning 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>

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 (<logical_exprs>) begin <func_stmts> end

| if (<logical_exprs>) begin <func_stmts> else <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 (<logical_exprs>) 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 (<decl_stmt> ; <logical_exprs> ; <assign_stmt>) begin <func_stmts> end

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

41. <return_stmt>::= return <value>

Return statements are written with a “return” token and return value.

42. <value>::= <primitive_value>

| <identifier>

| <primretfun_call>

| <returnfun_call>

A value can be a primitive value, or an identifier, or a returning function call.

43. <dec_param_list>::= <identifier>

| <identifier> , <dec_param_list>

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

44. <call_param_list>::= <value>

| <value> , <call_param_list>

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

45. <identifier>::= <word>

| <word> <identifier>

An identifier consists of one or more words.

46. <word>::=<letter>

| <word><digit>

| <word><letter>

A word consists of letters and optionally digits but it has to start with a letter.

47. <expression>::=<logical_exprs>

| <arith_exprs>

An expression can be a logical or an arithmetic expression.

48. <logical_exprs> ::= <logical_exprs> and <logical_expr>

| <logical_exprs> or <logical_expr>

| <logical_expr>

A group of logical expressions consists of logical expressions with “and” and “or” tokens among them. These tokens do not have priority over each other, their order is important.

49. <logical_expr> ::= (<logical_exprs>)

| <value>

| <comparison>

A logical expression can be a group of logical expressions in parentheses, or a value, or a comparison.

50. <comparison>::= <value> <logic_op> <value>

A comparison can be written with two values which have a logical operator between them.

51. <arith_expr>::= <arith_expr> + <term>

| <arith_expr> - <term>

| <term>

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

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

| <term> / <base_term>

| <base_term>

A term consists of <base_term>s multiplied and divided.

53. <base_term>::= (<arith_expr>)

| <value>

A base term can be an arithmetic expression in parentheses, or a value. 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> = <value>

| input var <identifier>

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> = <value>

| input const <identifier>

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

56. <primitive_value> ::= <integer> | <double> | <string> | <boolean> | <char>

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

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
token is placed before comments.
input token is used to prompt users to enter inputs to the program.
output token is used to display values on the console.

EVALUATION OF DROLAN

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. To avoid ambiguities caused by the conditional statements, we defined if statements in such a way that each if statement will start with “begin” token and end with “end” token. These words are chosen because they are understandable and easy to write (short) for everyone. These tokens make users track easily while the users are reading a program. Naming of the predefined functions is very similar to

natural language. For example, we have used “does”, “begin” and “end” tokens to make DROLAN understandable. Namely, we can say that DROLAN is a readable programing language. We have precedence rules for arithmetic expressions and logical expressions to prevent ambiguity and increase readability. For example, multiplication, division has precedence when we compare these expressions to addition and subtraction.

In terms of writability there are some advantages and disadvantages. For example, there are multiple ways to take input from the user. Programmers are able to use the “input” token to give the input value to a variable. They can also use the “input” token while declaring a variable or a constant. So they do not have to write multiple lines of code to use the input value for declaration. Matched if statements (which contain a matched “if”-“else” token pair) use only one “begin” and only one “end” token instead of using two of each. Namely, programmers do not have to write unnecessary “end” and “begin” tokens which come before and after the “else” token respectively. In DROLAN language, the returning function declarations have to have a return statement at the end of them to make them have at least one return statement. It is not good in terms of writability because the user might have to add a dummy return statement just to satisfy this condition.

EXAMPLE PROGRAMS WRITTEN IN DRALON

1.

```
begin
    ##This program will take temperature values from the user and will allow the drone to
    ## take on between the given temperature values while time is less than 5000.
    connectWifi()
    output(  “Please enter the temperature values which drone can take on between them
(First lower and second higher)” )
    input (var lower)
    input (var higher)
    while ( readTime() < 5000 )
    begin
        if( readTemp() < higher and readTemp() > lower )
        begin
            takeOf()
        else
```

```
                land()
            end
        end
        land()
end
```

2.

```
begin
        ##This program will show a menu to users to control the drone while time is less than
        5000.
        connectWifi()
        const time = 5000
        while( readTime() <  time )
        begin
            output( “Please press 1 to land” )
            output( “Please press 2 to take of “ )
            output( “Please press 3 to toggle camera” )
            output( “Please press 4 to take picture” )
            input ( var option )
            if( option == 1)
            begin
                land()
            end
            if(  option == 2 )
            begin
                takeOf()
            end
            if( option == 3 )
            begin
                toggleCam()
            end
            if( option == 4)
            begin
                takePic()
```

```
            end
        end
end
```

3.

```
begin
        ## Declare constants, modify variables
        ## Shows how a for loop operates


        const intVar = 1000
        const doubleVar = 4.10
        const stringVar = “string”
        const charVar = ‘O’

        var x = “string”
        var y = 4
        var z = .31

        for ( var  i = 1; i < y; i = i + 1) begin
            if ( i < 3)
            begin
                var param = i * 1000
                goForward ( param )
            end
        end
end
```

4.

```
begin
```

```
        ## This program is to show how to implement functions
        var param1 = 4
        var param2 = 7

        ## function with 2 parameters
        function func (param1,  param2 ) does
                if ( param1 == param2) begin
                        var temp = readTemp()
                        return param1 + param2
                end
                return   param1 - param2
        end

        ## function with no parameters
        function noParam() does
                if (param1 >= param2 )  begin
                        return param1
                end
                return param2
        end


        ## function with no return type
        function void noReturn () does
                param1 = param1 + param2
        end
end
```