Bilkent University
CS 315 - Programming Languages
Project 1 Report

Batuhan TOSYALI 21702055
Atay KAYLAR 21703284
İrem SEVEN 21704269

Name Of The Language:

M.U.D.O. (Multi Utility Drone Objectives)

1 BNF of the Language

```
<program> ::= <stmtList>
<stmtList> ::= <stmt> SEMICOLON <stmtList> | <stmt> SEMICOLON

<stmt>   ::= <declaration_stmt>
                  | <assign_stmt>
                  | <init_stmt>
                  | <if_stmt>
                  | <loop_stmt>
      | <expr>

<declaration_stmt>   ::= <funct_declaration>
                                                  | <var_declaration_list>

<var_declaration_list>   ::= <var_declaration>
                                                  |  <var_declaration>

<var_declaration> ::= TYPE  IDENTIFER_LITERAL

<funct_declaration> ::= FUNCTION IDENTIFER_LITERALLP <parameter_list> RP  TYPE
LCB <stmt_list> RETURN <expr> SEMICOLON RCB
                   | FUNCTION IDENTIFER_LITERALLP <parameter_list> RP LCB stmt_list
RCB

 <parameter_list>    :  IDENTIFIER
                                         |   IDENTIFER_LITERALCOMMA <parameter_list>

<init_stmt>   ::= TYPE <assign_stmt>

<if_stmt> ::= IF LP <logic_expr> RP LCB <stmt_list> RCB
          | IF LP <logic_expr> RP LCB <stmt_list> RCB ELSE LCB <stmt_list> RCB

<loop_stmt>   ::= <while_stmt>
                          | <for_stmt>

<while_stmt>   ::= WHILE LP <logic_expr> RP    LCB  <stmt_list> RCB

<for_stmt>   ::= FOR LP <init_stmt> SEMICOLON  <logic_expr> SEMICOLON  <stmt>  RP
LCB <stmt_list> RCB

<expr>   ::= <arithmetic_expr_level1>

<arithmetic_expr_level1> ::= <arithmetic_expr> PLUS <arithmetic_expr_level2>
                                                  | <arithmetic_expr> MINUS
<arithmetic_expr_level2>
```

```
                                                        | <function_call_expression>
                                                        | <arithmetic_expr_level2>

<arithmetic_expr_level2> ::= <arithmetic_expr_level2> MULT <arithmetic_expr_level3>
                                                        | <arithmetic_expr_level2> DIV
<arithmetic_expr_level3>
                                                        | <arithmetic_expr_level3>

<arithmetic_expr_level3> ::= <arithmetic_expr_level3>  POW <arithmetic_expr_level4>
                                                        | <arithmetic_expr_level4>

<arithmetic_expr_level4> ::=  LP <arithmetic_expr_level1> RP
                                                        |  <logic_expr>

<logic_expr> ::= <logic_expr> LAND <logic_expr>
   | <logic_expr> LOR <logic_expr>
   | <logic_expr> LEQ <logic_expr>
   | <logic_expr> LNEQ <logic_expr>
   | <logic_expr> LT <logic_expr>
   | <logic_expr> LTE <logic_expr>
   | <logic_expr> GT <logic_expr>
   | <logic_expr> GTE <logic_expr>
   | LNOT LP <logic_expr> RP
   | BOOLEAN_LITERAL
   | INTEGER_LITERAL
   | FLOAT_LITERAL
   | STRING_LITERAL
   | IDENTIFIER_LITERAL

<function_call_expression> ::=  IDENTIFER_LITERAL LP  <parameter_list>  RP
                              | IDENTIFER_LITERAL LP RP
                              | CONNECT LP IP_ADDRESS COLON port RP
                              | READALTITUDE LP RP
                              | READACCELERATION LP RP
                              | READTEMPERATURE LP RP
                              | TURNONCAMERA LP RP
                              | TURNOFFCAMERA LP RP
                              | TAKEPICTURE LP RP
                              | READFROMTIMER LP RP
                              | READSPEEDX LP RP
                              | READSPEEDY LP RP
                              | READSPEEDZ LP RP
                              | WAIT LP FLOAT_LITERAL RP
                              | WAIT LP IDENTIFER_LITERAL RP
                              | PRINT LP <parameter_list> RP
                              | SETSPEEDX LP FLOAT_LITERAL RP
                              | SETSPEEDX LP IDENTIFER_LITERALRP
                              | SETSPEEDY LP IDENTIFER_LITERALRP
                              | SETSPEEDY LP FLOAT_LITERAL RP
                              | SETSPEEDZ LP FLOAT_LITERAL RP
                              | SETSPEEDZ LP IDENTIFER_LITERALRP
                              | FOLLOWCURVE LP FLOAT_LITERAL RP
                              | FOLLOWCURVE LP IDENTIFER_LITERALRP
                              | DISCONNECT LP RP
                              | READCURVE LP RP
                              | SETHEIGHT LP FLOAT_LITERAL RP
                              | SETHEIGHT LP IDENTIFER_LITERALRP
                              | RISE LP FLOAT_LITERAL RP
                              | RISE LP IDENTIFER_LITERALRP
                              | DESCEND LP FLOAT_LITERAL RP
                              | DESCEND LP IDENTIFER_LITERALRP
                              | LAND LP RP
                              | MOVEFORWARD LP FLOAT_LITERAL RP
                              | MOVEFORWARD LP IDENTIFER_LITERALRP
                              | MOVEBACK LP FLOAT_LITERAL RP
                              | MOVEBACK LP IDENTIFER_LITERALRP
                              | MOVERIGHT LP IDENTIFER_LITERALRP
                              | MOVERIGHT LP FLOAT_LITERAL RP
                              | MOVELEFT LP IDENTIFER_LITERALRP
                              | MOVELEFT LP FLOAT_LITERAL RP
                              | STARTVIDEO LP RP
                              | PAUSEVIDEO LP RP
                              | STOPVIDEO LP RP
                              | READBATTERY LP RP
                              | STOPMOTOR LP RP
                              | READLOCATION LP RP
                              | STARTTIMER LP RP
                              | STOPTIMER LP RP
                              | SCAN LP  RP
                              | SAVE LP <parameter_list> RP
                              | ROTATELEFT LP <parameter_list RP
                              | ROTATELEFT LP <parameter_list> RP
                              | ROTATERIGHT LP FLOAT_LITERAL RP
                              | ROTATERIGHT LP IDENTIFIER_LITERAL RP
                              | READINCLINE LP RP

<assign_stmt> ::= IDENTIFER_LITERALEQ <expr>


<if_stmt> ::= <matched_if_stmt>
                              | <unmatched_if_stmt>


<matched_if_stmt> ::= IF LP <logic_expr> RP LCB<matched>RCB ELSE
LCB<matched>RCB
```

<unmatched_if_stmt> ::= IF LP <logic_expr> RP <stmt>
|IF LP <logic_expr> RP <matched_if_stmt> ELSE <unmatched_if_stmt>

2 BNF Explanation

2.1 Terminals

<program> : The first terminal to start the program, it is the root in the language tree.

<stmtList> : sum of all the statements in the program, it can reduce into stmt or stmtList in the end, gives SEMICOLON token which is ";" to the end of the statements. Which means every statement in the language we designed has to end with the ";" character. We wanted our statements to look like in C type languages to increase the writability and readability.

<stmt>: A statement in MUDO can be classified as "declaration" "assign" "init" "if" "loop" or "expr", these statement types should not be unfamiliar to any reader since these statement types are essential for most of the imperative programming paradigm.

<**declaration_stmt**> a declaration statement must be either a function declaration or a variable declaration. There is not any other type of declaration in this language.

<**var_declaration>** a declaration of variable must happen with the following grammar: TYPE IDENTIFIER, the examples are the following:

int x;

where int is one of the types and x is the identifier.
Unfortunately this statement itself does not allow us to use declaration and assignment at the same time such as:

int x = 5; #which is wrong with only this statement

#but init_stmt will handle that

<funct_declaration> You can declare a new function using the word “function”, this is more similar to javascript than C or Java, since the language does not give the information of the return type. A really simple example could be

function func1(){};

This is the simplest function that can be written in this language, as you can see we don’t have to give a return type or a parameter. But you can give infinite amount of parameters without their types just like in python and javascript.

i**function func2(speed, velocity){ return speed * velocity; };**

<parameter_list> Parameter List is simply arranges us to give parameters in functions, with that we can call functions with more than one parameter

function func1(<parameter_lsit>){};

may give us

function func1(a , b , c , d … n){};

<init_stmt> Initialise statement solves the issue of declaring a variable and assigning it to a value with one statement such as

int x = 1;

<if_stmt> This rule has been borrowed by the book in order not to be ambiguous. With this statement our language gets the simple “if” “else” decision component of C type languages. But it lacks the “else if” statement. An example for if_stmt is the following

```
if ( true ){
        #do something
}
else{
        #do something else
};
```

or it can just be

if(true){};

<loop_stmt> Our language supports both the conventional while and for statements, but it does not support “do while” since we believe it will complicate our bnf.

<while_stmt> While statement takes a logical expression inside between left and right brackets, and it will continuously execute until the logical expression becomes logically false.

while(true){};

<for_stmt> For statement is similar to a C type for statement, but with a major difference, in some languages you can pass some of the arguments between the left parenthesis and right parenthesis. But in our language you must have 3 statements in for statement declaration to happen.

```
        for( int i =0; i < 10; i++ ){
        } #works

        int i = 0;
        for( i < 10; i++){
        } #wrong usage
```

<expr> the implementation of expression is a vast majority of the MUDO language. Before moving further, we should discuss what an expression is in this language. A stream of characters are considered as expression when at least two literals or identifiers are combined with the following operators : “+” , “-” , “*” , “/” , “^” , “()” and the logical operators that we know and use such as “>” , “<” , “==” etc. We could have defined these expressions at the same level, but we have to consider operator precedence. Therefore we shall start with the operators that has the least precedence.

<arithmetic_expr_level1> this level has the least precedence operators which are “+” and “-”, in order to write expressions that have the more precedence we shall go to the level 2

<arithmetic_expr_level2> This level contains “*” and “/” operators for multiplication and division respectively.

<arithmetic_expr_level3> This is a deeper level that contains the “^” operator in order to take power of an expression.

<arithmetic_expr_level4> that is the deepest level for arithmetic expressions. This level contains the “(“ and “)” since parenthesis should have the most precedence among the arithmetic operators. This level also goes deeper down in the tree with logical expressions.

<logic_expr> Logic expression contains the traditional logical operators such as “<” , “>” and “==” etc. In our language the logical expressions gets more precedence even than the parenthesis

(x + y) | | k && y
|| statements executes first, && second, () third and lastly +

logic_expr also contains the literal nonterminal such as STRING_LITERAL and other literals in order to show that literals also have the highest precedence among the expressions.

<function_call_expression> function call expression shows us how to call functions grammar wise, which is similar to C type languages

function func1(){};

#calling the function func1();

func1();

That is for the user defined functions, our language also contains a set of built in functions. Such as:

Connect(Ip_address , port); which connects the drone to the computer or any other device given in the Ip address and port

Disconnect( ); Drone automatically comes back to the starting point, then it disconnects the device.

ReadAltitude( ); returns the altitude of the drone.

ReadAcceleration( ); returns the acceleration of the drone (total acceleration in both three directions)

TurnOnCamera( ); Turns the camera of drone on

TakePicture( ); Takes a picture from the camera of drone

ReadFromTimer( ); reads from the timer that is embedded to the drone and returns it.

ReadSpeedX( ); Reads the velocity in x direction.

ReadSpeedY( ); Reads the velocity in y direction.

ReadSpeedZ( ); Reads the velocity in z direction.

SetSpeedX(x); Sets the speed in x direction.

SetSpeedY(y); Sets the speed in the y direction.

SetSpeedZ(z); Sets the speed in the z direction.

SetSpeed(x , y , z); Sets the speed of both of the three dimensions.

FollowCurve(str); Follows the curve for the drone, takes a string as a parameter.

ReadCurve(); Reads the curve of the drone, returns the value

Rise(r); Rises the drone with the given meters.

Descent(d); Descents the drone with the given meters.

Land( ); Automatically lands the drone to its corresponding coordinates.

MoveForward(f); Moves the drone forward with the given meters.

MoveBack(b); Move the drone backwards with the given meters.

MoveRight(r); Moves the drones rightwards with the given meters.

MoveLeft(l); Moves the drones leftwards with the given meters.

StopMotor( ); Stops the motor immediately

Wait(s); It is a built in busy loop which stalls the program.

Log(p); logs whatever is inside the function.

Scan( ); scans the input given in the console. and returns the result

Save( ); saves the picture or video, whatever happened lastly.

RotateRight(q); rotates the drone right with q degrees .

RotateLeft(q); rotates the drone left with q degrees .

ReadInclination() returns the slope due to the gyroscope.

Disconnect() disconnects the drone

<assign_stmt> This statements assigns a variable a certain value.

int k; # declaration statement
k = 5; # assign statement

2.2 Non-Terminals

INT_LITERAL Traditional Integer type in language. Which can rane between -2^31 and 2^31
FLOAT_LITERAL Traditional float type in language which can be defined with a dot between two integers such as 13.123.
BOOLEAN LİTERAL Traditional boolean type in language , which takes “true” or “false” .
STRING_LITERAL Traditional String type in language It can start with the trivial string “” and can go big as the memory enables, it can also take any character. .
IDENTIFIER This nonterminal can be any set of string but it cannot start with any number or operators defined in this language.
PLUS is the + operator
MINUS is the - operator
DIV is the / operator
MUL is the * operator
POW is the ^ operator.
IF If keyword is reserved for the if start of if statement
ELSE else keyword is reserved for the second part of if-else statement.
LP is the left parenthesis (
RP is the right parenthesis)
LCB is the left curly brackets {
RCB is the right curly brackets }
FOR represented by for keyword in our language
WHILE represented by While keyword in our language
FUNCTION Is a keyword for declaring functions “function”
RETURN Is a keyword for returning literals or identifiers from functions “return”, a function can at most use 1 return, and should return at most 1 value.
SEMICOLON is the semicolon for ending a statement “;”
COLON is the colon for distinguishing between parameters “,”
LOR logical or “||”
LEQ logical eq “==”
LNEQ logical not equal “!=”
LT logical less than “<”
LTE logical less than or equal to “=<”
GT logical greater than “>”

GTE logical greater than or equal to “>=”
LNOT logical not equal “!=”

The other not_terminals are the primitive functions which are described at the end of non-terminals.

Lastly we have the commenting non trivial token .

COMMENT anything comes after # in the line is not considered, they are considered as a comment whole.

3 Evaluation of the Language

3.1 Readability

Overall Simplicity

The program does not hold the “feature multiplicity”, we did not include multiple ways to accomplish the basic arithmetic in our language. Therefore the reader doesn’t have to learn new features of the language, which will complicate the reading process. Also the language does not include complicated operator overloading, no complicated constructs(like arrays or vectors) are applied to themselves.

Orthogonality

The language is considered orthogonal, float and integer can be applied to themselves.

Control Statements

if-else: The statement can be read like a regular C-like language, however it lacks the else-if part from other C-like languages. Therefore, the if-else statements are less complicated, the user can read the code more easily.

for: The statement can be read like a regular C-like, however the parameters must be 3-tuple. The code will become more neat, because the reader doesn’t have to find the other parts of the parameters outside the parentheses. All the parameters involving the for statements, will be inside the parameter list. It will make the language more readable.

while: It operates like a regular C-like language, it can be considered as readable.

Data Types and Structures

Boolean types are written as “true” and “false”, making the language more readable.

Syntax considerations

Identifier length is not limited, it is considered more readable for languages. Also the language supports concatenating characters such as “_”, therefore the coder can create distinct variables, which makes the code more readable.

3.2 Writability

Simplicity and orthogonality

Our program does not include complex constructs also it only includes 4 primitive types, which makes the program less complicated to learn and write for the coder. Also integers and floats can apply to themselves arithmetically.

Support for abstraction

The language does not include arrays, pointers or any other abstract data types. However we have functions, which reduces code redundancy

Expressivity

As it is explained in the readability part, our language isn’t as expressive as other C-like languages (eg. feature multiplicity). Thus, the M.U.D.O does not have more convenient and shorter ways of specifying computations. However, we defined “for” statements, which makes the loops easier to write.

3.3 Reliability

Aliasing

The language did not implement pointers, two pointers can not point to the same memory location,which means the problem of aliasing is non-existent.

Readability and Writability

Previously, properties of readability and writability were explained. Loosely speaking, our language is less complicated than most of the other languages, because it services for a very specific scenario, which requires less language properties. Thus, our language becomes more readable and writable, from its less complicated nature.

Cost

As explained above, our language is not complicated, which makes it easy to learn. The cost of training programmers and writing the code should not be as expensive as other languages.