Bilkent University

Cs315 Project-2

Team19: MRE

Muhammed Emre YILDIZ 21702825 Section-01
Abdullah Can ALPAY 21702686 Section-03

BNF Description of MRE Language

```
<program> ::= FLY <stmts> LAND

<stmts> ::= <stmts><stmt> |
<stmt> ::= <if_stmt> | <nonif_stmt> | COMMENT
<if_stmt> ::= IF LP <logic_expr> RP BEGIN2 <stmts> END| IF LP<logic_expr>RP
                                                  BEGIN2 <stmts> END ELSE BEGIN2
                                                  <stmts> END
<nonif_stmt>  ::= <variable> | <loops> |<funct_dec> | <func_call> |
                  <var_dec> | <input_stmt> | <output_stmt>

<types> ::= CHAR | INT | STRING | BOOLEAN | DOUBLE

<var_dec> ::= IDENTIFIER | <assignment_stmt>

<loops> ::= <while_stmt> | <for_stmt>

<while_stmt> ::= WHILE LP <logic_expr> RP BEGIN2 <stmts> END

<for_stmt> ::=FOR LP <var_dec> SEMICOLON <logic_expr> SEMICOLON
                                            <assignment_stmt>RP BEGIN2 <stmts> END

<funct_dec> ::=  <func_name> LP <func_params>RP BEGIN2<stmts>
                                               <return_statement> END

<return_statement> ::= RETURN | RETURN IDENTIFIER

<func_call> ::= <func_name> LP RP | <func_name> LP <func_params>RP |
                <primitive_func>

<func_name> ::= IDENTIFIER

<func_params> ::= <func_param> | <func_param> COMMA <func_params>

<func_param> ::= <variable> | IDENTIFIER

<assignment_stmt> ::= IDENTIFIER ASSIGNMENT_OP <logic_expr>

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

<output_stmt> ::=  OUTPUT LP IDENTIFIER RP | OUTPUT LP <variable> RP | OUTPUT
                   LP<func_call>RP

<primitive_func> ::=  MOVE LP RP | GET_ALTITUDE LP RP | GET_TEMPERATURE LP
                      RP | SNAP LP RP | GET_ACCELERATION LP RP | GET_TIME LP
                      RP | GET_CONNECTION LP RP | CAMERA_ON LP RP |
                      CAMERA_OFF LP RP

<logic_expr> ::= <or_expr>
```

<or_expr> ::= <and_expr> | <or_expr>OR<and_expr>

<and_expr> ::= <inc_or_expr> | <and_expr>AND<inc_or_expr>

<inc_or_expr> ::= <exc_or_expr> | <inc_or_expr> INR_OR<exc_or_expr>

<exc_or_expr> ::= <inc_and_expr> | <exc_or_expr> XOR <inc_and_expr>

<inc_and_expr> ::= <eq_comp> | <inc_and_expr> INC_AND <eq_comp>

**<eq_comp> ::= <relat_comp> | <eq_comp>EQUAL <relat_comp>
| <eq_comp> NOT_EQUAL<relat_comp>**

**<relat_comp> ::= <add_expr> | <relat_comp> LESSTHAN<add_expr> | <relat_comp>
GREATERTHAN<add_expr> | <relat_comp>LTE <add_expr>
| <relat_comp> GTE <add_expr>**

**<add_expr>::= <mult_expr> | <add_expr> PLUS<mult_expr>
| <add_expr>MINUS<mult_expr>**

**<mult_expr> ::= <prim_expr> | <mult_expr>MULTIPLICATION<prim_expr>
| <mult_expr> MOD<prim_expr>
| <mult_expr> DIVISION<prim_expr>**

**<prim_expr> ::= IDENTIFIER | <types> | LP <logic_expr> RP | <func_call>
| <input_stmt>**

<variable> :: = <types>

BNF Explanation of MRE Language

<program> ::= FLY <stmts> LAND

Program starts only with the one reserved word fly and ends with the reserved word land. Basically the program has three elements which are fly, land and proper statements.

<stmts> ::= <stmt> |<stmts><stmt> |
Statements can either be a single statement or consist of multiple statements.

<stmt> ::= <if_stmt> | <nonif_stmt> | COMMENT
For a proper statement, it can either be an if statement, non if statement or a comment.

**<if_stmt> ::= IF LP <logic_expr> RP BEGIN2 <stmts> END | IF LP <logic_expr> RP
BEGIN2 <stmts> END ELSE BEGIN2
<stmts> END**
For if statements, it's start with the if condition, followed by begin, single or multiple statements and end statement. In some cases if-else conditions can be used. if the condition is false do the else statements instead.

<nonif_stmt> ::= <variable> | <loops> |<funct_dec> | <func_call> |
<var_dec> | <input_stmt> | <output_stmt>

When the grammar considers the non_if statements, it ends up with the possibilities shown above. It can be variable declaration, loops, function declaration, function call, variable itself, input or output statements.

<types> ::= CHAR | INTEGER | STRING | BOOLEAN | DOUBLE
The grammar works like python, thus, <types> is not one of the necessities of the grammar. However, in some cases <types> becomes useful.

<var_dec> ::= IDENTIFIER | <assignment_stmt>
In our language variable declarations are similar to Python. It can be declared as a single variable like 'x' or variables can be declared with an assignment such as 'x = 5'.

<loops> ::= <while_stmt> | <for_stmt>
Two loops are considered in our language: while or for.

<while_stmt> ::= while LP<logic_expr> RP BEGIN2 <stmts> END
While statement declaration is simple: starts with while and it's logical expression, continue with the begin token and statements. Then ends with the end token. The while statement declaration is similar to the C language declaration.

<for_stmt> ::= FOR LP<var_dec> SEMICOLON <logic_expr> SEMICOLON
<assignment_stmt> RP BEGIN2 <stmts> END
For statement declaration is similar to the C++ language declaration where for consists of three phases. In the first phase the variable is declared. Declaration phase's difference from C++ is that declaration is similar to Python. In the second phase normal logical expression is declared. In the third phase arithmetic expression is declared. The phases are divided with the semicolon like in C++.

<funct_dec> ::= <func_name>LP<func_params> RP begin <stmts>
<return_statement> end
The function declaration is similar to Python. Functions do not need to specify the return type.

<return_statement> ::= RETURN | RETURN IDENTIFIER
Return statement is similar to most of the languages where the identifier identifies the return parameter.

<func_call> ::= <func_name> LP RP | <func_name>LP<func_params> RP |
<primitive_func>
The function call is similar to most of the languages. Calling function with the wanted parameters with the right order or function call with no parameters.

<func_name> ::= IDENTIFIER
Function name identified with the identifier.

<func_params> ::= <func_param> | <func_param> COMMA <func_params>
For a single parameter in the <function params>, the grammar will choose the <func_param> and for the multiple parameters, lex recursively call itself.

<func_param> ::= <variable> | IDENTIFIER
Function parameters can be named according to the identifier, it can be just an identifier declaration or variable as a parameter.

<assignment_stmt> ::= IDENTIFIER ASSIGNMENT_OP <logic_expr>
Assignment statement assigns the identifier to the right side.

<input_stmt> ::= INPUT LP IDENTIFIER RP | INPUT LP RP
Declares which type of input will be scanned.

<output_stmt> ::= OUTPUT LP IDENTIFIER RP | OUTPUT LP<variable> RP | OUTPUT LP<func_call> RP
Declares which variable or identifier to print.

<primitive_func> ::= MOVE LP RP | GET_ALTITUDE LP RP | GET_TEMPERATURE LP RP | SNAP LP RP | GET_ACCELERATION LP RP | GET_TIME LP RP | GET_CONNECTION LP RP | CAMERA_ON LP RP | CAMERA_OFF LP RP
Primitive functions are the implemented functions. These functions will already be given to the user.

<logic_expr> ::= <or_expr>
This is the logic expression and it considers the priority of the operations. The definitions given below are listed from least to the highest priority. The reason it is defined in this way is to take into account the priorities of operations in mathematics over each other. It can be thought of as a tree.

<or_expr> ::= <and_expr> | <or_expr> OR <and_expr>
In order to avoid ambiguity and considering the priority of the functions, the or expression is defined firstly followed by an expression. The reason it is defined in this way is to take into account the priorities of operations over each other.

<and_expr> ::= <inc_or_expr> | <and_expr> AND <inc_or_expr>
In order to avoid ambiguity and considering the priority of the functions, the and expression is defined following by inclusive or expression. The reason it is defined in this way is to take into account the priorities of operations over each other.

<inc_or_expr> ::= <exc_or_expr> | <inc_or_expr> INC_OR <exc_or_expr>
In order to avoid ambiguity and considering the priority of the functions, the inclusive or expression is defined following by exclusive or expression. The reason it is defined in this way is to take into account the priorities of operations over each other.

<exc_or_expr> ::= <inc_and_expr< | <exc_or_expr> XOR <inc_and_expr>
In order to avoid ambiguity and considering the priority of the functions, the exclusive or expression is defined following by inclusive and expression. The reason it is defined in this way is to take into account the priorities of operations over each other.

<inc_and_expr> ::= <eq_comp> | <inc_and_expr> INC_AND <eq_comp>
In order to avoid ambiguity and considering the priority of the functions, the inclusive and expression is defined as follows by equality comparator expression. The reason it is defined in this way is to take into account the priorities of operations over each other.

<eq_comp> ::= <relat_comp> | <eq_comp> EQUAL <relat_comp>
| <eq_comp> NOT_EQUAL <relat_comp>
In order to avoid ambiguity and considering the priority of the functions, the equality comparator expression is defined following the relations comparator expression. The reason it is defined in this way is to take into account the priorities of operations over each other.

<relat_comp> ::= <add_expr> | <relat_comp> LESSTHAN <add_expr> | <relat_comp>
GREATERTHAN <add_expr> | <relat_comp> LTE <add_expr>
| <relat_comp> GTE <add_expr>
Rest of the relation comparators are defined in the same.

<add_expr>::= <mult_expr> | <add_expr> PLUS <mult_expr>
| <add_expr> MINUS <mult_expr>
In order to avoid ambiguity and considering the priority of the functions, the least priority functions are addition and subtraction. Therefore they are defined firstly. The reason it is defined in this way is to take into account the priorities of operations over each other.

<mult_expr> ::= <prim_expr> | <mult_expr> MULTIPLICATION <prim_expr>
| <mult_expr> MOD <prim_expr>
| <mult_expr> DIVISION <prim_expr>
In order to avoid ambiguity and considering the priority of the functions, more priority functions are multiplication and division. Therefore they are defined after the addition and subtraction. The reason it is defined in this way is to take into account the priorities of operations over each other.

<prim_expr> ::= IDENTIFIER | <types> | LP<logic_expr>RP | <func_call>
| <input_stmt>
In order to avoid ambiguity and considering the priority of the functions, the most priority function is functions that are defined into parentheses. Therefore parentheses are defined lastly. The reason it is defined in this way is to take into account the priorities of operations over each other.

<variable> :: = <types>
The types <denotes> the variation of the <variable>.

Precedence Rules in MRE

From the logic expressions it can understand that MRE has a presence for the operations. Just like most of the languages MRE also follows the following order: Multiplication, division operations are executed first. Then summation and subtraction are executed. XOR, inclusive and and inclusive or are higher priority than AND and OR which led And and or executed last.

Reserved Words used in MRE

fly, land, move, connection, altitude, time, temperature, snap, begin, end, while, if, for, int, double, char, string, return, camera_on, camera_off

Tokens in the MRE

IDENTIFIER
The identifier can be a single char, a char sequence or a sequence of char and a number (double or integer).

COMMENT
/* denotes the start of the comment and */ denotes the end of the comment.

STRING
Anything that is between “” denotes the string.

INTEGER
An integer can be single digit or a sequence of digits.

CHAR
Each letter in the alphabet denotes the char.

BOOLEAN
Boolean is detected when true-false statements are considered.

IF
Denotes the if condition.

ELSE
Denote the else condition in if-else statements.

FOR
Denotes the for loop.

WHILE
Denotes the while statement.

INPUT
Denote that the user will enter the value.

OUTPUT
Denotes that the value in the parentheses will be printed on to the console.

RETURN
Denotes the return statement.

LETTER
This token corresponds to the letters in the alphabet.

DIGIT
Denotes the numbers.

SIGN
Denotes the positive/negative sign.

ALPHANUMERIC
Denotes the words/variable names.

GET_ALTITUDE
Denotes the get_altitude() primitive function.

GET_TEMPERATURE
Denotes the get_temperature() primitive function.

SNAP
Denotes the snap() primitive function.

GET_ACCELERATION
Denotes the get_acceleration() primitive function.

GET_TIME
Denotes the get_time() primitive function.

GET_CONNECTION
Denotes the get_connection() primitive function.

CAMERA_ON
Denotes the camera_on() primitive function.

CAMERA_OFF
Denotes the camera_off() primitive function.

MOVE
Denotes the move() primitive function.

VOID
Denotes the void in C++.

BOOL
Denotes the bool operation.

LCB
Left curly braces denotes the Left Curly Braces (LCB)

RCB
Right curly braces denotes the Right Curly Braces (RCB)

LSB
Left brackets denote the LSB.

RSB
Right brackets denote the RSB.

COMMA
Comma is used to separate multiple elements.

PLUS
Plus is used for arithmetic operation which is summation.

MINUS
Minus is used for arithmetic operation which is subtraction.

MULTIPLICATION
Multiplication is used for arithmetic operation which is multiplication.

DIVISION
Division is used for arithmetic operation which is division.

MOD
Percentage is used for arithmetic operation which is mod operation.

OR
Used for logic or operation.

INC_OR
Used for inclusive or.

INC_AND
Used for inclusive and. This operation is used as a logical gate like x = a&b.

NOT_EQUAL
This operation is used as a conditional operation. It checks the inequality .

AND
Used for logic and operation.

XOR
Used for logic xor operation.

EQUAL
This is one of the other conditional operations. It checks the equality of the equation.

LESSTHAN
This operation determines whether the left side is smaller than the right side.

GREATERTHAN
This operation determines whether the left side is bigger than the right side.

LTE
This operation determines whether the left side is smaller or equal to the right side.

LP
Denotes the left parenthesis.

RP
Denotes the right parenthesis.

GTE
This operation determines whether the left side is bigger or equal to the right side.

NOT
This operator reverses the following variable or identifier.

SPACE
This operation indicates the space. Spaces are used widely during the programming.

DOT
Used in sentences or declaring double numbers.

ASSIGNMENT_OP
Used for assigning identifier to the other identifier, function return, expression or type.

SEMICOLON
Used for separating the phases in the for loop.

HASHTAG
Using for denoting the hashtag.

ENDLINE
Used for denoting a new line.

FLY
Used for starting to code. Without fly operation, the code will not be executed.

LAND
Denotes the end of the code.

BEGIN2
Used for indicating where the loops, functions or if statements start.

END
Used for indicating where the loops, functions or if statements finish.

Conflicts in the MRE

In the start, yacc file we encountered with the 150 reduce/reduce conflict error and 4 shift/reduce error. When we debugged the yacc and lex file we understand that these conflicts occur because of the parse ambiguity. We were able to solve the conflicts.

The lex and yacc file do not give any reduce/shift or reduce/reduce conflicts. During the test cases the program is able to succeed in the cases without any conflicts.