BILKENT UNIVERSITY

CS 315 - PROGRAMMING LANGUAGES

Project Group: 6

- Atasagun Şanap, 21902435, Section 2
- Melis Atun, 21901865, Section 1
- Öykü Erhan, 21901541, Section 2

Name of our Language: FLYRONE

BNF Description

<program> ::=
BEGIN_PR **<statement_block>** END

<statement_block> ::= **<statement>** | **<statement>** **<statement_block>**

<statement> ::= **<return_statement>** | **<conditional_statement>** |
<in_out_statement> | **<comment>** | **<assignment_statement>** |
<function_call_statement> | **<function_def_statement>** | **<loop_statement>** |
<primitive_function_call>

<return_statement> ::= RETURN LEFT_PAR **<expression>** RIGHT_PAR
SEMICOLON

<conditional_statement> ::=
IF LEFT_PAR **<expression>** RIGHT_PAR LEFT_BRACE **<statement_block>**
RIGHT_BRACE |
IF LEFT_PAR **<expression>** RIGHT_PAR LEFT_BRACE **<statement_block>**
RIGHT BRACE ELSE **<conditional_statement>** |
IF LEFT_PAR **<expression>** RIGHT_PAR LEFT_BRACE **<statement_block>**
RIGHT_BRACE ELSE **<statement_block>** RIGHT_BRACE

<in_out_statement> ::= **<input_statement>** | **<out_statement>**

<assignment_statement> ::=
<type> IDENTIFIER ASSIGNMENT_OP **<value>** SEMICOLON |
<type> IDENTIFIER ASSIGNMENT_OP IDENTIFIER SEMICOLON |
IDENTIFIER ASSIGNMENT_OP IDENTIFIER SEMICOLON |
<type> IDENTIFIER ASSIGNMENT_OP **<function_call_statement>** |
IDENTIFIER ASSIGNMENT_OP **<primitive_function_call>** |
<type> IDENTIFIER ASSIGNMENT_OP **<expression>** SEMICOLON |
IDENTIFIER ASSIGNMENT_OP **<expression>** SEMICOLON

<function_call_statement> ::= IDENTIFIER LEFT PAR RIGHT_PAR SEMICOLON | IDENTIFIER LEFT_PAR RIGHT_PAR **<parameters>** RIGHT_PAR SEMICOLON

<function_def_statement> ::= **<type>** IDENTIFIER LEFT_PAR LEFT_BRACE **<statement_block>** RIGHT_BRACE | **<type>** IDENTIFIER LEFT_PAR **<parameters>** RIGHT_PAR LEFT_BRACE **<statement_block>** RIGHT_BRACE **<statement_block>** RIGHT_BRACE | **<type>** IDENTIFIER LEFT_PAR RIGHT_PAR LEFT_BRACE **<statement_block>** RIGHT_BRACE

<loop_statement> ::= **<for_statement>** | **<while_statement>**

<input_statement> ::= IDENTIFIER ASSIGNMENT_OP INPUT LEFT_PAR RIGHT_PAR SEMICOLON | **<type>** IDENTIFIER ASSIGNMENT_OP LEFT_PAR RIGHT_PAR SEMICOLON

<out_statement> ::= PRINT LEFT_PAR **<value>** RIGHT_PAR SEMICOLON | PRINT LEFT_PAR IDENTIFIER RIGHT_PAR SEMICOLON | PRINT LEFT_PAR **<expression>** RIGHT_PAR SEMICOLON

<for_statement> ::= FOR LEFT_PAR **<assignment_statement>** **<comp_expression>** SEMICOLON **<statement>** RIGHT_PAR LEFT_BRACE **<statement_block>** RIGHT_BRACE

<while_statement>::= WHILE LEFT_PAR **<comp_expression>** RIGHT_PAR LEFT_BRACE **<statement_block>** RIGHT_BRACE
| WHILE LEFT_PAR **<boolean>** RIGHT_PAR LEFT_BRACE **<statement_block>** RIGHT_BRACE
| WHILE LEFT_PAR IDENTIFIER RIGHT_PAR LEFT_BRACE **<statement_block>** RIGHT_BRACE

<comp_expression> SC **<statement>** RIGHT_PAR LEFT_BRACE **<statement_block>** RIGHT_BRACE

<parameters> ::= <type> IDENTIFIER COMMA **<parameters>**
| IDENTIFIER COMMA **<parameters>**
| **<value>** COMMA **<parameters>**
| **<type> IDENTIFIER**
| **IDENTIFIER**
| **<value>**

<expression>::= <arith_expression> | <comp_expression>

<arith_expression> ::= <sum_expression>
| **<mult_expression>**
| **<sum_expression> <operation> <arith_expression>**
|**<mult_expression> <operation>arith_expression>**

<comp_expression>::= <arith_comp_expression> <arith_expression>
| **<arith_comp_expression> <factor>**
|**<factor_comp_expression> <arith_expression>**
|**<factor_comp_expression> <factor>**

<arith_comp_expression>::= <arith_expression> <general_comparator>

<factor_comp_expression>::= <factor> <general_comparator>

<mult_expression>::=<mult_operation_espression> <factor> |
<mult_operation_espression> <mult_expression>

<sum_expression> ::= **<sum_operation_expression> <factor> |**
<sum_operation_expression> <sum_expression>

<sum_operation_expression> ::= factor **<sum_sub_operation>**
<operation> ::= <mult_div_operation> | <sum_sub_operation>
<sum_sub_operation> ::= SUM | SUB

<mult_div_operation> ::= MULT | DIV

<factor> ::= IDENTIFIER | **<value>**
<boolean>::= true | false
<type> ::= **<double_type> | <int_type> | <string_type> | <bool_type>**
<double_type> ::= “double”
<int_type> ::= “int”
<string_type> ::= “string”
<bool_type> ::= “bool”
<general_comparator> ::= LT | LTE | GT | GTE | EE | NE | OR | AND
<value> ::= STRING | DOUBLE | **<boolean>** | INT

<primitive_function_call> ::= READ_HEADING LEFT_PAR RIGHT_PAR SEMICOLON
| READ_ALTITUDE LEFT_PAR RIGHT_PAR SEMICOLON
| READ_TEMPERATURE LEFT_PAR RIGHT_PAR SEMICOLON
| GO_UP LEFT_PAR INTEGER RIGHT_PAR SEMICOLON
| GO_UP LEFT_PAR IDENTIFIER RIGHT_PAR SEMICOLON
| GO_DOWN LEFT_PAR INTEGER RIGHT_PAR SEMICOLON
| GO_DOWN LEFT_PAR IDENTIFIER RIGHT_PAR SEMICOLON
| GO_FORWARD LEFT_PAR INTEGER RIGHT_PAR SEMICOLON
| GO_FORWARD LEFT_PAR IDENTIFIER RIGHT_PAR SEMICOLON
| GO_BACKWARD LEFT_PAR INTEGER RIGHT_PAR SEMICOLON
| GO_BACKWARD LEFT_PAR IDENTIFIER RIGHT_PAR SEMICOLON
| STOP_HORIZONTALLY LEFT_PAR RIGHT_PAR SEMICOLON
| MOVE_RIGHT LEFT_PAR INTEGER RIGHT_PAR SEMICOLON
| MOVE_RIGHT LEFT_PAR IDENTIFIER RIGHT_PAR SEMICOLON
| MOVE_LEFT LEFT_PAR INTEGER RIGHT_PAR SEMICOLON
| MOVE_LEFT LEFT_PAR IDENTIFIER RIGHT_PAR SEMICOLON
| STOP_TURN_LEFT LEFT_PAR INTEGER RIGHT_PAR SEMICOLON
| STOP_TURN_LEFT LEFT_PAR IDENTIFIER RIGHT_PAR SEMICOLON
| STOP_TURN_RIGHT LEFT_PAR INTEGER RIGHT_PAR SEMICOLON
| STOP_TURN_RIGHT LEFT_PAR IDENTIFIER RIGHT_PAR SEMICOLON
| TURN_ON_NOZZLE LEFT_PAR RIGHT_PAR SEMICOLON
| TURN_OFF_NOZZLE LEFT_PAR RIGHT_PAR SEMICOLON
| WAIT LEFT_PAR INTEGER RIGHT_PAR SEMICOLON
| WAIT LEFT_PAR IDENTIFIER RIGHT_PAR SEMICOLON

| TURN LEFT_PAR INTEGER RIGHT_PAR SEMICOLON
| TURN LEFT_PAR IDENTIFIER RIGHT_PAR SEMICOLON
| CONNECT_TO_BASE_COMPUTER_THROUGH_WIFI LEFT_PAR RIGHT_PAR SEMICOLON

print ::= print
return::= **return**
if ::= **if**
else ::= **else**
left_par ::= **"("**
right_par ::= **")"**
comma ::= **","**
dot ::= **"."**
mult ::= **"*"**
div ::= **"/"**
sum ::= \+
sub ::= \-
colon ::= **":"**
semicolon ::= **";"**
left_brace ::= **"{"**
right_brace ::= **"}"**
comment_line ::= **"//"**
digit ::= **[0-9]**
assignment_op ::= **"="**
int_type ::= **int**
string_type ::= **string**
bool_type ::= **bool**
double_type ::= **double**
true ::= **true**
false ::= **false**
while ::= **while**
for ::= **for**
equal ::= **"=="**
not_equal ::= **"!="**

greater_equal ::= **">="**
less_equal ::= **"<="**
greater ::= **">"**
less ::= **"<"**
or ::= **"||"**
and ::= **"&&"**
input ::= **input**
int ::= **{digit}+**
double ::= **{digit}*{dot}{digit}+**
read_heading::= READ_HEADING
read_altitude READ_ALTITUDE
read_temperature READ_TEMPERATURE
go_up GO_UP
go_down GO_DOWN
go_forward GO_FORWARD
go_backward GO_BACKWARD
stop_horizontally STOP_HORIZONTALLY
move_left MOVE_LEFT
move_right MOVE_RIGHT
stop_turn_left STOP_TURN_LEFT
stop_turn_right STOP_TURN_RIGHT
nozzle_turn_on TURN_ON_NOZZLE
Nozzle_turn_off ::= TURN_OFF_NOZZLE
wait ::= WAIT
turn ::= TURN
Connect_to_base_computer_through_wifi ::=
CONNECT_TO_BASE_COMPUTER_THROUGH_WIFI
begin ::= BEGIN_PR
end ::= END
comment ::= {comment_line}([^\n])*
identifier ::= [A-Za-z][A-Za-z0-9]*
string ::= \"([^\\\"]|\\\"|\\\n|\\\\)*\"
. ::= ;

Reserved Words

PRINT
RETURN
IF
ELSE
LEFT_PAR
RIGHT_PAR
COMMA
DOT
MULT
DIV
SUM
SUB
COLON
SEMICOLON
LEFT_BRACE
RIGHT_BRACE
COMMENT_BEGIN
COMMENT_END
ASSIGNMENT_OP
INT_TYPE
STRING_TYPE
BOOL_TYPE
DOUBLE_TYPE
TRUE
FALSE
WHILE
FOR
EQUAL
NOT_EQUAL
GREATER_EQUAL
LESS_EQUAL
GREATER
LESS

OR
AND
DOUBLE
INTEGER
INPUT
BEGIN_PR
END
IDENTIFIER
STRING
ANY_TEXT

READ_HEADING
READ_ALTITUDE
READ_TEMPERATURE
GO_UP
GO_DOWN
GO_FORWARD
GO_BACKWARD
STOP_HORIZONTALLY
STOP_TURN_LEFT
STOP_TURN_RIGHT
TURN_ON_NOZZLE
TURN_OFF_NOZZLE
WAIT
TURN
CONNECT_TO_BASE_COMPUTER_THROUGH_WIFI

Primitive Functions

<builtin_function_identification> ::= get_reading | get_alti | get_temp | go_vert | go_up | go_down | stop_horiz | go_forward | go_backward | stop_turn_left | stop_turn_right | turn_on_nozzle | turn_off_nozzle | connect_WIFI

read_heading
A function to read the heading.

read_altitude
A function to read the altitude.

read_temperature
A function to read the temperature.

go_up
A function to go in the upper direction.

go_down
A function to go in the downward direction.

stop_horizontally
A function to stop horizontally.

go_forward
A function to go forward.

go_backward
A function to go backward.

stop_turn_left
A function to stop and turn the heading left.

stop_turn_right

A function to stop and turn the heading right.

turn_on_nozzle
A function to turn on the nozzle.

turn_off_nozzle
A function to turn off the nozzle.

wait
A function to wait

turn
A function to turn around

connect_to_base_computer_through_wifi
A function to connect to the base computer through a WIFI.

Description of Non-Terminal Literals

<program> is the most general literal of the language. All of the codes that are written fall into this literal. This literal is composed of one or multiple **<statement_block>** or one **<statement>**.

<statement> statement is the building block of the whole language and is the most important one. It must be one of the five statement types. These are **<return_statement>**, **<conditional_statement>**, **<in_out_statement>**, **<comment>** or **<assignment_statement>**.

<statement_block> is the second most general literal. This refers to a group of code segments that is a part of the **<program>**. This literal is composed of a combination of a single **<statement>**, a group of **<statement>**, a group of **<function>**, and a group of **<loop>**.

<function> is a literal that is composed of statements or loops and their combinations. Note that this gives the flexibility of using loops in functions as well as in a proper language.

<loop_statement> is another complex literal that consists of **<for_statement>** loop or **<while_statement>** loop.

<for_statement> represents the for loop and is composed of one or more statements.
<while_statement> represents the while loop and is composed of one or more statements.

<return_statement> is as its name suggests a return statement that returns a value.

<comment> is a line that gives a comment and is not supposed to be recognized by the compiler. It must be done with the use of double slash at the beginning of the line or slash star and star slash at the beginning and at the end of the statement.

<in_out_statement> is either an **<in_statement>** or an **<out_statement>**.
<input_statement> either gets a boolean, double or string with their corresponding input words.
<out_statement> prints out an expression with PRINT keyword, with a LEFT_PAR and either a VAR (variable name), **<var>** or an **<expression>** ending with a RIGHT_PAR and SEMICOLON
<conditional_statement> is composed of a type of writing. First IF keyword and LEFT_PAR (left parenthesis) needs to be used. Then comes an expression with a RIGHT_PAR (right parenthesis). Then, after a **<statement_block>**, either the statement ends or an ELSE part comes in with either another conditional statement or a **<statement_block>**.

<assignment_statement> is either a **<double_assignment>**, a **<string_assignment>** or a **<boolean_assignment>**. All of these assignments are done by a variable (DOUBLE or STRING or BOOLEAN), then an ASSIGNMENT_OP (assignment operator), and then either a value of the same type, an input, another variable or a return statement.

<expression> is another very important literal. It expresses a group of values and their relations with each other. It is either a sum or multiple of variables, a comparison of expressions or variables, or a combination of these.

<function_call_statement> represents calling the function.
<function_def_statement> is used for defining a function.
<parameters> is either an IDENTIFIER or more than one IDENTIFIERS.

<comp_expression> is a comparative expression that compares two arithmetic expressions.
<arith_expression> is an arithmetic expression which includes **<sum_sub_operation>** and/or **<mult_div_operation>.**

<arith_comp_expression> is an arithmetic comparative expression which compares expressions.
<factor_comp_expression> is a factor comparative expressions which compares factors.

<mult_expression> is a multiplication expression which multiplicates two values or identifiers
<mult_operation_expression> is a multiplication operation expression which is for factor and mult/div.
<sum_expression> is a summation expression which sums two values or identifiers.
<sum_operation_expression> is a summation operation expression which is for factor and sum/sub.

<operation> is either mult/div operation or sum/sub operation.
<sum_sub_operation> is for characters which are + or -.
<mult_div_operation> is for characters which are * or /.

<factor> is either an IDENTIFIER or a VALUE.
<boolean> is either TRUE or FALSE.
<type> is a literal name for defining IDENTIFIERS which are int, boolean, string, and double types.
<general_comparator> is all comparator types like equal, greater than, etc.
<value> is int, string, double, and boolean.
<primitive_function_call> is for our special function names which we used in our drone program.

CONFLICTS

The language gives only 4 conflicts. The detailed output file shows that these are shift reduce conflicts which are related to the expression rule that is defined in the yacc file. In state 171 and 173, the shift reduce conflict is caused by the operations SUM SUB DIV and MULT used to create a rule of the language that is expression. Although we tried to do minor changes with the language to minimize, without a major change in the language, we could not find a way to further decrease these 4 conflicts. The source of the issue is that yacc looks only one token ahead and this is not simply enough to differentiate the rules. So the language is not **ambiguous** but it is due to the structure of the language. The output file gives out the following text to point out the conflicts.

```
171: shift/reduce conflict (shift 155, reduce 63) on MULT
171: shift/reduce conflict (shift 156, reduce 63) on DIV
state 171
        mult_expression : mult_operation_expression factor .  (63)
        mult_operation_expression : factor . mult_div_operation  (65)

        MULT  shift 155
        DIV  shift 156
        RIGHT_PAR  reduce 63
        SUM  reduce 63
        SUB  reduce 63
        SEMICOLON  reduce 63
        EQUAL  reduce 63
        NOT_EQUAL  reduce 63
        GREATER_EQUAL  reduce 63
        LESS_EQUAL  reduce 63
        GREATER  reduce 63
        LESS  reduce 63
        OR  reduce 63
        AND  reduce 63

        mult_div_operation  goto 166
```

173: shift/reduce conflict (shift 157, reduce 66) on SUM
173: shift/reduce conflict (shift 158, reduce 66) on SUB
state 173

sum_expression : sum_operation_expression factor . (66)
sum_operation_expression : factor . sum_sub_operation (68)

SUM shift 157
SUB shift 158
RIGHT_PAR reduce 66
MULT reduce 66
DIV reduce 66
SEMICOLON reduce 66
EQUAL reduce 66
NOT_EQUAL reduce 66
GREATER_EQUAL reduce 66
LESS_EQUAL reduce 66
GREATER reduce 66
LESS reduce 66
OR reduce 66
AND reduce 66

sum_sub_operation goto 167

Evaluation of our Language

Readability: In order to make our programming language readable, we avoided unnecessary details and tried to make it as simple as possible so that the reader can understand every detail clearly. For example, we used string as an enhanced type rather than giving place chars, making the language easier to read by categorizing it into a common variable. Furthermore, the names of all the literals are chosen in a way that they are understandable by the reader. Also, we made sure that they are resembling the most commonly used programming languages' literals as well, making it easier to adapt to for newcomers.

Writability: We made sure that our language is writable as we do not have a char type because the input is taken by a string which is way more enhanced than char. This allows our language to be written easily without the concerns of typecasting as much as possible. In the expression part, we made sure that almost any combination of an arithmetic operation is possible when declaring an assignment. Furthermore, for and while loop structures are added to our language hence loops are enhanced and detailed, which provides more ability to the writer of the program. In addition, there are a variety of combinations in our language that can be used by the statement block. For instance; functions, loops, and groups of statements or all of their combinations can be used by statement blocks. Therefore, this enhances the flexibility and use of our programming language and it is easier for the writer to accomplish what he/she needs to.

Reliability: There are some recursive definitions of non-terminal literals in our language that prevent ambiguity and enhance the reliability of the language. Furthermore, the simplicity of our language makes it easy to use and therefore reliable at the same time. There are only four shift-reduce conflicts which are not related to the ambiguity of the language but to its structure.