CS 315 – PROGRAMMING LANGUAGES

PROJECT 1

“AGA” Language

Ahmet Hakan Yılmaz 21803399 Section 1

Akın Kutlu 21803504 Section 1

Arda Güven Çiftçi 21801763 Section 1

Complete BNF Description of AGA Language

1.Program

<program>::=<BP> <program_with_main> <EP>

<program_with_main> ::= <main_program>|<statement_list> <main_program>|
<statement_list> <main_program> <statement_list>
|<main_program> <statement_list>

<main_program>::= main <LP> <RP> <LB> <statement_list> <RB>

<statement_list> ::= <statement> | <statement> <statement_list>

<statement> ::= <if_else_statement>| <assignment_statement>
|<initialize_statement>|<loop_statement>| <output_statement>
|<function_statement>| <comment_statement>

2.Types

<variable_type > ::= int|float|void|string|char|boolean

<literal> ::= <int_literal> | <float_literal> | <boolean_literal> |
<char_literal> | <string_literal>

<int_literal> ::= 0 | <non_zero_digit> <digit> | <non_zero_digit>
<digit_list>

<signed_int_literal>::= <sign><int_literal>

<digit_list> ::= <digit> | <digit_list>

<digit> ::= 0 | <non_zero_digit>

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

<float_literal> ::= <int_literal>.<int_literal> | <int_literal>.0 <int_literal>

<signed_float_literal>::= <sign><float_literal>

<char_literal> ::= ’ **<digit> ‘ | ‘ <symbol>’ | ‘<letters>’**

<*boolean_literal*> ::= **true** | **false**

<sign> ::= +|-

<other_character>::=$|£|_|@

<all_character>::= <letter> | <digit> | <other_character>

<word>::= <all_character> | <all_character> <word>

<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

<sentence>::= <word> | <word> <space> <sentence>

<string_literal>::= “<sentence>”

<comment> ::= <long_comment> | <short_comment>

<short_comment> ::= SHORT_COMMENT

<long_comment> ::= LONG_COMMENT

<sentence_list>::= <sentence> | <sentence_list> <newline>

<variable_name> ::= <word>

<term>::= <literal>|<variable_name>

3. Conditional Statements

<if_else_statement> ::= if <LP> <logic_expression_list> <RP> <LB>

<statement_list> <RB> else <LB> <statement_list> <RB>
| if <LP> <logic_expression_list> <RP> <LB> <statement_list> <RB>
| if <LP> <logic_expression_list> <RP> <LB> <statement_list> <RB>
<else_if_statement_list> else <LB> <statement_list> <RB>
| if <LP> <logic_expression_list> <RP> <LB> <statement_list> <RB>
<else_if_statement_list>

<else_if_statement_list> ::= <else_if_statement> | <else_if_statement>
<else_if_statement_list>

<else_if_statement>::= <elif> <LP> <logic_expression_list> <RP> <LB>
<statement_list> <RB>

4. Initialize and Assignment Statements

<assignment_statement> ::= <variable_name> <assignment_op> <cast_expression> <semicolon>| <variable_name> <assignment_op> <input_expression><semicolon>| <variable_name> <assignment_op> <function_call> <semicolon>|<variable_name> <assignment_op> <function_output> <semicolon>|<variable_name> <assignment_op> <string_concat> <semicolon>

<initialize_statement> ::= <variable_type> <variable_name> <semicolon> | <variable_type> <variable_name> <assignment_op> <cast_expression> <semicolon>| <variable_type> <variable_name> <assignment_op> <input_expression><semicolon>| <variable_type> <variable_name> <assignment_op> <function_call> <semicolon>| <variable_type> <variable_name> <assignment_op> <function_output> <semicolon>
| <variable_type> <variable_name> <assignment_op><string_concat> <semicolon>

5. Cast Expression

<cast_expression> : <LP> <variable_type> <RP> <term>

6. Loops

<loop_statement>::= <while_statement>

<while_statement> ::= while<LP> <logic_expression> <RP> <LB> <statement_list> <RB>

7. Input/Output

<output_statement>::=output<LP><term><RP><semicolon>

<input_expression>::=input<LP><RP>

8. Logic and Arithmetic Expressions

<logic_expression_list> ::= <logic_expression>| <logic_expression> <boolean_operator> <logic_expression_list>

<logic_expression> ::= <relational_exp>| <boolean>| <variable_name>| <not_logic_exp>

<return_expression_list> ::= <return_expression> | <return_expression>

<boolean_operator> <return_expression_list>

<return_expression> ::= <relational_exp>|<not_logic_exp>

<string_concat>::=<term><sharp><term>|<string_concat><sharp><term>

<not_logic_exp> ::= <not> <LP> <logic_expression_list> <RP>

<relational_exp> ::= <arithmetic_exp> <relational_operator> <arithmetic_exp>|<function_call> <relational_operator> <arithmetic_exp> |

<arithmetic_exp> <relational_operator> <function_call> |<function_call> <relational_operator> <function_call>

<arithmetic_exp> ::= <arithmetic_exp> <arithmetic_operator_1>

<arithmetic_exp_2> | <arithmetic_exp_2>

<arithmetic_exp_2> : <arithmetic_exp_2> <arithmetic_operator_2> <term>| <term>
<arithmetic_operator_1> ::= +| -

<arithmetic_operator_2>::= / | *|%

<boolean_operator> ::= &&| ||

<relational_operator> ::= >| >= | == | < | <=

<assignment_operator> ::= <equal>

9. Function

<function_statement>::= <function_def> | <function_call> <semicolon>

<function_def> ::= <variable_type> **fun_dec** <variable_name> <LP> < variable_type_list> <RP><LB> <function_body><RB> | <variable_type> **fun_dec** <variable_name> <LP> <RP> <LB> <function_body> <RB>

<function_body> ::= <statement_list> return <function_output> <semicolon>

|< function_body> <statement_list> return <function_output> <semicolon>

< variable_type_list> ::= < variable_type > <variable_name> | < variable_type> <variable_name> <comma> <variable_type_list>

<function_output> ::= <arithmetic_exp> | <return_expression_list>

<function_call> ::= **fun_call** <variable_name> <LP> <term_list> <RP>
 | **fun_call** <variable_name> <LP> <RP> | <primitive_functions>

<primitive_functions>::= <function_readTemperatureF>| <function_readAltitudeF>| <function_readHeadF> | <function_connectToBaseF> | <function_changeSprayF>|

<function_stopUpOrDownF>| <function_moveForwardOrBackF>|
<function_moveUpOrDownF>| <function_stopForwardOrBackF>|
<function_turnF>|<function_arcCosF>

<term_list> ::= <term> | <term><comma> <term_list>

10. Primitive Functions

<function_readTemperatureF> ::= **fun_call** readTemperatureF <LP> <RP>

<function_readAltitudeF> : **fun_call** readAltitudeF <LP> <RP>

<function_readHeadF> ::= **fun_call** readHeadF <LP> <RP>

<function_changeSprayF> ::= **fun_call** changeSprayF <LP> <term> <RP>

<function_connectToBaseF> ::= **fun_call** connectToBaseF <LP> <term> <comma> <term> <RP>

<function_stopUpOrDownF> ::= **fun_call** stopUporDownF <LP> <term> <RP>

<function_moveForwardOrBackF> ::= **fun_call** moveForwardOrBackF <LP> <term> <RP>

<function_moveUpOrDownF> ::= **fun_call** moveUporDownF <LP> <term> <comma> <term> <RP>

<function_stopForwardOrBackF> ::= **fun_call** stopForwardorBackF <LP> <term><RP>

<function_turnF> ::= **fun_call** turnF <LP> <term> <comma> <term> <RP>

<function_arcCosF> ::= < **fun_call>** arcCosF <LP> <term> <RP>

11. Symbols

<BP> ::= begin_program

<EP> ::= end_program

<LP> ::= (

<RP> ::=)

<LB>::= {

<RB>::= }

<LSB>::= [

<RSB>::=]

<newline>::=/n

<semicolon>::= ;

<comma>::= ,

<space>::=

<and> ::=&&

<or>::=||

<equal>::= ==

<not_equal>::= !=

<not> ::= !

<sharp>::=

EXPLANATIONS

Program

<program>::=<BP> <program_with_main> <EP>

In our language we have begin_program and end_program syntax decides where will be program start reading and ending.

<program_with_main> ::= <main_program>|<statement_list> <main_program>| <statement_list> <main_program> <statement_list> |<main_program> <statement_list>

In our language there are possible options for program with main. Either is has only main program or it has main and program and other statement lists.

<statement_list> ::= <statement> | <statement> <statement_list>

<statement> ::= <if_else_statement>| <assignment_statement>

|<initialize_statement>|<loop_statement>| <output_statement>

|<function_statement>| <comment_statement>

In our program there can be multiple statements and we call it statement lists. In those statement lists which are consist of statements there are 7 option for creating statement.

<statement> ::= <if_else_statement>| <assignment_statement>

|<initialize_statement>|<loop_statement>| <output_statement>

|<function_statement>| <comment_statement>

There can be statement for creating conditioning which is if-else statements. For assigning and initializing there are assignment and initialize statements. For loops we have loop statements. For comments, functions and output there are also statements.

Types

< variable_type > ::= int|float|void|string|char|boolean

In this terminal we define variable types with their normal names.

<*literal*> ::= <*int_literal*> | <*float_literal*> | <*boolean_literal*> | <*char_literal*> | <*string_literal*>

These are the literals of the variable types that program use.

<int_literal> ::= 0 | <*non_zero_digit*> <*digit*> | <*non_zero_digit*> <*digit_list*>

This explains the possible integer literal options in the program.

<float_literal> ::= <int_literal>.<int_literal> | <int_literal>.0 <int_literal>

This explains the possible float literal options in the program.

<char_literal> ::= ’ <digit> ‘ | ‘ <symbol>’ | ‘<letters>’

This explains the possible character options in the program.

<boolean_literal> ::= true | false

Terminal state for boolean values.

<digit_list> ::= <digit> | <digit_list>

This non-terminal allows user to create integer with recursively.

<digit> ::= 0 | <non_zero_digit>

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

This non-terminal (first one) and terminal (second one) allows user to create digits.

<other_character>::$|£|_|@
This terminal allows user to define other characters

<sentence>::= <word> | <word> <space> <sentence>

This allows user to create strings with one and more words recursively.

<comment> ::= <long_comment> | <short_comment>

This state shows two options for commenting in program.

<short_comment> ::= SHORT_COMMENT

Short comment starts with // and it ends at the end of the line. The between part will not printed or used.

<long_comment> ::= LONG_COMMENT

Long comment starts with "/*" and it ends with the symbol "*/". The between part will not printed or used.

<sentence_list>::= <sentence> | <sentence_list> <newline>

The purpose of this non-terminal state is to allow user to have sentence or multiple sentences created with the newline.

<term>::= <literal>|<variable_name>

This non-terminal allows us to choose one of the literals when it would be used in assignment or simply get variable name.

<signed_int_literal>::= <sign><int_literal>

<signed_float_literal>::= <sign><float_literal>

These non-terminals allow us to use signed numbers

Conditional Statements

<if_else_statement> ::= if <LP> <logic_expression_list> <RP> <LB> <statement_list> <RB> else <LB> <statement_list> <RB> | if <LP> <logic_expression_list> <RP> <LB> <statement_list> <RB> | if <LP> <logic_expression_list> <RP> <LB> <statement_list> <RB> <else_if_statement_list> else <LB> <statement_list> <RB> | if <LP> <logic_expression_list> <RP> <LB> <statement_list> <RB> <else_if_statement_list>

In our language there are options for creating nested if alongside with else if and else. In this non-terminal segment we create options for these conditions.

<else_if_statement_list> ::= <else_if_statement> | <else_if_statement> <else_if_statement_list>

<else_if_statement> ::= <elif> <LP> <logic_expression_list> <RP> <LB> <statement_list> <RB>

In order to have more condition and options we created else if statements. This non-terminal demonstrates the syntax of else if statements.

Initializing and Assignment

<initialize_statement> ::= <variable_type> <variable_name> <semicolon> | <variable_type> <variable_name> <assignment_op> <assignment_op> <input_expression><semicolon>| <variable_type> <variable_name> <assignment_op> <function_call> <semicolon>| <variable_type> <variable_name> <assignment_op> <function_output> <semicolon>| <variable_type> <variable_name> <assignment_op><string_concat> <semicolon>

This non-terminal shows user syntax of initializing and creating variables. Whether create variable with empty value or with assignment operator assigning different options to them.

<assignment_statement> ::= <variable_name> <assignment_op> <cast_expression> <semicolon>| <variable_name> <assignment_op> <input_expression><semicolon>| <variable_name> <assignment_op> <function_call> <semicolon>|<variable_name> <assignment_op> <function_output> <semicolon>|<variable_name> <assignment_op> <string_concat> <semicolon>

This non-terminal shows user syntax of assigning variables. There are 5 options available for assigning values.The available options are through cast expressions, inputs, calling functions, function outputs and combining strings via string_concat.

Cast Expression

<cast_expression> : <LP> <variable_type> <RP> <term>

This non-terminal shows user casting one variable to another. There are only one options available for casting operation.

Logic, Arithmetic Expressions and Operators

<or>::=||
This terminal is syntax for “or operation” that is used in logical expressions.

<equal>::= ==
This terminal is the syntax for “equal operation” that is used in logical expressions.

<not_equal>::= !=
This terminal is the syntax for “not equal operation” that is used in logical expressions.

<multiplication_op>::= *
This is the syntax for multiplication operation.

<divide_op>::= /
This is the syntax for division operation.

<and> ::=&&
This terminal is syntax for “and operation” that is used in logical expressions.

<logic_expression_list> ::= <logic_expression>| <logic_expression> <boolean_operator> <logic_expression_list>
This non-terminal shows how logic expression list recursively.

<logic_expression> ::= <relational_exp>| <boolean>| <variable_name>| <not_logic_exp>
The options for logic expressions are shown in this non-terminal. There are 4 options for logical expressions.

<return_expression_list> ::= <return_expression> | <return_expression> <boolean_operator> <return_expression_list>
This non-terminal shows return expression list.

<return_expression> ::= <relational_exp>|<not_logic_exp>
This non-terminal shows there are 2 options available for the return expression.

<string_concat>::=<term><sharp><term>|<string_concat><sharp><term>
String_concat is been used for combining strings.

<not_logic_exp> ::= <not> <LP> <logic_expression_list> <RP>
This is the basic syntax for “not” in logical expressions.

<relational_exp> ::= <arithmetic_exp> <relational_operator> <arithmetic_exp>|<function_call> <relational_operator> <arithmetic_exp> |<arithmetic_exp> <relational_operator> <function_call> |<function_call> <relational_operator> <function_call>
In our language relational expressions are used in arithmetic expressions and also function calls.

<arithmetic_exp>::= < arithmetic_exp > <arithmetic_operator_1> < arithmetic_exp_2> | < arithmetic_exp_2>
This non-terminal shows that there are 2 options for arithmetic expressions whether with addition, subtraction with recursively.

< arithmetic_exp_2>::= < arithmetic_exp_2> <arithmetic_operator_2> <term> | <term>
This non-terminal shows second part of the arithmetic expression with term and multiplication division operations recursively.

<relational_exp> ::= <arithmetic_exp> <relational_operator> <arithmetic_exp>
This non-terminal shows syntax of relational expressions by using relational operator between arithmetic expressions

<boolean_exp>::=<arithmetic_exp><boolean_operator><arithmetic_exp>
This non-terminal shows the syntax of Boolean expressions. Boolean operators are used between arithmetic expressions.

<arithmetic_operator_1> ::= +| -
This terminal shows the arithmetic operator syntax for addition and subtraction.

<arithmetic_operator_2>::= / | *|%
This terminal shows the arithmetic operator syntax for multiplication and division.

<boolean_operator> ::= AND|OR
This terminal shows the Boolean operators.

<relational_operator> ::= >| >= | == | < | <=
The syntax for relational operators are shown in this terminal.

<assignment_operator> ::= <equal>
This non-terminal shows the syntax of assignment operator.

Loops

<while_statement> ::= while<LP> <logic_expression> <RP> <LB> <statement_list> <RB>

This non-terminal shows the syntax of while statement. It uses reserved word “while” with logical expression between parentheses and statement list between brackets.

Input/Output

<input_expression>::=input<LP><RP>
In this non-terminal program allows user to enter input. The syntax for entering input is, firstly use reserved word for input “input” after that enter string literal inside of left and right parenthesis.

<output_statement>::=output<LP><term><RP>
In this non-terminal program shows user to print output. The syntax for printing output is, firstly use reserved word for output “output” after that entering desired string literal inside of left and right parenthesis.

Function Definitions and Function calls

Definitions

<function_statement>::= <function_def> | <function_call> <semicolon>

In our language there are 2 function statements which are defining and calling an function.

<function_def> ::= <variable_type> fun_dec <variable_name> <LP> < variable_type_list> <RP><LB> <function_body><RB> | <variable_type> fun_dec <variable_name> <LP> <RP> <LB> <function_body> <RB>

This non-terminal shows the main definition of functions which are divided into two options. Whether user can enter variable type list inside the function input or not. To enter the function first decide variable type and use “fun_dec” reserved word. Inside the function there is function body.

function_body> ::= <statement_list> return <function_output> <semicolon> |< function_body> <statement_list> return <function_output> <semicolon>

Function body allows us to use return and get the function output while there are statements that will be used inside of function.

< variable_type_list> ::= < variable_type > <variable_name> | < variable_type> <variable_name> <comma> <variable_type_list>

This non- terminal allows user to have one or more variable type with recursively.

<function_output> ::= <arithmetic_exp> | <return_expression_list>
This non-terminal allows function output can either be arithmetic expression or inside of return expression list.

<function_call> ::= fun_call <variable_name> <LP> <term_list> <RP>
| fun_call <variable_name> <LP> <RP> | <primitive_functions>

This non-terminal shows the syntax for function calling. It syntax starts with reserved word “fun_call” followed by variable name and either with a term list or empty inside of parenthesis. In the third option it can also select functions from primitive function list.

<term_list> ::= <term> | <term>, <term_list>
This non-terminal allows user to have one or more term with recursively.

PRIMITIVE FUNCTIONS

<primitive_functions>::=<function_readTemperatureF>|
<function_readAltitudeF>| <function_readHeadF> |
<function_connectToBaseF>|<function_changeSprayF>|<function_stopUpOrDownF>|<function_moveForwardOrBackF>|<function_moveUpOrDownF>| <function_stopForwardOrBackF>| <function_turnF>|
<function_arcCosF>

This is the program definition for primitive functions which are used for turn, moving up or down and moving back or forward.

<**function_turnF> ::= fun_call turnF <LP> <term> <comma> <term> <RP>**
This is the syntax for turning function which uses reserved word “fun_call” with two terms divided by comma for function input between parentheses.

<function_moveUpOrDownF> ::= fun_call moveUporDownF <LP> <term> <comma> <term> <RP>
This is the syntax for moving up or down function which uses reserved word “fun_call” with two terms divided by comma for function input between parentheses.

<function_moveForwardOrBackF> ::= fun_call moveForwardOrBackF <LP> <term> <RP>
This is the syntax for moving forward or back function which uses reserved word “fun_call” with term divided by comma for function input between parentheses.

<function_changeSprayF> ::= fun_call changeSprayF <LP> <term> <RP>
This is the syntax for changing spray function which uses reserved word “fun_call” with term for function input between parentheses.

<function_connectToBaseF> ::= fun_call connectToBaseF <LP> <term> <comma> <term> <RP>
This is the syntax for connecting to base function which uses reserved word “fun_call” with two terms divided by comma for function input between parentheses.

<function_readHeadF> ::= fun_call readHeadF <LP> <RP>
This is the function syntax for reading the heading.

<function_readAltitudeF> : fun_call readAltitudeF <LP> <RP>
This is the function syntax for reading the altitude.

<function_readTemperatureF> ::= fun_call readTemperatureF <LP> <RP>
This is the function syntax for reading the temperature.

<function_stopUpOrDownF> ::= fun_call stopUporDownF <LP> <term> <RP>

This function is used for stopping whether going up or down.

<function_stopForwardOrBackF> ::= fun_call stopForwardorBackF <LP> <term><RP>
The purpose of this function is to stop whether going forward or backward.
.
<function_arcCosF> ::= < fun_call> arcCosF <LP> <term> <RP>
This function is been used to calculate arccos of angle.

Where our tokens come from?

Most of our non-trivial tokens come from Java such as comments “//”, “/**/”. The main reason behind this is that we are more familiar with java more than any other programming language. But our tokens are not only from Java we are also inspired by C and Kotlin while creating our own input/output syntax. The other reason behind the usage of these is we also think that these options were better in terms of writability and readability so we took and put them into one language. Another inspiring language was Python. We used ELIF keyword like in python.

What we changed in the second part?

In the first design we wrote comments wrong. We changed them. Comments are terminals now and lex does not print in the comments, it only returns SHORT_COMMENT or LONG_COMMENT. We also fix MAIN keyword and start program of language. BEGIN_PROGRAM and END_PROGRAM was added to determine where program starts and ends. Aga language was not supporting else statements. We added ELIF keyword and else_if_statement to support it. We solved so many conflicts and now our language have no conflict. Finally, we made small changes on primitive functions.