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> ::= <statement_list>

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

<statement> ::= <comment> | <matched_statement> | <unmatched_statement>

<matched_statement> ::= if (<logic_expression>) <matched_statement>

else <matched_statement> |<other_statement>

<unmatched_statement> ::= **if** (<logic_expression>) <statement> |**if** (<logic_expression>) <matched_statement> **else** <unmatched_statement>

<other_statement> ::= <assignment_statement> | <initialize_statement> |<loop_statement> | <output_statement> |<function_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> ::= //< sentence >

<long_comment> ::= /* < sentence_list > */

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

<variable_name> ::= <word>

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

3.Initialize and Assignment statements

<initialize_statement> ::= <variable_type> <variable_name><semicolon> | <var_type> <variable_name> <assignment_op> <term><semicolon> |<var_type> <variable_name> <assignment_op> <function><semicolon> | <var_type> <variable_name> <assignment_op> <cast_expression><semicolon> | <variable_type> <variable_name> <assignment_op> <input_expression><semicolon>

<assignment_statement> ::= <variable_name> <assignment_op> <term> | <variable_name> <assignment_op> <function> | <variable_name> <assignment_op> <cast_expression>| variable_name> <assignment_op> <input_expression>

<expression>::= <logic_expression>| <cast_expression>|<arithmetic_exp>

<logic_expression> ::= <arithmetic_exp>|<relational_exp> | <boolean_exp>|<not_expression>

<not_expression>::= <not><LP><logic_expression><RP>| <not><LP><boolean_literal><RP> | <LP><variable_name<RP>

<cast_expression> ::= <LP> <var_type> <RP> <term>

<arithmetic_exp>::= < arithmetic_exp > <arithmetic_operator_1> < arithmetic_exp_2> | < arithmetic_exp_2>

< arithmetic_exp_2>::= < arithmetic_exp_2> <arithmetic_operator_2> <term> | <term>

<relational_exp> ::= <arithmetic_exp> <relational_operator> <arithmetic_exp>

<boolean_exp>::= <arithmetic_exp><boolean_operator><arithmetic_exp>|

<aritmetic_exp><boolean_operator><relational_exp>|<relational_exp><boolean_operator><relational_exp>|<arithmetic_exp><boolean_operator><arithmetic_exp>

<arithmetic_operator_1> ::= +| -

<arithmetic_operator_2>::= / | *|%

<boolean_operator> ::= &&| ||

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

<assignment_operator> ::= <equal>

3.Loops

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

4.Input/Output

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

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

5.Function Definitions and Function calls

<function_statement>::= <function_def> | <fun_call>

<function_def> ::=
<variable_type> **fun_dec** <variable_name> <LP> < variable_type_list> <RP>
<LB> <statement_list>
return <function_output> **;**
<RB> | <variable_type> **fun_dec** <variable_name> <LP> <RP>
{ <statement_list>
return <function_output> **;** }

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

<function_output> ::= <term>|<expression>

<function> ::= **fun_call** <variable_name> (<term_list>) | **fun_call**
<variable_name> ( )

<term_list> ::= <term> | <term>, <term_list>

PRIMITIVE FUNCTIONS

<primitive_functions> ::= <function_turn> | <function_ moveUporDown> | <function_moveBackorForward>|

<function_turn> ::= **fun_call** turnF(<term>, <term>);

<function_ moveUporDown> ::= **fun_call** moveUporDownF (<term>, <term>);

<function_moveForwardOrBack> ::= **fun_call** moveForwardOrBackF(<term>);

<function_changeSpray>::= **fun_call** changeSprayF(<term>, <term>);

<function_connectToBase>::= **fun_call** connectToBaseF(<term>, <term>);

<function_readHead>::= **fun_call** readHeadF();

<function_readAltitude>::= **fun_call** readAltitudeF();

<function_readTemperature>::= **fun_call** readTemperatureF();

SYMBOLS

<LP> ::= (

<RP> ::=)

<LB>::= {

<RB>::= }

<LSB>::= [

<RSB>::=]

<newline>::=/n

<semicolon>::= ;

<comma>::= ,

<space>::=

<and> ::=&&

<or>::=||

<equal>::= ==

<not_equal>::= !=

<not> ::= !

EXPLANATIONS

Program

<program> ::= <statement_list>

This is an non-terminal state. The purpose of this state is that program use statements that can be used inside of other statements.

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

This non-terminal state helps user to use multiple statements recursively.

<statement> ::= <comment> | <matched_statement> | <unmatched_statement>

This non-terminal state allows us to divide statements which are matched and unmatched. This state also allows to use comment line in program.

<matched_statement> ::= if (<logic_expression>) <matched_statement>

else <matched_statement> |<other_statement>

The purpose of this non-terminal state is to show that <matched_statement> can have 2 options and syntax of the matched if statement. This state also allows to use if statement inside of other if statement recursively.

<unmatched_statement> ::= if (<logic_expression>) <statement>|if (<logic_expression>) <matched_statement> else <unmatched_statement>

This state shows us syntax of unmatched “if statement“. This statement shows that there are 2 options whether unmatched if statement takes any statement or takes matched statement followed by an unmatched statement in else part. This is non-terminal state.

<other_statement> ::= <assignment_statement> | <initialize_statement> |<loop_statement> | <output_statement>

This non-terminal state allows us to use other statements than if statement which are loops, input, output, and initialize 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> ::= //< sentence >

This non-terminal shows the syntax for short commenting by "//".

<long_comment> ::= /* < sentence_list > */

This non-terminal shows user to syntax for long commenting by identifying beginning of the long comment symbol "/*" with the end of the long comment symbol "*/".

<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

Initializing Assignment

<initialize_statement> ::= <variable_type> <variable_name><semicolon> | <var_type> <variable_name> <assignment_op> <term><semicolon> |<var_type> <variable_name> <assignment_op> <function><semicolon> | <var_type> <variable_name> <assignment_op> <cast_expression><semicolon> | <variable_type> <variable_name> <assignment_op> <input_expression><semicolon>

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

<assignment_statement> ::= <variable_name> <assignment_op> <term> | <variable_name> <assignment_op> <function> | <variable_name> <assignment_op> <cast_expression>| variable_name> <assignment_op> <input_expression>

This non-terminal shows user syntax of assigning variables. There is 2 options available for assigning values. First with the assignment operator terms are assigned to the variables. In second option variables are getting assigned from functions with assignment operator.

Operators and Expressions

<and> ::=&&

This terminal is syntax for “and operation” that is used in logical expressions.

<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.

<expression>::= <cast_expression> | <arithmetic_exp> | <logic_expression>

This non-terminal allows user to have 2 options either arithmetic or logical expression.

<logic_expression> ::= <relational_exp> | <boolean_exp>|<not_expression>

This non-terminal shows logical expressions can have 2 options either relational or Boolean expression.

<not_expression>::= <not><LP><logic_expression><RP>| <not><LP><boolean_literal><RP> | <LP><variable_name<RP>

This non-terminal allow users to use not expression

<cast_expression> ::= <LP> <var_type> <RP> <term>

This non-terminal allow users to have cast expression

<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><term><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_def> ->
<variable_type> fun_dec <variable_name> <LP> < variable_type_list> <RP>
<LB> <statement_list>
return <function_output> ;
<RB> | <variable_type> fun_dec <variable_name> <LP> <RP>
{ <statement_list>
return <function_output> ; }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 use “fun” reserved word and to get the function output program uses reserved word “return”.
< 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> ::= <term>|<expression>
This non-terminal allows function output can either be term or expression.
<function> ::= fun_call <variable_name> (<term_list>) | fun_call <variable_name> ( )|<primitive_function>This non-terminal shows the syntax for function. It syntax starts with reserved word “fun” followed by variable name and either with a term list or empty inside of parenthesis.
<term_list> ::= <term> | <term>, <term_list>
This non-terminal allows user to have one or more term with recursively.

PRIMITIVE FUNCTIONS

<primitive_functions> ::= <function_turn> | <function_ moveUporDown>|<function_moveBackorForward>|<function_change Spray>|<function_connectToBase>|<function_readHead>|<function_r eadAltitude>|<function_readTemperature>

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

<function_turn> ::= funcall turnF(<term>, <term>);

This is the syntax for turning function which uses reserved word “fun” with two terms divided by comma for function input between parentheses.

<function_ moveUporDown> ::= funcall moveUporDownF (<term>, <term>);

This is the syntax for moving up or down function which uses reserved word “fun” with two terms divided by comma for function input between parentheses.

<function_moveBackorForward> ::= funcall moveBackorForwardF (<term>, <term>);

This is the syntax for moving back or forward function which uses reserved word “fun” with two terms divided by comma for function input between parentheses.

<function_changeSpray>::= funcall changeSprayF(<term>, <term>);

This is the syntax for changing spray function which uses reserved word “fun” with two terms divided by comma for function input between parentheses.

<function_connectToBase>::= funcall connectToBaseF(<term>, <term>);

This is the syntax for connecting to base function which uses reserved word “fun” with two terms divided by comma for function input between parentheses.

<function_readHead>::= funcall readHeadF();

This is the function syntax for reading the heading.

<function_readAltitude>::= funcall readAltitudeF();

This is the function syntax for reading the altitude.

<function_readTemperature>::= funcall readTemperatureF();

This is the function syntax for reading the temperature.

Readibility

In terms of readability, our language can be considered readable. Since we don't have detailed or complicated objectives our language mostly focused on basic functions and operations therefore isn't a much-complicated structure. Also, our language is similar to other popular programming languages; it doesn't have much difference in terms of basic operation syntax. In other words, our language can easily be familiar to people who used any of the popular programming languages. Since most of our reserved words are easily understandable and meaningful it is easy to memorize their functionality.

Reliability

The reliability of our language can depend mostly on different situations. For example, we take one type of input and cast it if the user wants, and since we don't have an exception handler or warning it can give errors to the user without knowing what caused them. Since the aim of our language isn't to complete the complicating tasks and projects it can be reliable when it is used in basic projects. Except for highly complicated projects our language can be easily read, written, and rely on.

Writability

In terms of writability, our language pretty much consists of basic operations and functions. As we mentioned in readability our language is similar and more basic when it is compared to popular languages such as Java and Kotlin. Inside our language structure, there is a small number of primitive data types and a small number of operations which makes users write more simple and easy programs. Since there is only one loop and one way to do arithmetic calculations and many other things there aren't any confusing different techniques to write a program in our language. Our

language can be useful in doing simple projects but it needs to be improved in terms of more complicated projects. So, the writability of our language can be considered simple and easy.

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.