CS315 Programming Languages

Project 2 Report

Team 12

Language Name: Glide

Team Members

Kamil Kaan Erkan 21___ Section 2

Zübeyir Bodur 21702382 Section 1

Ege Türker 21702993 Section 1

Instructor

Halil Altay Güvenir

Fall 2020

1. BNF Description of the Language

The following is the complete BNF description of the language.

<assign_op> ::= =

<letter> ::= <lowercase_let>

| <uppercase_let>

<lowercase_let> ::= a | b | ... | z

<uppercase_let> ::= A | B | ... | Z

<digit> ::= 0 | 1 | … | 9

<sign> ::= "+" | "-"

<point> ::= "."

<rp> ::= ")"

<lb> ::= "{"

<rb> ::= "}"

<end_stm> ::= ";"

<comment_sign> := “#”

<comment> ::= <comment_sign> (<chars>)<comment_sign>

<comma> ::= ","

<quote> ::= "\""

<literal> ::= <int_literal>

| <double_literal>

| <bool_literal>

| <string_literal>

<string_literal> ::= <quote>(<chars>)<quote>

<int_literal> ::= <sign>?<numeral>

<double_literal> ::= <sign>?<numeral><point>?<digit>*

<numeral> ::= <digit> | <numeral><digit>

<bool_literal> ::= true | false

<type> ::= <type_int> | <type_double> | <type_bool> | <type_string>

<type_int> ::= int

<type_double> ::= double

<type_bool> ::= bool

<type_string> ::= string

<char> ::= <digit> | <letter>

<chars> ::= <char> | <chars><char>

<identifier> ::= <letter> <ident_chars>

<identifiers> ::= <identifier> | <identifiers> <comma> <identifier>

<ident_chars> ::= <letter> <ident_chars>

| <digit> <ident_chars>

| "-"<ident_chars>

| "_"<ident_chars>

//removed the <end_stm>

<declaration_statement> ::= <type> <identifiers>

<statement_list>::= <expression_stm>

| <compound_stm>

| <if_stm> | <iteration_stm> | <comment_line> | <print_statement>

|<input_statement> | <function_call> | <function_declaration> | <predefined> |

<declaration_statement>

<statement> ::= <statement_list> <end_stm>

<statements> ::= <statement>

| <statement><statements>

Expressions

<expression_stm> ::= <expression>

<expression> ::= <assignment_expr>

//included <declaration_statement>

<assignment_expr> ::= <declaration_statement><assign_op><logical_expr> |

<identifier><assign_op><logical_expr>

<logical_expr> ::= <logical_expr> || <and_expr>

| <and_expr>

<and_expr> ::= <and_expr> && <equality_expr>

| <equality_expr>

<equality_expr> ::= <equality_expr> == <relational_exp>

| <equality_expr> != <relational_exp>

| <relational_exp>

<relational_exp> ::= <relational_exp> <<additive_expr>

| <relational_exp> > <additive_expr>

| <relational_exp> <= <additive_expr>

| <relational_exp> >= <additive_expr>

| <additive_expr>

<additive_expr> ::= <additive_expr> + <multiplicative_expr>

| <additive_expr> - <multiplicative_expr>

| <multiplicative_expr>

<multiplicative_expr> ::= <multiplicative_expr> * <prefix_expr>

| <multiplicative_expr> / <prefix_expr>

| <multiplicative_expr> % <prefix_expr>

| <prefix_expr>

// Added logical NOT

<prefix_expr> ::= <postfix_expr>

| ++ <prefix_expr>

| -- <prefix_expr>

| ! <prefix_expr>

<postfix_expr> ::= <primary_expr>

| <postfix_expr> ++

| <postfix_expr> --

//the value returned by the function can be assigned to a variable

<primary_expr> ::= <lp> <logical_expr> <rp>

| <identifier>

| <literal>

| <function_call>

| <predefined>

<compound_stm> ::= <lb><statements><rb>

Program Structure

<program> ::= <main>

<main> ::= "main"<lp><rp><lb><statements><rb>

Loops

//made some changes inside the parentheses

<iteration_stm> ::= <for_stm> | <while_stm>

<for_stm> ::=

for<lp><assignment_expr><end_stm><logical_expr><end_stm><assignment_expr><en

d_stm><rp><lb><statements><rb>

//deleted the <end_stm>

<while_stm> ::= while <lp><logical_expr><rp> <lb><statements><rb>

Conditional Statements

//made some changes inside the parentheses

<if_stm> ::= <if_alone> | <if_else>

<if_else> ::= <if_alone> else <lb><statements><rb>

//deleted the <end_stm>

<if_alone> ::= if <lp><logical_expr><rp> <lb><statements><rb>

Comments

<comment_statement> ::=<comment> <sentence> <comment>

<sentence> ::=<identifier> | <identifier><sentence>

Statements for Input/Output

<print_statement> ::="print"<lp><outputs><rp>

<input_statement> ::="input"<lp><user_prompt><rp>

<output> := <literal>|<identifier>|<function_call>|<predefined>

<outputs> := <output> | <outputs><output>

<user_prompt> := <string_literal>|<identifier>

Function Definitions and Function Calls

<function_call> ::=<identifier> <lp><argument_list><rp>
| <identifier> <lp><rp>
<argument_list> ::= <literal> | <identifier> | <literal>, <argument_list>
| <identifier>,<argument_list>
<function_declaration> ::=<type> <identifier> <lp> <parameter_list> <rp> <lb>
<function_body> <rb>
| <type> <identifier> <lp> <rp> <lb><function_body> <rb>
<parameter_list> ::=<parameter>|<parameter_list><comma> <parameter>
<parameter> ::= <type><identifier>
<function_body> ::=<statement> | <function_body> <statement>
<predefined> ::=<inclination>| <altitude>|<temperature>|<acceleration>|<camera>
|<photo>|<timestamp>|<connect>|<ascend>|<descend>
<ascend> ::= “ascend”<lp><double><rp>
<descend> ::= “descend”<lp><double><rp>
<inclination> ::= “getInclination” <lp><rp>
<altitude> ::=“getAltitude” <lp><rp>
<temperature> ::=“getTemp” <lp><rp>
<acceleration> ::=“getAcc” <lp><rp>
<camera> ::=“toggleCam”<lp><bool><rp>
<photo> ::=“takePhoto” <lp><rp>
<timestamp> ::=“getTime” <lp><rp>
<connect> ::=“connect”<lp><rp>

2. Explanation of the Language

Program Structure

<program> ::= <main>

<main> ::=<lp><rp><lb><statements><rb>

<statements> ::= <statement> | <statements> <statement>

Program is written inside the curly braces after stating main(), by using statements.

Expressions

<statement_list>::= <expression_stm>

| <compound_stm>

| <if_stm> | <iteration_stm> | <comment_line> | <print_statement>

|<input_statement> | <function_call> | <function_declaration> | <predefined> | <declaration_statement>

<statement> ::= <statement_list> <end_stm>

Statements include expressions, if statements, iteration statements, function call, function declaration, print statement, input statement, comment line and variable declarations. All statements end with <end_stm>:= “;” .

<assignment_expr> ::= <identifier><assign_op><logical_expr>

Assignment statements include assigning a value to a variable.

<logical_expr> ::= <logical_expr> || <and_expr>

| <and_expr>

<and_expr> ::= <and_expr> && <equality_expr>

| <equality_expr>

<equality_expr> ::= <equality_expr> == <relational_exp>

| <equality_expr> != <relational_exp>

| <relational_exp>

<relational_exp> ::= <relational_exp> < <additive_expr>

| <relational_exp> > <additive_expr>

| <relational_exp> <= <additive_expr>

| <relational_exp> >= <additive_expr>

| <additive_expr>

Logical and relational expressions include comparison of operands.

<additive_expr> ::= <additive_expr> + <multiplicative_expr>

| <additive_expr> - <multiplicative_expr>

| <multiplicative_expr>

<multiplicative_expr> ::= <multiplicative_expr> * <prefix_expr>

| <multiplicative_expr> / <prefix_expr>

| <multiplicative_expr> % <prefix_expr>

| <prefix_expr>

Arithmetic expressions include arithmetic operations.

<prefix_expr> ::= <postfix_expr>

| ++ <prefix_expr>

| -- <prefix_expr>

<postfix_expr> ::= <primary_expr>

| <postfix_expr> ++

| <postfix_expr> --

Prefix and postfix expressions include prefix and postfix operators. Prefix operator increments/decrements the variable by one and then returns this value, whereas, postfix operator returns the original value and then increments/decrements the original value by one.

Loops

<iteration_stm> ::= <for_stm> | <while_stm>

Iteration statements are composed of for statements and while statements.

<for_stm> ::= for<lp><expression_stm><end_stm><expression_stm><end_stm><expression_stm><end_stm><rp><lb><statements><rb>

“for” statement is an iteration statement. It takes an int identifier and its initial value (loop control variable), a relational/logical expression and an assignment expression to update the loop control variable. If the relational/logical expression is true, statements inside the braces are executed until relational/logical expression becomes false. If the relational/logical expression is false, statements inside the braces are not executed and flow of control jumps to the next statement after “for”.

<while_stm> ::=while <lp> <expression_stm><end_stm> <rp> <lb><statements><rb>

“while” statement is also an iteration statement. It takes a relational/logical expression. If the relational/logical expression is true, statements inside the braces are executed until relational/logical expression becomes false. If the relational/logical expression is false, statements inside the braces are not executed and flow of control jumps to the next statement after “while”.

Conditional Statements

<if_stm> ::= <if_alone> | <if_else>

Conditional statements are composed of if and if-else statements.

<if_else> ::= <if_alone> else <lb><statements><rb>

<if_alone> ::=if <lp> <expression_stm><end_stm><rp> <lb><statements><rb>

“if” statement takes a relational/logical expression. “else” statement is optional and it does not take any expressions. If the relational/logical expression inside the “if” statement is true,

statements inside the braces belonging to “if” are executed, if it is not true, then the first statement after the end of “if” statement is executed.

Statements for Input/Output

<user_prompt> := <string_literal>|<identifier>

<input_statement> ::="print"<lp>(<user_prompt>)<rp>

Input statements can be used to take input from the user. Programmers can also put an identifier or a string between the parentheses to prompt the user. The identifier or the string between the parentheses will be printed before asking the user for input.

Input statement returns the user’s input after taking it, so the programmer should assign it to a variable if the input needs storage.

Example: x = input(“Enter your input”)

The string “Enter your input” will be printed and the input taken from the user will be stored in x

<output> :=

<int_literal>|<double_literal>|<string_literal>|<function_call>|<predefined>|<identifier>

<print_statement> ::="print"<lp>(<output>)<rp>

Print statement is the output statement for the program. Programmers can put an identifier or a string between the parentheses to print that identifier or string to the console. Programmers can also call functions, use predefined functions or output numbers too.

Comments

<comment> ::=“#”

<comment_statement> ::=<comment> <sentence> <comment>

Comment statements can be used for explaining the code or preventing the execution of a block. Comment sign is the “#” character. Putting the comment sign before and after the sentence will prevent the execution of that sentence.

<sentence> ::=<identifier> | <identifier><sentence>

Sentences are defined as at least one identifier or multiple identifiers.

Function Definitions and Function Calls

<parameter_list> ::=**<parameter>|<parameter_list> <comma> <parameter>**

<parameter> ::=<literal><identifier>

Parameter is defined as a literal followed by an identifier. The literal defines the type of the identifier. Parameter list is defined as a single parameter or multiple parameters separated by the comma sign ",".

<function_body> ::=<statement> | <function_body> <statement>

Function body is defined as a single statement or multiple statements.

<function_declaration> ::=<literal> <identifier> <lp> <parameter_list> <rp> <lb>
<function_body> <rb>

Programmers can declare a function by specifying its return type(literal), the parameters the function takes inside parentheses, and writing the function body inside curly braces.

<argument_list> ::=<identifier> | <identifier>,<argument_list>

Argument list consists of one or more identifiers.

<function_call> ::=<identifier> <lp><argument_list><rp>

Programmers can call the functions they declared by using the function name(identifier) and the argument list inside parentheses.

Primitive Functions

<predefined::=

<ascend>|<descend>|<inclination>|<altitude>|<temperature>|<acceleration>|<camera>
|<photo>|<timestamp>|<connect>

Predefined consists of all the primitive functions defined in the language.

<ascend>::= "ascend"<lp><double><rp>

Programmers can use the ascend keyword followed by a double variable to make the drone ascend. This double variable indicates the distance to ascend.

<descend>::= “descend”<lp><double><rp>

Programmers can use the descend keyword followed by a double variable to make the drone descend. This double variable indicates the distance to descend.

<inclination> ::=“getInclination” <lp><rp>

Programmers can use the getInclination keyword followed by open and close parenthesis to read the inclination from the drone.

<altitude> ::=“getAltitude” <lp><rp>

Programmers can use the getAltitude keyword followed by open and close parenthesis to read the altitude from the drone.

<temperature> ::=“getTemp” <lp><rp>

Programmers can use the getTemp keyword followed by open and close parenthesis to read the temperature from the drone.

<acceleration> ::=“getAcc” <lp><rp>

Programmers can use the getAcc keyword followed by open and close parenthesis to read the acceleration from the drone.

<camera> ::=“toggleCam”<lp><bool><rp>

Programmers can use the toggleCam keyword followed by a bool variable inside parentheses to toggle the camera on or off. When bool variable true is used inside parentheses, the camera will be toggled on. When bool variable false is used inside parentheses, the camera will be toggled off.

<photo> ::=“takePhoto” <lp><rp>

Programmers can use the takePhoto keyword followed by open and close parenthesis to prompt the drone to take a photo.

<timestamp> ::=“getTime” <lp><rp>

Programmers can use the getTime keyword followed by open and close parenthesis to read the timestamp from the drone.

<connect> ::=“connect”<lp><rp>

Programmers can use the connect keyword followed by open and close parenthesis to connect to the base computer through wifi.

3. Evaluation of the Language

a) Readability

The language Glide is influenced by other languages such as C and Java. Therefore, those who know these languages already can read and understand the code. Glide uses similar special words to other languages (for, if, else) and similar data types (int, double, boolean, string) to increase the readability and to make transition from other languages smoother. However, similar to C, Glide also has feature multiplicity. Programmers can implement the same code in different ways (count++, count = count + 1 or count += 1) which reduces the readability of the code.

b) Writability

The language Glide has simple, short and self explanatory keywords for its functions, types and statements. Since the language is designed to be used with a drone, there aren't any unnecessary details present in it. The programmers can just think about what they want to prompt the drone to do, and translate it into the code by using the simple keywords of Glide. They can also get environmental data from drones by writing the one word primitive functions existing in Glide. The simple comment system also helps the writability a great deal, since programmers can comment a whole block during code writing for testing purposes, by adding comment sign (#) before and after the block. Programmers can also use this feature to write comments in their programs. Finally, since Glide is influenced by widely used languages like C, Java and Python, many programmers can easily adapt to writing code in Glide.

c) Reliability

Even though our language doesn't have built-in exception handling or type checking for expressions just like in Java, its reliability comes from its simplicity. Since our language is easy to read and write, it will be easy to modify the programs written in the long run. For instance, in Glide, there are no aliasing methods, such as pointers or reference types in C++, which allows dangerous operations in the memory. In our language, type checking could reduce the reliability. Even though type tokens are defined in this language, currently the outcome type of an expression is not computed in the description of the language. Since the <primary_expr> can be defined as an <identifier> or <literal>, it could be even a string variable or a string literal in an arithmetic or a logical expression. If this part was handled in the compile time, the programmer

would see the error without having a run-time error where it is ambiguous what type of error is encountered. Overall, Glide is unreliable in terms of errors and exceptions but it is reliable as it has a good overall readability & writability.