CS 315 - Project1

The Bee Language

A Programming Language for Drones and its Lexical Analyzer

Esad İsmail Tök - 21801679 - Section 3

Nima Ghaffarzadeh - 21801432 - Section 1

Note: The red writings show the parts that we added recently, after Project1.

Program

<program> -> <main> | <func_def> | <program> <func_def>
A Program can have only one main function, or only one or more function definitions, or it can have a combination of both of them.

<main> -> main <LP> <RP> <LB> <statement_list> <RB>
A main function is the function that is run first when the program is run.

<statement_list> -> <empty> | <statement> | <statement>; <statement_list>
A Statement list can be empty, or it can be one statement, or it can be multiple statements.

<statement> -> <var_declaration> | <const_declaration> | <assignment> | <const_assignment> | <condition> | <loop> | <funct_call> |<comment> | <input> | <output>
A statement can be a variable declaration, an assignment operation, a conditional statement, a loop, a function call, a commented phrase, input or output.

<non_if_statement> -> <var_declaration> | <const_declaration> |<assignment> | <const_assignment> | <loop> | <funct_call> |<comment> | <input> | <output>
A non_if_statement can be any statement, except conditional statements.

Declaration

<var_declaration> -> <type> <var_identifier> | <type> <assignment>
A var_declaration is either:

- Just declaring a variable
- Declaring a variable and assigning a value to it

<const_declaration> -> const <type> <const_identifier> | const <type> <const_assignment>
A const_declaration is either:

- Just declaring a constant
- Declaring a constant and assigning a value to it

<type> -> int | db | str | bool
A type can be integer, double, string, or boolean.

<var_identifier> -> <lowercase_letter> | <var_identifier> <under_score>
<lowercase_letter> | <var_identifier> <lowercase_letter> | <var_identifier> <digit>
A var_identifier can be in one of these forms:

- A lowercase letter
- Lowercase letters with under_scores and digits in between them

<const_identifier> -> <uppercase_letter> | <const_identifier> <under_score>
<uppercase_letter> | <const_identifier> <uppercase_letter> | <const_identifier> <digit>
A const_identifier can be in one of these forms:

- An uppercase letter
- Uppercase letters with under_scores and digits in between them

<assignment> -> <var_identifier> <assign_op> <var_identifier> | <var_identifier>
<assign_op> <expression> | <var_identifier> <assign_op> <input>
An assignment operation can be in one of the following forms:

- A variable being assigned to another variable
- An expression being assigned to a variable
- An input value being assigned to a variable

<const_assignment> -> <const_identifier> <assign_op> <const_identifier> |
<const_identifier> <assign_op> <expression> | <const_identifier> <assign_op> <input>

- A constant being assigned to another constant
- An expression being assigned to a constant
- An input value being assigned to a constant

Expression

<expression> -> <relational> | <arithmetic>
An expression is either relational or arithmetic.

<arithmetic> -> <arithmetic> <add_sub_op> <var_identifier>
| <arithmetic> <add_sub_op> <number>
| <arithmetic> <add_sub_op> <term>
| <arithmetic> <add_sub_op> <parentheses>

| <term>

An arithmetic expression is either:

- An addition/subtraction operation
- A term(A term is a multiplication or division operation)
- An arithmetic operation added to/subtracted from a parentheses(<parentheses> is considered as a separate token, in order to give priority to parentheses expressions.)

<term> -> <term> <mult_div_op> <var_identifier>
| <term> <mult_div_op> <number>
| <term> <rem> <var_identifier>
| <term> <rem> <number>
| <term> <mult_div_op> <parentheses>
| <term> <rem> <parentheses>

A term is either:

- A term being multiplied/divided/modulo by another term
- A term being multiplied/divided/modulo by a parentheses expression(<parentheses> is considered as a separate token, in order to give priority to parentheses expressions)

<number> -> <digit> | <nonzero_digit> <number>

A number is a series of digits that cannot start with 0.

<add_sub_op> -> <add_op> | <sub_op>

Add_sub_op is either an addition or a subtraction operator.

<mult_div_op> -> <mult_op> | <div_op>

Mult_div_op is either a multiplication or a division operator.

<parentheses> -> <LP> <arithmetic> <RP>
| <LP> <number> <RP>
| <LP> <var_identifier> <RP>

A parentheses expression can be in one of the following forms:

- An arithmetic expression inside of parentheses
- A number inside of parentheses
- A variable expression inside of parentheses

<relational> -> <arithmetic> <less_than> <arithmetic> | <arithmetic> <greater_than> <arithmetic> | <arithmetic> <less_or_equal> <arithmetic> | <arithmetic> <greater_or_equal> <arithmetic> | <arithmetic> <equal> <arithmetic> | <arithmetic>

<not_equal> <arithmetic> | <arithmetic> <or> <arithmetic> | <arithmetic> <and> <arithmetic>

A relational expression can be one of the following expressions:

- less than
- greater than
- less than or equal to
- greater than or equal to
- equality operation
- non-equality operation
- OR operation
- AND operation

Condition Statements

<if_statement> -> <matched> | <unmatched>

If statements can be either matched or unmatched (this is done in order to avoid ambiguity in statements with unmatched else statements).

<matched> -> if <relational> <matched> else <matched> | <LB> <non_if_statement> <RB>

A matched if statement is the one in which: starting from the right side of the statement, each else statement can be matched to an if statement, and there will be no dangling else statement left in the end.

<unmatched> -> if <relational> <LB> <statement> <RB> | if <relational> <matched> else <unmatched>

An unmatched if statement is the one in which the number of if statements and else statements do not match. In this case, we start from the right side of the statement, and match each unmatched else statement to the closest unmatched if statement.

Loops

<loop> -> <for> | <while>

A loop is either for or while loop.

<for> -> for <assignment> , <relational> , <assignment> <LCB> <statement_list> <RCB>
For loop consists of: “for” keyword, the base case of the loop, the condition, and then the operation to be done on each iteration of the loop, and then the block of the loop.

<while> -> while <relational> <LCB> <statement_list> <RCB>
While loop consists of: “while” keyword, the condition of the loop, and then the block of the loop.

Comments

<comment> -> <comment_sign> <phrase>
A comment consists of the comment sign, and the phrase of the comment.

<phrase> -> <letter> | <digit> | <phrase><letter> | <phrase><digit>
A phrase consists of letters and digits.

Function Definitions

<func_def> -> define <type> <func_identifier> <LP> <param_list> <RP> <func_body>
A function definition is written as follows: “define” keyword, type of return variable, function name, parameters list inside of parentheses, the function body.

<func_identifier> -> <lowercase_letter> | <func_identifier> <uppercase_letter> | <func_identifier> <lowercase_letter> | <func_identifier> <digit>
A func_identifier can be in one of these forms:

- A lowercase letter
- Starting with a lowercase letter, and then having uppercase letters and digits in between

An example of a function identifier convention would be: getHeightOfDrone

<func_body> -> <LB> <statement_list> return <var_identifier> <RB>
A function body is written as follows: open bracket, statements, return a variable, then close the bracket.

<param_list> -> <empty> | <type> <var_identifier> | <param_list> <comma> <type> <var_identifier>
Parameter list can be: empty, one variable, or multiple variables separated by comma.

Function Calls

<func_call> -> <var_identifier> <LP> <param_list> <RP>
Function call is written as follows: the function name, parameter list inside of parentheses.

Input and Output

<output> -> display <LP> <expression> <RP> | display <LP> <func_call> <RP> | display <LP> <quote> <phrase> <quote> <RP>
There are three different usages for output:
1 - “display” keyword, expression inside of parentheses.
2 - “display” keyword, function call; for outputting the returned value of a function.
3 - “display” keyword, a quote phrase inside of the parentheses; for displaying a string.

<input> -> take <LP><RP>
Input is used as follows: “take” keyword enables the user to enter his/her input via keyboard.

Primitive Functions

<readHeading> -> readHeading <LP><RP>
This function returns the heading direction of the drone, as an integer between 0 and 359.

<readAltitude> -> readAltitude <LP><RP>
This function returns the altitude of the drone.

<readTemperature> -> readTemperature <LP><RP>

This function returns the temperature of the environment around the drone.

<climbUp> -> climbUp <LP><RP>
This function moves the drone up with a speed of 0.1 m/s.

<dropDown> -> dropDown <LP><RP>
This function moves the drone down with a speed of 0.1 m/s.

<stopVertical> -> stopVertical <LP><RP>
This function stops the movement of the drone in the vertical direction.

<moveForward> -> moveForward <LP><RP>
This function moves the drone in the forward direction with a speed of 1 m/s.

<moveBackward> -> moveBackward <LP><RP>
This function moves the drone in the backward direction with a speed of 1 m/s.

<stopHorizontal> -> stopHorizontal <LP><RP>
This function stops the movement of the drone in the horizontal direction.

<turnHeadingRight> -> turnHeadingRight <LP><RP>
This function turns the heading right for 1 degree.

<turnHeadingLeft> -> turnHeadingLeft <LP><RP>
This function turns the heading left for 1 degree.

<sprayOn> -> sprayOn <LP><RP>
This function turns on the spray nozzle on the drone.

<sprayOff> -> sprayOff <LP><RP>
This function turns off the spray nozzle on the drone.

<connectToBase> -> connectToBase <LP><RP>
This function connects to the base computer through wifi.

<readHorizontalSpeed> -> readHorizontalSpeed <LP><RP>
This function returns the horizontal speed of the drone.

<readVerticalSpeed> -> readVerticalSpeed <LP><RP>
This function returns the vertical speed of the drone.

<climbUp> -> climbUp <LP><number><RP>
This function sends the drone up; to a desired height(in meters).

<climbDown> -> climbDown <LP><number><RP>
This function brings the drone down; to a desired amount(in meters).

<hitSide> -> hitSide <LP><RP>
This function returns a boolean: true if the function is on one of the sides of the field, false otherwise.

Terminals

<lowercase_letter> -> a|b|c…|z

<uppercase_letter> -> A|B|C…|Z

<letter> -> <lowercase_letter> | <uppercase_letter> | <space>

<under_score> -> _

<digit> -> 0|1|2…|9

<nonzero_digit> -> 1|2|...|9

<sign> -> + | -

<comment_sign> -> ##

<dot> -> .

<comma> -> ,

<semicolon> -> ;

<space> -> “ ”

<LP> -> (

<RP> ->)

<LB> -> {

<RB> -> }

<LSB> -> [

<RSB> ->]

<add_op> -> +

<sub_op> -> -

<mult_op> -> *

<div_op> -> /

<rem_op> -> %

<assign_op> -> <-

<equal> -> ⇔

<not_equal> -> <^=>

<greater_than> -> >

<less_than> -> <

<greater_or_equal> -> >=

<less_or_equal> -> <=

<not> -> ^

<and> -> &

<or> -> |

<quote> -> “

<empty> ->

Reserved words

main: A reserved word for the main function.

if: A reserved word for if statements.

else: A reserved word for else statements.

for: A reserved word for for loops.

while: A reserved word for while loops.

int: A reserved word for integer variable type.

db: A reserved word for double variable type.

str: A reserved word for string variable type.

bool: A reserved word for boolean variable type.

define: A reserved word for declaring a function.

return: A reserved word for the return statement of a function.

take: A reserved word for getting an input from the user.

display: A reserved word for printing an output.

Non-trivial Tokens

- A variable identifier is written with lowercase letters, with digits and underscores in between the letters. The variable identifier cannot start with a digit or an underscore, it can start only with a lowercase letter. Also, a variable identifier

cannot end with an underscore. An example of a variable identifier would be this: number_of_words.
    - The fact that we don't allow upper-case letters in defining variables improves **writability**, because using only lower-case letters increases simplicity in writing.
    - It also improves **readability** of the language, because having less variety in variable identifier names makes the code easier to read.
    - But it is not good in terms of **reliability**, because there might be variable cases(such as constant variables) that could be distinguished better from other variables if they were written in upper-case letters.

- A constant variable is written in the exact same way that a variable identifier is written, except that all the letters of a constant variable should be uppercase letters. An example of a constant variable would be this: NUMBER_OF_WORDS.
    - Using only uppercase letters when defining a constant variable, increases **readability**, because the reader can easily differentiate between constant variables and non-constant variables in a piece of code.
    - It might reduce **writability**, because it is a condition that the programmer should care about while writing the code.
    - It also increases **reliability,** because it makes a clear distinction between constant and non-constant variables.

- Comment sign is ##, and commenting a block(such as /* */ in Java), is not supported in this language.
    - This improves **readability** of the language, because the reader only needs to know one comment notation.
    - A possible disadvantage in **writability** might be that the programmer needs to use comment signs multiple times if he/she wants to comment a block of code.
    - Also for **reliability**, it is disadvantageous, because various comment notations help the programmer to distinguish the type of the commented lines.

- There are four different variable types in the language: integer, double, boolean, and string. Having different variable types increases **readability** because the

reader can know the variable types easier. It decreases **writability** because the programmer has to define the type of the variables. It increases the **reliability** of the language, because it considers multiple cases when dealing with variables.

- Most of the reserved words are chosen to be similar with popular programming languages(such as Java or Python). This increases **readability**, **writability**, and **reliability** of the Bee language. Because programmers who have experience with other languages can easily read, write, and use the Bee language's reserved words without confusion. Other reserved words(take, display, and define) are different from popular languages. However these words reflect their functionalities well, which increases **readability**, **writability**, and **reliability**.

Evaluation

Based on the descriptions above, the Bee Language is a readable and writable language. This is because it removes details that are not necessary. For example, in conditional statements and loops, Bee does not use parentheses in front of the reserved words(if-else, while, for). Another example: in function declaration, Bee does not distinguish between private or public functions. These simplifications reduce the reliability of the language. Generally, Bee can afford to be a simple language, because it is a language only for controlling drones.