CS 315 Project 1

2021-2022 Fall

Fam Programming Language

Arda Önal 21903350 Section 1
Cemal Faruk Güney 21903474 Section 3
Mert Barkın Er 21901645 Section 1

Table Of Contents

Grammar in Backus-Naur Form

Program

<program> -> <stmts>
<stmts> -> <stmt><stmts> | <stmt> |
<stmt> -> <assgmnt_stmt> | <loop_stmt> | <if_stmt> | <fnc_stmt> | <in_stmt> |
<out_stmt> | <comment>

Assignment Statements

<assgmnt_stmt> -> <declaration_stmt> | <stmt_with_operand>

<declaration_stmt> ->
  <var_type><identifier><assgmnt_op><variadic_right_side><end_stmnt_op>
 | <var_type><identifier><end_stmnt_op>

<stmt_with_operand> -> <unary_stmt> | <variadic_stmt>

<unary_stmt> -> <identifier><unary_postfix_op><end_stmnt_op>
                | <identifier> <unary_op> <const><end_stmnt_op>
                | <reserved_void_fnc>

<variadic_stmt> -> <identifier><assgmnt_op><variadic_right_side><end_stmnt_op>

Conditional Statements

<if_stmt> -> is <condition> ?? <stmts> endis
            | is <condition> ?? <stmts> endis <else_stmt>
            | is <condition> ?? <stmts> endis <or_stmt>
            | is <condition> ?? <stmts> endis <or_stmt> <else_stmt>

<or_stmt> -> or <condition> ?? <stmts> endor
              | or <condition> ?? <stmts> endor <or_stmt>
<else_stmt> -> else ?? <stmts> endelse

Loop Statements

<loop_stmt> -> <for_stmt> | <while_stmt>

<while_stmt> -> while <condition> ?? <stmts> endwhile

<for_stmt> ->
for <declaration_stmt> <comma> <condition> <comma>
<stmt_with_operand> ??
<stmts> endfor

Function Definitions and Function Calls

<fnc_dec> ->
fnc::<var_type> <identifier> <LP><fnc_params><RP><LB> <stmts> return
<variadic_right_side> <RB>
| fnc::<var_type> <identifier> <LP><fnc_params><RP><LB> return
<variadic_right_side> <RB>
| fnc::void <identifier> <LP><fnc_params><RP><LB> <stmts> <RB>

<fnc_params> -> <var_type><identifier> | <var_type><identifier> <comma>
<fnc_params> |

<fnc_stmt> -> <identifier><LP><list><RP>

Input/Output Statements

<in_stmt> -> in<LP><RP>
<out_stmt> -> out<LP><exp><RP>

Comments

<comment> -> <comment_op> <sentence> <comment_op>

Operators

<low_precedence_op> -> + | -

<high_precedence_op> -> * | /

<bool_op> -> && | || | == | > | < | <= | >= | !=

<unary_postfix_op> -> ++ | -- | **

<unary_op> -> += | -= | *= | /= | %=| ^=

<assgmnt_op> -> =

<end_stmnt_op> -> <semicolon>

Variables

<exp> -> <const> | <identifier> | <LP> <variadic_right_side><RP>

<list> -> <identifier><comma><list> | <identifier> |

<identifier> -> <word><number> | <word><number><identifier> | <word>

<const> -> <number> | <bool_var> | <string> | <double>

<bool_var> -> true | false

<string> -> <str_op> <sentence> <str_op>

<double> -> <number>.<number> | .<number>

<number> -> <sign><number> | <digit> | <digit><number>

<digit> -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<word> -> <letter> | <letter><word> | <letter><underscore><word> |
<word><underscore>

<sentence> -> <word><sentence> | <number><sentence> | <symbol><sentence> |
<word> | <number> | <symbol>

<condition> -> <exp> <bool_op> <exp>
| <exp> <bool_op> <exp> <bool_op> <condition>
| <exp>

<var_type> -> int | double | string | bool

<variadic_right_side> -> <variadic_right_side><low_precedence_op><term>
| <term> | <fnc_stmt> | <in_stmt> |
| <reserved_return_fnc>

<reserved_return_fnc> -> getHeading<LP><RP>
| getAltitude<LP><RP>
| getTemperature<LP><RP>
| connect<LP> <string>,<string>,<bool_var><RP>

<reserved_void_fnc> -> move_vertical<LP><number><RP>
| move_horizontal<LP><number><RP>
| turn<LP><bool_var><RP>
| move<LP><number>,<number>,<bool_var><RP>
| toggle_spray<LP><bool_var><RP>

<term> -> <term><high_precedence_op><factor>
| <factor>

<factor> -> <exp> ^ <factor> | <exp>

Symbols

<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

<comment_op> -> $$

<symbol> -> <LP> | <RP> | <LB> | <RB> | <LSB> | <RSB> | <comma> | <semicolon> | <underscore> | <equal> | <sign> | <hashtag> | <endline> | <dot>

<LP> -> (
<RP> ->)
<LB> -> {
<RB> -> }
<LSB> -> [
<RSB> ->]
<comma> -> ,
<semicolon> -> ;
<underscore> -> _
<equal> -> =
<sign> -> +|-
<hashtag> -> #
<endline> -> \n
<dot> -> .
<str_op> -> \”

Primitive Functions

Reading the heading

```
fnc::int getHeading(){
    return heading;
```

```
}

fnc::double getAltitude(){
	return altitude;
}

fnc::double getTemperature(){
	return temperature;
}

$$ direction_ver should be 1 for moving upwards, -1 for moving downwards and 0 for
stopping $$
fnc::void move_vertical(int direction_ver){
	is direction_ver == -1 ??
		drone_moving_vertical = true;
		drone_vertical_speed = -0.1;
		drone_moving_upwards = false;
	endis
	or direction_ver == 1 ??
		drone_moving_vertical = true;
		drone_vertical_speed = 0.1;
		drone_moving_upwards = true;
	endor
	else ??
		drone_vertical_speed = 0;
		drone_moving_vertical = false;
	endelse
}

$$ direction_hor should be 1 for moving forward, -1 for moving backward and 0 for
stopping $$
fnc::void move_horizontal(int direction_hor){
	is direction_hor == -1 ??
		drone_moving_horizontal = true;
		drone_horizontal_speed = -1;
		drone_moving_forward = false;
	endis
	or direction_hor == 1 ??
		drone_moving_horizontal = true;
		drone_horizontal_speed = 1;
		drone_moving_forward = true;
	endor
	else ??
		drone_moving_horizontal = false;
```

```
            drone_horizontal_speed = 0;
        endelse
}

$$ right should be true for increasing the heading and false for decreasing it $$
fnc::void turn(bool right){
        is right ??
            heading++;
            is heading > 359 ??
                heading = 0;
            endis
        endis
        else ??
            is heading < 0 ??
                heading = 0;
            endis
            heading--;
        endelse
}

$$ This function calls all moving functions with three given parameters $$
fnc::void move(int direction_ver, int direction_hor, bool right){
        move_vertical(direction_ver);
        move_horizontal(direction_hor);
        turn(right);
}

fnc::void toggle_spray(bool on){
        spray = on;
}

$$ This function connects the drone to the base computer with a given network
name and network password for wi-fi. Different helper functions are called for
Desktop and mobile base computers. $$
fnc::bool connect(string network_name, string network_pass, bool isDesktop){
        is isDesktop ??
            connectToDesktop(network_name, network_pass);
            return true;
        endis
        else ??
            connectToMobile(network_name, network_pass);
            return true;
        endelse
        return false;
```

```
}

fnc::void timer(){
$$ This function makes the code execution wait for one second. The implementation is done on the base computer in which the drone is connected through wifi. $$
}
```

Explanation of Grammar Constructs

<program> -> <stmts>

Program non-terminal consists of statements, which contains every feature that we added to our language. Our language does not require the user to specify where a program starts or ends.

<stmts> -> <stmt><stmts> | <stmt> |

This non-terminal consists of individual statements, which create our language. It can be a single statement, a list of statements or just an empty line.

<stmt> -> <assgmnt_stmt> | <loop_stmt> | <if_stmt> | <fnc_stmt> | <in_stmt> | <out_stmt> | <comment>

This non-terminal can be a variety of different non-terminals depending on the situation. It is used for distinguishing different types of statements.

<assgmnt_stmt> -> <declaration_stmt> | <stmt_with_operand>

This non-terminal is used for any declaration and value assigning operations. If a statement has an operand on its right hand side (RHS) it is classified as <stmt_with_operand>. If it does not have any operand it is classified as <declaration_stmt>. <stmt_with_operand> is generally used to change the value of a variable.

<declaration_stmt> ->
<var_type><identifier><assgmnt_op><variadic_right_side><end_stmnt_op>
| <var_type><identifier><end_stmnt_op>
| <var_type><identifier><assgmnt_op><fnc_dec><end_stmt_op>
| <var_type><identifier><assgmnt_op><in_stmt> <end_stmt_op>

This non-terminal is used to declare a variable that did not exist previously. To declare a variable the developer has to type its type and an identifier (name) to call the variable.

<stmt_with_operand> -> <unary_stmt> | <variadic_stmt>

This non-terminal is used for previously declared variables. Statements that match with these non-terminal are expected to assign a new value to a variable. These statements consist of two types of statements, unary and variadic statements.

<unary_stmt> -> <identifier><unary_postfix_op><end_stmnt_op>
| <identifier> <unary_op> <const><end_stmnt_op>

Unary statements are used to change a variable's value using its existing value. When the developer needs to increment, reduce, multiply, divide or get a power of the value of a variable by a constant number they shall use these types of statements. If the constant number they want to use is 1 and they are trying to increment or reduce the value they can use the statement with postfix operators ("++" or "--"). Other than that they should use other unary operators.

<variadic_stmt> ->
<identifier><assgmnt_op><variadic_right_side><end_stmnt_op>

Variadic statements are used to assign a new value to an existing variable. This new value can be different from the variable's existing value. The variable on the left hand side and the value on the right hand side should be compatible.

<if_stmt> -> is <condition> ?? <stmts> endis
| is <condition> ?? <stmts> endis <else_stmt>
| is <condition> ?? <stmts> endis <or_stmt>
| is <condition> ?? <stmts> endis <or_stmt> <else_stmt>

This non-terminal is used to indicate conditional statements. Every conditional statement starts and ends with given keywords. An is statement can have both or statements and an else statement matched with it or it can have one of them or it can be on its own.

<or_stmt> -> or <condition> ?? <stmts> endor
| or <condition> ?? <stmts> endor <or_stmt>

This statement is usually referred to as an "else if" statement in other programming languages. It is used after an "is" statement. If the condition of the matched "is" statement is not satisfied "or" statement is checked. If the conditions given by the "or" statement are met the statements between the keywords are executed.

<else_stmt> -> else ?? <stmts> endelse

Else statements come after “is” or “or” statements. They do not have a specific condition. If none of the previous conditions are satisfied the statements between question marks and “endelse” are executed.

<loop_stmt> -> <for_stmt> | <while_stmt>

This non-terminal is used to combine all loop statements. There are two types of loops. One of them is for statements and the other is while statements. The common do while statement is not implemented because there is no need for it in the domain.

<while_stmt> -> while <condition> ?? <stmts> endwhile

While statements give a condition and start executing the statements between question marks and endwhile keywords if the condition is satisfied and keeps executing these statements until the condition is not satisfied.

<for_stmt> ->
for <declaration_stmt> <comma> <condition> <comma> <stmt_with_operand> ??
<stmts> endfor

For statements consist of three different statements. First of them is a declaration statement that is used to declare a new variable and give it a value. Second of them is a condition, the for loop will start if this condition is met and it will keep running until the condition is wrong. Third statement is a statement that changes the value of a variable; this statement will be executed on every iteration. If the condition is met, statements between question marks and “endfor” keyword will be executed until the condition is not satisfied.

<fnc_dec> ->
fnc::<var_type> <identifier> <LP><fnc_params><RP><LB> <stmts> return <variadic_right_side> <RB>
| fnc::<var_type> <identifier> <LP><fnc_params><RP><LB> return <variadic_right_side> <RB>
| fnc::void <identifier> <LP><fnc_params><RP><LB> <stmts> <RB>

This non-terminal is used to declare a new function. A function declaration must start with “fnc::” so that it could be fastly recognized by anyone reading the code. Functions must have a return type either void or a variable type. If the return type is void no return statement will be needed otherwise, the function must return a value that is compatible with its return type. Functions have parameters that are given to the function upon calling.

<fnc_params> -> <var_type><identifier> | <var_type><identifier> <comma> <fnc_params> |

Function parameters is a list containing parameter values for that will be given to a function. The list can have 0 or more elements. If there are more elements than 1, comma will be used to separate these elements.

<fnc_stmt> -> <identifier><LP><list><RP>

This non-terminal is used to call functions from any place in functions scope. The identifier and the list together create the function signature. This signature must match a previous function declaration for the call to be useful.

<in_stmt> -> in<LP><RP>

This non-terminal indicates that there is an input statement. Input statements are used to get a value from the user.

<out_stmt> -> out<LP><exp><RP>

This non-terminal is used to prompt the user with an output. The statement must have an expression between parentheses. This expression can be a variable, a constant or an operation.

<comment> -> <comment_op> <sentence> <comment_op>

This non-terminal shows the structure of a comment in the programming language. Anything between comment operators is classified as a comment and it does not affect the program.

<low_precedence_op> -> + | -

This non-terminal consists of the operations that do not have priority over other operations. When a parse tree of the program is drawn, statements with these operators should be high up on the parse tree.

<high_precedence_op> -> * | /

This non-terminal consists of the operations that do have priority over low precedence operators. On the parse tree of the program statements with these operators should be lower in the tree than the statements with low precedence operators.

<bool_op> -> && | || | == | > | < | <= | >= | !=

Boolean operators are used to create logical equations. The type of operators are and, or, equals, more than, less than, less than or equal to, more than or equal to, not equal respectively.

<exp> -> <const> | <identifier> | <LP> <variadic_right_side><RP>

This non-terminal is used to combine the non-terminals that can be the right hand side of an assignment or can be returned in a function.

<list> -> <identifier><comma><list> | <identifier> |

This non-terminal shows the structure of a list in the programming language. A list consists of zero or more identifiers separated by a comma.

<identifier> -> <word><number> | <word><number><identifier> | <word>

This non-terminal shows the rules to create an identifier. An identifier is used for referring to a variable or a function. The identifier can consist of both numbers and words (letters and underscore symbol); however, it can only start with a word. It can not start with a number.

<const> -> <number> | <bool_var> | <string> | <double>

This non-terminal indicates the types of constants in the programming language. A constant's value can not be changed with assignment arguments. They can not be on the left hand side of any argument. They can not be declared. The types of constants are numbers, boolean variables, strings and doubles.

<bool_var> -> true | false

A boolean variable is a variable used in conditional arguments. It can be either true or false.

<string> -> <str_op> <sentence> <str_op>

Any character combination between string operators is considered as a string in the programming language.

<double> -> <number>.<number> | .<number>

A double is basically a 64 bit floating number. If the number before the dot sign is 0 it can be skipped. Otherwise, doubles are created by two numbers.

<number> -> <sign><number> | <digit> | <digit><number>

A number can contain a single digit or multiple digits. It can also have a preceding sign. If the sign is omitted it is assumed as a plus sign.

<digit> -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

This non-terminal contains all digits available in the Fam programming language. These digits can be used to create numbers.

<word> -> <letter> | <letter><word> | <letter><underscore><word> | <word><underscore>

Words in the Fam language consist of letters in the English alphabet. They can also have underscore characters between or after the letters however, they can not start with underscore.

<sentence> -> <word><sentence> | <number><sentence> | <symbol><sentence> | <word> | <number> | <symbol>

This non-terminal is used for sentences. A sentence can consist of any character in the language such as letters, numbers and symbols.

<condition> -> <exp> <bool_op> <exp>
| <exp> <bool_op> <exp> <bool_op> <condition>
| <exp>

A condition can consist of two expressions separated with a boolean operator or it can just have one expression. Condition statements also can have more than one condition in it.

<term> -> <term><high_precedence_op><factor>
| <factor>

This non-terminal is used for left recursive high precedence operations such as multiplication and division.

<factor> -> <exp> ^ <factor> | <exp>

This non-terminal is used for exponent operation. The operation is right recursive.

<symbol> -> <LP> | <RP> | <LB> | <RB> | <LSB> | <RSB> | <comma> | <semicolon> | <underscore> | <equal> | <sign> | <hashtag> | <endline> | <dot>

This non-terminal is used for combining all symbols included in Fam language.

Tokens

<DOT>
<INT_TYPE>
<BOOLEAN_TYPE>
<STRING_TYPE>
<DOUBLE_TYPE>
<ASSGMNT_OP>
<BOOL_OP>
<COMMENT>
<ELSE_STMT>
<LP>
<RP>
<LB>
<RB>
<LSB>
<RSB>
<COMMA>
<INT>
<DOUBLE>
<IF>
<WHILE>
<FOR>
<TRUE>
<FALSE>
<RETURN>
<FUNC_DEC>
<QUESTION_MARKS>
<END_STMNT_OP>
<ENDIS>
<ENDOR>
<OR_STMT>
<ENDELSE>
<WHILE_END>
<FOR_END>
<IN>
<OUT>
<VOID>
<STR_OP>
<LOW_PRECEDENCE_OP>
<HIGH_PRECEDENCE_OP>
<UNARY_POSTFIX_OP>
<UNARY_OP>
<POWER_OP>
<RESERVED_RETURN_FNC>
<RESERVED_VOID_FNC>
<IDENTIFIER>
<STRING>

Non-Trivial Tokens

<IF>: “is” keyword is used for representing the if statements. The reason for this is the conditional statements that usually come after the if statements are questions that can be started with “is”. By doing this, we tried to make our language more readable and unique.
<OR_STMT>: “or” keyword is used for representing else if statements. The reason for this is if we read a code written in our language, the conditional statements sound like real life conditions. For example: is x == 5?? or x ==7??. We can say that this both increases and decreases the readability of the language because using or statement for else if statements may seem confusing to first time learners especially those who know other programming languages however, it also increases the readability because this makes the programming language closer to natural languages. The or statement also increases the writability compared to other programming languages because “else if” has a lot more characters than “or” hence, harder to write.

<FUNC_DEC>: “fnc::” keyword is used for defining languages. The reason for this is to improve the readability of our language since in most of the programming languages, function definitions start with return type and this often causes confusions at first glance. By adding a keyword when defining a new function, it is easy to notice that there will be a new function signature after the keyword.

<QUESTION_MARKS>: “??” sign is used for defining the beginning of if, else if, else, while, for statements. The reason for this is the conditional statements are actually undetermined which can be true or false. The question marks indicate that there is a conditional statement which can lead to different outcomes. This increases the readability of our language by making the arguments closer to real life human language.

<ENDIS>: “endis” keyword marks the end of “is” (if) statements. While this increases the readability of the language by specifying the end of the if statements, it decreases the writability because most common programming languages use brackets which are one char, endis is 5 characters which takes more time to write.

<ENDOR>: “endor” marks the end of or (else if) statements. Similar to endis, this increases the readability but decreases writability.

<ENDELSE>: “endelse” marks the end of else statements. Similar to endis and endor, this increases the readability but decreases writability.

<WHILE_END>: “endwhile” marks the end of while statements. Similar to endis, endor and endelse this increases the readability but decreases writability.

<FOR_END>: “endfor” marks the end of for statements. Similar to endis, endor, endwhile and endelse this increases the readability but decreases writability.

<IN>: “in” keyword is used for getting input from the console. This increases the readability a lot since in most common programming languages, getting input from the console requires high effort such as using the Scanner class in Java. In our programming language, this is achieved by simply two characters. The downside is that just the word in does not make sense which decreases the readability of the program.

<OUT>: “out” keyword is used for outputting information to the console. Similar to “in”, this increases the readability and decreases the readability.

<UNARY_POSTFIX_OP>: The only different operator to other programming languages is the use of “**”. This allows it to multiply the number by itself and assign it to itself. For example, “int x = 3; x**;” After the execution of this, the value of x becomes 9. This increases the writability by allowing easier computation however, it decreases the readability because the syntax may be confusing to first time users/readers.

<UNARY_OP>: The different operators that are added are “%=” and “^=”. These have similar usage to “+=”, “-=”, “*=” in most common programming languages. “%” is for mod and “^” is for taking the power. These operators once again increase the writability by allowing easier computation; however, it decreases the readability because the syntax may be confusing to first time users/readers.

<POWER_OP>: The power operator is one of the most important operators in mathematics yet, in most of the programming languages, it is very hard to compute. Our language provides this with a simple “^” sign. This is very easy to understand in mathematical context and easy to write hence, it increases the readability and writability.

<RESERVED_RETURN_FNC>: This token indicates the reserved functions with a return (non-void) that will be used in the drone which are getHeading, getAltitude, getTemperature, connect. These increase the writability of the functions because the application domain of it is used on drones and built in functions makes it easier to use them. They also make the language more reliable because it allows the users to use the drone functions very easily and correctly.

<RESERVED_VOID_FNC>: This token indicates the reserved functions without a return (void) that will be used in the drone which are move_vertical, move_horizontal,

turn, move, toggle_spray. These increase the writability of the functions because the application domain of it is used on drones and built in functions makes it easier to use them. They also make the language more reliable because it allows the users to use the drone functions very easily and correctly.

Language Evaluation

While implementing the language, we tried to make our language unique, readable, writeable and reliable. In the “Non-Trivial Tokens” heading, each tokens that are unique to our language is explained in detail with why they were chosen and how they impact the writability, readability and reliability. In the following subsections, the language is evaluated in terms of writability, readability and reliability more generally.

Readability: The readability of a language implies how easily it is understood when it is read. We tried to make our language as readable as possible by making the programming language statements similar to already widely used programming languages and also, similar to statements in natural languages. Our language is similar to C group languages in terms of using curly brackets, semicolons, assignment operations and function calls. However, we have unique features that increase the readability such as using “is” and “or” keywords and “??” sign to denote conditional statements. These are closer to natural languages which increases the readability. However, there are components which decrease readability. These are made to increase writability. There has to be a balance between writability and readability and our language I believe achieves this. Under the “Non-Trivial Tokens” section, every token that is unique to our language is explained in detail in terms of why they were chosen and how they impact readability. For specific examples please refer to the “Non-Trivial Tokens” section. Lastly, our language is case sensitive which means that there can be variables that have the same names but with different cased characters. All in all, we believe that the readability of our language is higher than most common programming languages.

Writability: The writability of a language is how easily the code is written. It is determined by comparing similar code written in different languages and how many characters it requires to code a program that does the same things. The application domain of our language is drones that will be used for spraying pesticides or fertilizers over grain or vegetable fields or fruit plantations. The commands that drones can do are already built in our language which means that our language is probably one of the most writable languages for the application domain. If we were to compare the other language components, our language offers a very high readability on some features which decreases the writability however, there are also other features which are far easier to write compared to the most common programming languages. The specifics of the impacts of different new tokens on writability are

discussed under the section “Non-Trivial Tokens”. We believe that our language is definitely not hard to write in general usage and is very useful for its application domain. Hence, we can say that our language has good writability for general usage and excellent writability for drones that will be used for spraying pesticides or fertilizers over grain or vegetable fields or fruit plantations.

Reliability: The reliability of a language is determined by considering the ability of programs written in that language to comply with its specifications, under all conditions. Especially for drone usage, our language is very reliable and the code written in it will be able to achieve what the programmer desires. Furthermore, all the other statements of the language do what they are actually supposed to do which means that it is reliable. The reason for this is that the interpreter of our language will be more advanced and predict the possible incorrect cases. For example, our language will not allow a statement like “x < y < z” where we compare integers because the result of “x<y” returns a boolean type and comparison of a boolean with an integer will be illegal. This issue is not checked in for example the C language which makes it unreliable. We will try to solve the most common unreliability issues of the common programming languages by implementing a very good interpreter.

Sample Programs

Sample Program 1

```
$$ This program makes the drone spray in a user specified region $$

$$ direction should be 1 for moving forward, -1 for moving backward and 0 for
stopping $$
fnc::void goHorizontal(int meters, int direction){
        int distanceTraveled = 0;
        move_horizontal(direction);
        while distanceTraveled  < meters ??
                timer(); $$ program waits for one second $$
                distanceTraveled++;
        endwhile
        $$ drone went the amount we wanted at this point, so we stop the drone $$
        move_horizontal(0);
}

$$ direction should be 1 for moving upwards, -1 for moving downwards and 0 for
stopping $$
fnc::void goVertical(int meters, int direction){
```

```
        is direction != 0 ??
                int initialHeight = barometer.getAltitude();
                move_vertical(direction);

                $$ If moving downwards, direction is -1 and (barometer.getAltitude() -
                initialHeight) value will be negative so we multiply by direction $$
                while direction*(barometer.getAltitude() - initialHeight) < meters ??
                endwhile
        endis
        $$ drone went the amount we wanted at this point, so we stop the drone $$
        move_vertical(0);
}


$$ Program start $$
out("Welcome, this program allows the drone to spray in a rectangular area.\n");
out("Note: It assumes that the drone is already positioned on the bottom left corner
of the field.\n");
out("The current temperature outside is: ");
out(getTemperature());
out(" degrees Celcius\n");

out("Are you sure you want to irrigate the field? Enter Y for yes, N for no:");
string input = in();
while input != "Y" || input != "N" ??
        out("Error: Invalid input, please type Y or N:\n");
        input = in();
endwhile

is input == "N" ??
        out("Ending program.\n");
endis
or input == "Y" ??
        out("Please enter the x dimension of the field:\n");
        int x = in();
        while x < 0 ??
                out("x must be positive. Please re enter x:\n");
                x = in();
        endwhile

        out("Please enter the y dimension of the field:\n");
        int y = in();
        while x < 0 ??
                out("y must be positive. Please re enter x:\n");
```

```
        y = in();
    endwhile

    goVertical(10, 1); $$ Drone will spray from a height of 10 meters. $$

    toggle_spray(true); $$ Start the spray of the drone. $$
    out("The drone has started irrigating. \n");

    for int i = 0; i < x / 2; i++ ??
        $$ Turn the drone 90 degrees $$
        while getHeading() != 90 ??
            turn(true);
        endwhile

        goHorizontal(y,1); $$ spray for y meters $$

        while getHeading() != 0 ??
            turn(false);
        endwhile

        goHorizontal(1,1); $$ spray for one meter upwards $$

        while getHeading() != 270 ??
            turn(true);
        endwhile

        goHorizontal(y,1); $$ spray for y meters $$

        while getHeading() != 0 ??
            turn(true);
        endwhile

        goHorizontal(1,1); $$ spray for one meter upwards $$
    endfor

    toggle_spray(false); $$ Start the spray of the drone. $$
    out("The irrigation is done.");

    goVertical(10, -1); $$ Drone return to the ground. $$
endor
```

Sample Program 2

$$ This program includes useful functions for a drone company to use in their implementations. $$

```
fnc::void startNozzle(bool flag, int time){
        is tank.getVolume() == 0 ??
                is flag == true??
                        return;
                endis
                return;
        endis
        tank.startSpray();
        tank.setVolume(tank.getVolume - time * tank.getSpeed());
}
```

$$ This function connects the drone to the base computer with a given network name and network password for wi-fi. Different helper functions are called for Desktop and mobile base computers. $$

```
fnc::bool connect(string network_name, string network_pass, bool isDesktop){
        is isDesktop ??
                return connectToDesktop(network_name, network_pass);
        endis
        else ??
                return connectToMobile(network_name, network_pass);
        endelse
        return false;
}

fnc::bool connectToDesktop(string network_name, string network_pass){
        bool flag = doesWifiExist(network_name, drone.getRange());
        is flag == true??
                drone.connect(network_name,network_pass);
                return true;
        endis
        else ??
                return false;
        endelse
}

fnc::bool connectToMobile(string network_name, string network_pass){
        bool flag = doesWifiExist(network_name, drone.getRange());
```

```
	is flag == true??
		drone.connect(network_name,network_pass);
		return true;
	endis
	else ??
		return false;
	endelse
}

fnc::bool doesWifiExist(string s, int r)
{
	$$ looks around the radius for available wifis and returns true if there is one
with the same name$$
	for int i = 0, i < r^2, i++ ??
		is Router.getName() == s ??
			return true;
		endis
	endfor
	$$ if the function comes to this point, it couldn't find the wifi so it does not exist
$$
	return false;
}
```

Sample Program 3

```
$$ A program to calculate the average of any number of integers. This is made to
display that the application domain is not just drones, but any other program can be
written. $$
int x = in();
int val = 0;

for int i = 0, i < x, i++ ??
	val += in();
	endfor
out(val / x);
```