BİLKENT UNIVERSITY
CS 315 PROGRAMMING LANGUAGES COURSE
TERM PROJECT- 2
TEAM 35

Burak Öçalan	Section 3	21703769
Melike Fatma Aydoğan	Section 1	21704043
Ömer Faruk Akgül	Section 1	21703163

CLOUD

I. **Language Name:** Cloud

II. BNF Description of Cloud

```
<program> ->begin <stmts> end | <function_def> <program>

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

<stmt> -> <function_call>; | <assignment>; | <while_stmt> | <if_stmt> |
          <for_stmt> | <variable_declaration>; | <return_stmt>;

<variable_declaration> -> <variable_type> <ident_init> | <variable_declaration> ,
<ident_init>
<variable_type> -> int | char | string | bool
<ident_init> -> <assignment> | IDENTIFIER

<assignment> -> IDENTIFIER <assignment_op> <assignment_expr>

<expression> -> <assignment_expr>

<assignment_expr> -> <logical_or> | <assignment>
<assignment_op> -> = | += | -= | *= | /= | %=

<logical_or> -> <logical_or> OR <logical_and> | <logical_and>
<logical_and> -> <logical_and> AND  <equality_expr> | <equality_expr>

<equality_expr> -> <equality_expr> <equality_op> <relat_expr> | <relat_expr>
<equality_op> -> == | !=

<relat_expr> -> <relat_expr> <relat_op> <additive_expr> | <additive_expr>
<relat_op> -> <|<=|>|>=

<additive_expr> -> <additive_expr> <additive_op> <multiplicative_exp> |
<multiplicative_exp>
<additive_op> -> + | -

<multiplicative_exp> -> <multiplicative_exp > <multiplicative_op> <unary_exp> |
<unary_exp>
<multiplicative_op> -> * | / | %

<unary_exp> -> <unary_op> <unary_exp> | <parenth>
<unary_op> -> + | - | NOT

<parenth> -> (<expression>) | <function_call> | <const> | IDENTIFIER
<const> -> INT_LITERAL | STRING_LITERAL | CHAR_LITERAL | <bool_literal>
```

```
<bool_literal> -> True | False

<while_stmt> -> while ( <expression>) <stmts> endwhile
<for_stmt> -> for ( <for_init> ; <expression>; <for_assignment>) <stmts> endfor
<for_init> -> <variable_declaration> | <assignment>
<for_assignment> -> | <assignment>

<if_stmt> -> if (<expression>) <stmts> endif |
        if (<expression>) <stmts>else <stmts> endif |
        if (<expression>) <stmts> <elif_stmts> endif

<elif_stmts> -> elif(<expression>) <stmts> |
                elif(<expression>) <stmts> else <stmts> |
                elif(<expression>) <stmts> <elif_stmts>

<function_def> -> <func_return_type> <func_name> (<parameters>) <stmts> endfunc
<func_return_type> -> void | <variable_type>
<parameters> -> | <parameter_list>
<parameter_list> -> <parameter> | <parameter> , <parameter_list>
<parameter> -> <variable_type> IDENTIFIER
<func_name> -> IDENTIFIER

<return_stmt> -> return <expression> | return

<function_call> -> <defined_function_call> | <primitive_function_call> |
<input_output_function_call>

<defined_function_call> -> <defined_function_name> (<arguments>)
<arguments> -> | <argument_list>
<argument_list> -> <argument> | <argument>, <argument_list>
<argument> -> <expression>
<defined_function_name> -> IDENTIFIER

<input_output_function_call> -> <input_output_function_name> (<arguments>)
<input_output_function_name> -> print | scan | print_to_file | scan_from_file

<primitive_function_call> -> <primitive_function_name> ()
<primitive_function_name> -> get_inclination | get_altitude
                        get_temperature | get_acceleration
                        turn_on_camera  | turn_off_camera
                        take_picture    | get_timestamp
                        connect_base | is_camera_active
```

III. Explanations of Language Constructs

Tokens used in bnf description:

alphabetic	[A-Za-z_$]
digit	[0-9]
alphanumeric	({alphabetic}|{digit})

IDENTIFIER:	{alphabetic}{alphanumeric}*
INT_LITERAL:	{digit}+
STRING_LITERAL:	\".*?\"
CHAR_LITERAL:	\'.\'

<program> : Represents the Cloud program. A Cloud file can only have a single Cloud program. It starts with "begin" and ends with "end" reserved words. When the program execution begins, the computer executes the statements between "begin" and "end" words in order. It is allowed to define functions before Cloud program, i.e. before the begin keyword. These functions can be called in the program. Note that a Cloud program works similar to the "main" function in other languages. However, we differentiate our syntax of program definition from function definitions.

<stmts> : Represents a bunch of statements (<stmt>'s). Used in program-function definitions, loops and conditional statements. Note that statements can also be empty.

<stmt> : Represents a statement. Building block of our <stmts> nonterminal. Cloud supports many different statement types, which are function calls, expressions (arithmetic, relational, boolean expressions and combinations of them.), loops (for and while statements), conditional statements (if statements), variable declarations and returns statements for function definitions. Most of the statements, excluding loops and if statements, end with a semicolon. Loops and if statements have their own reserved words to represent their end. We will explain each statement detailly.

<variable_declaration>, <variable_type>, <ident_init>: In Cloud language, each variable has a name and a type. Variables might also have values if assigned. <variable_type> defines the possible variable types of a variable, which is "int, char, string and bool". Variables' names can only be IDENTIFIERs. Variables may or may not be assigned to a value in declaration. <ident_init> represents those two possibilities. Note that <assignment> is merely assigning right-hand value to a variable. We may also have multiple declarations of variables with the same type in a single variable declaration statement. Those declarations are separated by comma (,).

<expression> : In our language, expressions include assignment, relational, equality, additive, multiplicative and unary expressions, parenthesis, constants and identifiers. The precedence of these expressions is listed as increased order in the previous sentence. Also,

expressions with lower precedence include both their rule and expressions with higher precedence to operate the precedence rule.

<assignment> <assignment_expr> : Expressions are either assignments, or non-assignments. This non-terminal represents the assignment part. As we said before, assignments are merely assigning right-hand value to a variable, whose name is an IDENTIFIER. This assignment is operated by some <assignment_op>'s. The right hand value, <assignment_expr>, can actually be any expression, either assignment or non-assignment. Note that <logical_or> represents the all non-assignment expressions. So, our assignments return values and these values can be used just like other expressions. Assignment expressions have the lowest precedence. Also, they have right to left associativity.

<assignment_op> : Assignment operators are assignment (=), multiplication assignment (*=), division assignment (/=), modulus assignment (%=), addition assignment (+=), subtraction assignment (-=). We add the last five assignment operators to increase the writability.

<logical_or> : In Cloud language, <logical_or> nonterminal represents two types of expressions. The first one is left recursive, which is used for statements in the form of x OR y. Since it is a recursive expression, any expression with high precedence can be applied to either side of the OR. The second one represents expressions with the higher precedence, i.e. <logical_and>. This case is used for the expressions excluding OR.

<logical_and> : As in <logical_or>, <logical_and> can be used for two types of expressions. One of these two rules contains the AND operator in it, while the other is a rule that does not contain an AND operator and has higher precedence. When statements with a value of true come to both sides of AND, the truth value of the expression is 1. Also, since the rule containing AND is recursive, it can generate expressions containing multiple ANDs.

<equality_expr> : This expression has two operands and one operator and checks the equality or non-equality of the two expressions. For the first operand, we have an expression with higher or equal precedence. For the second operand, we have an expression with higher precedence. This expression has left to right associativity, that's why the first operand can be an expression with the same precedence, but the second cannot be. Also, if a <equality_expr> is not matched with the given rule, it will continue with higher precedences as <relat_expr>.

<equality_op> : This construct is the different types of equality operators which are equality(==) and inequality(!=). With the equality operator(==), language checks if the result of the two given expressions are equal or not. With inequality operator(!=), language checks if the result of the two given expressions are inequal or not.

<relat_expr> : When one of the relational operators is used in an expression, <relat_expr> compares the value of statements and returns the truth value of the expression. Thanks to the recursive property, there can be more than one <relat_expr> on the left side of the operator. If there is no <relat_op> in the expression; expression is redirected to <additive_expr> which has a higher precedence.

<relat_op> : Relational operators mainly consist of four types. < (less than) operator checks whether the left side of the operator is less than the other side. Unlike, > (greater than) operator checks if the left side of the > is greater than the right side. When = (equal) sign is used after either of the < and >, the language additionally checks whether both sides are equal.

<additive_expr> : Additive expressions includes addition and subtraction rules. Its general rule has two expressions and an additive operator. The first expression can be <additive_expr> since this rule's associativity is, also, left to right. If language cannot detect the additive rule from the given <additive_expr>, then it will continue with multiplicative rules as <multiplicative_exp>.

<additive_op> : The Cloud language has two additive operators, addition (+) and subtraction (-). These two operators should not be confused with unary sign operators. The addition operator finds the summation of the two expressions whereas the subtraction operator finds the subtraction.

<multiplicative_exp> : This nonterminal is used for operations with higher precedence than addition and subtraction, such as multiplication, division, modulus. If the expression contains one of the <multiplicative_op> s, the first rule of this nonterminal is taken into account. According to this rule, the left side of the operator is recursive, so there can be new expressions containing <multiplicative_op>. The second rule is used for expressions with higher precedence, in which there are no operators mentioned.

<multiplicative_op> : In Cloud language, there are three types of multiplicative operators. * is used for multiplication, while / represents division. Lastly, % is an operator used to calculate the remainder.

<unary_exp> : The unary expression includes one of the unary operators and an <unary_expr>. This expression's associativity is right to left. With this associativity, if a unary expression has inner unary expressions, firstly, the inner one will be calculated and then the outer ones. In the operand of a unary expression, we can just use another unary expression, expressions in parentheses, constants and identifiers. Since parentheses already include constants and identifiers, we defined <unary_exp> as including parenthesis to continue with the highest precedence operator if a unary expression rule is not found in the <unary_exp>.

<unary_op> : Unary expression has two sign operators (+, -) and the logical not operator (NOT). With the positive sign operator (+), we can show a positive integer constant's sign and with the negative sign operator (-), we can both show a negative integer constant's sign and calculate the expression value as multiplied with -1 which means calculated with the changed sign.

<parenth> : Expressions given in parentheses have higher precedence than any of the additive and multiplicative operators in the Cloud. Moreover, <function_call>, <const> and IDENTIFIERs have equal degree of precedence with parentheses. When (<expression>) is given all the steps described after <expression> can be repeated.

<const>: This nonterminal is used as the main title of the literals, which have 4 types in Cloud. INT_LITERAL is the token used for integers. Similarly, STRING_LITERAL and CHAR_LITERAL are the tokens used for strings and chars, respectively. Lastly, <bool_literal> is not a token, it is a nonterminal.

<bool_literal>: The Boolean literals are the keywords True and False. They are right-hand side values of the type bool.

<while_stmt> : Our first loop statement is while statement. Its semantics is the same as most known languages. It executes the statements inside the while loop until the expression between brackets returns false. Our language does not use curly brackets to determine the scope of the while statement. Instead, it has a reserved word "endwhile" which represents the end of a while loop. Note that in our current description of while, it seems that it can take any kinds of expressions, including the expressions returning non-boolean values, between brackets. However, we are planning to accept only boolean values in while loop conditions by considering the value of <expression>.

<for_stmt>, <for_init>: Our second loop is for statement. Its semantics is again the same as most known languages. There are three parts in between brackets, separated by two semicolons. The first part is executed before the loop starts. As described in <for_init>, this part can be either an assignment or a variable declaration. Note that variable declarations can also include assignments. The second part is checked before each iteration of the loop. Loop continues until this part returns False. Note that similar to the while statement, we are planning to disable non-boolean values in this part. The third part is executed after each iteration of the loop. This part can have only an assignment, or it can be empty. Note that in the first and third parts, Cloud doesn't support multiple statements as it would complicate the code. Instead, we recommend the consideration of using a while loop. As a last note, similar to while statement, for statement also uses a reserved word to indicate its end, named as "endfor".

<if_stmt>, <elif_stmts>: If statement makes use of four reserved words, "if", "elif", "else", "endif". elif stands for "else if" in many languages. Before starting the explanation, note that similar to for and while statements, we are again planning to disallow non-boolean values in <expression>'s. Our general syntax is as follows:

```
if ( <expression> )
        <stmts>
elif ( <expression> )
        <stmts>
...
... ===========> Continous elif statements. Number of the elif statement n can be from 0
to plus infinity -in theory-.
...
else
        <stmts>
endif
```

Note that an if statement must include the "if" part at the beginning and "endif" keyword at the end. However, "elif" and "else" are optional. The semantics of if statement is the same as other most known languages. Because all of the potential readers of this report know how the semantics of if statement works in most known languages, we decided not to bore you with a detailed description of if statement. As a final reminder about loops and the if statement, because they all are <stmt>'s, they can be used intertwined. For example, a for loop inside a while loop may include 2 if statements which again include a while loop.

<function_def>, <parameters>, <parameter_list>, <parameter>, <func_var_type>, <func_name> : Our language allows defining new functions. As it is said before, these functions must be defined before our Cloud program (which begins with "begin" and ends with "end"). A function basically accepts some <parameters>, executes some <stmts> and returns a result. A <parameter> is merely a variable name preceded by its type. Note that <parameter> can have any type defined in Cloud language. <parameter_list> is <parameter>s separated by comma. <parameters> is either a list of parameters, or empty if this function does not accept any parameter. A function must have a name and this name must be IDENTIFIER. A function must also have a return type, which is represented as <func_return_type>. A function can return a value with any defined variable type in Cloud language, or it can return "void" which actually means it returns nothing. As a last note, <stmts> in functions can also include a <return_stmt>, which is used by functions to return values and which will be explained separately next.

<return_stmt> : A return statement starts with reserved word "return" and may or may not be followed by an expression. The return statement can be used in both functions and the Cloud program. When used in functions, the function stops its execution of statements and returns the corresponding value. Note that there might be statements after return statements but they will not be executed. Some functions don’t have to return a value. These functions have a return type "void" as mentioned before. In these functions, the return statement is used without an expression. Return statements can also be used in the Cloud program. Similar to functions, Cloud program stops its execution after encountering the return statement. Originally, Cloud programs don't return any value, however, it is syntactically correct for a return statement to be followed by an expression. In this case, it works the same as void return statements.

<function_call> : Our language has three different types of function calls; defined, input-output, primitive. We will explain all in detail, but as a general rule, they are called with their function names and their arguments - if they take any -

<defined_function_call>, <defined_function_name>, <arguments>, <argument_list>, <argument> : As its name suggests, defined functions are the ones that the programmer defines with the function definition rules given above. After defined, these functions can be called in the Cloud program or in another function definition. Note that a function can only call functions defined before it. We are planning to carry out this rule in our future work. Defined functions are called with their names and arguments. A function name, denoted <func_name> is an IDENTIFIER, just like in function definition. A function call also has some <arguments> which are coupled with the <parameters> in function definition. <arguments> is merely a list that consists of expressions. Note that <arguments> can also be empty just like <parameters>.

<primitive_function_call>, <primitive_function_name> : Unlike the defined functions, primitive functions are pre-defined in the language. In other words, they can be called in anywhere in the code without defining them. Actually, they cannot be defined as a function because their names are reserved words, which are given by <primitive_function_name>. Note that <func_name>s in <function_def> cannot be reserved words because IDENTIFIERs also can't be. The last difference of primitive function calls from defined function calls is that they cannot have arguments.

<input_output_function_call>, <input_output_function_name> : In cloud language, inputs can be scanned from a keyboard or a file, and outputs can be printed to a console or a file. Input output functions are in a place between defined functions and primitive functions. Similar to primitive functions, they are pre-defined functions and their names are reserved words. However, they can take arguments like defined functions.

IV. Definitions of Tokens

Comments: We have both single line and multi-line comments. Single line comments simply start with a '#' character and ends with a newline. Between those, it can take any character besides a newline character. Multi-line comments start with '#-' and end with '-#' characters. Multi-line comments can naturally have newline characters, but '#' character is not allowed in multi-line comments in order not to confuse single-line comments with multi-line comments. Unlike some other languages, our comment specifier character '#' cannot be used elsewhere in the code. This greatly helps to distinguish comments from codes and increases the readability. Note that comments are ignored in lexical analyzer and not passed to the parser.

Literals: We have four types of literals, integer, char, bool, and string. Integers are merely numbers. String literals are specified with double quotes ("") and character literals with single quotes (''). Note that string and char literals may have any characters inside them besides the newline character because writing string literals in multiple lines would make the code less clear and readable. Bool literals consist of two values: True and False. Instead of representing boolean variables with integer literals, using bool literals increases the readability of the code.

Identifiers: In our language, identifiers consist of digits, upper and lower case characters, underscore (_) and the dollar sign ($). Identifiers can't start with digits so that it is easier to distinguish them from integer literals. Enabling underscores increases both writability and readability in a way that they help separating words from each other.

Reserved words: As described in the previous part, we have reserved words for many different purposes, such as variable types, loops or conditional statements. They are merely strings which consist of only letters. Because reserved words are frequently used, we didn't use any digits, underscore or any other characters in their definitions, so that they can be written easily. Most of them consist of only lower case letters which are even easier to write then upper case letters. However, we have some upper case reserved words (AND, OR, NOT) because their lower case form was unreadable. So we decreased their writability a little bit. But in return, we gained much more readable reserved words. In the lexical analysis file, we defined our reserved words before identifier definition, so that they are distinguished from identifiers.

V. Evaluation of Language

This language is designed for drone controlling. So, programmers will not be the only users of this language. Even a highschool student may own a drone these days. Of course there will be a console to help this student control the drone. But if this console doesn't encounter this curious young man's or lady's needs or expectations from this drone, Cloud language enters the stage. They can use our language to have a superior control over their drone. Our purpose is to ensure that they learn and use Cloud language with the minimum effort and pain. Our language must also be used by programmers to design the interior logic of a drone, which might not be as simple as controlling a designed robot with simple code patterns. For this reason, our language must also have some necessary functionalities. When designing our language, we tried to look at it from both a programmer's and user's perspective and find a midway between them. In our intense efforts, we considered all language criterias (Readability - Writability and Reliability). Among those three, the most important one was Readability for us...

Readability: It would be pretentious to say that our language is the most readable language in computer science history. However, we believe that it is definitely more readable than many other languages. First of all, it uses a different indicator for nearly every different construct. For example, the character '#' is only used in comments, which enables them to be easily distinguishable and readable. As you might be realized by now, we do not have curly brackets in our language. Using curly brackets in every scope definition would confuse coder's mind, especially if they are beginners like our young student example. Instead, we use different keywords for different types of scopes. We end a for loop with "endfor", while loop with "endwhile" and if statement with "endif". In that way, we guarantee that the readers of a code written in our language are not drawn in the curly brackets of multiple nested loops and if statements, and also it is easy to remember that "endif" is used at the end of an if statement. Secondly, we don't allow some complicated and hard to read code blocks. For example, we don't allow arithmetic expressions as control expressions of for, while and if statements. Because reading if (True) is much easier than reading if (1). We also don't allow more than three statements in the condition part of for loop, because it is already hard to read for beginners and it might even be strange for experienced coders to see five or six statements in condition part of a for loop.

Writability: Although it may seem from the previous part that we abandoned writability for a better readability, it is not the case. We only restricted some complicated and not very commonly used structures to standardize our language. However, we are still allowing many useful and easy to write styles. For example, we allow multiple variable declarations in the same statement. Otherwise, it would be quite boring to write "int" before each variable name. It is also readable, you only need to read the part between commas. We also have some powerful assignment operators; +=, -=, *=, /=, %=. Thanks to them, you don't need to write the counter word twice in "counter = counter + 1". Well, it might be argued that these operators decrease readability of assignments. However, their contribution to the writability is so powerful that we believe they can't be abandoned. Note that we don't use "counter++;", so it is still more readable than many other languages. Furthermore, removing curly brackets from our language made it also more writable. Writing just a single "endif" to the end of our if statement is highly easier than putting two curly brackets for "if" and two more curly brackets for each use of the "elif" word.

Reliability : For reliability, during the compilation process our program will detect some type mismatches, these mismatches are listed in the below. Assignments only can be done within the same variable types. Also, relational operators can be used only within integers and characters. Also, equality and inequality operators should compare the same types. For the if, while and for statements, the results of the control expressions should be boolean value. Besides that, in our Cloud program, we do not allow any direct memory accessing or any other kinds of low level stuff, that’s why this language does not have any problem with aliasing. Not using the aliasing, also, increases the reliability since memory accessing might cause run time errors. Lastly, our language is designed as much as easy to both write and read. We discard the curly bracket usage during the blocks to make the block much understandable and ease the writing process. Therefore, if people can write this code as much as understandable they are less likely to make errors and they can maintain their programs. That's why our language is designed to be reliable as much as possible.

VI. Conflicts

Our parser has no conflicts.