CS 315

Programming Languages

Project Part 2

SL++

(Speaking Language++)

Group No: 36

Ege Özcan 21602150 Section 01

Mustafa Bayraktar 21604079 Section 01

Language Name = SL++ (Speaking Language++)

**BNF DESCRIPTION OF SL++**

<program> ::= <statements> END\_SIGN

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

<statement> ::= <non\_if\_statement> | <matched> | <unmatched>

<non\_if\_statement> ::= <assignment\_statement>

| <while\_statement >

| <constant\_statement>

| <function\_call\_statement>

| <function\_define\_statement>

| <read\_statement>

| <write\_statement>

<matched> ::=  IF <logic\_expression> COMMA <statements> ELSE <statements> EXCLAMATION\_MARK

<unmatched> ::= IF  <logic\_expression> COMMA <statements> EXCLAMATION\_MARK

<assignment\_statement> ::= IDENTIFIER IS <expression>

| IDENTIFIER IS <function\_call\_statement>

| IDENTIFIER IS <read\_statement>

| IDENTIFIER IS <logic\_expression>

<basic\_logic> ::=  <factor> <comparator> <factor> | BOOLEAN

<logic\_expression> ::= <logic\_expresion> <logic\_operator> <basic\_logic> | <basic\_logic>

<while\_statement> ::= WHILE  <logic\_expression> COMMA <statements> EXCLAMATION\_MARK

<constant\_statement> ::= CONSTANT CONST\_IDENTIFIER IS <factor>

<function\_call\_statement> ::= CALL FUNC\_IDENTIFIER WITH parameters

| CALL FUNC\_IDENTIFIER

| CALL MOVE WITH FLOAT

| CALL TURN WITH FLOAT

| CALL GRAB

| CALL RELEASE

| CALL READ\_DATA WITH INTEGER

| CALL SEND WITH INTEGER

| CALL RECEIVE

<function\_define\_statement> ::= NEW\_FUNCTION\_IS FUNC\_IDENTIFIER WITH <parameters> COLON <statements> RETURN <return\_type> EXCLAMATION\_MARK

| NEW\_FUNCTION\_IS FUNC\_IDENTIFIER COLON <statements> RETURN <return\_type> EXCLAMATION\_MARK

<read\_statement> ::= READ

<write\_statement> ::= WRITE <return\_type>

<return\_type> ::= <factor>

|<logic\_expression>

| <expression>

| NOTHING

<comparator> ::= GREATER | LESS\_THAN | EQUAL\_TO | GREAT\_EQ | LESS\_EQ

<expression> ::= <expression> PLUS <term> |  <expression> MINUS <term> | <term>

<term> ::=  <term> TIMES  <factor> | <term> DIVIDED\_BY <factor>  | <factor>

<factor> ::= IDENTIFIER | INTEGER | FLOAT

<parameters> ::= <factor> | <factor> COMMA <parameters>

<logic\_operator> ::= AND | OR | EQUALS

**Description**

Speaking Language++ aims to replicate the natural language that humans use to communicate. By making our code similar to natural language we aim to make it easy to understand. As a result, the user spends less time decrypting the language and more time decrypting the algorithm during debugging. Also new users can easily understand code written in this language even though they don’t have any experience with it. Our naming conventions were based on the education we were given in our first year; variables have all lower-case notation while constants have all uppercase notation. Functions have a camel notation. We converted the reserved signs used by many languages into reserved words (e.g ‘=’ became ‘is’). Every statement in our program ends with a dot, just like how every sentence ends with a dot. Our loops end with an exclamation mark, we chose exclamation marks to show actions just like in natural languages. Loops end with exclamation marks but the loops are still statements so we also need a full stop at the end of a loop statement, this results in if and while statements ending with “!.”. Our language allows the user to define new functions which makes our language extendible. Functions don’t need to have parameters and they also don’t have any limit imposed by the language (a compiler might limit the amount of parameters). This makes our language flexible. Our language also has read and write statements which allows user interaction with programs that allows users to make use of the terminal. Our language uses the pound ‘#’ sign to show the end of the program. It is easy to distinguish from other signs and we used it as the greatest full stop, the full stop that shows the end of the program. Our efforts have been mostly around increasing readability of our language. While doing so we also tried to improve reliability and writability as much as we could.

**Explanation of BNF**

**<program> is made up of one or more statements. These statements can be of any type such as assignment statements, while statements and if statements(matched or unmatched). Every statement ends with a dot. Program ends with “#”.**

number is 8.

number2 is 9.

sum is 8 plus 9.

while number3 lessThan 14,

if sum equalTo 8,

        number3 is 13.

sum is 8.

else

        number3 is 15.!.

!.

number 7 is 8.8.

#

**<statements> have three different types. They can be a non-if statement, a matched if statement or an unmatched if statement.  The non-if statement contains many general use statements. The matched if statement is the if statement with an else statement and the unmatched if statement is the if statement without an else statement.**

if red equalTo 13,

    number1 is 9.!

blue is 7.12.

if red equalTo 15,

    blue is red.!

else

    blue is 8.!

**<statement> can show many different statements. It can be an assignment statement where a variable is given a value. It can be a while statement which is used for looping. It can be a constant statement which is used for setting a constant. It can be a function call statement which is used for calling predefined functions and also the functions defined by the function define statement. A statement can also correspond to read and write statements.**

number is 2.

newFunctionIs Calculate with number1, number2: number3 is number1 plus number2. return number3!

**<matched> corresponds to an if statement with an else statement. We can have nested if-else statements as a matched if statement. A matched statement can also correspond to a non if statement so that we can have if statements that are not nested.**

if number1 equalTo 9,

    write number1.

else

    write number2. ps. The dot shows the end of write statement

! ps. The exclamation mark shows the end of loop

. ps. The dot shows the end of the matched if statement

**<unmatched> corresponds to if statements without an else statement. This allows for nested if statements without else statements.**

if number1 equalTo 9,

    write number1.!.

**<assignment\_statement> is where an identifier/variable is given a value. This value can be the result of 3 different things: an expression, a function call or user input.**

count is 10.

count is call Calculate with 1, 5.

**<basic\_logic> is the where number comparisons are made. Basic logic also contains the boolean values True and False.**

number1 equalTo number2

**<logic\_expression> is where logic comparisons are made. Comparisons between boolean types are made here. A logic expression can consist of many different logic and number comparisons. In order to prevent ambiguity we used left recursion.**

True equals number1 equalTo number2.

number1 equalTo number2 equals True.

**<while\_statement> is the statement used for executing certain commands in a loop as long as the given criteria(logic expression) is satisfied.**

while count lessThan 14,

    count is count plus count.

!.

**<constant\_statement> is used for declaring constants that can’t be changed once they are set.**

Constant LIMIT is 40.

Constant PI is 3.14 .

**<function\_call\_statement> is used for calling functions. These functions can either be predefined like the move function or they can be functions that have been defined using the function define statement. Functions can be called with zero or more parameters and can return a single value or void.**

call Move with 15.0.

call Grab.

Call myFunction with 5, 6.

**<function\_define\_statement> is used for defining functions. Functions can have zero or more parameters and can return a single value or void.**

newFunctionIs Decrease with number1, number2:

number1 is number1 minus number2.

!.

newFuctionIs Summation with number1, number2:

sum is number1 plus number2.

return num

!.

**<read\_statement> is used for getting user input.**

A is Read.

**<write\_statement> is used for displaying things.**

B is 10.

Write B.

**<return\_type> contains different constructs that can be returned by functions.**

newFuctionIs Summation with number1, number2:

sum is number1 plus number2.

**return num**!.

**<comparator> contains number comparison operators.**

a **lessThan** 3.

b **greaterThan** 5.

**<expression> contains addition and subtraction operations. It is left recursive**

b is **a plus c.**

c is **a minus a.**

**<term> contains multiplication and division operations. It is left recursive**

a is **a times c.**

b is **a dividedBy 6.**

**<factor> contains anything that can have a numerical value**

a is **5.**

b is **6.5.**

a is **b.**

**<parameters> can contain one or multiple factors separated by a comma. Parameters are the arguments that are used when defining and calling functions**

Call Move with **15.**

**<logic\_operator> consists of and, or, equals operations. Logic operators are used for comparing boolean values. The equals logic operator should only be used to compare boolean expressions and not numerical values.**

a is True.

b is 7.

c is 7.

result is True **equals** a **equals** b equalTo c.

**Explanation of Nontrivial Tokens**

**IS**

Our main goal while designing this language was to obtain a language that is written in a similar manner to how we talk. So we use a notation like “it is apple” when assigning values to variables. We think that it makes the language much more readable even though it means sacrificing writability.

When compared to other languages our IS token is similar to “=”.  
  
**BOOLEAN**

We use Booleans for logic operations and their common language counterparts are True and False therefore we use them.

Other languages have similar approaches when defining Boolean values. It is very easy to understand this notation therefore both readability and writability is ideal. It is also very straightforward and therefore reliable.

**WHILE / RETURN / IF / ELSE**

These are used for conditional and loop statements. They are very similar to other programming languages and they exist in our daily language, therefore these statements are suitable for Speaking Language++ (C) .These provide a perfect balance between readability, reliability and writability.

**IDENTIFIER/CONST\_IDENTIFIER/FUNC\_IDENTIFIER**

We decided that identifiers should be all lowercase and constants should be all uppercase so that it is easy to distinguish between them.

Function identifiers must start with an uppercase letter and continue with lowercase letters or an underscore.

We followed the naming conventions that we learned while deciding on identifier naming schemes. In CS101 we were advised to name our constants in uppercase and our variables in lowercase. We were also advised to name functions in a way that resembled mini sentences that explained what the function did, we were taught to separate the elements of these mini sentences with underscores(e.g Do\_This)

No identifier can begin with a number to prevent misunderstandings. But identifiers can have numbers in other places so that we don’t limit the user in naming identifiers.

This naming convention needs some getting used to but we believe it makes things much easier in the long run.

**COMMENT**

We decided on a radical way of declaring comments. Postscript is an additional remark at the end of a letter, after the signature and introduced by ‘PS’. Any line that starts with ps is treated as a comment in our language.

It is similar to other programming languages in the way it functions, so essentially what we do is change the “//” in c++ with “ps”

**NEW\_FUNCTION\_IS / CALL**

Instead of using robotic ways to define a function we decided to use a sentence structure. To declare a new function the user needs to write code that resembles a natural sentence (e.g newFunction is Calculate\_Area with radius: …) The overall form of defining a new function is similar to Python (e.g def Calculate\_Area(radius)) unlike other languages we don’t use parentheses and we have parameters as a part of the sentence. We believe that this approach makes our language easier to read as it becomes closer to a natural language.

We call functions by using the call token followed by the name of the function. In other languages functions are usually called by just writing their name. We believe that our approach increases readability.

**GREATER/ LESS\_THAN/EQUAL\_TO/GREAT\_EQ/LESS\_EQ**

These are comparators. They are used for comparing numerical values. Rather than using symbols to denote these comparators like other languages we used reserved words. We are aware of the writability problems this brings up, however in order remain consistent in our language design approach we used sentences instead of signs. We believe this slightly increases readability too.

**PLUS/MINUS/TIMES/DIVIDED\_BY**

These are arithmetic operators. They are used for doing operations with numerical values. Rather than using symbols to denote these operators like other languages we used reserved words. It increases readability. Also by using plus and minus rather than signs, the user is able to distinguish between arithmetic operations and the positivity/negativity of numbers. (5 plus -2) This improves reliability.

**AND/OR/EQUALS**

These are logic operators. They are used for doing operations with logical values. Rather than using symbols to denote these operators like other languages we used reserved words. This approach not only increases the readability but also increases the writability of our program. Compared to C++ the user does not have to remember signs like && and || to make logical operations. OR and AND are simple and short words that are very easy to remember.

**NOTE: There is no conflict left.**