CS315: Programming Languages

PROJECT - 1

A Boolean Language: RUSSELL

Group - 15
Anıl İlağa - 22002044
Zeynep Begüm Kara - 22003880

Section - 1

March 11, 2023

A. Language Design of RUSSELL

a. The Complete BNF Description

Constants

<LC> ::= “{”

<RC> ::= “}”

<LP> ::= “(”

<RP> ::= “)”

<LS> ::= “[”

<RS> ::= “]”

<new_line> ::= “\n”

<semicolon> ::= “;”

<comma> ::= “,”

<quote> ::= “\””

<boolean> ::= true | false

<letter> ::= [a-zA-Z]

<digit> ::= [0-9]

<char> ::= [^\”\\\/]

<assign> ::= “=”

<and> ::= “&&”

<or> ::= “||”

<not> ::= “!”

<eql> ::= “==”

<neq> ::= “!=”

<imply> ::= “=>”

<iff> ::= “⇔”

<dec_sym> :: “$bool”

<func_> ::= F:

<array_> ::= A:

<while> ::= while

<for_each> ::= for_each

<for_each_sym> ::= “@”

<return> ::= return

<break> ::= break

<in_> ::= :in:

<out_> ::= :out:

<io_sym> ::= “::”

<comment_sym> ::= “--\””

Program

<program> ::= <LC><RC> | <LC><stmnt_list><RC | <func_list><LC><RC>
                    | <func_list><LC><stmnt_list><RC>

<stmnt_list> ::= <stmnt_list><stmt> | <stmnt>

<func_list> ::= <func_list> <func> | <func>

Identifiers and Variables

<identifier> ::= <letter> | <identifier><letter>

<func_identifier> ::= <func_><identifier>

<array_identifier> ::= <array_><identifier>

<identifier_list> ::= <identifier>, <identifier_list> | <identifier>

<boolean> ::= <true> | <false>

Operators

<logic_opr> ::= <not> | <binary_opr>

<binary_opr> ::= <and> | <or> | <imply> | <iff>

Expressions

<logic_expr> ::= <compound_expr> | <atomic_expr>

<atomic_expr> ::= <identifier> | <array_element> | <boolean>

<compound_expr> ::= <LP><logic_expr><binary_opr><logic_expr><RP>
                              | <not><logic_expr >

Statements

<stmnt> ::= <matched_stmnt> | <unmatched_stmnt>

Conditional Statements

<unmatched_stmnt>::= <if><LP><logic_expr><RP><LC><stmnt><RC>
                    | <if><LP><logic_expr><RP><LC><matched_stmnt><RC>
                    <else> <unmatched_stmnt>

<matched_stmnt> ::= <if><LP><logic_expr><RP><matched_stmnt>
                              <else><matched_stmnt>

| <loop_stmnt>

| <assign_stmnt>

| <io_stmnt>

| <decl_stmnt>

| <function_call>

| <comment>

| <return_stmnt>

Declaration Statements

<decl_stmnt> ::= <var_decl> | <arr_decl> | <funct_decl>

<var_decl> ::= <dec_sym><identifier><semicolon>

| <dec_sym> <identifier_list><semicolon>

| <dec_sym>><identifier> = <boolean><semicolon>

<arr_decl> ::= <array_identifier> = <numeric><semicolon>

| <array_identifier> = <numeric><LC><logic_expr_list> <RC> ;

<numeric> ::= <numeric><digit> | <digit>

<funct_decl> ::= <function_signature><LC><stmnt_list> <RC>

<function_signature> ::=<function_identifier> <LP><parameter_list><RP>

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

<parameter>::= <identifier> | <array_identifier>

Assignment Statements

<assign_stmnt> ::= <var_assign_stmnt> | <arr_assign_stmnt>

<var_assign_stmnt> ::= <identifier> = <logic_expr><semicolon>

<arr_assign_stmnt> ::= <array_element> = <logic_expr><semicolon>

| <LC><identifier_list><RC><semicolon>

Loop Statements

<loop_stmnt> ::= <while_stmnt> | <for_each_stmnt>

<while_stmnt> ::= <while><LP><logic_expr><RP> <LC> <stmnt_list><RC>

<for_each_stmnt> ::= <for_each><LP><identifier><for_each_sym>

<array_identifier><RP> <LC> <stmnt_list><RC>

Input / Output Statements

```
<io_stmnt> ::= <in_stmnt> | <out_stmnt>
<in_stmnt> ::= <identifier><in_><semicolon>
<out_stmnt> ::= <out_><io_expr><semicolon>
                |<variable_out__stmnt>
<variable_out__stmnt> ::= <identifier><out_><semicolon>
<io_expr> ::= <io>  | <io> <io_sym> <io_expr>
<io> :: = <string_literal> | <logic_expr>
<string_literal> ::= "<str>"
<str> ::= <str><char> | <char>
```

Return Statement

```
<return_stmnt> ::= <return> <logic_expr><semicolon>
```

Function Calls

```
<func_call> ::= <function_signature><semicolon>
```

Data Structures

```
<array_assign> ::= <array_identifier> =
<numeric><LC><logic_expr_list><RC><semicolon>
<logic_expr_list> ::=  <logic_expr_terms>
                     | <logic_expr_terms> , <fill_with> , <logic_expr_terms>
                     |<fill_with> , <logic_expr_terms>
                     | <logic_expr_terms> , <fill_with>
                     | <fill_with>
<fill_with> ::= …<boolean>
<logic_expr_terms> ::= <logic_expr> | <logic_expr>, <logic_expr_terms>
<array_element> ::= <array_identifier><LS><numeric><RS>
```

Comments

```
<comment> ::= <comment_sym><str><comment_sym>
```

b. List of Features and Explanations

Non Trivial Constants

```
<dec_sym> ::= $bool
```

Defined to indicate declaration of variables. Although only boolean type variables exist in the language , for readability purposes we have chosen the symbol $bool.

```
<func_> ::= F:
```

Defined to distinguish function identifiers, syntax has been chosen similar to common mathematical notation of functions.

```
<array_> ::= A:
```

Defined to distinguish array identifiers, syntax has been chosen for consistency with <func_>.

```
<for_each_sym> ::= @
```

Defined to indicate iterating element in for_each loop. *for_each (x @ A:arr)* means for each element at A:arr. Such syntax has been chosen to distinguish identifiers and array_identifiers.

```
<in_> ::= :in:
```

Defined to get input stream. Such syntax has been chosen for concatenation, and shortcuts. See i/o statements for further explanation.

```
<out_> ::= :out:
```

Defined to display output stream. Such syntax has been chosen for concatenation, and shortcuts. See i/o statements for further explanation.

```
<io_sym> ::= "::"
```

Defined to concatenate input and output streams, syntax has been chosen for consistency with <in_> and <out_>.

```
<comment_sym> ::= "--"
```

Defined to indicate beginning and end of comment statements, a simple syntax has been chosen for increasing writing ability.

Program

```
<program> ::= <LC><RC> | <LC><stmnt_list><RC>
                       | <func_list><LC><RC>
                       | <func_list><LC><stmnt_list><RC>
```

Start variable of this grammar is the non-terminal <program>. Statements to be executed must be declared inside curly braces. User defined functions, if any, must be defined before this structure.

```
<stmnt_list> ::= <stmnt_list><stmt> | <stmnt>
```

At least one <stmnt> defines a <stmnt_list>. <stmnt_list> cannot be empty. Observe that this is a left recursive definition. As a design choice, if both left and right recursion is possible, we will use left recursive grammars for consistency.

```
<func_list> ::= <func_list><func> | <func>
```

At least one <func> defines a <func_list>.

Identifiers and Variables

```
<identifier> ::= <letter> | <identifier><letter>
```

In this grammar, <identifier> is the name of a boolean variable, or simply a variable since there only exist boolean type variables. Identifiers can only contain letters and they cannot be empty. Even though convention in first order logic is to use a single uppercase letter to represent atomic propositions, this language allows users to have lowercase letters and identifier names longer than single letter. As a result, one can define as many as necessary propositions and give them meaningful names, which increases expressivity and readability where overall simplicity is decreasing.

```
<func_identifier> ::= <func_><identifier>
```

<func_identifier> is the name of a function. Function identifiers always begin with <func_>, i. e. "F:" and followed by an identifier. Having a restriction in function identifiers allows us to distinguish function identifiers from variable identifiers or array identifiers. Even though this decreases the writability and simplicity, it increases reliability by distinguishing those structures.

```
<array_identifier> ::= <array_><identifier>
```

<array_identifier> is the name of a data structure, array. Array identifiers always begin with <array_>, i. e. "A:" and followed by an identifier. Having a restriction in array identifiers allows us to distinguish array identifiers from variable identifiers or function identifiers. Even though this decreases the writability and simplicity, it increases reliability by distinguishing those structures.

```
<identifier_list> ::=  <identifier><comma> <identifier_list>
                    |  <identifier>
```

One <identifier> or consecutive <identifier>s separated by a comma defines a <identifier_list>. <identifier_list> cannot be empty.

```
<boolean> ::= <true> | <false>
```

Boolean variables, which is the only type in RUSSELL language, has only two values: true or false.

Operators

```
<logic_opr> ::= <not> | <binary_opr>
```

Operators can be classified as *unary operators*, which is applicable to a single proposition, and *binary operators,* which requires two propositions as operands. Since the language RUSSELL deals with first order logic, it has only one unary operator, which is operator <not>.

```
 <binary_opr> ::= <and> | <or> | <imply> | <iff>
```

This language has binary operations *and, or, implication* and *if and only if.* Recall that binary operations <and>, <or> and <iff> are *fully associative* whereas operator <imply> is *right associative*. However, since compound propositions are defined with parentheses in our language, we will not worry about this distinction.

Expressions

```
<logic_expr> ::= <compound_expr> | <atomic_expr>
```

Logical expressions can be classified as *compound expressions* and *atomic expressions.* In first order logic, this distinction respectively corresponds to prepositions with <logic_opr> and prepositions with no <logic_opr>.

```
<atomic_expr> ::= <identifier> | <array_element> | <boolean>
```

An atomic expression, which represents prepositions with no connectives, is either an identifier, element of an array or a truth value. In both cases they do not include any <logic_opr>.

```
<compound_expr> ::= <LP><logic_expr><binary_opr><logic_expr><RP>
                  | <not><logic_expr>
```

A compound expression is either a preposition with <binary_opr> that connects two logical expressions and such expressions are bounded by left, (, and right,), parentheses, or it is a negation of a proposition, whose truth value is opposite of the previous one. Observe that operation <not> does not require parentheses, which increases readability and writability.

Statements

```
<stmnt> ::= <matched_stmnt> | <unmatched_stmnt>
```

Statements are either matched or unmatched statements.

Conditional Statements

```
<unmatched_stmnt>::= <if><LP><logic_expr><RP><LC><stmnt><RC>
                   |<if><LP><logic_expr><RP><LC><matched_stmnt><RC>
                    <else><LC><unmatched_stmnt><RC>
```

An unmatched statement is either an if with no else statement followed or if statement followed by an else statement that has an unmatched statement inside.

```
<matched_stmnt> ::=  <if><LP><logic_expr><RP><LC><matched_stmnt><RC>
                     <else><LC><matched_stmnt><RC>
                   | <loop_stmnt>
                   | <assign_stmnt>
                   | <io_stmnt>
                   | <decl_stmnt>
                   | <function_call>
                   | <comment>
                   | <return_stmnt>
                   | <break_stmnt>
```

A statement is matched if it has an equal number of *if* and *else* keywords. Observe that all statements except conditional statements are matched since it has an equal number of *if* and *else*'s, which is zero.

Declaration Statements

```
<decl_stmnt> ::= <var_decl> | <arr_decl> | <funct_decl>
```

Declaration statements can be variable declarations, array declarations, or function declarations.

```
<var_decl> ::= <dec_sym><identifier><semicolon>
             | <dec_sym><identifier_list><semicolon>
             | <dec_sym><identifier> = <boolean><semicolon>
```

In our language variables can be declared by just passing a declaration symbol $bool (since we don’t have any other type) and identifier. Also, you can initialize the variable in the declaration statement.

```
<arr_decl> ::= <array_identifier> = <numeric><semicolon>
```

An array is declared by passing the <array_identifier> (A:<identifier>), assignment operator, and a numeric value that will be the size of the array.

```
<numeric> ::= <numeric><digit> | <digit>
```

Numeric values consist of at least one digit and it is defined with left recursion.

```
<funct_decl> ::= <function_signature><LC><stmnt_list><RC>
```

A function is declared by a signature followed by statements which are inside curly braces.

```
<function_signature> ::= <function_identifier><LP><parameter_list><RP>
```

Function signature contains the identifier “F: functionName” and the parameters for that function inside braces.

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

A parameter list is the plural form of a parameter followed by commas.

```
<parameter>::= <identifier> | <array_identifier>
```

A function parameter can be a boolean or an array. On the contrary with C like languages, The language does not allow passing a function as parameters for reliability concerns.

Assignment Statements

```
<assign_stmnt> ::= <var_assign_stmnt> | <arr_assign_stmnt>
```

Two types of assignment statements are variable assignment (boolean) and array assignment.

```
<var_assign_stmnt> ::= <identifier> = <logic_expr><semicolon>
```

Variables can be assigned by the variable name followed by the expression to be assigned.

```
<arr_assign_stmnt> ::= <array_element> = <logic_expr><semicolon>
```

An element of an array can be changed by passing the array identifier and the expression.

Loop Statements

```
<loop_stmnt> ::= <while_stmnt> | <for_each_stmnt>
```

The RUSSEL language has two loop statements that are while loop and for each loop.

```
<while_stmnt> ::= <while><LP><logic_expr><RP><LC><stmnt_list><RC>
```

Statement type for while loops are similar to traditional while loops in commonly used languages to increase readability and writability purposes.

```
<for_each_stmnt> ::= <for_each><LP><identifier><for_each_sym>
                     <array_identifier><RP><LC><stmnt_list><RC>
```

The language has a for loop statement to go through each element of an array. Again, syntax has chosen to be intuitively.

```
<break_stmnt> ::= <break><semicolon>
```

Break statement to get out of the loop. This statement can be used to avoid infinite loops.

Input/Output Statements

```
<io_stmnt> ::= <in_stmnt> | <out_stmnt>
```

An I/O statement is either an input statement or an output statement.

```
<in_stmnt> ::= <identifier><in_><semicolon>
```

As a design choice, this language only accepts only boolean type inputs. An input statement assigns user input to the identifier before the input indicator :in:. Unlike some languages, it is not allowed to get an input and assign it after a variable such as using a scanner object. Even though it might decrease writability, we believe that it is a good practice to write reliable programs.

```
<out_stmnt> ::= <out_><io_expr><semicolon>|<variable_out_stmnt>
```

Output statements either begin with a constant ":out:" and might contain I/O expressions followed by a semicolon or it is a shortcut to display variables.

```
<variable_out_stmnt> ::= <identifier><out_><semicolon>
```

This statement is a shortcut to commonly used notation in displaying variables. It is called by an identifier :out: followed by a semicolon and it will print out the identifier and its value. To be more precise:

P :out: ;   prints   p = true

```
<io_expr> ::= <io>  | <io> <io_sym> <io_expr>
```

These expressions can contain just one expression or expressions followed by io constant "::". This concatenation property helps programmers to write output statements efficiently.

```
<io> :: = <string_literal> | <logic_expr>
```

Either strings or expressions can be used in io expressions.

```
<string_literal> ::= "<str>"
```

String literals are strings surrounded by quotation marks.

```
<str> ::= <str><char> | <char>
```

Strings are chars following each other. We have distinguished strings and string literals to avoid nested quotation marks in recursive definition.

Return Statement

```
<return_stmnt> ::= <return><logic_expr><semicolon>
```

Return statements can be used to return logic expressions.

Function Calls

```
<func_call> ::= <function_signature><semicolon>
```

Parameters are passed in the function signature to call a function.

Data Structures

```
  <array_assign> ::= <array_identifier><assign><numeric><LC>
                     <logic_expr_list> <RC> <semicolon>
```

Assigning values inside an array by putting all the data inside curly braces.

```
  <logic_expr_list> ::=  <logic_expr_terms>
         | <logic_expr_terms>,  …<boolean>, <logic_expr_terms>
         | …<boolean>, <logic_expr_terms>
         | <logic_expr_terms>, …<boolean>
         | …<boolean>
```

A logical expression list may or may not contain a "…<boolean>" structure which means to assign the same value through the end of the array if no other value passed after or until the last few elements.

```
<logic_expr_terms> ::= <logic_expr> | <logic_expr>, logic_expr_terms>
```

Logical expression terms are one expression or expressions separated by commas.

```
 <array_element> ::= <array_identifier><LS><numeric><RS>
```

Accessing a specific element of an array is done by passing the numeric value inside square brackets.

Comments

```
 <comment> ::= <comment_sym><str><comment_sym>
```

Comments begin and end with a comment symbol which allows line comments or comment blocks.

B. Language Evaluation

RUSSELL programming language is designed to be used in applications of first-order logic. It is named after the memory of famous logician, mathematician and philosopher *Bertrand Russell* (1872 - 1970).

The crucial aspect of the language is that it only contains boolean types, true and false. Therefore, this language is designed to be very compact, yet quite ***expressive*** in terms of propositions. Note that every variable in this language can be considered as a proposition, whose truth value is either true or false. Recall that in first-order logic as well as being *atomic* statements, propositions might be *compound*, i.e. they might be combinations of atomic statements connected by logical operations. RUSSELL also allows these propositions and for ***readability*** purposes, such propositions are bounded by left, (, and right,), parentheses. Furthermore, parentheses are necessary to indicate operator precedence so that ***an unambiguous grammar*** is formed.

Even though we know that the each of the following logical operation sets {and, not}, {or, not}, {implication, not}, {nand (or as logicians called it "the Sheffer stroke")}, and {nor} is *expressively complete*, meaning that every compound proposition can be expressed by only using logical operations in one of these small sets, we have commonly used logical operations *and, or, iff,* and *imply* in RUSSELL for ***writability*** purposes. The purpose of our design choice is to avoid complex and long expressions. However, we do not include every operator such nand or nor in order not to decrease ***readability*** too much. Also, even though negation of a proposition forms a compound statement, our definition does not require parentheses in such statements to increase simplicity.

Although we know that every variable is a boolean, to distinguish declaration and assignment statements RUSSELL still requires the defined keyword *$bool* in declaration statements because of ***reliability*** concerns. Distinguishing declaration statements from assignment statements, RUSSELL checks if the variable is previously declared to protect misleading declarations in assignment. This design choice prevents users from bugs resulting from basic typos by throwing a syntax error. Additionally, the language doesn't have the *aliasing* feature, which may cause hard to detect bugs, as a design choice to have more reliable programs.

Control flow of the program can be changed using conditional statements, break statements in loops, while loop and for each loop. The syntax of the loops is close to some well-known languages. Therefore, it increases readability, writability and reliability. Ambiguity of *the dangling else problem* has been solved defining matched and unmatched terms. Also, for each loop gives the ability to the language to iterate over a collection of booleans, an array, easily. The language does not have functionality of *go to, jump* or *jal* statements to ease the traceability of the program and enhance reliability.

As a data structure, the language provides a one dimensional array whose elements are boolean type. Because first order logic is interested in truth values of set propositions and their relations, we might have a large dataset. To efficiently store and retrieve these propositions, this language has an array structure. Recall that retrieval operation is O(1) in arrays. Due to this structure we can perform complex and long logical operations easily. Recall that elements of arrays might be only logical expressions, neither any array or a function. One might argue that this choice directly forbids multidimensional arrays, and damages orthogonality; this design choice resulted from reliability concerns. Additionally, since the propositions which will be given to the program might be a long list, instead of writing every value one by one, the programmer can assign one interval in the array as specified value. To be more precise:

A:myArray = 10{true, true, false, false, …true}

which corresponds to

A:myArray = 10{true, true, false, false, true, true, true, true, true, true}

or an another example might be

B:yourArray = 13{false, true, false, true, …false, true, true, true}

which corresponds to

B:yourArray = 13{false, true, false, true, false, false, false, false, false, false, true, true, true}

this notation also includes

C:theirArray = 14{...true}

C:theirArray = 14{ true, true, true, true, true, true, true, true, true, true, true, true, true, true }

This feature of the language increases the writability of it.

C. Test Programs for RUSSELL

- **Program - 1**

```
{
        $bool x, y, z;
         x :in: ;
         y :in: ;
         z :in: ;

        while (true)
        {
                if ( ((x && y) && z) )
                {
                        :out: “Please enter at least one false value”;
                         x :in: ;
                         y :in: ;
                         z :in: ;
                }
                else
                {
                        break;
                }
        }

        :out: ((x => y) && !(x⇔z)) ;
 }
```

- **Program - 2**

```
F:foo( p, q, r)
{
        p :out: ;
        q :out: ;
        r :out: ;

        :out: “F:foo”;
        if ( r ) { return (p => ( q || r )); }
        else { return (q ⇔r); }
}

{
        A:list = 5{true, false, …false};
        $bool c;
        for_each ( a @ A:list )
        {
                for_each(b @ A:list)
                {
                        c = F:foo( a, b, false);
                        a :out: ;
                        b :out: ;
                        c :out: ;
                }
        }
}
```

- **Test - 1: Football Team: Array declaration and assignment**
  This test program creates an array with 11 elements each of which represents a football team player's health state (true: infected) (false: healthy)
  If the player is infected it means the player next to that (only the right one) is also infected.
  The changed array is the final health state of the team.

```
{
        A:myArray = 11;
        A:myArray = {true, false, ...true, false, false, true, false};
        $bool prev = true;
        for_each(i @ A:myArray){
                if(prev){
                        prev = i;
                        i = true;
                }
                else{
                        prev = i;
                }
        }
}
```

- **Test - 2 : Array manipulation and display statements**

```
{
        –"this is a comment"–
        A:myArray = 6{true,false, …true, false};
        A:myArray[2] = (A:myArray[0] && A:myArray[1]);
        A:myArray[5] = (A:myArray[3] || A:myArray[4]);

        $bool x = (A:myArray[2] ⇔ A:myArray[5]);
        :out: “Boolean at index 2 is” :: A:myArray[2];
        :out: “Boolean at index 5 is” :: A:myArray[5];
        :out: “The result is”;
        :out:  x;
}
```

- **Test - 3: Logical Operations, Compound Statements Function Calls and Loops**

```
F:changeArray(A:array){
        for_each(x @ A:array){
                x :in: ;
        }
}

F:thereExistTrue(A:array){
        for_each(x @ A:array ){
                if( x ){ return true; }
        }
        return false;
}

{
        $bool P;
        P :in: ;
        $bool Q;
        Q :in: ;
        $bool R = (P =>Q);

        Q = ( !(R ⇔ P) => ( false || (!Q && R ));
        Q : out: ;

        if (Q) {
                A:myArr = 5;
                F:changeArray(A:myArr);
                $bool anyTrue = F:thereExistTrue(A:myArr);
                anyTrue :out: ;
        }
}
```

D. Non - Trivial Tokens

integer	[0-9]+	: Token to represent array size.
lowerLetter	[a-z]	: Token to represent lowercase letters.
upperLetter	[A-Z]	: Token to represent uppercase letters.
identifier	({upperLetter}|{lowerLetter})*	: Token to represent identifiers which is a combination of lowercase and uppercase letters.
boolean	(true|false)	: Token to represent boolean values.
return	return	: Token to return variables in functions.
assign	"="	: Token to represent assignment operator.
and	"&&"	: Token to represent logical AND operator.
or	"||"	: Token to represent logical OR operator.
not	"!"	: Token to represent logical NOT operator.
eql	"=="	: Token to represent logical EQUALITY operator.
fillWith	"..."{boolean}	: Token to partially fill an array.
neq	"!="	: Token to represent logical NOT EQUAL operator.
imply	"=>"	: Token to represent logical IMPLIES operator.
iff	"<=>"	: Token to represent logical IFF operator.
dec_sym	$bool	: Token to represent variable declarations.
func_	F:	: Token to identify function identifiers
array_	A:	: Token to identify array identifiers
in_	":in:"	: Token to take an input.
out_	":out:"	: Token to print an output.
double_column	"::"	: Token to concatenate io expressions.
while	while	: Token to while loop.
break	break	: Token to break statement in loops.
for_each	for_each	: Token to for_each loop.
for_each_sym	"@"	: Token to represent elements to iterate in for_each loop.
if	if	: Token to indicate conditions in conditional statements.
else	else	: Token to indicate else in conditional statements.
comment_sym	"--"	: Token to identify comments.
logical_op	({and}|{or}|{not}|{eql}|{neq}|{imply}|{iff})	: Token to all logical operations.

Logical operation tokens have been chosen with respect to commonly used notations in computer science and logic. Reserved words are defined in a very similar way in commonly used languages for readability and writability purposes. For detailed explanations see ***B.***