CS 315 - Programming Languages

Project 1 Report

2022-2023 Spring

Programming Language: Veritas

Team 07 Members:
Selin Bahar Gündoğar 22001514, Section 01
İdil Atmaca 22002491, Section 01
Emre Karataş 22001641, Section 01

Instructor: H. Altay Güvenir
Teaching Assistant: Dilruba Sultan Haliloğlu

1.Complete BNF Description

1.1 Program Structure

<program> → <start> <stmt_list> <finish>

<stmt_list> → <stmt> <SC> | <stmt> <SC> <stmt_list>

<stmt> → <matched> | <unmatched>

1.2 If Statements

<matched> → if <left_paranthesis> <operation> <right_paranthesis> <left_braces> <matched> <right_braces> else <left_braces> <matched> <right_braces> | <non–if-statements>

<unmatched> → if <left_paranthesis> <operation> <right_paranthesis> <left_braces> <stmt> <right_braces> | if <left_paranthesis> <operation> <right_paranthesis> <left_braces> <matched> <right_braces> else <left_braces> <unmatched> <right_braces>

1.3 Non-If Statements

<non_if_statement> → <non_if_expression> | <non_if_expression> <SC> <non_if_statement>

<non_if_expression> → <assign_stmt>

|<declaration_stmt>

| <operation>

| <loop_stmt>

| <method_declare>

| <method_call>

| <comment>

<assign_stmt> → <identifier> <assign_op> <identifier>

| <identifier> <assign_op> <method_call>

| <identifier> <assign_op> <operation>

| <identifier> <assign_op> <type>

<declaration_stmt> → <declaration> | <declaration_assign> | <hash_array_declaration>

<declaration> → <tag> <identifier_list>

<declaration_assign>→ <tag> <assign_stmt> | const <tag> <assign_stmt>

<identifier_list> → <identifier> <SC> | <identifier> <comma> <identifier_list>

<identifier> → <letter> | <letter> <identifier_rest>

1.4 Operators

<operation> → <basic_operation> | <left_paranthesis> <operation> <right_paranthesis> | <not_operation> <basic_operation> | <not_operation> <left_paranthesis> <operation> <right_paranthesis> | <not_operation> <identifier>

<identifier_combinations> → <identifier> | <basic_operation> | <not_operation> <identifier> | <not_operation> <basic_operation>

<basic_operation> → <logical_operation> | <implication> | <double-implication> | <left_paranthesis> <basic_operation> <right_paranthesis>

<logical_operation> → <identifier_combinations> <logical_op>
<identifier_combinations>

<logical_op> → <or> | <and> | <equality_op> | <not_equal_op>

<implication> → <identifier_combinations> <implication_single_symbol>
<identifier_combinations>

<double-implication> → <identifier_combinations>
<implication_double_symbol><identifier_combinations>

1.5 Looping Statements

<loop_stmt> → <while_loop> | <for_loop>

<while_loop> → while <left_paranthesis> <operation> <right_paranthesis>
<left_braces> <stmt> <right_braces> | do <left_braces> <stmt> <right_braces> while
<left_paranthesis> <operation> <right_paranthesis> <SC>

<for_loop> → foreach <identifier> in <identifier> <left_braces> <stmt> <right_braces> |
<identifier> in <hash_array> <left_braces> <stmt> <right_braces>

1.6 Function Definitions

<method_declare> → <return_type> <identifier> <left_paranthesis> <parameter_list>
<right_paranthesis> <left_braces> <stmt> <return> <return_stmt> <right_braces>

<parameter_list> → <parameter> | <parameter> <comma> <parameter_list> | <empty>

<parameter> → <tag> <identifier>

<method_call> → <identifier> <left_paranthesis> <parameter_identifier> <right_paranthesis> <SC> | <identifier> <left_paranthesis> <empty> <right_paranthesis> <SC>

<parameter_identifier> → <identifier> | <identifier> <comma> <parameter_identifier>

1.7 Comments

<comment> → <line_comment> <comment_description> <line_comment>

1.8 Data Structure

<hash_array_declaration> →   HashArray <identifier> <equal_sign> <hash_array> | HashArray <identifier> <equal_sign> <method_call>

<hash_array> →<left_curly> <item_list> <right_curly>

<item_list> → <item> | <item> <comma> <item_list>

<item> → <identifier> | <boolean> | <empty>

1.9 Primitive Functions

<primitive_methods> → <scan_input> | <display> | <print_hash>|<get_name> | <get_function_name> | <add_to_hash> | <delete_all_false> | <delete_all_true> | <all_true_hash> | <all_false_hash> | <is_empty>

<scan_input> → scanInput <left_paranthesis> <io_str> <right_paranthesis>

<display> → display <left_paranthesis> <output> <right_paranthesis>

<output> → <io_str> | <identifier> | <boolean>

<print_hash>→ printHash <left_paranthesis> <identifier> <right_paranthesis> | printHash <left_paranthesis> <hash_array> <right_paranthesis>

<add_to_hash> → <identifier> <DOT> addHash <left_paranthesis> <identifier> <right_paranthesis> | <identifier> <DOT> addHash <left_paranthesis> <boolean> <right_paranthesis>

<delete_all_false> →<identifier> <DOT> deleteAllFalse <left_paranthesis> <right_paranthesis>

<delete_all_true> →<identifier> <DOT> deleteAllTrue <left_paranthesis> <right_paranthesis>

<all_true_hash> → <identifier> <DOT> allTrueHash <left_paranthesis> <right_paranthesis>

<all_false_hash> → <identifier> <DOT> allFalseHash <left_paranthesis> <right_paranthesis>

<is_empty> → <identifier> <DOT> isEmpty <left_paranthesis> <right_paranthesis>

1.10 Symbols and Constants

<start> → start

<finish> → finish

<SC> → ;

<comma> → ,

<assign_op> → =

<equality_op> → ==

<not_equal_op> → !!

<not_operation> → !

<and> → &&

<or> → ||

<left_braces> → [

<right_braces> →]

<left_curly> → {

<right_curly> → }

<left_paranthesis> → (

<righ_paranthesis> →)

<empty> →

<letter> → a - z | A-Z

<digit> → 0-9

<identifier_symbols> → _

<symbols> → ~ | ! | @ | # | $ | % | ^ | & | * | (|) | - | _ | = | + | [|] | { | } | \ | | | ; | : | ' | " | , |
. | / | ? | < | > |

<DOT> → .

<QUOTE> → “

<type> → <boolean> | <hash_array>

<tag>→ boolean | HashArray

<const> → <constant_true> | <constant_false>

<constant_true> → veritas

<constant_false> → falsus

<implication_single_symbol> → ->

<implication_double_symbol> → <->

<identifier_rest> → <identifier_char> | <identifier_char> <identifier_rest>

<identifier_char> → <letter> | <digit> | <identifier_symbols>

<io_str> → <QUOTE> <chars> <QUOTE>

<chars> → <char> | <char> <chars>

<char> → <letter> | <digit> | <symbols>

<return_type> → boolean | void | HashArray

<return> → return

<return_stmt> → <identifier> | <boolean> | <method_call> | <operation> | <hash_array>

<boolean> → true | false

<line_comment> → ##

<comment_description> → letter | digit | symbols

<NL> → \n

2. Explanation of BNF Description

2.1 Program Structure

<program> → <start> <stmt_list> <finish>

This non-terminal is the core of the program, which starts with a start statement and ends with a finish statement. Between them, there is a list of statements.

<stmt_list> → <stmt> <SC> | <stmt> <SC> <stmt_list>

This non-terminal indicates that statement lists consist of statements or statements with a statements list. Besides that, the <SC> (semicolon) is the indicator for the end of a statement, as shown above.

<stmt> → <matched> | <unmatched>

This non-terminal shows that statements are labeled as either matched or unmatched statements. Details of matched and unmatched statements are described in the coming section.

2.2 If Statements

<matched> → if <left_paranthesis> <operation> <right_paranthesis> <left_braces> <matched> <right_braces> else <left_braces> <matched> <right_braces> | <non–if-statements>

The non-terminal known as a "Matched statement" is utilized for if statements that have a corresponding else statement. This differentiation between matched and unmatched statements resolves the problem of uncertainty in if or if-else statements, where it may not always be evident which "if" and "else" pertains to. In the Veritas programming language, an if statement necessitates parentheses for the control expression and braces for the statements inside. Furthermore, a matched statement can include non-if statements such as assignments, loops, function definitions, and other statements that do not include "if".

<unmatched> → if <left_paranthesis> <operation> <right_paranthesis> <left_braces> <stmt> <right_braces> | if <left_paranthesis> <operation> <right_paranthesis> <left_braces> <matched> <right_braces> else <left_braces> <unmatched> <right_braces>

This particular non-terminal indicates that an unmatched statement contains a greater number of "if" statements than "else" statements. These statements either have an "if" without a corresponding "else" or consist of another unmatched statement within an "else" block. Unmatched statements, along with matched statements, play a crucial role in determining which “if” statement an else statement corresponds to.

2.3 Non-If Statements

<non_if_statement> → <non_if_expression> | <non_if_expression> <SC> <non_if_statement>

This non-terminal is used for statements that do not include if statement(s). These statements can consist of single or multiple non-if expressions, which are described below.

<non_if_expression> → <assign_stmt> | <declaration_stmt> | <operation>|<loop_stmt> | <method_declare> | <method_call> | <comment>

Non-terminals made from non-if expressions can consist of different parts of statements, such as assignments, declarations, loops, declarations of methods, operations, comments, and method calls. Every single non-if expression can be described as non-terminal in its main field. Details of every non-terminal are identified below.

<assign_stmt> → <identifier> <assign_op> <identifier> | <identifier> <assign_op> <method_call> | <identifier> <assign_op> <operation> | <identifier> <assign_op> <type>

This assignment non-terminal describes that identifiers may be assigned to each other, to a method call or an operation or to its type. Such examples are given:

```
a = b;
c = foo();
d = e && f;
g = true;
```

<declaration_stmt> → <declaration> | <declaration_assign> | <hash_array_declaration>

This non-terminal explains that declaration statements may just be declarations, declarations with the assignment operation, or a HashArray declaration.

<declaration> → <tag> <identifier_list>

This non-terminal identifies declarations with their tags and the list of identifier items. The tag can be a boolean or HashArray.

<declaration_assign>→ <tag> <assign_stmt> | const <tag> <assign_stmt>

This non-terminal expression describes the declaration together with the assignment to boolean type variables such as true or false. This declaration can be done in two different situations. Statement with their tag, or statement with their tag along with const prefix (reserved word) to emphasize the value is constant.

```
boolean a = true;
const boolean b = false;
```

<identifier_list> → <identifier> <SC> | <identifier> <comma> <identifier_list>

This non-terminal describes that multiple identifiers in Veritas language can be separated by commas and can be just in one line of declaration. Such example can be:

```
boolean a;
boolean d,e,f;
```

<identifier> → <letter> | <letter> <identifier_rest>

This non-terminal identifies the BNF structure of identifiers. Identifiers may consist of just one letter or a letter can be followed by a combination of letters, numbers, or identifier symbols. The first character of an identifier must start with a letter.

2.4 Operators

<operation> → <basic_operation> | <left_paranthesis> <operation> <right_paranthesis> | <not_operation> <basic_operation> | <not_operation> <left_paranthesis> <operation> <right_paranthesis> | <not_operation> <identifier>

This non-terminal describes the basic structure of operations in the Veritas language. Operations may be enclosed with many matching parentheses. Operations may just consist of a single basic operation such as and, or, single and double implication operations. Operations can also include “not” operation to change result operation from/to true/false. Such example is:

```
boolean a = true;

a = !a;
```

<identifier_combinations> → <identifier> | <basic_operation> | <not_operation> <identifier> | <not_operation> <basic_operation>

This non-terminal mentions identifier combinations, which is an umbrella term for describing operations between identifiers or an identifier itself. Basic operation includes logical operations as well as implication rules, as described below in detail.

<basic_operation> → <logical_operation> | <implication> | <double-implication> | <left_paranthesis> <basic_operation> <right_paranthesis>

This non-terminal expresses how logical operations are formed in the core structure. Such an approach gives the program flexibility to add parentheses between operations in case operations inside the main operation are working correctly. Such a statement can be:

```
boolean a,b,c,d;
boolean e,f;
a = true;
b = true;
c = true;
d = true;
e = false;
f = false;

boolean result = (((a&&b) -> (c||e))<->((d<->f)->a));
```

<logical_operation> → <identifier_combinations> <logical_op> <identifier_combinations>

This non-terminal describes that logical operation symbols can be used between identifier combinations, which is an umbrella term to give freedom to the user about adding an unlimited number of logical operations.

<logical_op> → <or> | <and> | <equality_op> | <not_equal_op>

This non-terminal explains logical operations in the Veritas language. These logical operations are similar to the ones in other imperative languages such as Java.

```
boolean a = true;
boolean b = true;
boolean r1,r2,r3,r4;
r1 = a || b;
r2 = a && b;
r3 = a == b;
r4 = a !! b;
```

<implication> → <identifier_combinations> <implication_single_symbol> <identifier_combinations>

This non-terminal describes a single implication rule in Veritas language. Single implication truth table as follows:

p	q	p -> q
true	true	true
true	false	false
false	true	true
false	false	true

Table 1: Truth Table of Single Implication [1]

Implication rules may be assigned with other logical operations in the Veritas language as long as rules are applied.

<double-implication> → <identifier_combinations> <implication_double_symbol><identifier_combinations>

This non-terminal describes a double implication rule in Veritas language. Double implication truth table as follows:

p	q	p <-> q
true	true	true
true	false	false
false	true	false
false	false	true

Table 2: Truth Table of Double Implication [1]

Implication rules may be assigned with other logical operations in the Veritas language as long as rules are applied.

2.5 Looping Statements

<loop_stmt> → <while_loop> | <for_loop>

This non-terminal explains loop statements in Veritas language. Veritas has two different types of loop statements, while and for loop. Loop statements can be used to iterate over the elements of data structure in Veritas, which is HashArray.

<while_loop> → while <left_paranthesis> <operation> <right_paranthesis> <left_braces> <stmt> <right_braces> | do <left_braces> <stmt> <right_braces> while <left_paranthesis> <operation> <right_paranthesis> <SC>

This non-terminal identifies while loop rules in Veritas language. Veritas language has two types of while loops, do-while loop and while loop. Do-while loop gives freedom to the user to complete statement(s) before checking the statement condition, similar to the applied rules in other imperative programming languages. Such while loop examples are:

```
## do-while example ##
HashArray arr  = { false, false, false , true };
Do [
      ## do some statements ##

] while (arr == true );
## while example ##
While ( arr !! true )
[
## do something... #
]
```

<for_loop> → foreach <identifier> in <identifier> <left_braces> <stmt> <right_braces> | <identifier> in <hash_array> <left_braces> <stmt> <right_braces>

This non-terminal describes a for-loop for Veritas language. Foreach is an enhanced looping statement that iterates over elements of the data structure from beginning to end. Such example code may be:

```
HashArray hashArr = {true, false, true, true, false, true};
foreach val in hashArr
[
      ## do something... ##
]
```

2.6 Function Definitions

<method_declare> → <return_type> <identifier> <left_paranthesis> <parameter_list> <right_paranthesis> <left_braces> <stmt> <return> <return_stmt> <right_braces>

This non-terminal describes how functions are declared in the Veritas language. Specifying the return statement. In the parentheses, parameter(s) are explained. In the braces ([ ]), needed function statement(s) are listed.

```
boolean getTruth (boolean a, boolean b)
[
     boolean c;

      ## statements are there
     c = (a&&b)->a;

      return c

]
```

<parameter_list> → <parameter> | <parameter> <comma> <parameter_list> | <empty>

This non-terminal describes the parameter list in the function definition. In Veritas language, the parameter(s) can be empty, single, or multiple. In multiple-parameter situations, they should be separated by comma(s).

<parameter> → <tag> <identifier>

This non-terminal expresses how parameters are formed in Veritas language. The parameter's type and identifier name should be given.

<method_call> → <identifier> <left_paranthesis> <parameter_identifier> <right_paranthesis> <SC> | <identifier> <left_paranthesis> <empty> <right_paranthesis> <SC>

This non-terminal describes how methods are called in the Veritas language. Method identifiers must be given. Between parenthesis, there can be an identifier'(s) list or no identifier at all. Examples are:

```
getTruth(a, b, c);
noElementFunc();
```

<parameter_identifier> → <identifier> | <identifier> <comma> <parameter_identifier>

This non-terminal mentions parameter identifiers listed in the function call. These parameter identifiers should be separated by a comma if they are multiple in the parameter definition.

2.7 Comments

<comment> → <line_comment> <comment_description> <line_comment>

This non-terminal describes how comments are formed in Veritas language. Beginning with a double dash line, a comment description is given by the program user. The comment should be completed with a double-dash line at the end to indicate the end of the comment.

2.8 Data Structure

<hash_array_declaration> → HashArray <identifier> <equal_sign> <hash_array> | HashArray <identifier> <equal_sign> <method_call>

This non-terminal explains how HashArray, the data structure of Veritas language, is formed. HashArray should have an identifier name with the HashArray itself. Such example is:

```
HashArray test = { true, false, true };
```

<hash_array> →<left_curly> <item_list> <right_curly>

This non-terminal describes the format of the elements of the HashArray. Elements of HashArray should be enclosed by curly braces.

<item_list> → <item> | <item> <comma> <item_list>

This non-terminal identifies an item list that is located inside curly braces. There can be just one element in HashArray or a combination of elements that are divided by commas. The comma is the indicator for Veritas language for separation.

<item> → <identifier> | <boolean> | <empty>

This non-terminal describes how HashArray elements are structured. A user may give just a true/false value as a boolean or the same user may give an identifier for the HashArray element. Such example is:

```
boolean a,b,c,d;
a = true;
b = true;
c = false;
d = false;
HashArray withIdentifiers = {a,b,c,d};
```

2.9 Primitive Functions

<primitive_methods> → <scan_input> | <display> | <print_hash>|<get_name> | <add_to_hash> | <delete_all_false> | <delete_all_true> | <all_true_hash> | <all_false_hash> | <is_empty>

This non-terminal is for identifying different types of primitive methods, which are methods that are predefined in Veritas.

<scan_input> → scanInput <left_paranthesis> <io_str> <right_paranthesis>

This non-terminal defines the primitive function scanInput which takes a string parameter as a message. Then, the function gets the value from the user. It is inspired by the Scanner class of Java. The method returns a boolean value, which is the output of the method.

<display> → display <left_paranthesis> <output> <right_paranthesis>

This non-terminal acts as a print line method in Java. The method takes in the value of a string, true, false, or a boolean variable and prints it. The method returns void.

<output> → <io_str> | <identifier> |<boolean>

This non-terminal acts as a parameter list for the display functions and takes in string, variable, or true or false values.

<print_hash>→ printHash <left_paranthesis> <identifier> <right_paranthesis> | printHash <left_paranthesis> <hash_array> <right_paranthesis>

This non-terminal is a print method for a HashArray. It takes in a HashArray variable or an initialized HashArray and prints the elements inside the data structure. The method returns void.

<add_to_hash> → <identifier> <DOT> addHash <left_paranthesis> <identifier> <right_paranthesis> | <identifier> <DOT> addHash <left_paranthesis> <boolean> <right_paranthesis>

This non-terminal adds either true, false, or an identifier that carries a boolean value or a variable into a HashArray. The HashArray is indicated before the method and then a dot is put. After the dot, the parameter input is given and the HashArray adds the parameter value into its elements. The method returns true if the procedure was carried out successfully.

<delete_all_false> →<identifier> <DOT> deleteAllFalse <left_paranthesis> <right_paranthesis>

This non-terminal deletes all false values inside the HashArray. The HashArray is indicated before the method and then a dot is put. The method does not take in any parameters and returns true to indicate that the procedure was carried out correctly.

<delete_all_true> →<identifier> <DOT> deleteAllTrue <left_paranthesis> <right_paranthesis>

This non-terminal deletes all true values inside the HashArray. The HashArray is indicated before the method and then a dot is put. The method does not take in any parameters and returns true to indicate that the procedure was carried out correctly.

<all_true_hash> → <identifier> <DOT> allTrueHash <left_paranthesis> <right_paranthesis>

This non-terminal returns true if all the elements inside the given HashArray are true. The function returns false otherwise. The HashArray is indicated before the method and then a dot is put. The method does not take in any parameters.

<all_false_hash> → <identifier> <DOT> allFalseHash <left_paranthesis> <right_paranthesis>

This non-terminal returns true if all the elements inside the given HashArray are false. The function returns false otherwise. The HashArray is indicated before the method and then a dot is put. The method does not take in any parameters.

<is_empty> → <identifier> <DOT> isEmpty <left_paranthesis> <right_paranthesis>

This non-terminal returns true if there are no elements inside the given HashArray. The function returns false otherwise. The HashArray is indicated before the method and then a dot is put. The method does not take in any parameters.

3.Non-Trivial Tokens

3.1 Identifiers

Identifiers can be written as a combination of letters, digits, and identifier-specific symbols, though the beginning of an identifier must be a letter. This simplifies the look of an identifier and is a feature inspired from Java. Moreover, a single letter can also be an identifier. Lastly, **Veritas** is a case-sensitive language which means that a variable named **a** and another variable named **A** are two different variables. There are also two constant variables decided by the program which are **veritas** and **falsus**. As these names are taken by the programming language, programmers cannot use the taken reserved names.

3.2 Comments

Veritas only allows single-line comments as the simplicity of the programming language does not allow for complicated algorithms to be produced. Single-line comments can be made by putting **##** and then the comment description and finishing it by putting **##** at the end of the comment.

3.3 Literals

There are only two data types, boolean and HashArray, in which HashArray was made specific to Veritas. HashArray can only contain a series of boolean variables. Furthermore, there are other predefined literal values such as **veritas** and **falsus** as true and false. These literals are used as constants in the language.

3.4 Reserved Words

There are reserved words in Veritas such as the word **start**, **finish**, **if**, **else**, **boolean**, **void, true, false, veritas, falsus, for**, **foreach**, **in**, **while**, **do**, **HashArray**, **return**, **scanInput**,

display, **printHash**, **addToHash**, **deleteAllTrue**, **deleteAllFalse**, **isEmpty**, **allTrueHash**, **allFalseHash.**

4. Evaluation

We have only considered the lexical analyzer in this part of the report; though, we completed this portion of the Veritas while also considering the parser implementation. For instance, we decided to keep a track of special reserved constants that have boolean values: **veritas** and **falsus**. These words will be handled by the parser as automatic true or false values. In addition, the default value of boolean variables will be evaluated based on the combination of the variable names. For instance, variable names that start with vowels will have a true value whereas variable names that start with consonants will have a false value, except for the reserved constants **veritas** and **falsus**. Lastly, the parser will take the variables that will be acted on by a function, put a dot, and write the function and its necessary parameters. This style of writing is consistent with Java, instead of taking pointers as parameters like in C++. The code example of such case is as the following:

```
boolean a = true;
HashArray array = {true, false};
array.addHash(a);
```

By such an example, it can be seen that the array is taken not inside the method as a parameter but as listed before the method and indicated that the variable is connected with the method by putting a dot on it.

4.1 Readability

Veritas is a simplified imperative programming language that only supports boolean data types and operations. The language is purely focused on readability and understandability and contains two constants that also count as Boolean values: **veritas** and **falsus**. The language can only modify boolean values and the HashArray data structure that only contains boolean-type variables. Moreover, there are functions added by default to make most common functions easily available to users such as **scanInput()**, **display(),** or **printHash()**. These methods can be both used in other methods or used as sole functions. The names of all primitive functions were made to indicate the function of the method as clearly and shortly as possible. Other users can make custom methods in Veritas and use them on the two data variables. In addition, there are several methods created to work with our new data type: HashArray. The embedded functions for the data type will enable the users to add variables, delete all false or all true boolean values, and check if all values in the data type are true or false.

Furthermore, Veritas' variable names can be declared as letter/s or a combination of letter and digits with the letter being the first character in the name. Only the symbol _ is also allowed to be in the variable names. All constants take on a **true** or a **false** value based on their initial value decided by the program. For instance, **veritas** is a reserved word that takes on the True value and the other reserved word **falsus** takes on the False value.

Moreover, brackets were opted to be used instead of curly braces to indicate the beginning and end of an if-statement, method definition, etc. As integers are not part of our programming language, the traditionally used for-loops are replaced with for-each, while, and do-while type loops. This makes the programming more readable as there are not as many types of for-loops as the traditional programming languages. Lastly, **start** and **finish** were defined as reserved words to indicate the beginning and the end of a program.

4.2 Writability

Veritas has predetermined reserved keywords in the Boolean type that allow users to use instead of just true or false. Moreover, the primitive functions added to the language allow basic functionality to occur without the user having to define additional methods. These functions were named in a way to be easily memorized by the user. Thus, programmers can hide the details of code in custom methods and call their custom methods in their programs. In addition, easy placement of parentheses is allowed in Veritas, which means that programmers can put parentheses in front of their logical expression and change the precedence of logical expressions in such a way. This allows programmers to program more freely and rearrange their logical expressions not just in a left-to-right fashion. Lastly, there is minimal feature multiplicity to prevent the writing of the code being a tedious process. There are only 2 types of for-loops, there are no confusing data structures, and the learning curve of Veritas is rapid for people who already know other imperative languages.

4.3 Reliability

Although Veritas only has 2 data types, variables must be declared with their types before assigning it a value, unlike the programming languages such as Python. This forces the programming language to check each data type before doing any operations, in which this step will be checked by Yacc. Furthermore, identifiers cannot be named in a certain way. The starting character of a variable must be a letter and then digits or the symbol _ can be used in the rest of the name, a feature which was inspired by Java. This allows the language to identify the identifiers as identifiers and not as a random string. Furthermore, although the Lex specification does not include type checking, the BNF evaluation checks constant true and constant false values as terminals. Thus, making the program more reliable in terms of keeping constant values as the same value as it was assigned.

5. References

[1] H. Kwong, “2.3: Implications,” *Mathematics LibreTexts*, Jul. 07, 2021. [Online]. Available: https://math.libretexts.org/Bookshelves/Combinatorics_and_Discrete_Mathematics/A_Spiral_Workbook_for_Discrete_Mathematics_(Kwong)/02%3A_Logic/2.03%3A_Implications. [Accessed: 09-Mar-2023].