CS 315 - Programming Languages

Project 2 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> <stms_list> | <stmt>

<stmt> → < if_stmt> | <non_if_statement> | <comment>

1.2 If Statements

<if_stmt> → if <left_paranthesis> <logical_combination> <right_paranthesis> <left_braces> <block> <right_braces> else <left_braces> <block> <right_braces>

|

if <left_paranthesis> <logical_combination> <right_paranthesis> <left_braces> <block> <right_braces> <else_if_stmts>

|

if <left_paranthesis> <logical_combination> <right_paranthesis> <left_braces> <block> <right_braces> <else_if_stmts> else <left_braces> <block> <right_braces>

|

if <left_paranthesis> <logical_combination> <right_paranthesis> <left_braces> <block> <right_braces>

<else_if_stmts> → <else_if_stmt> | <else_if_stmt> <else_if_stmts>

<else_if_stmt> → elseif <left_paranthesis> <logical_combination> <right_paranthesis> <left_braces> <block> <right_braces>

<block> → <start_symbol> <stmt_list> <finish_symbol> | <start_symbol> <return_block> <finish_symbol> | <start_symbol> <stmt_list> <return_block> <finish_symbol>

<return_block> → <return> <return_stmt>

<return_stmt> → <boolean> | <logical_combination> | <hash_array>

1.3 Non-If Statements

<non_if_statement> → <expression> <sc> | <method_declare> | <loop_stmt>

<expression> → <assign_stmt> | <declaration_stmt> | <logical_combination> | <empty>

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

| <identifier> <assign_op> <data_type>

<boolean> → true| false

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

<declaration> → <boolean_tag> <identifier_list>

<declaration_assign>→ <boolean_tag> <assign_stmt> | const <boolean_tag> <assign_stmt>

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

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

1.4 Operators

<logical_combination> → <logical_combination> <implication_double_symbol> <logical_combination2> | <logical_combination2>

<logical_combination2> → <logical_combination2> <implication_single_symbol> <logical_combination3> | <logical_combination3>

<logical_combination3> → <logical_combination3> <or> <logical_combination4> | <logical_combination4>

<logical_combination4>→<logical_combination4> <and> <logical_combination5> | <logical_combination5>

<logical_combination5> → <logical_combination5> <equality_check> <logical_combination6> | <logical_combination6> | <logical_combination5> <equality_check> <boolean>

<logical_combination6> → <variable_combo> | <left_paranthesis> <logical_combination> <right_paranthesis> | <not_operation> <left_paranthesis> <logical_combination> <right_paranthesis>

<equality_check> → <equality_op> | <not_equal_op>

<variable_combo> → <not_operation> <variable> | <variable>

<variable> → <identifier> | <method_call> | <primitive_methods>

1.5 Looping Statements

<loop_stmt> → <while_loop> | <for_loop>

<while_loop> → while <left_paranthesis> <logical_combination> <right_paranthesis> <left_braces> <stmt_list> <right_braces> | do <left_braces> <stmt_list> <right_braces> while <left_paranthesis> <logical_combination> <right_paranthesis>

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

1.6 Function Definitions

<method_declare> → <void_tag> <identifier> <left_paranthesis> <parameter_list> <right_paranthesis> <left_braces> <stmt_list> <right_braces>

| <void_tag> <identifier> <left_paranthesis> <empty> <right_paranthesis> <left_braces> <stmt_list> <right_braces>

| <boolean_tag> <identifier> <left_paranthesis> <parameter_list> <right_paranthesis> <left_braces> <stmt_list> return <return_stmt> <right_braces>

| <boolean_tag> <identifier> <left_paranthesis> <empty> <right_paranthesis> <left_braces> <stmt_list> return <return_stmt> <right_braces>

| <hasharray_tag> <identifier> <left_paranthesis> <parameter_list> <right_paranthesis> <left_braces> <stmt_list> return <return_stmt> <right_braces>

| <hasharray_tag> <identifier> <left_paranthesis> <empty> <right_paranthesis> <left_braces> <stmt_list> return <return_stmt> <right_braces>

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

<parameter> → <boolean_tag> <identifier> | <hasharray_tag> <identifier>

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

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

1.7 Comments

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

1.8 Data Structure

<hash_array_declaration> → <hasharray_tag> <identifier> <assign_op> <hash_array> | <hasharray_tag> <identifier> <assign_op> <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> | <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> | <logical_combination> | <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_list> <right_paranthesis> | <identifier> <DOT> addHash <left_paranthesis> <boolean_tag> <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>

<data_type> → <hash_array> | <boolean>

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> → .

<data_type> → <boolean> | <hash_array>

<boolean_tag> → boolean

<void_tag> → void

<hasharray_tag> → HashArray

<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>

<QUOTE> → “

<chars> → <char> | <char> <chars>

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

<return> → return

<return_stmt> → <boolean> | <logical_combination> | <hash_array>

<boolean> → TRUE | FALSE

<line_comment> → ##

<comment_description> → <chars> | <chars><comment_description>

<start_symbol> → #*

<finish_symbol> → *#

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> <stms_list> | <stmt>

This non-terminal indicates that statement lists consist of statements or statements with a statements list.

<stmt> → < if_stmt> | <non_if_statement> | <comment>

This non-terminal shows that statements are labeled as if statements, non-if statements or comments.

2.2 If Statements

<if_stmt> → if <left_paranthesis> <logical_combination> <right_paranthesis> <left_braces> <block> <right_braces> else <left_braces> <block> <right_braces>

|

if <left_paranthesis> <logical_combination> <right_paranthesis> <left_braces> <block> <right_braces> <else_if_stmts>

|

if <left_paranthesis> <logical_combination> <right_paranthesis> <left_braces> <block> <right_braces> <else_if_stmts> else <left_braces> <block> <right_braces>

|

if <left_paranthesis> <logical_combination> <right_paranthesis> <left_braces> <block> <right_braces>

The non-terminal known as if statement is utilized for if statements or for if statements that have a corresponding else statement. In the Veritas programming language, an if statement necessitates parentheses for the logical combinations and braces for the statements (block) inside. An corresponding “if” statement may also include else-if statement(s) and/or as well as else statements included by Veritas language.

<else_if_stmts> → <else_if_stmt> | <else_if_stmt> <else_if_stmts>

This non-terminal indicated as “else-if” statement is used to test a new condition if the first condition provided is false. “Else-if” statements may take more than one else-if statements as long as the above conditions in the code are evaluated as “if” and “else-if” statements.

<else_if_stmt> → elseif <left_paranthesis> <logical_combination> <right_paranthesis> <left_braces> <block> <right_braces>

This particular non-terminal shows else-if statements used in Veritas language. Veritas language evaluates conditions in order and the evaluated conditions are more than two, at least one of the conditions should be evaluated in “else-if” condition. This situation can be distinguishable as the “elseif” keyword in Lex evaluation. “Elseif” is similar to the “if” condition described above: checking condition (logical combination) should be enclosed with parentheses, statement block should be enclosed with braces.

<block> → <start_symbol> <stmt_list> <finish_symbol> | <start_symbol> <return_block> <finish_symbol> | <start_symbol> <stmt_list> <return_block> <finish_symbol>

This non-terminal shows the block statements used inside if statements. Beginning inside if,elseif and else statements there should be start_symbol **#*** and finish_symbol ***#** to differentiate statements inside blocks from other statements. It also enables users to insert return statements inside if statements.

<return_block> → <return> <return_stmt>

This particular non-terminal is used to emphasize return statement blocks in the Veritas language. A function may take one or more return values depending on if statements inside. By this approach, users may define more than one return statement inside the program.

<return_stmt> → <boolean> | <logical_combination> | <hash_array>

This particular non-terminal shows that return statements inside return blocks can be boolean, logical combination (combination of operations) or HashArray.

2.3 Non-If Statements

<non_if_statement> → <expression> <sc> | <method_declare> | <loop_stmt>

This non-terminal is used for statements that do not include if statement(s). These statements can consist of expressions, method declarations or loop statements.

<expression> → <assign_stmt> | <declaration_stmt> | <logical_combination> | <empty>

This particular non-terminal expression shows that expression may consist of assignment statements, declaration statements, logical combinations or simply empty.

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

| <identifier> <assign_op> <data_type>

This assignment non-terminal describes that identifiers may be assigned to a logical combination or to its data type. Such examples are given:

```
boolean a,b,c,d;
a = c && d;
b = true;
```

<boolean> → true| false

This particular non-terminal shows that Veritas language takes 2 tokens for boolean non-terminal. These tokens are “true” and “false” in the language.

<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> → <boolean_tag> <identifier_list>

This non-terminal identifies declarations with their boolean tag and the list of identifier items.

<declaration_assign>→ <boolean_tag> <assign_stmt> | const <boolean_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 boolean 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> | <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

<logical_combination> → <logical_combination> <implication_double_symbol> <logical_combination2> | <logical_combination2>

This particular non-terminal indicates precedence rules in Veritas language, beginning from the lowest order in the language, double implication operation. Note that this and below BNF rules are listed according to ascending operation order, in the tree structural BNF. Details about double implication rules can be found below expressions.

Double implication truth table as follows:

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

Table 1: 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.

<logical_combination2> → <logical_combination2> <implication_single_symbol> <logical_combination3> | <logical_combination3>

This particular non-terminal refers to precedence rules in Veritas language, beginning from the second lowest order in the language, single implication operation.

Single implication truth table as follows:

p	q	p -> q

true	true	true
true	false	false
false	true	true
false	false	true

Table 2: 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.

<logical_combination3> → <logical_combination3> <or> <logical_combination4> | <logical_combination4>

This non-terminal shows precedence rules in Veritas language, beginning from the third lowest order in the language, or operation. “OR” operation result is evaluated as true even if one of the entries in the condition is true.

<logical_combination4>→<logical_combination4> <and> <logical_combination5> | <logical_combination5>

This non-terminal shows precedence rules in Veritas language, beginning from the third highest order in the language, AND operation.And operation result is evaluated true if all of the entries in the operation is true, otherwise result is false.

<logical_combination5> → <logical_combination5> <equality_check> <logical_combination6> | <logical_combination6> | <logical_combination5> <equality_check> <boolean>

This non-terminal points out precedence rules in Veritas language, beginning from the second highest order in the language, equality check operations.This operation considers both expressions in the sides of operator are equal or not equal to each other.

<logical_combination6> → <variable_combo> | <left_paranthesis> <logical_combination> <right_paranthesis> | <not_operation> <left_paranthesis> <logical_combination> <right_paranthesis>

This particular non-terminal shows that the not operation and matching parentheses have higher precedence order than any other operations. Such an approach gives users the flexibility to manipulate order of precedence according to her/his needs.

<equality_check>→ <equality_op> | <not_equal_op>

This non-terminal points out the equality check conditions in the Veritas language. Two identifier combinations can be checked by their equality or inequality by == and !! operators.

<variable_combo> → <not_operation> <variable> | <variable>

This particular non-terminal indicates that variables in the preceding operations may take variables with their not operation or just themselves.

<variable> → <identifier> | <method_call> | <primitive_methods>

This particular non-terminal shows that variables in the Veritas language can be an identifier (a), method calls (foo()) or primitive methods (display(“hello”)).

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> <logical_combination> <right_paranthesis> <left_braces> <stmt_list> <right_braces> | do <left_braces> <stmt_list> <right_braces> while <left_paranthesis> <logical_combination> <right_paranthesis>

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

(logical_combination), similar to the applied rules in other imperative programming languages. Such while loop examples are:

```
## do-while example ##
HashArray arr  = { 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_list> <right_braces> | foreach <identifier> in <hash_array> <left_braces> <stmt_list> <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};
foreach val in hashArr
[
      ## do something... ##
]
```

2.6 Function Definitions

<method_declare> → <void_tag> <identifier> <left_paranthesis> <parameter_list> <right_paranthesis> <left_braces> <stmt_list> <right_braces>

| <void_tag> <identifier> <left_paranthesis> <empty> <right_paranthesis> <left_braces> <stmt_list> <right_braces>

| <boolean_tag> <identifier> <left_paranthesis> <parameter_list> <right_paranthesis> <left_braces> <stmt_list> return <return_stmt> <right_braces>

| <boolean_tag> <identifier> <left_paranthesis> <empty> <right_paranthesis> <left_braces> <stmt_list> return <return_stmt> <right_braces>

| <hasharray_tag> <identifier> <left_paranthesis> <parameter_list> <right_paranthesis> <left_braces> <stmt_list> return <return_stmt> <right_braces>

| <hasharray_tag> <identifier> <left_paranthesis> <empty> <right_paranthesis> <left_braces> <stmt_list> return <return_stmt> <right_braces>

This non-terminal shows how methods are declared in the Veritas language. Method tags can be void, boolean or HashArray. Parameter(s) are listed inside parentheses, or there can be no parameter inside a method calling. Method declares that are not void also include the return tag and return statements.

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

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

<parameter> → <boolean_tag> <identifier> | <hash_array_tag> <identifier>

This non-terminal expresses how parameters are formed in Veritas language. The parameter's tag (boolean or HashArray) and identifier name should be given.

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

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> | <boolean> | <boolean> <comma> <parameter_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. There can also be boolean variables such as true and/or false.

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_tag> <identifier> <assign_op> <hash_array> | <hasharray_tag> <identifier> <assign_op> <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. Hash array can also be set equal to a method call.

Such example is:

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

<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;
a = true;
b = true;
HashArray withIdentifiers = {a,b};
```

2.9 Primitive Functions

<primitive_methods> → <scan_input> | <display> | <print_hash> | <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> | <logical_combination> |<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_list> <right_paranthesis> | <identifier> <DOT> addHash <left_paranthesis> <boolean_tag> <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.

<data_type> → <hash_array> | <boolean>

This particular non-terminal shows the type of the data (variable) in the Veritas language. It can be HashArray data structure type or simply boolean type.

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.

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 which can be represented as {true,false}.

3.4 Reserved Words

There are reserved words in Veritas such as the word **start**, **finish**, **if**, **else**,**elseif, boolean**, **void, true, false, foreach**, **in**, **while**, **do**, **HashArray**, **return**, **scanInput**, **display**, **printHash**, **addToHash**, **deleteAllTrue**, **deleteAllFalse**, **isEmpty**, **allTrueHash**, **allFalseHash.**

It should be emphasized that Veritas language starts with a **start** word and ends with a **finish** word. Every statement(s) between start and finish keywords are evaluated according to

BNF grammar rules. If there is an error detected by the parser, it prints out syntax errors with their line numbers. Else, the program prints out that it is valid.

4. Evaluation

4.1 Parameter Passing and Scope

Veritas language follows a strict pass-by-value policy in its methods, meaning that if a variable is passed to a method and is changed within its scope, the change will not be reflected outside the method. In addition, the language adopts a block-scoped variable declaration similar to Java, where variables declared within a block are not recognized outside of it. However, they can change the value of a variable declared outside of the block. If a variable with the same name is declared both inside and outside the block, a completely new variable is created for the inside block that is independent of the outside scope.

4.2 Operation Evaluation Order

Veritas language enables users to perform operations on their precedence rules. Order of evaluation of operations in the Veritas language from highest precedence to lowest precedence as follows:

1. Parentheses and NOT operation
2. Equality Checking (== and !!)
3. AND operation
4. OR operation
5. SINGLE Implication
6. DOUBLE Implication

This precedence rule applied within the language gives freedom to users to apply combinations of operations together on their importance level. Veritas executes the code from left to right; thus, the precedence of parentheses and not operation is evaluated based on a left-to-right reading fashion. Moreover, probabilities of different combinatioral operations increase, which will lead to more modular and applicable language rules similar to Imperative languages such as Java, C++.

4.3 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. 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 named to indicate the function of the method as clearly and understandably as possible. Other users can make custom methods in Veritas and use them on the two data variables: boolean and HashArray. 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.

Moreover, a new feature was added to Veritas to prevent confusion within the program for the if statements. Users must specify the beginning and the end of statements within an if-statement by typing the symbol #* and *#. The statements written in between the symbols will be executed by the program. This feature enables adding several components with no limitations inside the conditional statements without causing any conflicts.

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.

4.4 Writability

Veritas has predetermined primitive functions added to the language that 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. Moreover, brackets were opted to be used

instead of curly braces to indicate the beginning and end of an if-statement, method definition, etc. We believe that it creates a difference from other programming languages and it is easier to type brackets in Linux and MacOS compared to curly braces as shift key also must be pressed to type curly braces compared to a bracket having its own key. Lastly, there is minimal feature multiplicity to prevent the writing of the code being a tedious process. There are only 3 types of loops, there are no confusing data structures, and the learning curve of Veritas is rapid for people who already know other imperative languages.

4.5 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.

Since Veritas does not have pointers, it eliminates the problems that are caused by memory leaks and enables a smaller memory usage.Moreover, Veritas checks for return types in its method declarations, which can include boolean, HashArray, or void. If a method is declared as void, no return statement is required. For all other types, a return statement is necessary. This ensures that the program always returns a value. While Veritas does not check for type consistency when it comes to reassignments or assigning through a method call or identifier, this can actually offer flexibility and ease of use in certain programming scenarios. In addition, Veritas allows users to include statements within conditional statements, and even return a value outside of the conditional statement. This ensures that the program always returns a value, no matter what, and helps to eliminate errors and confusion during the development process.

Finally, Veritas programming language has no unresolved conflicts in its YACC execution, which increases the overall reliability.

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].