2021-2022 Spring CS315 Programming Languages Project II

Group 14

Dominica Language

Members

Ferhat Korkmaz 21901940 Section 1

Melih Fazıl Keskin 21901831 Section 1

Kylian Djermoune 22103997 Section 1

Name of the language: Dominica

BNF Description of Dominica Language

1) Types and Constants

<set_indicator> ::= “_”

<LB> ::= “{“

<RB> ::= “}”

<LP> ::= “(“

<RP> ::= “)”

<return> ::= “return”

<void> ::= “void”

<end_of_stmt> ::= “;”

<line_comment_start> ::= “#”

<new_line> ::= “\n”

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

<ascii_char> ::= [-!#-~]

<digit> ::= [0-9]

<quote> ::= “"”

<column> ::= “:”

<sign> ::= + | -

<exclamation_mark> ::= “!”

<and_op> ::= “&&”

<or_op> ::= “||”

<equals_op> ::= “==”

<func_type> ::= “func_”

<not_equals_op> ::= “!=”

<greater_than_op> ::= “>”

<greater_equal_op> ::= “>=”

<less_than_op> ::= “<”

<less_equal_op> ::= “<=”

<comma> ::= “,”

<mul_div_sign> ::= * | /

<dot> ::= “.”

<empty_grammar_rule> ::= “”

<assignment_op> ::= “=”

<type_name> ::= “int” | “string” | “bool” | “double” | “elm”

<set_type> ::= “set”

<set_add_op> ::= “++”

<set_remove_op> ::= “--”

<set_union_op> ::= “U”

<set_intersection_op> ::= “^"

<set_difference_op> ::= “\”

<set_delete_op> ::= “delete”

<set_is_subset> ::= “C”

<set_is_superset> ::= “@”

<set_is_an_element_of> ::= “E”

<set_are_disjoint> ::= “D”

<main> ::= “main()”

2) Program

<Dominica> ::= <main><LB><RB>
| <main><LB><stmt_list><RB>
| <main><LB><RB><function_list>
| <main><LB><stmt_list><RB><function_list>

<function_list> ::= <function> | <function><function_list>
<stmt_list> ::= <stmt> | <stmt><stmt_list>

3) Identifiers and Variables

-Identifiers

<function_identifier> ::= <func_type><identifier>

<identifier> ::= <letter_char><letter_char> | <letter_char><digit> |<identifier><letter_char>
| <identifier><digit>

<set_identifier> ::= <set_indicator><letter_char> | <set_indicator><digit>
| <set_identifier><letter_char> | <set_identifier><digit>

-Variables

<set> ::= <LB><RB> | <LB><set_init_list><RB>

<set_init_list> ::= <identifier> | <type> | <identifier><comma><set_init_list>
| <type><comma><set_init_list>

<type> ::= <integer> | <string> | <double> | <bool>

<string> ::= <quote><quote> | <quote><string_term><quote>

<string_term> ::= <ascii_char> | <string_term><ascii_char>

<integer> ::= <digit> | <sign><digit> | <digit><integer> | <sign><digit><integer>

<double> ::= [<sign>]{<digit>}[<dot>]{<digit>}

<bool> ::= true | false

4) Statements

<stmt> ::= <expression><end_of_stmt>
| <conditional_stmt>
| <loop_stmt>
| <commentt>

5) Loops

<loop_stmt> ::= <for_loop> | <while_loop>

<for_loop> ::= <reg_for_loop> | <set_for_loop>

<reg_for_loop>::=
for<LP><variable_declaration><end_of_stmt><logical_arithmetic_expression><end_of_stmt
> <assignment_expression><RP><LB><stmt_list><RB>
|for<LP><variable_declaration><end_of_stmt><logical_arithmetic_expression><end_of_stm
t> <assignment_expression><RP><LB><RB>

<set_for_loop>::=
for<LP><set_identifier><column><set_identifier><RP><LB><stmt_list><RB>
|for<LP><set_identifier><column><integer><column><set_identifier><RP><LB><stmt_list
><RP>

|for<LP><set_identifier><column><integer><column><identifier><RP><LB><stmt_list>
<RP>
|for<LP><set_identifier><column><set_identifier><RP><LB><RB>
|for<LP><set_identifier><column><integer><column><set_identifier><RP><LB><RP>
|for<LP><set_identifier><column><integer><column><identifier><RP><LB><RP>

<while_loop> ::= while<LP><logical_arithmetic_expression><RP><LB><stmt_list><RB>

|while<LP><logical_arithmetic_expression><RP><LB><RB>

6) Conditionals

<conditional_stmt> ::= if<LP><logical_arithmetic_expression><RP><LB><stmt_list><RB>
|if<LP><logical_arithmetic_expression><RP><LB><RB>
|if<LP><logical_arithmetic_expression><RP><LB><stmt_list><RB>else<LB><stmt_list><
RB>
|if<LP><logical_arithmetic_expression><RP><LB><stmt_list><RB>else<LB><RB>
|if<LP><logical_arithmetic_expression><RP><LB><RB>else<LB><stmt_list><RB>
|if<LP><logical_arithmetic_expression><RP><LB><RB>else<LB><RB>
|if<LP><logical_arithmetic_expression><RP><LB><stmt_list><RB>else<conditional_stmt>
|if<LP><logical_arithmetic_expression><RP><LB><RB>else<conditional_stmt>

-Logical Expressions

<logical_arithmetic_expression> ::= <regular_logic_arithmetic> | <set_logic>
| <logical_arithmetic_expresion><and_op><regular_logic_arithmetic>
| <logical_arithmetic_expresion><or_op><regular_logic_arithmetic>
| <logical_arithmetic_expresion><and_op><set_logic>
| <logical_arithmetic_expresion><or_op><set_logic>

<regular_logic_arithmetic> ::= <bool> | <bool><equals_op><bool>
| <bool><not_equals_op><bool> | <arithmetic_operation>
| <exclamation_mark><LP><logical_arithmetic_expression><RP>
| <arithmetic_operation><equals_op><bool>
| <arithmetic_operation><not_equals_op><bool>
| <arithmetic_operation><equals_op><arithmetic_operation>
| <arithmetic_operation><not_equals_op><arithmetic_operation>
| <arithmetic_operation><greater_equal_op><arithmetic_operation>
| <arithmetic_operation><less_equal_op><arithmetic_operation>
| <arithmetic_operation><greater_than_op><arithmetic_operation>
| <arithmetic_operation><less_than_op><arithmetic_operation>

<set_logic> ::= <identifier><set_is_an_element_of><set>

| <type><set_is_an_element_of><set>
| <identifier><set_is_an_element_of><set_identifier>
| <type><set_is_an_element_of><set_identifier>
| <set_operation><set_is_subset><set_operation>
| <set_operation><set_is_superset><set_operation>
| <set_operation><set_are_disjoint><set_operation>

7) Expressions

<expression> ::= <variable_declaration> | <assignment_expression> | <function_call> |
<set_expression> | <set_declaration> | <set_assignment>

<variable_declaration>::= <type_name><identifier><assignment_op><string>
| <type_name><identifier><assignment_op><logical_arithmetic_expression>
| <type_name><identifier><assignment_op><function_call>

<assignment_expression> ::= <identifier><assignment_op><string>
| <identifier><assignment_op><logical_arithmetic_expression>
| <identifier><assignment_op><function_call>

8) Sets

<set_declaration> ::= <set_type><set_identifier><assignment_op><set_operation>
| <set_type><set_identifier><assignment_op><function_call>

<set_assignment> ::= <set_identifier><assignment_op><set_operation>
| <set_identifier><assignment_op><function_call>

<set_expression> ::= <set_identifier><set_add_op>< type >
| <set_identifier><set_remove_op>< type >
| <set_identifier><set_add_op><identifier>
| <set_identifier><set_remove_op><identifier>
| <set_delete_op><set_identifier>

<set_operation> ::= <set_term> | <set_operation><set_intersection_op><set_term>
| <set_operation><set_difference_op><set_term>
| <set_operation><set_union_op><set_term>

<set_term> ::= <set_identifier> | <set> | <LP><set_operation><RP>

9) Arithmetic Operations

<arithmetic_operation> ::= <arithmetic_term>
| <arithmetic_operation><sign><arithmetic_term>

<arithmetic_term> ::= <arithmetic_factor>

|<arithmetic_term><mul_div_sign><arithmetic_factor>

<arithmetic_factor> ::= <LP><arithmetic_operation><RP> | <identifier> | <integer>
| <double>

10) Functions

-Function Definition

<function> ::=
<type_name><function_identiifer><LP><RP><LB><stmt_list><return><identifier><end_of_stmt><stmt_list><RB>
|<type_name><function_identiifer><LP><RP><LB><stmt_list><return><identifier><end_of_stmt><RB>
|<type_name><function_identiifer><LP><RP><LB><return><identifier><end_of_stmt><stmt_list><RB>
|<type_name><function_identiifer><LP><RP><LB><return><identifier><end_of_stmt><RB>
|<type_name><function_identiifer><LP><RP><LB><stmt_list><return><type><end_of_stmt><stmt_list><RB>
|<type_name><function_identiifer><LP><RP><LB><stmt_list><return><type><end_of_stmt><RB>
|<type_name><function_identiifer><LP><RP><LB><return><type><end_of_stmt><stmt_list><RB>
|<type_name><function_identiifer><LP><RP><LB><return><type><end_of_stmt><RB>
|<type_name><function_identiifer><LP><param_list><RP><LB><stmt_list><return><identifier><end_of_stmt><stmt_list><RB>
|<type_name><function_identiifer><LP><param_list><RP><LB><stmt_list><return><identifier><end_of_stmt><RB>
|<type_name><function_identiifer><LP><param_list><RP><LB><return><identifier><end_of_stmt><stmt_list><RB>
|<type_name><function_identiifer><LP><param_list><RP><LB><return><identifier><end_of_stmt><RB>
|<type_name><function_identiifer><LP><param_list><RP><LB><stmt_list><return><type><end_of_stmt><stmt_list><RB>
|<type_name><function_identiifer><LP><param_list><RP><LB><stmt_list><return><type><end_of_stmt><RB>
|<type_name><function_identiifer><LP><param_list><RP><LB><return><type><end_of_stmt><stmt_list><RB>
|<type_name><function_identiifer><LP><param_list><RP><LB><return><type><end_of_stmt><RB>
|<set_type><function_identiifer><LP><RP><LB><stmt_list><return><set_identifier><end_of_stmt><stmt_list><RB>
|<set_type><function_identiifer><LP><RP><LB><stmt_list><return><set_identifier><end_of_stmt><RB>
|<set_type><function_identiifer><LP><RP><LB><return><set_identifier><end_of_stmt><stmt_list><RB>
|<set_type><function_identiifer><LP><RP><LB><return><set_identifier><end_of_stmt><RB>
|<set_type><function_identiifer><LP><RP><LB><stmt_list><return><set><end_of_stmt><stmt_list><RB>
|<set_type><function_identiifer><LP><RP><LB><stmt_list><return><set><end_of_stmt><RB>

|<set_type><function_identiifer><LP><RP><LB><return><set><end_of_stmt><stmt_list><RB>
|<set_type><function_identiifer><LP><RP><LB><return><set><end_of_stmt><RB>
|<type_name><function_identiifer><LP><param_list><RP><LB><stmt_list><return><set_identifier><end_of_stmt><stmt_list><RB>
|<set_type><function_identiifer><LP><param_list><RP><LB><stmt_list><return><set_identifier><end_of_stmt><RB>
|<set_type><function_identiifer><LP><param_list><RP><LB><return><set_identifier><end_of_stmt><stmt_list><RB>
|<set_type><function_identiifer><LP><param_list><RP><LB><return><set_identifier><end_of_stmt><RB>
|<set_type><function_identiifer><LP><param_list><RP><LB><stmt_list><return><set><end_of_stmt><stmt_list><RB>
|<set_type><function_identiifer><LP><param_list><RP><LB><stmt_list><return><set><end_of_stmt><RB>
|<set_type><function_identiifer><LP><param_list><RP><LB><return><set><end_of_stmt><stmt_list><RB>
|<set_type><function_identiifer><LP><param_list><RP><LB><return><set><end_of_stmt><RB>
|<void><function_identifier><LP><RP><LB><stmt_list><RB>
|<void><function_identifier><LP><RP><LB><RB>
|<void><function_identifier><LP><RP><LB><stmt_list><return><end_of_stmt><stmt_list><RB>
|<void><function_identifier><LP><RP><LB><return><end_of_stmt><stmt_list><RB>
|<void><function_identifier><LP><RP><LB><stmt_list><return><end_of_stmt><RB>
|<void><function_identifier><LP><RP><LB><stmt_list><end_of_stmt><RB>
|<void><function_identifier><LP><param_list><RP><LB><stmt_list><RB>
|<void><function_identifier><LP><param_list><RP><LB><RB>
|<void><function_identifier><LP><param_list><RP><LB><stmt_list><return><end_of_stmt><stmt_list><RB>
|<void><function_identifier><LP><param_list><RP><LB><return><end_of_stmt><stmt_list><RB>
|<void><function_identifier><LP><param_list><RP><LB><stmt_list><return><end_of_stmt><RB>
|<void><function_identifier><LP><param_list><RP><LB><stmt_list><end_of_stmt><RB>

-Function Call

<function_call> ::= <function_identifier><LP><RP>
|<function_identifier><LP><arg_list><RP>
| <primitive_function_call>

<primitive_function_call> ::= n<LP><set_operation><RP> | pow<LP><set_operation><RP>
| cartesian<LP><set_operation><comma><set_operation><RP>
| print<LP><set_operation><RP>
| print<LP><string><RP>
| print<LP><identifier><RP>
| scan<LP><identifier><RP>
| scan<LP><set_identifier><RP>
| fread<LP><string><RP>
| fwrite<LP><string><comma><string><RP>
| fwrite<LP><set_operation><comma><string><RP>

| fwrite<LP><identifier><comma><string><RP>

-Lists

<arg_list> ::= <logical_arithmetic_expression> | <string> | <set_identifier>
| <logical_arithmetic_expressin> <comma> <arg_list>
| <string> <comma> <arg_list>
| <set_identifier> <comma> <arg_list>

<param_list> ::= <type_name><identifier> | <set_type><set_identifier>
| <type_name><identifier><comma><param_list>
| <set_type><set_identifier><comma><param_list>

10) Comments

<comment> ::= <line_comment_start><string_term>

Explanation of Dominica Language Constructs

Constants & Symbols

- <set_indicator> ::= “_”

This construct represents “_” symbol and it can be used at the beginning of <set_identifier> to make it distinct than <identifier>

- <LB> ::= “{“

This construct represents the left curly braces.

- <RB> ::= “}”

This construct represents the right curly braces.

- <LP> ::= “(“

This construct represents the left parenthesis.

- <RP> ::= “)”

This construct represents the right parenthesis.

- <return> ::= “return”

This construct is used to represent “return” for returning a variable or set in the functions.

- <end_of_stmt> ::= “;”

This construct represents semi column and it is used at the end of statements.

- <line_comment_start> ::= “#”

This construct represents ‘#’ and used at the beginning of line comments.

- <new_line> ::= “\n”

This construct represents the new line character.

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

This construct represents a single letter.

- <ascii_char> ::= [-!#-~]

This construct represents all of the ASCII characters except quote. It is because of preventing the confusion with string indicator <quote>

- <digit> ::= [0-9]

This construct represents a single digit.

- <quote> ::= “"”

This construct represents quote sign and it is used for indicating strings.

- <column> ::= “:”

This construct represents column and it is used in set for loops.

- <sign> ::= + | -

This construct represents the minus and plus to indicate the sign of numbers.

- <exclamation_mark> ::= “!”

This construct represents the exclamation mark and it is used for reversing the result of a logic operation.

- <and_op>::= “&&”

This construct represents ‘&&’ symbol and it is used as AND operation in logic operations.

- <or_op>::= “||”

This construct represents ‘||’ symbol and it is used as OR operation in logic operations.

- <equals_op> ::= “==”

This construct represents ‘==’ symbol and it is used for checking equality in logical operations.

- <func_type> ::= “func_”

This construct represents the “func_” word and it is used at the beginning of <function_identifier> to make it distinct from <identifier>

- <not_equals_op> ::= “!=”

This construct represents '!=' symbol and it is used for checking equality in logical operations.

- <greater_than_op> ::= ">"

This construct represents '>' symbol and it is used for checking if the left-hand side is greater than the right-hand side in logical operations.

- <greater_equal_op> ::= ">="

This construct represents '>=' symbol and it is used for checking if the left-hand side is greater than or equal to the right-hand side in logical operations.

- <less_than_op> ::= "<"

This construct represents '<' symbol and it is used for checking if the left-hand side is less than the right-hand side in logical operations.

- <less_equal_op> ::= "<="

This construct represents '<=' symbol and it is used for checking if the left-hand side is less than or equal to the right-hand side in logical operations

- <comma> ::= ","

This construct represents the comma and it is used for separating identifiers or types in set declaration or <param_list> or <arg_list>.

- <mul_div_sign> ::= * | /

This construct represents the symbols for multiplication and division.

- <dot> ::= "."

This construct represents the dot that is used in the representation of doubles.

- <empty_grammar_rule> ::= ""

This construct represents nothing. It makes our grammar more readable when we have an empty rule.

- <assignment_op> ::= "="

This construct represents the '=' symbol which is used for assigning values to identifiers.

- <type_name> ::= "int" | "string" | "bool" | "double" | "elm"

This construct represents the reserved words for type names of variables.

- <set_type> ::= "set"

This construct represents the reserved word for type name of set.

- <set_add_op> ::= "++"

This construct represents the symbol '++', which is used for adding an element to a set.

- <set_remove_op> ::= “--”

This construct represents the symbol ‘--’, which is used for removing an element from a set.

- <set_union_op> ::= “U”

This construct represents the symbol ‘U’ which is used for creating a union of two sets. We selected this symbol because it is similar to its mathematical notation.

- <set_intersection_op> ::= “^"

This construct represents the symbol ‘^’ which is used for taking the intersection of two sets. We selected this symbol because it is similar to its mathematical notation.

- <set_difference_op> ::= “\”

This construct represents the symbol ‘\’ which is used for finding the difference of two sets. We selected this symbol because it is similar to its mathematical notation.

- <set_delete_op> ::= “delete”

This construct represents the reserved word for the deletion of a set.

- <set_is_subset> ::= “C”

This construct represents ‘C’ symbol and it is used in logical set operations for determining if the left-hand side is a subset of the right-hand side. We selected this symbol because it is similar to its mathematical notation.

- <set_is_superset> ::= “@”

This construct represents ‘@’ symbol and it is used in logical set operations for determining if the left-hand side is the superset of the right-hand side. We selected this symbol because it is looks like there are two circles and one is inside another. It is similar to Venn Diagram.

- <set_is_an_element_of> ::= “E”

This construct represents ‘E’ symbol and it is used in logical set operations for determining if the left-hand side is an element of right-hand side. We selected this symbol because it is similar to its mathematical notation.

- <set_are_disjoint> ::= “D”

This construct represents ‘D’ symbol and it is used in logical set operations for determining if the two sets are disjoint. We selected this symbol because as the first letter for disjoint.

- <main> ::= “main()”

This construct represents the reserved word for the main call in the program.

- <void> ::= “void”

This is a reserved for the void return type of the functions.

Program Structure

- <Dominica> ::= <main><LB><RB>
  | <main><LB><stmt_list><RB>
  | <main><LB><RB><function_list>
  | <main><LB><stmt_list><RB><function_list>

Dominica language has to consist of a <main> means main() and it has to be followed by <LB> and <RB>, which are curly braces.

- <stmt_list> ::= <stmt> | <stmt><stmt_list>

Inside the curly braces, there can be <stmt_list>. It consists of multiple <stmt>, which are statements. These statements are the instructions which the user wants to execute during work time.

- <function_list> ::= <function> | <function><function_list>

Also, function definitions can be included in <function_list>, at outside of the curly braces. It consists of multiple <function>, which are function definitions. Keeping function definitions outside of the main program increases the writability.

Variables

We have set, integer, double, string, bool, and elm as variables in our language.

- <set> ::= <LB><RB> | <LB><set_init_list><RB>

The variable type set it is indicated with left and right braces. The reason why we choose having braces to indicate sets is that it is similar to mathematical set notations. Also, it increases the readability and reliability. Inside these braces, it may have <set_init_list>. Otherwise, it is an empty set.

- <set_init_list> ::= <identifier> | <type> | <identifier><comma><set_init_list>
  | <type><comma><set_init_list>

<set_init_list> can be consist of all the other variable types or their identifiers other than sets. They are separated with comma to increase readability and reliability.

- <type> ::= <integer> | <string> | <double> | <bool>

<type> is for all variables except sets in Dominica language. Also, it is equivalent to variable type elm, to which any type of variable other than set can be assigned.

- <string> ::= <quote><quote> | <quote><string_term><quote>

String is a variable which consists of combinations of ASCII characters(except quote). It is indicated with quotes. Inside the quotes, it can contain <string_term>. Otherwise, it is an empty string.

- <string_term> ::= <ascii_char> | <string_term><ascii_char>

<string_term> is the combination of ASCII characters(except quote). The reason why quote mark is not included is that it may cause a confusion with string indicator <quote>.

- <integer> ::= <digit> | <sign><digit> | <digit><integer> | <sign><digit><integer>

The representation of integer variables can be with sign or without a sign.

- <double> ::= [<sign>]{<digit>}[<dot>]{<digit>}

The representation of double variables can be with sign or without a sign. Also, it can be without any digit before or after the dot. Also, it can be without a dot just like an integer. These options increase the writability.

- <bool> ::= true | false

Bool variables can have two values. They are true or false.

Identifiers

- <function_identifier> ::= <func_type><identifier>

<function_identifier> is used to identify functions. It has to start with “func_” reserved word which makes the distinction between it and the <identifier>. This condition increases reliability and readability but there is a little decrease in writability.

- <identifier> ::= <letter_char><letter_char> | <letter_char><digit>
  | <identifier><letter_char> | <identifier><digit>

Identifier is used to identify <type> variables. It has to consist of minimum of two characters and the first one has to be <letter_char>. It is for making the distinction between identifiers and operators we used like ‘C’ or ‘U’. This increases reliability but decreases writability.

- <set_identifier> ::= <set_indicator><letter_char> | <set_indicator><digit>
  | <set_identifier><letter_char> | <set_identifier><digit>

<set_identifier> has to start with an the ‘_’ char which is the <set_indicator>. In that way, the distinction between <set_identifier> and <identifier> is made. It makes it easy to do not to include any set in a set. Therefore, the reliability increases.

Statements

- <stmt> ::= <expression><end_of_stmt>
  | <conditional_stmt>
  | <loop_stmt>
  | <commentt>

<stmt> holds for one instruction in Dominica. For expressions, they have to followed by ‘;’ character, which is indicated by <end_of_stmt>. Other possibilities like <conditional_stmt> or <loop_stmt> has their own rule in their definition.

Expressions

- <expression> ::= <variable_declaration> | <assignment_expression> | <function_call | <set_expression> | <set_declaration> | <set_assignment>

<expression> is the common instructions in a programming language which generally occupies only one line. It can be one of the constructs which are indicated in its BNF definition.

Declarations

- <variable_declaration>::= <type_name><identifier><assignment_op><string> | <type_name><identifier><assignment_op><logical_arithmetic_expression> | <type_name><identifier><assignment_op><function_call>

Variable declaration for the types has to be done by assigning a value. The reason behind this is increasing reliability. To be more precise:

~~int count;~~ #This is not correct
int count = 5 * 4; #This is correct

Assignment operator is used in the declarations and the left-hand side has to be <type_name> followed by <identifier>. The right-hand side can be one of the constructs in the BNF definition. <identifier> and other types other than strings like int are included with <logical_arithmetic_expression> definition to prevent ambiguity. The right-hand side should match the variable type on the left-hand side unless the <type_name> is “elm”. “elm” type can contain any of the types among integer, string, double, or string.

- <set_declaration> ::= <set_type><set_identifier><assignment_op><set_operation> | <set_type><set_identifier><assignment_op><function_call>

Set declaration for the sets has to be done by assigning a set similar to variable declaration. The reason behind this is increasing reliability. To be more precise:

~~set _mySet;~~ #This is not correct
set _mySet = { count, true, 5, “Dominica” }; #This is correct

The assignment operator is used in the declaration of the set and the left-hand side has to be a <set_type> reserved word followed by a <set_identifier>. The right-hand side can be one of

the constructs in the BNF definition. <set> is included in <set_operation> to prevent ambiguity.

Assignments

- <assignment_expression> ::= <identifier><assignment_op><string>
  | <identifier><assignment_op><logical_arithmetic_expression>
  | <identifier><assignment_op><function_call>

The left-hand side of the assignment expression has to be an identifier. The right-hand side can be one of the constructs in the BNF definition. The right-hand side should match the variable type on the left-hand side unless the identifier is declared as “elm” type. “elm” type can contain any of the types among integer, string, double, or string. <identifier> and other types other than strings like int are included with <logical_arithmetic_expression> definition to prevent ambiguity.

- <set_assignment> ::= <set_identifier><assignment_op><set_operation>
  | <set_identifier><assignment_op><function_call>

Left-hand side of the assignment expression for the <set_assignment> has to be a <set_identifier>. Right-hand side can be one of the constructs in the BNF definition. <set> is included in <set_operation> to prevent ambiguity.

Sets

- <set_expression> ::= <set_identifier><set_add_op>< type >
  | <set_identifier><set_remove_op>< type >
  | <set_identifier><set_add_op><identifier>
  | <set_identifier><set_remove_op><identifier>
  | <set_delete_op><set_identifier>

<set_expression> holds for the expressions which are related to sets and which can not be written in assignment or declaration expressions. It contains three operations. One of them is adding an item to the set. ‘++’ is used as <set_add_op> and the left-hand side has to be a <set_identifier>. The right-hand side can be a type or an identifier.
One of the others is deleting an item from the set. ‘--’ is used as <set_remove_op> and the left-hand side has to be a set identifier. Right, hand-side can be a type or an identifier.

The other is deleting a set. ‘delete’ reserved keyword is used as <set_delete_op> and the expression has to start with it. Then, it has to be followed by a set identifier.

- <set_operation> ::= <set_term> | <set_operation><set_intersection_op><set_term>
  | <set_operation><set_difference_op><set_term>
  | <set_operation><set_union_op><set_term>

Set operations are the operations that return another set. It is defined as left-associative. To make this happen, it has <set_term> constructed after the operators. The operator for the intersection is '^', the operator for the union is 'U' and the operator for the difference is '\'. The selection of these symbols is because of the fact that they resemble their mathematical notations.

For precedence, it is left-associative but the parenthesis can be used because of the definition of <set_term>. Usage of parenthesis to help precedence increases the writability of Dominica.

- <set_term> ::= <set_identifier> | <set> | <LP><set_operation><RP>

<set_term> is defined for making the set operations left associative and also including precedence of parentheses. To be able to make this, <set_term> can be a <set_operation> with parenthesis.

Arithmetic Operations

- <arithmetic_operation> ::= <arithmetic_term>
  | <arithmetic_operation><sign><arithmetic_term>

Arithmetic operations in Dominica are left-associative. Also, there is the precedence of multiplication and division if there is not any parenthesis. If there are parentheses, the expression with it has precedence. To make the arithmetic operation left-associative, we include <arithmetic_term> in the BNF definition of <arithmetic_operation> and it is at the right of <sign>(+ or -).

- <arithmetic_term> ::= <arithmetic_term><mul_div_sign><arithmetic_factor>
  | <arithmetic_factor>

<arithmetic_term> construct includes <mul_div_sign> in its definition. This is for the precedence of multiplication and division to addition and subtraction. It also includes <arithmetic_factor> construct, which helps for the precedence of parentheses to multiplication and division.

- <arithmetic_factor> ::= <LP><arithmetic_operation><RP> | <identifier> | <integer>
  | <double>

<arithmetic_factor> is either one of the numerical variables or an identifier or an <arithmetic_operation> inside parenthesis. This is because of the precedence of parenthesis. To be more precise, it is either a base case for the variables or a call to an arithmetic operation which is in the form of (<arithmetic_operation).

Loops

- <loop_stmt> ::= <for_loop> | <while_loop>

Loops are divided into two categories, for loop, and while loop, in Dominica.

- <for_loop> ::= <reg_for_loop> | <set_for_loop>

For loop is also divided into two subcategories which are regular for loop and set for loop. Both of these for loops should start with the keyword “for”

- <reg_for_loop>::=
  for<LP><variable_declaration><end_of_stmt><logical_arithmetic_expression><end_of_stmt><assignment_expression><RP><LB><stmt_list><RB>
  |for<LP><variable_declaration><end_of_stmt><logical_arithmetic_expression><end_of_stmt><assignment_expression><RP><LB><RB>

After the keyword, the user should open a parenthesis pair, “(“, “)”. Inside of these parentheses, the user needs to declare a variable followed by a semicolon, “;”. For example,“int counter = 0;”. After the declaration of the variable, the loop needs a condition which is <logical_arithmetic_expression> followed by a semicolon, “;”. Finally, inside of the parentheses, the loop action, that is executed at the end of each iteration, should be determined by assigning a new expression to the initial variable. After the parentheses, curly braces, “{“, “}”, should contain at least one statement that is executed in each iteration.

- for<LP><set_identifier><column><set_identifier><RP><LB><stmt_list><RB>
  |for<LP><set_identifier><column><integer><column><set_identifier><RP><LB><stmt_list><RP>
  |for<LP><set_identifier><column><integer><column><identifier><RP><LB><stmt_list> <RP>
  |for<LP><set_identifier><column><set_identifier><RP><LB><RB>
  |for<LP><set_identifier><column><integer><column><set_identifier><RP><LB><RP>
  |for<LP><set_identifier><column><integer><column><identifier><RP><LB><RP>

Set for loop is designed in order for the user to iterate some subsets more easily. There are two possible ways to execute this loop. Inside of the curly braces, “{“, “}”, the first thing that is needed is to have a set whose subsets are wondered. After that, we can have a set that does not have to be defined previously. For example, “for(_mySet : _subsetsOfMySet)” will allow the user to print every subset of the _mySet by just accessing the _subsetsOfMySet iterator. This looks like the enhanced for loop in other programming languages. Also, another way to execute this loop is to have an integer constant between the first set and second iterator set. In this way, n element subsets can be displayed in each iteration. For example, “for(_mySet : 2: _twoElementSubsets)” will allow the user to print twoElementSubsets of _mySet variable. If the integer constant is 1, the user can have the type “elm” instead of a set in the very right of the set for loop’s parentheses. After the parentheses, the loop body should be inside of the curly braces, “{“, “}”.

- <while_loop> ::=
  while<LP><logical_arithmetic_expression><RP><LB><stmt_list><RB>
  | while<LP><logical_arithmetic_expression><RP><LB><RB>

A while loop should start with the keyword “while”. After that, the user should define a loop condition inside of parentheses, “(“, “)”. The loop condition is just a logical expression that is defined and included <logical_arithmetic_expression>. After the parentheses, the loop statements should be inside of the curly braces, “{“, “}”.

Conditions

- <conditional_stmt> ::=
  if<LP><logical_arithmetic_expression><RP><LB><stmt_list><RB>
  |if<LP><logical_arithmetic_expression><RP><LB><RB>
  |if<LP><logical_arithmetic_expression><RP><LB><stmt_list><RB>else<LB><stmt_list><RB>
  |if<LP><logical_arithmetic_expression><RP><LB><stmt_list><RB>else<LB><RB>
  |if<LP><logical_arithmetic_expression><RP><LB><RB>else<LB><stmt_list><RB>
  |if<LP><logical_arithmetic_expression><RP><LB><RB>else<LB><RB>
  |if<LP><logical_arithmetic_expression><RP><LB><stmt_list><RB>else<conditional_stmt>
  |if<LP><logical_arithmetic_expression><RP><LB><RB>else<conditional_stmt>

The conditional statements are “if”, “else”, and “else if” statements. Each conditional block should start with one of the keywords above and should specify the <logical_arithmetic_expression> inside of the parenthesis, “(“, “)”. After that, the whole block must be inside of curly braces opening and closing, “{“, “}”. <logical_arithmetic_expression> allows users to write if blocks like “if(5 + 2)”. However, this decreases the reliability but increases writability. This is a language design trade-off we encountered to resolve ambiguity. If we had not merged logical and arithmetic expressions, the users would not have put any identifier, including bool, inside of the if statement. Also, if statements like “if(5 + 2)” are also allowed in C group languages in light of our observations. Value zero is considered as “false” and others are considered as “true”.

Logical Expressions

- <logical_arithmetic_expression> ::= <regular_logic_arithmetic> | <set_logic>
  | <logical_arithmetic_expresion><and_op><regular_logic_arithmetic>
  | <logical_arithmetic_expresion><or_op><regular_logic_arithmetic>
  | <logical_arithmetic_expresion><and_op><set_logic>
  | <logical_arithmetic_expresion><or_op><set_logic>

A <logical_arithmetic_expression> consists of one or more <regular_logic_arithmetic> and <set_logic> following each other, separated by an <or_op> (which is “||”) or by an <and_op> (which is “&&”). <set_logic> and <regular_logic> are separated because sets cannot be used with the same operators as other types. Each logical expression value may be inverted if an

<exclamation_mark>(which is “!”) is placed before it; then if the value of the logical expression is true it will become false, and vice versa.

- <regular_logic_arithmetic> ::= <bool> | <bool><equals_op><bool>
  | <bool><not_equals_op><bool> | <arithmetic_operation>
  | <exclamation_mark><LP><logical_arithmetic_expression><RP>
  | <arithmetic_operation><equals_op><bool>
  | <arithmetic_operation><not_equals_op><bool>
  | <arithmetic_operation><equals_op><arithmetic_operation>
  | <arithmetic_operation><not_equals_op><arithmetic_operation>
  | <arithmetic_operation><greater_equal_op><arithmetic_operation>
  | <arithmetic_operation><less_equal_op><arithmetic_operation>
  | <arithmetic_operation><greater_than_op><arithmetic_operation>
  | <arithmetic_operation><less_than_op><arithmetic_operation>

<regular_logic> uses <arithmetic_operation> or <bool> as operand. The operators are <equals_op> (“==”),<not_equals_op> (“!”) <greater_equal_op>, <less_equal_op>, <greater_than_op> and <less_than_op> (respectively “>=”, “<=”, “>” and “<”). Since <arithmetic_operation> includes <identifier> in its definition, it is not repeated separately to prevent ambiguity. Of course, out of the scope of grammar definition, compared types should match and comparisons have to be logical (example: bool <= bool does not make any sense).

- <set_logic> ::= <identifier><set_is_an_element_of><set>
  | <type><set_is_an_element_of><set>
  | <identifier><set_is_an_element_of><set_identifier>
  | <type><set_is_an_element_of><set_identifier>
  | <set_operation><set_is_subset><set_operation>
  | <set_operation><set_is_superset><set_operation>
  | <set_operation><set_are_disjoint><set_operation>

The operator <set_is_an_element_of> ‘E’ is used to know if the <identifier> or the <type> placed before it, belongs to the set placed after. The operator <set_is_subset> ‘C’ is used when we have to know whether all the elements of the set placed before it also belongs to the set after. ‘@’ operator is the contrary, it is used to know if the first set is a superset of the second one. And finally, the <set_are_disjoint> operator ‘D’ is to know if two sets are disjoint. These symbols enhance the readability of the language because they look like mathematical ones, but also decrease the writability, forcing us to have <identifier> that are at least 2-characters-long.

Functions

- <function> ::=
  <type_name><function_identiifer><LP><RP><LB><stmt_list><return><identifier><end_of_stmt><stmt_list><RB>
  |<type_name><function_identiifer><LP><RP><LB><stmt_list><return><identifier><end_of_stmt><RB>
  |<type_name><function_identiifer><LP><RP><LB><return><identifier><end_of_stmt><stmt_list><RB>

|<type_name><function_identiifer><LP><RP><LB><return><identifier><end_of_st
mt><RB>
|<type_name><function_identiifer><LP><RP><LB><stmt_list><return><type><end
_of_stmt><stmt_list><RB>
|<type_name><function_identiifer><LP><RP><LB><stmt_list><return><type><end
_of_stmt><RB>
|<type_name><function_identiifer><LP><RP><LB><return><type><end_of_stmt><
stmt_list><RB>
|<type_name><function_identiifer><LP><RP><LB><return><type><end_of_stmt><
RB>
|<type_name><function_identiifer><LP><param_list><RP><LB><stmt_list><return
><identifier><end_of_stmt><stmt_list><RB>
|<type_name><function_identiifer><LP><param_list><RP><LB><stmt_list><return
><identifier><end_of_stmt><RB>
|<type_name><function_identiifer><LP><param_list><RP><LB><return><identifier
><end_of_stmt><stmt_list><RB>
|<type_name><function_identiifer><LP><param_list><RP><LB><return><identifier
><end_of_stmt><RB>
|<type_name><function_identiifer><LP><param_list><RP><LB><stmt_list><return
><type><end_of_stmt><stmt_list><RB>
|<type_name><function_identiifer><LP><param_list><RP><LB><stmt_list><return
><type><end_of_stmt><RB>
|<type_name><function_identiifer><LP><param_list><RP><LB><return><type><en
d_of_stmt><stmt_list><RB>
|<type_name><function_identiifer><LP><param_list><RP><LB><return><type><en
d_of_stmt><RB>
|<set_type><function_identiifer><LP><RP><LB><stmt_list><return><set_identifier
><end_of_stmt><stmt_list><RB>
|<set_type><function_identiifer><LP><RP><LB><stmt_list><return><set_identifier
><end_of_stmt><RB>
|<set_type><function_identiifer><LP><RP><LB><return><set_identifier><end_of_st
mt><stmt_list><RB>
|<set_type><function_identiifer><LP><RP><LB><return><set_identifier><end_of_st
mt><RB>
|<set_type><function_identiifer><LP><RP><LB><stmt_list><return><set><end_of_
stmt><stmt_list><RB>
|<set_type><function_identiifer><LP><RP><LB><stmt_list><return><set><end_of_
stmt><RB>
|<set_type><function_identiifer><LP><RP><LB><return><set><end_of_stmt><stmt
_list><RB>
|<set_type><function_identiifer><LP><RP><LB><return><set><end_of_stmt><RB>
|<type_name><function_identiifer><LP><param_list><RP><LB><stmt_list><return
><set_identifier><end_of_stmt><stmt_list><RB>

|<set_type><function_identiifer><LP><param_list><RP><LB><stmt_list><return><set_identifier><end_of_stmt><RB>
|<set_type><function_identiifer><LP><param_list><RP><LB><return><set_identifier><end_of_stmt><stmt_list><RB>
|<set_type><function_identiifer><LP><param_list><RP><LB><return><set_identifier><end_of_stmt><RB>
|<set_type><function_identiifer><LP><param_list><RP><LB><stmt_list><return><set><end_of_stmt><stmt_list><RB>
|<set_type><function_identiifer><LP><param_list><RP><LB><stmt_list><return><set><end_of_stmt><RB>
|<set_type><function_identiifer><LP><param_list><RP><LB><return><set><end_of_stmt><stmt_list><RB>
|<set_type><function_identiifer><LP><param_list><RP><LB><return><set><end_of_stmt><RB>
|<void><function_identifier><LP><RP><LB><stmt_list><RB>
|<void><function_identifier><LP><RP><LB><RB>
|<void><function_identifier><LP><RP><LB><stmt_list><return><end_of_stmt><stmt_list><RB>
|<void><function_identifier><LP><RP><LB><return><end_of_stmt><stmt_list><RB>
|<void><function_identifier><LP><RP><LB><stmt_list><return><end_of_stmt><RB>
|<void><function_identifier><LP><RP><LB><stmt_list><end_of_stmt><RB>
|<void><function_identifier><LP><param_list><RP><LB><stmt_list><RB>
|<void><function_identifier><LP><param_list><RP><LB><RB>
|<void><function_identifier><LP><param_list><RP><LB><stmt_list><return><end_of_stmt><stmt_list><RB>
|<void><function_identifier><LP><param_list><RP><LB><return><end_of_stmt><stmt_list><RB>
|<void><function_identifier><LP><param_list><RP><LB><stmt_list><return><end_of_stmt><RB>
|<void><function_identifier><LP><param_list><RP><LB><stmt_list><end_of_stmt><RB>

Dominica's function declaration should start with the return type which can be a set, regular variables, or void for not returning anything. After that, we need to have the indicator "func_" at the beginning of the naming of the function. After that we open left parentheses, "(" to start defining parameters that the function will get. After defining zero or more parameters, we can close the parentheses with right parentheses, ")". After that, the function block should be inside of curly braces with at least one statement. The return reserved word is like a break statement in Java, Python, and C group languages. Here is an example of a function declaration:

```
int func_double(int myInt)
```

```
{
        int newInt = myInt * 2;
        return newInt;
}
```

This implementation is close to some well-known languages. Therefore, it increases readability, writability, and reliability.

- <function_call> ::= <function_identifier><LP><RP>
  | <function_identifier><LP><arg_list><RP>
  | <primitive_function_call>

<function_call> may be used when calling a function that was declared. To call a function, the function name must be written in the first place followed by parentheses that contain or do not contain an argument list. Moreover, it can also be a call to a primitive function already defined in Dominica.

- <primitive_function_call> ::= n<LP><set_operation><RP>
  | pow<LP><set_operation><RP>
  | cartesian<LP><set_operation><comma><set_operation><RP>
  | print<LP><set_operation><RP>
  | print<LP><string><RP>
  | print<LP><identifier><RP>
  | scan<LP><identifier><RP>
  | scan<LP><set_identifier><RP>
  | fread<LP><string><RP>
  | fwrite<LP><string><comma><string><RP>
  | fwrite<LP><set_operation><comma><string><RP>
  | fwrite<LP><identifier><comma><string><RP>

<primitive_function_call> refers to a call to any of the primitive functions. They are called the same way as non-primitive functions.

n function returns the cardinality (number of elements) of the set given in the argument.

the cartesian function returns the cartesian product of the two sets given as arguments and displays them on the console. Since we are not supposed to have any set in a set, we display the results instead of returning a set.

pow function displays all the power sets of the set given as an argument on the console. Since we are not supposed to have any set in a set, we display the results instead of returning a set.

The print function displays every element of the set given in the argument on the console.

If an <identifier> is put as an argument, it will just display its value on the console.

The scan function is used for taking an input from the user on the console.

The fread function reads the file given as an argument and returns its result.

And finally, fwrite function is used to write a <string>, <identifier> or a <set_operation> to a file. Also note that <set_operation> contains the set_identifier itself.

Lists

- <arg_list> ::= <logical_arithmetic_expression> | <string> | <set_identifier>
  | <logical_arithmetic_expressin> <comma> <arg_list>
  | <string> <comma> <arg_list>
  | <set_identifier> <comma> <arg_list>

In Dominica, while calling primitive or user-defined functions, users may pass some arguments to the function. Arguments may be the regular identifiers such as integer, string, bool, double, or elm as well as the set identifiers which start with the indicator "_". More than one argument should be separated by commas.

- <param_list> ::= <type_name><identifier> | <set_type><set_identifier>
  | <type_name><identifier><comma><param_list>
  | <set_type><set_identifier><comma><param_list>

In Dominica, while defining some functions, the user may want to have some parameters that is passed when the function is called. All of the parameters may be the regular identifiers as well as the set identifiers. All of the parameters should define the type followed by the identifier. If there is more than one parameter, they should be separated by commas.

Comments

- <comment> ::= <line_comment_start><string_term>

All of the comments in Dominica have to start with <line_comment_start>, which is the symbol '#'. This obligation increases the readability and reliability because it prevents mixing the identifiers and comments. Then, it has to be followed by a <string_term>.

Readability, Writability, and Reliability of Dominica

To increase readability in Dominica, semicolon char at the end of expressions is forced. In that way, it is easier to understand when a new expression starts. However, this decreases the writability since an extra character is required for the programmer.

Also, to increase readability, we defined the symbols of set operations close to their mathematical representations(e.g., set union symbol is U). This also increases the writability since most programmers are familiar with math and symbols are in one character length. To make distinguishable this one char length symbols from the variable identifiers, a minimum two-character length rule is forced for identifiers (e.g., “x” is not a valid name for an identifier but “xa” is valid). Moreover, identifiers have to start with a letter character to prevent them from being full of digits. Without this, there would be confusion about if they are integers or identifiers. Therefore this rule increases reliability.

Furthermore, the cardinality primitive function is defined as n(<set_identifier>), which is exactly the same with mathematical notation. Therefore this increases readability and writability.

To increase reliability in Dominica, we have <identifier>, <set_identifier>, <func_identifier> separately. <set_identifier> starts with “_” character and <func_identifier> starts with “func_”. This rule also increases the readability but it decreases the writability since the programmer has to obey more rules.

To increase writability in Dominica, we let the programmer write complex logical and arithmetic expressions which include more than one operator and parentheses for precedence. Also, they are able to write a specially defined for loop specific to sets. However, these nice futures decrease the readability of Dominica. In for loops, it also decreases the reliability.

Overall, Dominica is mostly a readable and reliable language. To make these possible, we compromised the writability of Dominica.

How Precedence and Ambiguity Are Solved in Dominica

To solve the precedence in arithmetic and logical operations, we followed what was taught to us in the lectures. We created sub-variables where associativity is important eg., <arithmetic_term>, <arithmetic_factor> and <regular_logic_arithmetic>. We converted our initial grammar in part 1 into the yacc specification file, and compiled it. It gave 20 reduce/reduce conflicts at first, we solved these issues by interpreting the y.output file. Most of the conflicts are caused by reaching the token IDENTIFIER from two possible parse trees. That is why we merged logical and arithmetic expression variables in one variable, <logical_arithmetic_expression>.

For precedence in arithmetic operations, priority is on the operations in parentheses firstly, then it is followed by multiplication/division and then summation/subtraction. For precedence in logical operations, precedence is in parentheses firstly and then it is followed by && (and) and || (or) operations. For operations with the same priorities, left associativity is considered. We have designed the set operations as left-associative as it is in math.

Non-Trivial Tokens of Dominica

MAIN: Token to indicate the start of the program.

IF: Token to indicate conditional statement if.

ELSE: Token to indicate conditional statement else.

WHILE: Token to indicate while loop.

FOR: Token to indicate for loop

VOID: Token to define void return type functions.

RETURN: Token to return variable in functions.

BOOL_VALUE: Token to indicate bool values “true” and “false”.

LB: Token to indicate left braces “{“.

RB: Token to indicate right braces “}”.

LP: Token to indicate left parenthesis “(“.

RP: Token to indicate right parenthesis “)”.

COMMA: Token to indicate comma “,”.

COLUMN: Token to indicate column “:”.

NOT_OP: Token to indicate not operation symbol “!” for logical operations.

AND_OP: Token to indicate and operation symbol “&&” for logical operations.

OR_OP: Token to indicate or operation symbol “||” for logical operations.

ASSIGNMENT_OP: Token to indicate assignment operations symbol “=”.

EQUALS_OP: Token to indicate equals operation symbol “==” for logical operations.

NOT_EQUALS_OP: Token to indicate not equals operation symbol “!=” for logical operations.

GREATER_THAN_OP: Token to indicate greater than operation symbol “>” for logical operations.

GREATER_EQUAL_OP: Token to indicate greater than or equal operation symbol “>=” for logical operations.

LESS_THAN_OP: Token to indicate less than operation symbol “<” for logical operations.

LESS_EQUAL_OP: Token to indicate less than or equal operation symbol “<=” for logical operations.

VARIABLE_TYPE: Token to indicate reserved words “int”, “double”, “string”, “bool, for variables.

SET_ELEMENT_TYPE: Token to indicate reserved word “elm” for element variable, which can be any type among integer, double, string and bool.

SET_VARIABLE_TYPE: Token to indicate reserved word “set” for set variable.

NEW_LINE: Token to indicate newlines.

ADDITION_OP: Token to indicate summation symbol “+” for arithmetic operations.

SUBTRACTION_OP: Token to indicate subtraction symbol “-“ for arithmetic operations.

DIVISION_OP: Token to indicate division symbol “\” for arithmetic operations.

MULTIPLICATION_OP: Token to indicate multiplication symbol “*” for arithmetic operations.

SET_ADDITION_OP: Token to indicate the symbol “++” for adding a new element to a set.

SET_REMOVE_OP: Token to indicate the symbol “--“ for removing an element from the set.

SET_UNION_OP: Token to indicate the symbol “U” for taking the union of two sets.

SET_INTERSECTION_OP: Token to indicate the symbol “^” for taking the intersection of two sets.

SET_DIFFERENCE_OP: Token to indicate the symbol “\” for finding the difference of two sets.

SET_CARDINALITY: Token to indicate the reserved word “n” for primitive function identifier of cardinality function.

SET_POWER_SET: Token to indicate reserved word “pow” for primitive function identifier of the power function.

SET_CARTESIAN: Token to indicate reserved word “cartesian” for primitive function identifier of cartesian function.

SET_DELETE_SET: Token to indicate the reserved word “delete” for deleting a set.

SET_IS_SUBSET_OF: Token to indicate the symbol “C” for set logic operations which checks if the first set is a subset of the second set.

SET_IS_SUPERSET_OF: Token to indicate the symbol “@” for set logic operations which checks if the first set is the superset of the second set.

SET_IS_AN_ELEMENT_OF: Token to indicate the symbol “E” for set logic operations. which checks if a variable is the element of a given set.

SET_ARE_DISJOINT: Token to indicate the symbol “D” for set logic operations which checks if the given sets are disjoint.

SCAN_INPUT: Token to indicate reserved word “scan” for primitive function identifier of scan function, which takes input from the user.

FILE_WRITE: Token to indicate the reserved word “fwrite” for primitive function identifier of file write function, which writes data to a file.

FILE_READ: Token to indicate the reserved word “fread” for primitive function identifier of file read function, which reads data from a file.

PRINT: Token to indicate the reserved word “print” for primitive function identifier of the print function, which prints the string or set content to console.

END_OF_STMT: Token to indicate symbol “;” which has to be put at the end of statements.

INTEGER_CONST: Token to indicate integers.

DOUBLE_CONST: Token to indicate doubles.

COMMENT: Token to indicate comments.

STRING_CONST: Token to indicate strings.

FUNCTION_IDENTIFIER: Token to indicate function identifiers that are for only functions. They start with “func_”.

SET_IDENTIFIER: Token to indicate set identifiers which are only for sets. They start with “_”.

IDENTIFIER: Token to indicate variable identifiers. They have to start with a letter and they have to be a minimum of two-character length.