Department of Computer Engineering
CS 315 - Programming Languages
Project 1

CardiLan - An Imperative Set Manipulation Language

Team 3:

Yiğit Gürses 21702746
Ahmet Faruk Ulutaş 21803717
Efe Korkmazhan 21802712

Spring 2022

Table of Contents

1 Name of the Language

The name of our language is *CardiLan*.

2 BNF Description of the Language

2.1 Program Structure

1. ⟨program⟩ ⟶ ⟨imports⟩ ⟨stmts⟩
   | ⟨imports⟩ ⟨stmts⟩ ⟨main⟩

2. ⟨main⟩ ⟶ main ⟨stmts⟩ endmain

3. ⟨imports⟩ ⟶ ⟨empty⟩
   | import{⟨empty⟩}
   | import{⟨import_list⟩}

4. ⟨importList⟩ ⟶ ⟨string⟩
   | ⟨import_list⟩ , ⟨string⟩

5. ⟨stmts⟩ ⟶ ⟨empty⟩
   | ⟨stmts⟩ ⟨stmt⟩

6. ⟨stmt⟩ ⟶ ⟨if_stmt⟩
   | ⟨declaration_stmt⟩
   | ⟨return_stmt⟩
   | ⟨loop_stmt⟩
   | ⟨func_call_stmt⟩
   | ⟨comment_stmt⟩
   | ⟨assign_stmt⟩
   | ⟨expr_stmt⟩

2.2 Statements

7. ⟨if_stmt⟩ ⟶ if ⟨logic_expr⟩ : ⟨stmts⟩ ⟨elseif_list⟩ endif
   | if ⟨logic_expr⟩ : ⟨stmts⟩ ⟨elseif_list⟩ else ⟨stmts⟩ endif

8. ⟨elseif_list⟩ ⟶ ⟨empty⟩
   | ⟨elseif_list⟩ elseif ⟨logic_expr⟩ : ⟨stmts⟩

9. ⟨declaration_stmt⟩ ⟶ ⟨var_declaration⟩
   | ⟨set_declaration⟩
   | ⟨func_declaration⟩

10. ⟨var_declaration⟩ ⟶ ⟨var_type⟩ ⟨var_declaration_list⟩;

11. ⟨var_declaration_list⟩ ⟶ ⟨var_declaration_list_elem⟩
    | ⟨var_declaration_list⟩ , ⟨var_declaration_list_elem⟩

12. ⟨var_declaration_list_elem⟩ ⟶ ⟨var_id⟩
    | ⟨var_id⟩ ⟨assign_op⟩ ⟨expr⟩

13. ⟨set_declaration⟩ ⟶ set<⟨var_type⟩>⟨set_declaration_list⟩;

14. ⟨set_declaration_list⟩ ⟶ ⟨set_declaration_list_elem⟩
    | ⟨set_declaration_list⟩ , ⟨set_declaration_list_elem⟩

15. ⟨set_declaration_list_elem⟩ ⟶ ⟨set_id⟩
    | ⟨set_id⟩ ⟨assign_op⟩ ⟨set_expr⟩

16. ⟨func_declaration⟩ ⟶ func ⟨func_name⟩ (⟨param_list⟩) : ⟨stmts⟩ endfunc
    | func ⟨func_return_list⟩ ⟨func_name⟩ (⟨param_list⟩) : ⟨stmts⟩ endfunc

17. ⟨func_return_list⟩ ⟶ ⟨func_return_list_elem⟩
    | ⟨func_return_list⟩ , ⟨func_return_list_elem⟩

18. ⟨func_return_list_elem⟩ ⟶ ⟨var_type⟩
| set<⟨var_type⟩>

19. ⟨param_list⟩ ⟶ ⟨param_list_elem⟩
| ⟨param_list⟩ , ⟨param_list_elem⟩

20. ⟨param_list_elem⟩ ⟶ ⟨var_declaration_list_elem⟩
| ⟨set_declaration_list_elem⟩

21. ⟨return_stmt⟩ ⟶ return ⟨expr_list⟩;

22. ⟨expr_list⟩ ⟶ ⟨expr_list_elem⟩
| ⟨expr_list⟩ , ⟨expr_list_elem⟩

23. ⟨expr_list_elem⟩ ⟶ ⟨expr⟩
| ⟨set_expr⟩
| ⟨general_term⟩

24. ⟨loop_stmt⟩ ⟶ ⟨while_loop⟩
| ⟨dowhile_loop⟩
| ⟨for_loop⟩

25. ⟨general_logic_expr⟩ ⟶ ⟨logic_expr⟩ | ⟨general_term⟩

26. ⟨while_loop⟩ ⟶ while ⟨general_logic_expr⟩ : ⟨stmts⟩ endwhile

27. ⟨dowhile_loop⟩ ⟶ do : ⟨stmts⟩ while ⟨general_logic_expr⟩;

28. ⟨for_loop⟩ ⟶ for ⟨general_logic_expr⟩ , ⟨assignments⟩ : ⟨stmts⟩ endfor

29. ⟨func_call_stmt⟩ ⟶ ⟨func_call⟩ ;

30. ⟨func_call⟩ ⟶ ⟨func_name⟩()
| ⟨func_name⟩(⟨expr_list⟩)
| ⟨prim_func_call⟩

31. ⟨general_expr⟩ ⟶ ⟨expr⟩ | ⟨general_term⟩

32. ⟨assign_stmt⟩ ⟶ ⟨assignments⟩ ;

33. ⟨assignment⟩ ⟶ ⟨var_id⟩ ⟨assign_op⟩ ⟨general_expr⟩
| ⟨set_id⟩ ⟨assign_op⟩ ⟨set_expr⟩

34. ⟨assignments⟩ ⟶ ⟨assignment⟩
| ⟨var_id⟩ , ⟨assignments⟩ , ⟨general_expr⟩
| ⟨set_id⟩ , ⟨assignments⟩ , ⟨set_expr⟩
| ⟨id_list⟩ , ⟨assign_op⟩ , ⟨func_call⟩

35. ⟨id_list⟩ ⟶ ⟨var_id⟩ | ⟨set_id⟩ | ⟨id_list⟩ , ⟨var_id⟩ | ⟨id_list⟩ , ⟨set_id⟩

36. ⟨expr_stmt⟩ ⟶ ⟨expr⟩ ; | ⟨set_expr⟩ ;

2.3 Expressions

37. ⟨expr⟩ ⟶ ⟨arith_expr⟩
| ⟨logic_expr⟩

38. ⟨general_term⟩ ⟶ ⟨var_id⟩ | ⟨func_call⟩ | (⟨general_term⟩)

39. ⟨arith_general_term⟩ ⟶ ⟨general_term⟩ | - ⟨general_term⟩

40. ⟨arith_expr⟩ ⟶ ⟨arith_term_1⟩
| ⟨arith_expr⟩ ⟨additive_op⟩ ⟨arith_term_1⟩
| ⟨arith_expr⟩ ⟨additive_op⟩ ⟨arith_general_term⟩
| ⟨arith_general_term⟩ ⟨additive_op⟩ ⟨arith_general_term⟩
| ⟨arith_general_term⟩ ⟨additive_op⟩ ⟨arith_term_1⟩

41. ⟨arith_term_1⟩ ⟶ ⟨arith_term_2⟩
| ⟨arith_term_2⟩ ⟨multiplicative_op⟩ ⟨arith_term_1⟩
| ⟨arith_term_2⟩ ⟨multiplicative_op⟩ ⟨arith_general_term⟩
| ⟨arith_general_term⟩ ⟨multiplicative_op⟩ ⟨arith_general_term⟩
| ⟨arith_general_term⟩ ⟨multiplicative_op⟩ ⟨arith_term_1⟩

42. ⟨arith_term_2⟩ ⟶ ⟨arith_term_3⟩
| ⟨arith_term_3⟩ ⟨exp_op⟩ ⟨arith_term_2⟩
| ⟨arith_term_3⟩ ⟨exp_op⟩ ⟨arith_general_term⟩
| ⟨arith_general_term⟩ ⟨exp_op⟩ ⟨arith_general_term⟩
| ⟨arith_general_term⟩ ⟨exp_op⟩ ⟨arith_term_2⟩

43. ⟨arith_term_3⟩ ⟶ (⟨arith_expr⟩)
| ⟨numeric⟩
| ⟨set_expr⟩.cardinality()
| - ⟨arith_expr⟩

44. ⟨logic_expr⟩ ⟶ ⟨logic_term_1⟩
| ⟨logic_expr⟩ ⟨binary_logic_op⟩ ⟨logic_term_1⟩
| ⟨logic_expr⟩ ⟨binary_logic_op⟩ ⟨general_term⟩
| ⟨general_term⟩ ⟨binary_logic_op⟩ ⟨general_term⟩
| ⟨general_term⟩ ⟨binary_logic_op⟩ ⟨logic_term_1⟩

45. ⟨logic_term_1⟩ ⟶ ⟨logic_term_2⟩
| ⟨unary_logic_op⟩ ⟨logic_term_2⟩
| ⟨unary_logic_op⟩ ⟨general_term⟩

46. ⟨logic_term_2⟩ ⟶ (⟨logic_expr⟩)
| ⟨boolean⟩
| ⟨set_expr⟩.subsetof(⟨set_expr⟩)
| ⟨set_expr⟩.properSubsetof(⟨set_expr⟩)
| ⟨set_expr⟩.supersetof(⟨set_expr⟩)
| ⟨set_expr⟩.contains(⟨expr⟩)
| ⟨set_expr⟩.isempty()
| ⟨set_expr⟩.equals(⟨set_expr⟩)
| ⟨relational_expr⟩

47. ⟨general_set_term⟩ ⟶ ⟨set_id⟩ | ⟨func_call⟩ | (⟨general_set_term⟩)

48. ⟨set_expr⟩ ⟶ ⟨set_id⟩.⟨set_func_call⟩
| ⟨set_id⟩⟨powerset_op⟩⟨arith_expr⟩
| ⟨prim_set_declaration⟩
| ⟨foreach_set_declaration⟩
| (⟨set_expr⟩)
| set_input()
| ⟨general_set_term⟩

49. ⟨set_func_call⟩ ⟶ insert(⟨expr⟩)
| remove(⟨expr⟩)
| union(⟨set_expr⟩)
| intersect(⟨set_expr⟩)
| cartesian(⟨set_expr⟩)
| diff(⟨set_expr⟩)

50. ⟨prim_set_declaration⟩ ⟶ { }
| {⟨element_list⟩}

51. ⟨element_list⟩ ⟶ ⟨element⟩
| ⟨element_list⟩ , ⟨element⟩

52. ⟨element⟩ ⟶ ⟨expr⟩

53. ⟨foreach_set_declaration⟩ ⟶ ⟨expr⟩ foreach(⟨foreach_list⟩)

54. ⟨foreach_list⟩ ⟶ ⟨foreach_list_element⟩
| ⟨foreach_list⟩ , ⟨foreach_list_element⟩

55. ⟨foreach_list_element⟩ ⟶ ⟨var_id⟩ in ⟨set_id⟩

56. ⟨relational_expr⟩ ⟶ ⟨relational_expr⟩ ⟨relational_op⟩ ⟨general_expr⟩

2.4 Primitive Functions

57. ⟨prim_func_call⟩ ⟶ ⟨delete_call⟩
| ⟨print_call⟩
| ⟨input_call⟩
| ⟨file_input_call⟩
| ⟨file_output_call⟩

58. ⟨delete_call⟩ ⟶ delete(⟨set_id⟩)

59. ⟨print_call⟩ ⟶ print(⟨set_expr⟩) | print(⟨expr⟩) | print(⟨general_term⟩)

60. ⟨input_call⟩ ⟶ input()

61. ⟨file_input_call⟩ ⟶ file_input(⟨string⟩)

62. ⟨file_output_call⟩ ⟶ file_output(⟨set_expr⟩, ⟨string⟩)
| file_output(⟨general_expr⟩, ⟨string⟩)

2.5 Trivial Variables Derived From Lex Tokens

63. ⟨var_type⟩ ⟶ INT_TYPE | FLOAT_TYPE | BOOL_TYPE | CHAR_TYPE | STRING_TYPE
| SET_TYPE | GENERIC

64. ⟨numeric⟩ ⟶ INT | FLOAT

65. ⟨string⟩ ⟶ STRING

66. ⟨comment_stmt⟩ ⟶ COMMENT

67. ⟨boolean⟩ ⟶ TRUE | FALSE

68. ⟨assign_op⟩ ⟶ ASSIGNOP

69. ⟨relational_op⟩ ⟶ LESS | LEQ | EQ | GEQ | GREATER

70. ⟨binary_logic_op⟩ ⟶ AND | OR | XOR | IFF | IMP

71. ⟨unary_logic_op⟩ ⟶ NOT

72. ⟨additive_op⟩ ⟶ PLUS | MINUS

73. ⟨multiplicative_op⟩ ⟶ STAR | SLASH | PERCENT

74. ⟨exp_op⟩ ⟶ DOUBLESTAR

75. ⟨powerset_op⟩ ⟶ DOUBLESTAR

76. ⟨var_id⟩ ⟶ VARID

77. ⟨set_id⟩ ⟶ SETID

78. ⟨func_name⟩ ⟶ ⟨var_id⟩

79. ⟨empty⟩ ⟶

2.6 Terminals

80. ⟨<⟩ ⟶ ⟨smaller than operator⟩
81. ⟨>⟩ ⟶ ⟨greater than operator⟩
82. ⟨<=⟩ ⟶ ⟨smaller than or equal to operator⟩
83. ⟨>=⟩ ⟶ ⟨greater than or equal to operator⟩
84. ⟨=⟩ ⟶ ⟨equal operator⟩
85. ⟨!=⟩ ⟶ ⟨not equal operator⟩
86. ⟨AND⟩ ⟶ ⟨and operator⟩
87. ⟨OR⟩ ⟶ ⟨or operator⟩
88. ⟨XOR⟩ ⟶ ⟨xor operator⟩
89. ⟨NOT⟩ ⟶ ⟨not operator⟩
90. ⟨IFF⟩ ⟶ ⟨iff operator⟩
91. ⟨IMP⟩ ⟶ ⟨imp operator⟩
92. ⟨(⟩ ⟶ ⟨left parantheses⟩
93. ⟨)⟩ ⟶ ⟨right parentheses⟩
94. ⟨{⟩ ⟶ ⟨left bracket⟩
95. ⟨}⟩ ⟶ ⟨right bracket⟩
96. ⟨//⟩ ⟶ ⟨double slash⟩
97. ⟨**⟩ ⟶ ⟨double star⟩
98. ⟨%⟩ ⟶ ⟨percent⟩
99. ⟨/⟩ ⟶ ⟨slash⟩
100. ⟨.⟩ ⟶ ⟨dot⟩
101. ⟨,⟩ ⟶ ⟨comma⟩
102. ⟨:⟩ ⟶ ⟨colon⟩
103. ⟨;⟩ ⟶ ⟨semicolon⟩
104. ⟨_⟩ ⟶ ⟨underscore⟩

2.7 Non-terminals

2.8 Types

Descriptions of non-terminal literals are explained below. Other than the terminals above all the other literals are non-terminal. Starting from the ¡program¿ until the most spesific literal they are all non-terminals.

3 Explanations of Language Constructs

3.1 Types

CardiLan has the following built-in types: int, float, string, boolean, set, generic and char. Sets are an abstraction of bounded sets in mathematics. All variables except set are actually elements. Thus, elements can contain all symbols that a finite mathematical set can have.

3.2 Program Structure

1. ⟨program⟩ ⟶ ⟨imports⟩ ⟨stmts⟩
   | ⟨imports⟩ ⟨stmts⟩ ⟨main⟩
   A program consists of import, a pre-main section, and then a main section. Imports are for using outside library and piece of code. The stmts section before main is for function declarations and general expressions. The main section is for starting the program and contains code blocks, which are also composed of stmts.

2. ⟨main⟩ ⟶ main ⟨stmts⟩ endmain
   The program is a list of statements written from main to endmain.

3. ⟨imports⟩ ⟶ ⟨empty⟩
   | import{⟨empty⟩}
   | import{⟨import_list⟩}
   Import keyword may not be used. It can be used and left empty. Or, code can be exported by adding a list of libraries inside curly brackets.

4. ⟨importList⟩ ⟶ ⟨string⟩
   | ⟨import_list⟩ , ⟨string⟩
   The import list consists of a string or comma-separated strings written one after the other.

5. ⟨stmts⟩ ⟶ ⟨empty⟩
   | ⟨stmts⟩ ⟨stmt⟩
   Statements can be empty or consist of cascading statements.

6. ⟨stmt⟩ ⟶ ⟨if_stmt⟩
   | ⟨declaration_stmt⟩
   | ⟨return_stmt⟩
   | ⟨loop_stmt⟩
   | ⟨func_call_stmt⟩
   | ⟨comment_stmt⟩
   | ⟨assign_stmt⟩
   | ⟨expr_stmt⟩
   Statement can be a condition, declaration, return, loop, func call, comment, assign, or expr statement.

3.3 Statements

1. ⟨if_stmt⟩ ⟶ if ⟨logic_expr⟩ : ⟨stmts⟩ ⟨elseif_list⟩ endif
   | if ⟨logic_expr⟩ : ⟨stmts⟩ ⟨elseif_list⟩ else ⟨stmts⟩ endif
   The if statement contains a list of logical statements, colon statements, and elseif, beginning with an if statement, followed by a logical statement. It terminates when it sees the endif keyword.

2. ⟨elseif_list⟩ ⟶ ⟨empty⟩
   | ⟨elseif_list⟩ elseif ⟨logic_expr⟩ : ⟨stmts⟩
   The elseif list can be empty, or the elseif logical expression can be a block containing colons and statements, respectively. Elseifs can be listed one after the other.

3. ⟨declaration_stmt⟩ ⟶ ⟨var_declaration⟩
   | ⟨set_declaration⟩
   | ⟨func_declaration⟩
   The Declaration Statement can be a variable, set, or function declaration.

4. ⟨var_declaration⟩ ⟶ ⟨var_type⟩ ⟨var_declaration_list⟩;
   Variable declaration can be given with variable type and variable list.

5. ⟨var_declaration_list⟩ ⟶ ⟨var_declaration_list_elem⟩
   | ⟨var_declaration_list⟩ , ⟨var_declaration_list_elem⟩
   The variable declaration list can be a list element. In addition, more than one declaration list element can be defined, separated by commas.

6. ⟨var_declaration_list_elem⟩ ⟶ ⟨var_id⟩
   | ⟨var_id⟩ ⟨assign_op⟩ ⟨expr⟩
   Variable declaration list element can consist of variable id or variable id, assign operator and expr.

7. ⟨set_declaration⟩ ⟶ set<⟨var_type⟩>⟨set_declaration_list⟩;
   Set declaration can be defined as set⟨generic or variable type⟩(giving a set declaration list).

8. ⟨set_declaration_list⟩ ⟶ ⟨set_declaration_list_elem⟩
   | ⟨set_declaration_list⟩ , ⟨set_declaration_list_elem⟩
   A set declaration list can consist of one or more than one set declaration list element separated by commas.

9. ⟨set_declaration_list_elem⟩ ⟶ ⟨set_id⟩
   | ⟨set_id⟩ ⟨assign_op⟩ ⟨set_expr⟩
   The set declaration list element can consist of set id or set id, assign operator, set expr.

10. ⟨func_declaration⟩ ⟶ func ⟨func_name⟩ (⟨param_list⟩) : ⟨stmts⟩ endfunc
    | func ⟨func_return_list⟩ ⟨func_name⟩ (⟨param_list⟩) : ⟨stmts⟩ endfunc
    Function declaration is made with keyword func, function name, left parenthesis, parameter list, right parenthesis, colon, statements, endfunc respectively.

11. ⟨func_return_list⟩ ⟶ ⟨func_return_list_elem⟩
    | ⟨func_return_list⟩ , ⟨func_return_list_elem⟩
    A function return list can consist of one or more comma-separated function return list elements.

12. ⟨func_return_list_elem⟩ ⟶ ⟨var_type⟩
    | set<⟨var_type⟩>
    The function return list element can consist of variable type, set⟨generic or variable type⟩.

13. ⟨param_list⟩ ⟶ ⟨param_list_elem⟩
    | ⟨param_list⟩ , ⟨param_list_elem⟩
    The parameter list consists of one or more comma-separated parameter list elements.

14. ⟨param_list_elem⟩ ⟶ ⟨var_declaration_list_elem⟩
    | ⟨set_declaration_list_elem⟩
    A parameter list element consists of a variable or set definition list element.

15. ⟨return_stmt⟩ ⟶ return ⟨return_list⟩;
    The return statement is created with the return keyword and the return list, respectively.

16. ⟨expr_list⟩ ⟶ ⟨expr_list_elem⟩
    | ⟨expr_list⟩ , ⟨expr_list_elem⟩
    Expression list consists of one or more comma-separated expression list elements.

17. ⟨expr_list_elem⟩ ⟶ ⟨expr⟩
    | ⟨set_expr⟩
    | ⟨general_term⟩
    Expression list element consists of expr, set expr or general term.

18. ⟨loop_stmt⟩ ⟶ ⟨while_loop⟩
    | ⟨dowhile_loop⟩
    | ⟨for_loop⟩
    The loop statement consists of while, do while, and for loops.

19. ⟨general_logic_expr⟩ ⟶ ⟨logic_expr⟩ | ⟨general_term⟩
    General logic expression, logic expression or general term.

20. ⟨while_loop⟩ ⟶ while ⟨general_logic_expr⟩ : ⟨stmts⟩ endwhile
    The while loop is created with the keyword while, the general logic expression, the colon, the statements, and the endwhile keyword, respectively.

21. ⟨dowhile_loop⟩ ⟶ do : ⟨stmts⟩ while ⟨general_logic_expr⟩;
    The do While loop is constructed with the do keyword, colon, statements, while keyword, general logic expression, and semicolon, respectively.

22. ⟨for_loop⟩ ⟶ for ⟨general_logic_expr⟩ , ⟨assignments⟩ : ⟨stmts⟩ endfor
The for loop is created with the keyword for, the general logic expression, the comma, assignments, colon, statements, and the endfor keyword, respectively.

23. ⟨func_call_stmt⟩ ⟶ ⟨func_call⟩ ; Function call statement consists of function call and semicolon respectively.

24. ⟨func_call⟩ ⟶ ⟨func_name⟩()
| ⟨func_name⟩(⟨expr_list⟩)
| ⟨prim_func_call⟩
The function call statement is executed by the function name, the left parenthesis, the expression list if any, the right parenthesis, or the primitive function call, respectively.
The comment line consists of two slash signs followed by words.

25. ⟨general_expr⟩ ⟶ ⟨expr⟩ | ⟨general_stmt⟩
General expression, respectively expression or general statement.

26. ⟨assign_stmt⟩ ⟶ ⟨assignments⟩ ;
The assignment statement consists of assignments and a semicolon, respectively.

27. ⟨assignment⟩ ⟶ ⟨var_id⟩ ⟨assign_op⟩ ⟨general_expr⟩
| ⟨set_id⟩ ⟨assign_op⟩ ⟨set_expr⟩
Assignment consists of variable name, comma, assignments, comma, generic expression, respectively. Or it consists of set name, assignment operator, set expression, respectively.

28. ⟨assignments⟩ ⟶ ⟨assignment⟩
| ⟨var_id⟩ , ⟨assignments⟩ , ⟨general_expr⟩
| ⟨set_id⟩ , ⟨assignments⟩ , ⟨set_expr⟩
| ⟨id_list⟩ ⟨assign_op⟩ ⟨func_call⟩
Assignments can be defined as assignment or variable name comma assignments comma general expression or set name comma assignments comma set expressions or id list assignment operator function call.

29. ⟨id_list⟩ ⟶ ⟨var_id⟩ | ⟨set_id⟩ | ⟨id_list⟩ , ⟨var_id⟩ | ⟨id_list⟩ , ⟨set_id⟩
List id can be defined using either variable id, set id, list id comma variable id or idlist comma set id.

30. ⟨expr_stmt⟩ ⟶ ⟨expr⟩ ; | ⟨set_expr⟩ ;
To write an expression statement, expression or set expression followed by a semicolon.

3.4 Expressions

1. ⟨expr⟩ ⟶ ⟨arith_expr⟩
| ⟨logic_expr⟩
To summarize arithmetic expressions in general, there is priority to avoid operational and logical errors in expressions. The exponential part is right associative. Other parts are left associative. Also, operations like a + funcCall + 5 are allowed. It provides ease of operation.

2. ⟨general_term⟩ ⟶ ⟨var_id⟩ | ⟨func_call⟩ | (⟨general_term⟩)
General term occurs as variable id or function call or general term.

3. ⟨arith_expr⟩ ⟶ ⟨arith_term_1⟩
| ⟨arith_expr⟩ ⟨additive_op⟩ ⟨arith_term_1⟩
| ⟨arith_expr⟩ ⟨additive_op⟩ ⟨general_term⟩
| ⟨general_term⟩ ⟨additive_op⟩ ⟨general_term⟩
| ⟨general_term⟩ ⟨additive_op⟩ ⟨arith_term_1⟩

4. ⟨arith_term_1⟩ ⟶ ⟨arith_term_2⟩
| ⟨arith_term_2⟩ ⟨multiplicative_op⟩ ⟨arith_term_1⟩
| ⟨arith_term_2⟩ ⟨multiplicative_op⟩ ⟨general_term⟩
| ⟨general_term⟩ ⟨multiplicative_op⟩ ⟨general_term⟩
| ⟨general_term⟩ ⟨multiplicative_op⟩ ⟨arith_term_1⟩

5. ⟨arith_term_2⟩ ⟶ ⟨arith_term_3⟩
   | ⟨arith_term_3⟩ ⟨exp_op⟩ ⟨arith_term_2⟩
   | ⟨arith_term_3⟩ ⟨exp_op⟩ ⟨general_term⟩
   | ⟨general_term⟩ ⟨exp_op⟩ ⟨general_term⟩
   | ⟨general_term⟩ ⟨exp_op⟩ ⟨arith_term_2⟩

6. ⟨arith_term_3⟩ ⟶ (⟨arith_expr⟩)
   | ⟨numeric⟩
   | ⟨set_expr⟩.cardinality()
   | - ⟨arith_term⟩

7. ⟨logic_expr⟩ ⟶ ⟨logic_term_1⟩
   | ⟨logic_expr⟩ ⟨binary_logic_op⟩ ⟨logic_term_1⟩
   | ⟨logic_expr⟩ ⟨binary_logic_op⟩ ⟨general_term⟩
   | ⟨general_term⟩ ⟨binary_logic_op⟩ ⟨general_term⟩
   | ⟨general_term⟩ ⟨binary_logic_op⟩ ⟨logic_term_1⟩
   It covers logic expressions, logic term1, and binary logic operations.

8. ⟨logic_term_1⟩ ⟶ ⟨logic_term_2⟩
   | ⟨unary_logic_op⟩ ⟨logic_term_2⟩
   | ⟨unary_logic_op⟩ ⟨general_term⟩
   Logic term1 covers the relationship between single expressions and boolean expressions, and single expressions and general terms.

9. ⟨logic_term_2⟩ ⟶ (⟨logic_expr⟩)
   | ⟨boolean⟩
   | ⟨set_expr⟩.subsetof(⟨set_expr⟩)
   | ⟨set_expr⟩.properSubsetof(⟨set_expr⟩)
   | ⟨set_expr⟩.supersetof(⟨set_expr⟩)
   | ⟨set_expr⟩.contains(⟨expr⟩)
   | ⟨set_expr⟩.isempty()
   | ⟨set_expr⟩.equals(⟨set_expr⟩)
   | ⟨relational_expr⟩
   Logic term2 contains the operator list containing boolean types. There are expressions such as subset, proper subset, superset, contains, isempty, equals.

10. ⟨general_set_term⟩ ⟶ ⟨set_id⟩ | ⟨func_call⟩ | (⟨general_set_term⟩)
    General set term can be a set id, or function call.

11. ⟨set_expr⟩ ⟶ ⟨set_id⟩.⟨set_func_call⟩
    | ⟨set_id⟩⟨powerset_op⟩⟨arith_expr⟩
    | ⟨prim_set_declaration⟩
    | ⟨foreach_set_declaration⟩
    | (⟨set_expr⟩)
    | set_input()
    | ⟨general_set_term⟩
    Set expression can be used with set name and function call. The power set is provided by the set name, the powerset operator, and the arithmetic expression, respectively. Primitive set declaration can be made. With the foreach set declaration, set operations can be performed close to mathematical use. Set input can be taken. Set expression and general set term can be created.

12. ⟨set_func_call⟩ ⟶ insert(⟨expr⟩)
    | remove(⟨expr⟩)
    | union(⟨set_expr⟩)
    | intersect(⟨set_expr⟩)
    | cartesian(⟨set_expr⟩)
    | diff(⟨set_expr⟩)
    Set Function calls include insert to add elements to set, remove to remove, union to combine sets, intersect to get their intersection, cartesian to create a cartesian product, and diff to get the difference.

13. ⟨prim_set_declaration⟩ ⟶ { }
    | {⟨element_list⟩}
    Primitive set declaration is done by leaving curly brackets blank or by typing set elements.

14. ⟨element_list⟩ ⟶ ⟨element⟩
    | ⟨element_list⟩ , ⟨element⟩
    The element list consists of one or more comma-separated elements.

15. ⟨element⟩ ⟶ ⟨expr⟩
    The element is expression.

16. ⟨foreach_set_declaration⟩ ⟶ ⟨expr⟩ foreach(⟨foreach_list⟩)
    Foreach set declaration is used as expression, foreach keyword, left parenthesis, foreach list, right parenthesis.

17. ⟨foreach_list⟩ ⟶ ⟨foreach_list_element⟩
    | ⟨foreach_list⟩ , ⟨foreach_list_element⟩
    The foreach list is used by providing one or more comma-separated foreach lists.

18. ⟨foreach_list_element⟩ ⟶ ⟨var_id⟩ in ⟨set_id⟩
    Foreach list element is used by giving variable id, in keyword and set id.

19. ⟨relational_expr⟩ ⟶ ⟨relational_expr⟩ ⟨relational_op⟩ ⟨general_expr⟩
    Relational expression is created using relational expression, relational operator, and general expression.

3.5 Primitive Functions

1. ⟨prim_func_call⟩ ⟶ ⟨delete_call⟩
    | ⟨print_call⟩
    | ⟨input_call⟩
    | ⟨file_input_call⟩
    Primitive function call does one of delete, print, input, file input call.

2. ⟨delete_call⟩ ⟶ delete(⟨set_id⟩)
    The delete call is used to delete sets.

3. ⟨print_call⟩ ⟶ print(⟨set_expr⟩) | print(⟨expr⟩) | print(⟨general_term⟩)
    With the print call, any expression or term can be printed.

4. ⟨input_call⟩ ⟶ input()
    Input is received with the input call.

5. ⟨file_input_call⟩ ⟶ file_input(⟨string⟩)
    With the file input call, input is obtained from the file.

6. ⟨file_output_call⟩ ⟶ file_output(⟨set_expr⟩, ⟨string⟩);
    | file_output(⟨general_expr⟩, ⟨string⟩);
    File Output Call provides output to file.

3.6 Special Signs

1. ⟨**assign_op**⟩ ⟶ ⟨**<<**⟩
    It is the operator that performs initializations in CardiLan language.

2. ⟨**set_token**⟩ ⟶ ⟨**@**⟩
    This mark is placed in front of the sets to identify the sets.

3. ⟨**comment_sign**⟩ ⟶ ⟨**//**⟩
    In CardiLan, this flag is used to indicate a comment line.

4. ⟨**end_stmt**⟩ ⟶ ⟨**;**⟩
    In CardiLan, the semicolon removes ambiguity and confusion from the code and is used at the end of statements.

- **Identifiers:** To speak in general terms, identifiers can include all alphanumeric characters, as well as underscores. Though, to increase readability, we decided to make it easier to differentiate between set identifiers and variables. Thus, set identifiers in our language have to start with the symbol *@*.
- **Literals:** The literals pertaining to this programming language can be of the following types: int, float, string, boolean, set, generic and char.
- **Comments:** In our language, comments start with *//*, and are contained in a single line. Therefore, multi-line comments are not allowed.
- **Reserved Words:** Constructs like loop and conditional statements have reserved words for both their beginnings and ends. In our system, the reserved word designated for the end of a structure is the reserved word for the beginning, concatenated after the word *end*. For example, a for loop begins with the reserved word *for* and ends with the reserved word *endfor*. The purpose behind using such words instead of curly brackets was to increase the readability and the writability of our language, which is discussed further in the next section. (Note that these are in addition to trivial reserved words such as int, float and set etc.)
- **Assignment Operator:** The assignment operator in our language is << instead of =.

4 Evaluation of the Language

4.1 Readability

An essential evaluation metric for programming languages is readability. The readability of a given programming language typically depends on its orthogonality, syntax and intelligibility. To have a simpler, more intelligible language, one would need to keep the number of features to a minimum, and make sure that their multiplicity is low. In order to increase the readability of our language as much as possible, we have made the following decisions:

- chose symbols as intuitively as possible (e.g. using curly brackets and commas when defining sets, similar to how it is normally done in mathematics),
- used different identifiers for variables vs. sets (where the distinction is made by the symbol *@*),
- used reserved words such as *endif* and *endfor* when terminating conditional statements and loops (as opposed to using curly brackets like in Java) so that it gets easier to read which end belongs to which structure,
- implemented the *foreach* construct to allow certain tasks to be implemented in single lines that would otherwise require complex structures. (Further explanation for this construct can be found in the Writability subsection)

In general, to increase readability, we have tried to make sure that there is only one (or few) way(s) to achieve a task, as opposed to having a multitude of options.

4.2 Writability

Another crucial metric for the evaluation of programming languages is writability. To increase the writability of our programming language, we have taken the following actions:

- implemented three different types of loops (while, do while, for) to make coding loops easier,
- allowed the use of all alphanumeric characters and underscore in variable identifiers to ensure that users have a wide selection of names,
- used reserved words such as *endfor* and *endif* instead of curly brackets so that the coders can see which ends correspond to which loop/conditional statements more easily.
- Finally, perhaps the most important feature we have added to our language for the sake of writability was the *foreach* construct. An example use of this construct is as follows: *a + 2*b foreach(a in @A, b in @B)* returns the set *a + 2*b* for every (a,b) tuple from sets A and B. This structure should increase the writability of set operations greatly.

4.3 Reliability

Reliability is of utmost importance when it comes to the evaluation of a programming language. Provided that it does not tamper with the readability or the writability of the language, improving its reliability should result in easier maintenance. That is why, reliability was one of our main priorities during the design of our language. Exemplary actions we took to increase reliability include:

- made set elements immutable to prevent other pointers from changing their values,
- required prior type specifications for parameters and return values of functions,
- did not let imported files run the main section to avoid unwanted modifications,
- defined precedence rules well to avoid any mix-ups,
- used distinct identifiers for sets and variables (through the use of the symbol @).