
**Bilkent University**

**Department of Computer Engineering**

# CS 315 Programming Languages

Beste Güney 21901631

Uygar Onat Erol 21901908

Hakan Gülcü 21702275

Section 01

## “JarSet”

27.02.2022

# Name of the Language: JarSet

## 1) BNF Description of The Language

### Constants and Types

**<digit> ::= 0 | 1 | 2 | 3 | … | 9**

**<positive_digit> ::= 1 | 2 | 3 | … | 9**

**<sign> ::= + | -**

**<number> ::= <digit> | <number> <digit>**

**<int> ::= <sign> <number> | <number>**

**<float> ::= <int> “.” <number>**

**<lowercase_letter> ::= a | b | c | d | ….| z**

**<uppercase_letter> ::= A | B | C | D | …. | Z**

**<letter> ::= <lowercase_letter> | <uppercase_letter>**

**<at> ::= “@”**

**<character> ::= <digit> | <letter> | <at>**

**<assignment_operator> ::= “=”**

**<comment_symbol> ::= “//”**

**<boolean> ::= TRUE | FALSE**

**<null> ::= NONE**

**<identifier> ::=  <letter> | <identifier> <character>**

**<set_identifier> ::= “#”**

**<set> ::= <set_identifier><identifier>**

**<set_element> ::= <identifier> | <int> | <float> | <string>**

**<end> ::= “;”**

**<LCB> ::= “{”**

**<RCB> ::= “}”**

**<LP> ::= “(”**

**<RP> ::= “)”**

**<and> ::= “&”**

**<or> ::= “|”**

**<string_character> ::= <character> | <string_character><character>**

**<string> ::= “” | “<string_character>”**

**<is_equal> ::= “==”**

**<not_equal> ::= “!=”**

**<smaller_than> ::= “<”**

**<smaller_than_or_equal> ::= “<=”**

**<greater_than> ::= “>”**

**<greater_than_or_equal> ::= “>=”**

**<comparator> ::= <smaller_than> | <smaller_than_or_equal> | <greater_than>**
**| <greater_than_or_equal>**

## Set Operations

**<set_intersection> ::= “&=&”**

**<set_union> ::= “+=+”**

**<set_difference> ::= “-=-”**

**<set_complement> ::= “!=!”**

**<set_cartesian> ::= “X=X”**

**<create_set_keyword> ::= createSet**

**<delete_set_keyword> ::= deleteSet**

**<subset> ::= “<<”**

**<superset> ::= “>>”**

**<subset_or_equal> ::=“<<=”**

**<superset_or_equal> ::=“>>=”**

**<set_cardinality_keyword> ::= getSetSize**

**<set_empty_keyword> ::= isEmpty**

**<set_has_keyword> ::= setHas**

**<set_singleton_keyword> ::= isSingleton**

**<set_powerset_keyword> ::= getPowerset**

**<set_retrieve_keyword> ::= retrieve**

**<set_remove_keyword> ::= remove**

**<set_add_keyword> ::= add**

**<set_relation_operator> ::= <subset> | <subset_or_equal> | <superset> | <superset_or_equal>**

**<set_operation_symbol> ::= <set_intersection> | <set_union> | <set_difference> | <set_cartesian> | set_complement**

# Expressions

**<set_primaries> ::= <set_element> | <int> | <boolean> | <identifier> | <string> | <float>**

**<set_member_list> ::= <set_primaries> | <set_member_list>,<set_primaries>**

**<set_assignment> ::= <set><assignment_operator><LCB><set_member_list><RCB>**
**| <set><assignment_operator><set_operator_expression>**

**<set_boolean_expression> ::= <set><set_relation_operator><set>**
**| <set_has_expression>**
**| <set_singleton_check_expression>**
**| <set_empty_check_expression>**
**| <set_cardinality_expression> <comparator><int>**
**| <set_cardinality_expression> <is_equal><int>**
**| <set_cardinality_expression> <not_equal><int>**

**<boolean_expression> ::= <boolean> | <set_boolean_expression> | <int><comparator><int>**
**| <int><is_equal><int> | <int><not_equal><int>**
**| <int><comparator><identifier>**
**| <identifier><comparator><identifier>**
**| <identifier><comparator><int>**
**| <identifier><or><identifier> | <identifier><and><identifier>**
**| <identifier><is_equal><int> | <identifier><not_equal><int>**
**| <int><is_equal><identifier> | <int><not_equal><identifier>**
**| <identifier><is_equal><identifier> | <identifier><not_equal><identifier>**
**| <identifier><is_equal><null> | <identifier><not_equal><null>**
**| <boolean><is_equal><boolean> | <boolean><not_equal><boolean>**
**| <function_call>**

**<boolean_list> ::= <boolean_expression> | <boolean_list> <and> <boolean_expression> | <boolean_list> <or>**
**<boolean_expression> | <LP> <boolean_list> <RP>**

**<assignment> ::= <identifier><assignment_operator><assignment_arg>**

**<assignment_arg>::= <boolean_list> | <int> <set_cardinality_expression> | <float> | <string> | <null>**

**<set_create_expression> ::= <create_set_keyword><LP><set><RP>**

**<set_delete_expression> ::= <delete_set_keyword><LP><set><RP>**

**<set_operator_expression> ::= <set><set_operation_symbol><set>**
**| <LP><set_operator_expression><RP>**
**| <set_powerset_keyword><LP><set><RP>**
**| <set_operator_expression><set_operation_symbol><set>**

**<set_retrieve_operation> ::= <set_retrieve_keyword><LP><set>, <int><RP>**
**| <set_retrieve_keyword><LP><set><RP>**

**<set_cardinality_expression> ::= <set_cardinality_keyword><LP><set><RP>**

**<set_empty_check_expression> ::= <set_empty_keyword><LP><set><RP>**

**<set_has_expression> ::= <set_has_keyword><LP><set>, <set_primaries><RP>**

**<set_singleton_check_expression> ::= <set_singleton_keyword><LP><set><RP>**

**<set_remove_expression> ::= <set_remove_keyword><LP><set>,<set_primaries><RP>**

**<set_add_expression> ::= <set_add_keyword><LP><set>,<set_primaries><RP>**

**<set_expression> ::= <set_create_expression> | <set_delete_expression>**
**| <set_operator_expression> | <set_assignment> | <set_boolean_expression>**
**| <set_cardinality_expression>| <set_remove_expression>**
**| <set_add_expression> | <set_keyboard_input> | <set_retrieve_operation>**
**| <set_console_output> | <set_file_input> |<set_file_output>**

## Function Calls

**<break_statement> ::= stop<end>**

**<continue_statement> ::=continue<end>**

**<return_statement> ::= return<arg><end> | return <end>**

**<arg> ::= <set> | <set_element> | <int> | <boolean_list> | <float> | <string>**

**<arg_list> ::= <arg> | <arg_list>,<arg>**

**<simple_declaration> ::= <identifier><end> | <set><end> | <set_element><end>**

**<print_statement> ::= print<LP><identifier><RP>**
**| print<LP><string><RP>**

**<block> ::= <continue_statement> | <break_statement> | <statement_list>**

**<function_definition> ::= define<function_identifier><LCB><statement_list><RCB>**

**<function_operator> ::= “->”**

**<function_identifier> ::= <identifier><LP><arg_list><RP> | <identifier><LP><RP>**

**<function_call> ::= <identifier><function_operator><function_identifier>**
**| <set><function_operator><function_identifier>**
**| <function_identifier>**

## Loops and Conditionals

**<basic_for_loop> ::=**
**for<LP><assignment><end><boolean_list><end><assignment><end><RP><LCB><block><RCB>**

**<set_for_loop> ::= <for><LP><identifier>:<set><RP><LCB><block><RCB>**

**<for_loop> ::= <basic_for_loop> | <set_for_loop>**

**<while_loop> ::= while <LP> <loop_cond_arg><RP><LCB><block><RCB>**

**<loop_statement> ::= <for_loop> | <while_loop>**

**<if_statement>::= if<LP><loop_cond_arg><RP><LCB><block><RCB><else_statement>**
**| if<LP><loop_cond_arg><RP><LCB><block><RCB>**

**<else_statement> ::=else<LCB><block><RCB>**

**<loop_cond_arg> ::= <boolean_list>**

## Input Output

**<set_keyboard_input> ::= enterSet<LP><set><RP>**

**<set_console_output> ::= printSet<LP><set><RP>**

**<set_file_input> ::= fReadSet<LP><set>,<identifier><RP>**

**<set_file_output> ::= fWriteSet<LP><set>,<identifier><RP>**

## Program

**<program> ::= <main>**

**<main> ::= main<LP><RP><LCB><statement_list><RCB>**

**<statement_list> ::= <statement> | <statement_list> <statement>**

**<statement> ::= <assignment><end> | <set_expression><end> | <function_call><end>**
**| <loop_statement> | <if_statement> | <comment_statement> | <simple_declaration>**
**| <print_statement><end> | <return_statement> | <function_definition>**

**<comment_statement> ::= <comment_symbol><comment><comment_symbol>**

**<comment> ::= <comment><identifier> | <identifier>**

# 2) Explanations of NonTerminals

## Program:

**<program>** : Program is the start variable of the language. It is defined by the main program.

**<main>** : JarSet language programs are built on a main program. Main program is defined as a statement list.

**<statement_list>** : Statement list defines the statements that make the program. It is either defined as a statement or left recursively followed by other statements.

**<statement>** : The main statement types in JarSet language are assignment statements, expressions related to set operations, function calls, while and for loops, conditional statements and comments.

**<comment_statement>** : Comment statements represent the comments in a JarSet program. Comments are written between "//" symbols.

**<comment>** : Comments are sentences. They are either identified in the single identifier syntax or right recursively with many identifiers.

## Constants and Types:

In JarSet language there are 5 main types. Integer, Boolean, Float, String and Set. Integers and floats can be negative or positive. Boolean values are defined as either true or false. Sets can contain set elements, integers, floats, strings and booleans.

**<int>:** Integer defines the integers in language. It is defined as a number. Number can have a sign or not.

**<float>:** Float defines the floats in language. It is defined as a float number.

**<letter>**: Letter is defined as the all letter characters in the English alphabet both uppercase and lowercase.

**<identifier>** : Identifier represents the variables in JarSet language. It must start with a letter character to be an identifier. It can have digit characters, @ symbol and letters.

**<set>**: Set type is used to distinguish the set type variables in a program. For an identifier to represent a set type, it must start with a “#”character.

**<set_element>:** Set element represents set elements that a set can have. Set elements have the syntax of identifiers which are wrapped between apostrophes.

**<comparator>**: Comparators are defined as the operations that are used in comparisons. For readability and writability, the generally used relational operators are used as comparators.

**<string>:** Strings are the expressions which are printed inside the print method. They are characters inside apostrophes and can be inside sets.

# Set Operations

**<set_relation_operator>** : Set relation operators are defined as the operators that are used to understand the relation between two sets. A set can be a subset, superset of another set or can be equal to another set when all elements are the same. Set relational operators are chosen to have similar syntax with regular relational operators to make it easy to understand.

**<set_operation_symbol>:** Set operation symbols are used to define the operations that result in a third set. Intersection, union, difference and Cartesian product operations are used to define operations between sets. Again, for these operators the analogy in the operation is considered. For example, intersection operation has operators which have & symbol inside.

# Expressions

**<set_member_list>**: Set member list is used to initialize the set type elements in a program. It can consist of a single set element or left recursively it can consist of many set elements.

**<set_assignment>**: This statement is used to initialize the sets by entering the list or assigning to the result of a set operation.

**<set_boolean_expression>**: This expression is defined as the set expressions that return a boolean type answer.

- When a set is compared with another set by using the set relation operators
- When a set is checked for whether it has a certain element inside
- When a set is checked if it has just a single element
- When a set is checked if it is empty

These statements will return a boolean answer.

**<boolean_expression>** : Boolean expressions are the expressions that return a boolean answer. When elements of the same type are compared to each other or when a set boolean operation is conducted, it is expected to get a boolean answer.

**<assignment>**: Assignment expressions are defined as the operation of giving a value to a variable. A variable can be assigned to be an integer or a boolean value.

**<set_create_expression>**: This expression is used to create a new empty set and assign it to the provided set type variable.

**<set_delete_expression>:** This expression is used to delete a set which is provided at set type variable.

**<set_operator_expression>**: These expressions show performing operations between two sets and returning a third set which is the result of the applied operation. Two sets can either be intersected, unified, differed or their cartesian product can be taken. In addition to that, a power set can also be derived from a single set.

**<set_retrieve_operation>:** This operation is defined for retrieving the first element or an element at a specified index of a set. By using the retrieve token and giving the set variable to retrieve from inside parentheses, the resultant element can be taken.

**<set_cardinality_expression>**: This operation is defined for retrieving the size of the given set.

**<set_empty_check_expression>:** This operation is defined for checking whether a given set is empty or not.

**<set_singleton_check_expression>**: This operation is defined for checking whether a given set contains a single element or not.

**<set_add_expression>:** This operation is defined for adding a new element to the end of the given set.

**<set_expression>**: Set expressions defines all the expressions that are valid at set operations.

**<set_primaries>: Set primaries are the all types of elements that a set can have. They are used at the initialization of set type.**

**<boolean_list>: By using left recursion, boolean expressions are chained at boolean lists in other words it allows to use multiple boolean expressions side by side. When many boolean expressions are chained, the operations are executed from left to right.**

**<assignment_arg>: Assignment argument is the value that an identifier can take.**

# Function Calls

**<break_statement> :** Break statements indicate the times that the execution of a loop must be stopped. In JarSet, break statements are represented by keyword *stop.*

**<continue_statement> :** Continue statements indicate the times that execution of a loop must continue from the next iteration. In JarSet, continue statements are represented by the keyword *continue*

**<return_statement> :** Return statements indicate the returning conditions from functions. In JarSet, return statements are represented by the keyword *return* followed by the possible returning arguments.

**<print_statement> :** Print statements are used for printing sentences or the values of variables. When an identifier is passed into a print statement, it displays the content of the variable whereas when a string is passed in quotation marks, the exact wording is printed out.

**<arg>:** Argument is the parameter used in function calls. Arguments are defined as either set type variables, integers, booleans or other variables.

**<arg_list>:** Argument list is the list of parameters passed to the functions at function calls. Argument lists can consist of only a single argument or many arguments which are separated by “,”.

**<block>:** Block represents the statements that can be inside function calls, conditional or loop statement bodies. Blocks can be a single continue, break, return statement or many statements which are defined as a statement list.

**<function_definition>:** Functions are defined by the keyword “define” followed by an identifier which represents the name of the function. Functions may or may not receive arguments inside parentheses. In their body, they carry block statements.

**<function_identifier>:** Function calls are defined as the function name followed by parentheses. Inside parentheses they can carry arguments. Other variables call functions through this identifier.

**<function_call>**: Function calls can be made at different times.

-Sets can call functions through their definitions.
- Other variables can call functions through their definitions.
- Some functions can directly be called by just their names.

## Loops and Conditionals

**<basic_for_loop>:** Basic for loop defines the for loop which iterates through integer values. At the first part of the for loop an identifier is assigned to an integer, at the boolean expression part it is compared with a certain value and lastly at each iteration it gets updated at the last part of the for loop. At the body of the for loop, block statements are executed.

**<set_for_loop>:** Set for loop is a special type of for loop which is specifically to iterate through set elements. By separating the iterated value and the set, using a java-like colon syntax, sets can be iterated.

**<for_loop>:** For loop can be either basic or for set.

**<while_loop>:** While loops can take boolean returning function calls or boolean expressions in their declarations. In their body, they can execute block statements.

**<loop_statement>:** Loops can either be for loop or while loop.

**<if_statement>:** If statements can either take a boolean returning function call or a boolean expression in their declarations. They can have an else statement part. In their body, they execute the block statements.

**<else_statement>:** Else statements are defined as the else keyword followed by block statements given in curly braces. They can be inside if statements for representing another option.

**<loop_cond_arg>: Loop condition argument is to keep loop conditions in for and while loops.**

## Input Output

**<set_keyboard_input>:** By passing the set variable inside parentheses to the enterSet keyword, the user can enter new set elements to the given set.

**<set_console_output>:** By using printSet statements, the elements of a set can be printed out to the console.

**<set_file_input>**: By using fReadSet statements, the contents of a set variable can be taken from a specified file.

**<set_file_output>:** By using fWriteSet statements, the contents of a set variable can be written to a specified file.

# 3) Explanation of NonTrivial Tokens

## 3.1 Comments

In this language, the comments are used between “//” symbols. Comments are considered to be necessary in this language, because they increase the readability and writability of the language. By putting comments to appropriate places, users can understand the written program easier and better.

## 3.2 Identifiers

In this language, identifiers are chosen to start with letters only. Letters are indicated by uppercase or lowercase letters in English. After starting with a letter, identifiers can be followed by characters which are defined as either @ symbol, letters or digits. By starting with letters and not using so many characters in identifiers, the goal was to contribute to the writability of a language. From the readability perspective, although function names and variable names seem similar, due to the fact that parentheses are not allowed in identifier names, by simply checking parentheses the function definitions can be understood easily.

## 3.3 Literals

- int: Integer literals can be given as negative or positive numbers in this language. If a number is not given with a sign, it is still valid according to the production rules and the int is considered to be positive in this case.

- float: Float literals carry a dot sign between the integers as in mathematics. Floats can have signs as integers and also every integer is a float.

- boolean: There are two types of boolean values defined in this language. A boolean can be either true or false. This increases the writability of the programs because many already existing programming languages follow a similar approach to the definition of boolean type. This similarity makes it easy to comprehend and write the language.

- null: Null literal shows that an identifier has not been assigned to any other value and it doesn't carry any type information.

- string: String literals are a sequence of characters between apostrophes. In this language, empty strings are also valid as well.

## 3.4 Reserved Words

**TRUE-FALSE :** These reserved words are used to show boolean expressions and their similarity to many other programming languages increases the writability.

**NONE:** This word is used to indicate null variables.

**createSet:** This word is used to create a new set at the program. Its name increases the readability of the language because it is easy to understand.

**deleteSet:** This word is used to delete a set from the program. Again, Its name increases the readability of the language because it is easy to understand.

**getSetSize:** This word is used to get the size information of a set. The similarity of this name with java-like languages' getter-setter approach, makes it easy to read and write. It returns an integer.

**getPowerset:** This word is used to get the power set of a given set. Again, the similarity of this name with java-like languages' getter-setter approach, makes it easy to read and write.

**isEmpty-isSingleton:** These words are used to get the size information of a set. The syntax of the name resembles a question and indicates a boolean result. This contributes to the readability of the program.

**setHas:** This keyword is used to check whether an element is contained in a set.

**retrieve:** This keyword is used to get the first element or an element at the specified index from a set.

**remove:** This keyword is used to remove a set element from a given set.

**add:** This keyword is used to add a new element to the given set.

**stop:** This keyword is used for breaking out of loops. Although many traditional programming languages use break statements, we consider *stop* to be more clear to be understood and so increasing the readability of the language.

**continue:** This keyword is used to indicate continuing the execution of a loop statement from the next iteration.

**return:** This keyword is used to return from function blocks.

**while:** Used in while loops.

**for:** Used in for loops.

**if:** Used in conditional statements.

**else:** Used in conditional statements.

**enterSet:** This reserved word is used to create a set by entering values from the keyboard.

**printSet:** This reserved word is used to print set elements to the console. By using the print keyword, it is aimed to resemble C like languages and increase readability.

**fReadSet:** This reserved word is used to create a set by reading values from a file. The ‘f’ at the beginning, indicates that this is a file process, as in C like languages.

**fWriteSet:** This reserved word is used to save the contents of a set to a specified file. The ‘f’ at the beginning, indicates that this is a file process, as in C like languages.

**integer:** This reserved word is used to show integer type

**boolean:** This reserved word is used to show boolean type

**set_type:** This reserved word is used to show set type

**string:** This reserved word is used to show string type

**float:** This reserved word is used to show float type

# 4) Conflicts

At the first compilation of our project, we have received four conflict messages. When the y.output file is examined, we have seen that we have two rules which are never reduced. Although we have declared these rules, we did not use them at the right side anywhere at our bnf. These rules were about ‘set_retrieve_operation’ and we have added this operation to our set_expressions so that the program can reduce it as a statement. Also, we have realized that there is a reduce/reduce conflict at our project. We have declared statements in our program which included return_statement as well. In addition to that, we have block statements which represent the blocked parts of the program such as inside if and while statements. Because we have declared that block statements can be statements and return statement, this resulted in a reduce/reduce conflict because return statements were already declared at statements. Because of that, the excessive return is removed from the block statement.

In addition to that, when we tried to add features for chained operations we received 2 shift/reduce conflict. In order to be able to create chained boolean expressions, we have created a new term and created a left recursive rule. In this left recursive rule we had a structure like:

boolean_list: boolean_expression | boolean_list AND boolean_expression | boolean_list OR boolean_expression

Because of the fact that, boolean_expression both contained boolean and statements like (boolean and boolean), this created a shift/reduce conflict because the boolean in boolean_expression is enough to create and operations in boolean_list. In order to solve that, the redundant boolean and/or boolean operations were eliminated from the boolean_expression statement.

# 5) Precedence Rules and Ambiguities

In our language, the ambiguities are solved with creating new terms and writing recursive rules. For example, in order to make a chained boolean expression as it is mentioned earlier, a new term is created. In our language, the chained operations are executed from left to right. For example, if a expression such as

#set1 &=& #set2 -=- #set3

is given, it means to intersect set1 and set2 first and then take the difference from set3. So the set operations are left associative. This is also true for chained boolean expressions as well.