
# CS315 Project 2 Report

**Date: 08.04.2020**
**Language name: zetz**
**Group number: 22**


| **Group members:** | **Full name** | **Student ID** | **Section** |
|---|---|---|---|
| | Halil İbrahim Karakoç | 21702419 | 1 |
| | Suleyman Rahimov | 21701671 | 1 |
| | Uğur Erdem Seyfi | 21801744 | 1 |

## Part A - Revised and Augmented Language Design

Since the last project report, we made some changes on the grammar in order to make it more understandable and simple at the same time. These changes might be summed up as follows:

1. Added cartesian product feature to our language. Now it is allowed to take the cartesian product of two sets by using the "^" operator.

2. In order to achieve a more structured way of evaluating expressions, we changed the grammar in a way that is somehow like a staircase. Right now each expression can be obtained from each other in a certain order of priority.

3. Although our language allowed the "input()" in the form of a function expression, there was not any specifically reserved grammar for "input", we added this to the grammar alongside making possible the expressions of union and intersect also fall under the name of function expression.

### General description of the non-terminals in the grammar

**Statements:** The rules for the statements are pretty straightforward, we have different kinds of statements and all of these statements are contained in a single rule called **<statement>**. Most of the rules here are straightforward and easy to understand.

**Expressions:** This is the most tricky part of the grammar. We modified our grammar in a way that most of the expressions can be obtained from one and another since we wanted to allow compact expressions composed of different types of expressions such as boolean expression that includes two arithmetical expressions such as "(3 + 5) * 3 < 10". This is why **<expression>** has only two expressions: **<assignment_expression>** and **<boolean_expression>**. All of the other expressions are implemented in a way that can also be obtained from **<boolean_expression>** rule.

We also formed the grammar of expressions in a way that would prioritize some of the actions from the others. The rules or expression types that need to be evaluated first are the ones that are in the deeper levels of rules. For example, in our language expressions in parentheses are the most prioritized kind of expressions. In addition, we have taken into account the priority of different types of operators such as not (**!**) operator, addition (**+**) and multiplication (*****) operators, comparison (**<, <=, >, >=, ==,**

**!=**) operators and binding (**&&, ||**) operators and set operators such as set subtraction (\) and cartesian product (**^**).

In part 2.5, one might ask why there are different lists for parameter lists ( **<parameter_list>** and **<set_op_parameters>**). The reason for this is that we wanted to enforce the users to use 2 or more parameters when they use the intersect and union operators.

**Literals:** In terms of grammar, the only change we have done here was to remove **<unsigned>** token. Also in the last report we have shown the grammar for these literals, however, right now we are treating the following rules as tokens (which are all of them are denoted by capital letters only) that are given to us by lexer: **<IDENTIFIER>**, **<STRING>**, **<CHAR>**, **<FLOAT>**, **<INTEGER>**, **<BOOLEAN>**.

## Grammar of the Language

Note: ***ε*** denotes the empty string.

Our program starts with the following rule:

<program> → <statement_list>

## 1 Statements

<statement_list> → <statement> | <statement_list statement>

<statement> → <declaration_statement>

| <assignment_statement>

| <if_statement>

| <while_statement>

| <do_while_statement>

| <for_statement>

| <print_statement>

| <set_property_statement>

| <delete_statement>

| <function_declaration_statement>

| <function_call_statement>

| <return_statement>

### 1.1 Declaration/Assignment Statements

<declaration_statement> → <declaration_expression> ;

<assignment_statement> → <assignment_expression> ;

<print_statement> → print (<expression>) ;

### 1.2 If/Else Statements

<if_statement> → if (<boolean_expression>) { <statement_list> } <else_if> <else>

<else_if> → <else_if> else if ( <boolean_expression> ) { <statement_list> } | ε

<else> → else { <statement_list> } | ε

### 1.3 Loops

<while_statement> → while ( <boolean_expression> ) { <statement_list> }

<do_while_statement> → do { <statement_list> } while ( <boolean_expression> ) ;

<for_statement> → for ( <declaration_expression> ; <boolean_expression> ; <assignment_expression> ) { <statement_list> }

### 1.4 Methods (Functions)

<parameter_list> → <parameter_list> , <type> IDENTIFIER | type IDENTIFIER | ε

<function_declaration_statement> → <type> IDENTIFIER ( <parameter_list> ) { statement_list }

<return_statement> → return <expression> ;

<function_call_statement> → <function_expression> ;

### 1.4.1 Special Method Statements

<set_property_statement> → <IDENTIFIER> . add ( <expression_list> ) ;

| <IDENTIFIER> . remove ( <expression_list> ) ;

| <IDENTIFIER> . size ( ) ;

<delete_statement> → delete <IDENTIFIER>;

## 2. Expressions

<expression> → <assignment_expression> | <boolean_expression>

### 2.1 Assignment Expression

<assignment_expression> → <IDENTIFIER> = <expression>

<declaration_expression> → <type> <IDENTIFIER>

| <type> <assignment_expression>

### 2.2 Boolean & Comparison Expressions

<boolean_expression> → <boolean_expression> **||** <boolean_and_expression>

| <boolean_and_expression>

<boolean_and_expression> → <comparison_expression>

| <boolean_and_expression> **& &** <comparison_expression>

<comparison_expression> → <comparison_expression> **<=** <arithmetic_expression>

| <comparison_expression> **>=** <arithmetic_expression>

| <comparison_expression> **>** <arithmetic_expression>

| <comparison_expression> **<** <arithmetic_expression>

| <comparison_expression> **==** <arithmetic_expression>

| <comparison_expression> **!=** <arithmetic_expression>

| <arithmetic_expression>

### 2.3 Arithmetic expressions

<arithmetic_expression> → <term> + arithmetic_expression

| <term> - <arithmetic_expression>

| <term>

<term> → <term> * <set_expression> | <term> / <set_expression>

| <set_expression>

### 2.4 Set Operation Expressions

<set_expression> → <set_expression> \ <cartesian_expression> |
<cartesian_expression>

<cartesian_expression> → <cartesian_expression> ^ <function_expression>

| <function_expression>

## 2.5 Function expression

<function_expression> → <IDENTIFIER> ( <input_parameters> )

| <IDENTIFIER> ( )

| <input_expression>

| <simple_expression>

| <union_expression>

| <intersect_expression>

<input_parameters> → <expression> | <input_parameters> , <expression>

<set_op_parameters> → <set_op_parameters> , <expression> | <expression
com_expr

<com_expr> → , <expression>

<input_expression> → input ( )

<union_expression> → union( <set_op_parameters> )

<intersect_expression> → intersect ( <set_op_parameters> )

## 2.6 Unit Expression

<simple_expression> → ! <unit_expression> | <unit_expression>

<unit_expression> → ( <expression> ) | <literal> | <IDENTIFIER>

### 3. Literals

literal → <INTEGER> | <FLOAT> | <set> | <STRING> | <CHAR> | <BOOLEAN>

type → int | float | boolean | set | char | string

<set> → { <expression_list> } | { }

<expression_list> → <expression_list> , <expression> | <expression>

### 4. Example Program

```
/*
  A test program to test the various components of the language.
 */

/*
        This part tests the set operations in the language.
*/

set set1 = { 1, 2, 7 };                     // set of integers
set set2 = {1,7,2};                         // set of integers
set set3 = { "Mesut", "Husnu", "Riza" };        // set of strings
set set4 = { set1, set2 };                  // set of sets
set set5 = { true, true, true, false };     // set of booleans
set set6 = { true, false };
set stringSet2 = input();                   // storing the input in the stringSet2 variable



if ( set1 == set2 ) {
        print( "Two sets are equal" );
}

else {
        print( "Two sets are not equal" );
}


/*
```

```
Note: the sets are considered equal when they have the same elements. The number of
elements do not matter.
*/

if ( set5 == set6 ) {
        print( "Given boolean sets are equal" );
}

else {
        print( "Given boolean sets are not equal" );
}

set1.add( 87 );         // adding an element to a set
set2.remove( 7 );       // removing an element from the set

set set7 = union( set1, set2);
set set8 = intersection( set1, set2 );

if ( set1 > set2 ) {
        print( "set1 is a superset of set2" );
}

if ( set1 < set8 ) {
        print( "set1 is a subset of set8" );
}

delete set1;            // to remove all elements from the set and setting it equal to {}(empty
set)
set2.add(input());      // adding the elements, that user entered, to the set

delete set6;            // set 6 is now equal to empty set

set2 = set2 \ set1;   // set subtraction

/*
        This part tests the set decision statements and loops in the language.
*/

int num1 = 3;
int num2 = 8;
int num3 = 27;
int num4 = -72;
float realNum = 0.99;
char character = 'y';
string str = "Mesut";    // string declaration
```

```
if ( ( num1 != num2 ) && ( num1 == num3 ) ) {
        print ("num1 is not equal to num2 and is equal to num3" );
}

else if ( ( num2 == num4 ) || ( num3 == num4 ) ) {
        print ("num2 is equal to num4 or num3 is  equal to num4");
}

else {
   if ( ( num3 > 0 ) && (! (num3 >= 8)) ) {
         print("num3 is greater than num0 and num3 is not greater than or equal to 8");
   }
}

// incrementing num2 by i multiple times using a for loop
for ( int i = 0; i < num3; i = i + 3 ) {
        num2 = num2 + i;
}

// incrementing num2 until it reaches 1000 using while loop
while ( num2 < 1000 ) {
        num2 = num2 + 10;
}

// incrementing num4 until it is zero using do-while loop
do {
   num4 = num4 + 1;
} while ( num4 < 0 );


/*
        This part tests the function definition and implementation in the language.
*/

int sum( int a, int b ) {
        return a + b;
}

int number1 = 7;
int number2 = 12;

int number3 = sum( number1, number2 );   // number3 is 19 after this method call


/*
        New tests for the project2 report
```

```
*/


set A = {1, 2, 3};
set B = {‘a’, ‘b’};

set C = A ^ B; // Now C is the cartesian product of A & B.
                    // C = { (1,’a’), (1, ‘b’), (2,’a’), (2, ‘b’), (3,’a’), (3, ‘b’) }

// we can also get the cartesian product of more than one sets

print( A ^ {1, 3} ^ {“selam”, foo(), a, {3} } ); // this is a valid expression

int newNumber = 5 * input();    // we can directly use input in an expression
                                        // newNumber is 5 times larger than the input entered.
```