

Roman to Integers Converter

JOIE ANN M. MAC
Instructor

PROGRAMMING PARADIGMS**Table of Contents**

The Machine Problem.....	3
Language Definition.....	9
Evaluation.....	22
Concluding Statements.....	23
Bibliographic Sources.....	23
Appendices.....	23

Group Members

Khan, Imroz Mae S.
Sedoriosa, Febron Jr. B.
Jimenez, Cherry Lee H.
Dacapio, Ella Norienne C.
Preciado, Via Nicole A.
Zarate, Ismael Sulpicio G.

The Machine Problem

- A. The Problem Statement – *Develop a program to convert Roman numerals into their equivalent integer values. Implement solutions using four different programming paradigms: Imperative, Object-Oriented, Logic, and Functional.*
- Input – *A Roman numeral string provided by the user.*
A termination condition, such as the phrase "end conversion", to exit the program.
 - Output – *The corresponding integer value of the Roman numeral.*
For invalid inputs, display an error message such as "Invalid Roman numeral."

B. Solutions

a. Imperative Paradigm Solution

```
// ROMAN TO INTEGERS IN C
#include <stdio.h>
#include <string.h>
#include <ctype.h>

// Function to get the integer value of a Roman numeral
character
int romanToInt(char c) {
    switch (toupper(c)) {
        case 'I': return 1;
        case 'V': return 5;
        case 'X': return 10;
        case 'L': return 50;
        case 'C': return 100;
        case 'D': return 500;
        case 'M': return 1000;
        default: return 0; // Invalid character
    }
}

// Function to convert Roman numeral to integer
int convertRomanToInt(const char *roman) {
    int total = 0;
    int prevValue = 0;

    for (int i = strlen(roman) - 1; i >= 0; i--) {
        int currentValue = romanToInt(roman[i]);

        // If the current value is less than the previous value,
        subtract it; otherwise, add it
        if (currentValue < prevValue) {
            total -= currentValue;
        } else {
            total += currentValue;
        }
    }
}
```

```
        prevValue = currentValue;
    }

    return total;
}
int main() {
    char roman[20];

    printf("Enter Roman numerals to convert them to integers.\n");
    printf("Type 'end conversion' to stop the program.\n");

    while (1) {
        printf("\nEnter a Roman numeral: ");
        scanf("%s", roman);

        // Check if the user wants to end the conversion
        if (strcasecmp(roman, "end") == 0) {
            char secondWord[20];
            scanf("%s", secondWord);
            if (strcasecmp(secondWord, "conversion") == 0) {
                printf("Ending conversion.\n");
                break;
            }
        }

        int result = convertRomanToInt(roman);

        // Check for invalid input
        if (result == 0) {
            printf("Invalid Roman numeral.\n");
        } else {
            printf("The integer value is: %d\n", result);
        }
    }
    return 0;
}
```

b. Object-Oriented Paradigm Solution

```
// ROMAN TO INTEGER IN JAVA (change the file name to public name)

import java.util.Scanner;

public class Main {
    // Method to get the integer value of a Roman numeral
```

```
character
    public static int romanToInt(char c) {
        switch (Character.toUpperCase(c)) {
            case 'I': return 1;
            case 'V': return 5;
            case 'X': return 10;
            case 'L': return 50;
            case 'C': return 100;
            case 'D': return 500;
            case 'M': return 1000;
            default: return 0; // Invalid character
        }
    }

    // Method to convert a Roman numeral string to an integer
    public static int convertRomanToInt(String roman) {
        int total = 0;
        int prevValue = 0;

        for (int i = roman.length() - 1; i >= 0; i--) {
            int currentValue = romanToInt(roman.charAt(i));

            // If the current value is less than the previous
            value, subtract it; otherwise, add it
            if (currentValue < prevValue) {
                total -= currentValue;
            } else {
                total += currentValue;
            }

            prevValue = currentValue;
        }

        return total;
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.println("Enter Roman numerals to convert
them to integers.");
        System.out.println("Enter Roman numerals to convert
them to integers.");
        System.out.println("Type 'end conversion' to stop the
program.");

        while (true) {
            System.out.print("\nEnter a Roman numeral: ");
            String input = scanner.nextLine().trim();
```

```

        // Check if the user wants to end the conversion
        if (input.equalsIgnoreCase("end conversion")) {
            System.out.println("Ending conversion.");
            break;
        }

        // Convert Roman numeral to integer
        int result = convertRomanToInt(input);

        // Check for invalid input
        if (result == 0) {
            System.out.println("Invalid Roman numeral.");
        } else {
            System.out.println("The integer value is: " +
result);
        }
    }

    scanner.close();
}
}

```

c. Logical Paradigm Solution

```

% Facts defining the integer values of Roman numeral
characters
roman_value('I', 1).
roman_value('V', 5).
roman_value('X', 10).
roman_value('L', 50).
roman_value('C', 100).
roman_value('D', 500).
roman_value('M', 1000).

% Base case: an empty list has a value of 0
roman_to_int([], 0).

% Recursive case: calculate the value of a Roman numeral list
roman_to_int([H|T], Value) :-
    roman_value(H, Current),           % Get value of the
current Roman numeral
    roman_to_int(T, NextValue),        % Recursively
calculate the rest of the list
    ( T = [NextHead|_],                % Check if there is a
next character
        roman_value(NextHead, Next),  % Get the value of the

```

```

next Roman numeral
    Current < Next                                % Subtraction case:
smaller value before larger
    -> Value is NextValue - Current
    ; Value is NextValue + Current                % Normal case: add the
value
    ).

% Predicate to validate Roman numeral characters
valid_roman([]).                                % An empty list is
valid
valid_roman([H|T]) :-
    roman_value(H, _),                          % Check if the
character is a valid Roman numeral
    valid_roman(T).                             % Recursively validate
the rest of the list

% Predicate to process user input
process :-
    write('Enter a Roman numeral (or type "end" to stop): '),
    read_line_to_string(user_input, Input),      % Read input as
a string
    string_upper(Input, NormalizedInput),        % Convert input
to uppercase
    ( NormalizedInput == "END"                   % Check for the
end condition
    -> write('Ending conversion.'), nl
    ; string_chars(NormalizedInput, RomanChars), % Convert
string to list of characters
    ( valid_roman(RomanChars)                    % Validate if all
characters are Roman numerals
    -> roman_to_int(RomanChars, Result),
        format('The integer value is: ~d~n', [Result])
    ; write('Invalid Roman numeral. Please try
again.'), nl
    ),
    process                                       % Repeat the
process
    ).

% Start the program
start :-
    write('Roman Numeral to Integer Converter'), nl,
    write('Type "end" to stop the program.'), nl,
    process.

```

d. Functional Paradigm Solution

```

(defun roman-value (char)
  "Return the integer value of a Roman numeral character."
  (ecase (char-upcase char) ; Ensure the character is
    converted to uppercase
    (#\I 1)
    (#\V 5)
    (#\X 10)
    (#\L 50)
    (#\C 100)
    (#\D 500)
    (#\M 1000)))

(defun roman-to-int (roman)
  "Convert a Roman numeral string to an integer."
  (let ((total 0)
        (prev-value 0))
    ;; Loop through each character in the string
    (loop for char across roman
          for current-value = (roman-value char)
          do (if (> prev-value current-value)
                (setf total (+ total current-value)) ;; Add
                if the current value is less than or equal to the previous
                (setf total (+ total (- current-value (* 2
prev-value))))) ;; Adjust for subtraction rule
                (setf prev-value current-value)) ;; Track
the previous value
    total))

(defun process ()
  "Read Roman numeral input, convert to integer, and repeat
until 'end' is entered."
  (loop
    (format t "Enter a Roman numeral (or type 'end' to stop):
")
    (let ((input (read-line)))
      (if (string= (string-upcase input) "END") ; Check if
          user entered 'end' (case-insensitive)
          (progn
            (format t "Ending conversion.~%")
            (return))
          (handler-case
            (let ((result (roman-to-int input)))
              (format t "The integer value is: ~D~%"
result))
            (error ()
              (format t "Invalid Roman numeral. Please try
again.~%"))))))))

```



```
(defun start ()
  "Start the Roman numeral to integer converter program."
  (format t "Roman Numeral to Integer Converter~%")
  (format t "Type 'end' to stop the program.~%")
  (process))

(start)
```

Language Definition

A. Syntax

a. BNF from basic syntax until control structures

```
<program> ::= <statements>

<statements> ::= <statement> ";" <statements> |
<statement> ";"

<statement> ::= <assignment> | <if_statement> |
<while_statement> | <function_call> |
<function_declaration>

<assignment> ::= <identifier> "=" <expression>
<expression> ::= <roman_numeral> | <identifier> |
<binary_expression>
<binary_expression> ::= <expression> <operator>
<expression>
<operator> ::= "+" | "-" | "*" | "/"

<if_statement> ::= "if" "(" <condition> ")" "{"
<statements> "}" <else_part>
<else_part> ::= "else" "{" <statements> "}" | " "

<while_statement> ::= "while" "(" <condition> ")"
"{" <statements> "}"
<condition> ::= <expression> <comparison_operator>
<expression>
<comparison_operator> ::= "==" | "!=" | "<" | ">" |
"<=" | ">="

<function_call> ::= <identifier> "(" <arguments>
")" ";"
<arguments> ::= <expression> "," <arguments> |
<expression> | " "

<function_declaration> ::= "function" <identifier>
 "(" <parameters> ")" "{" <statements> "}"
```

```

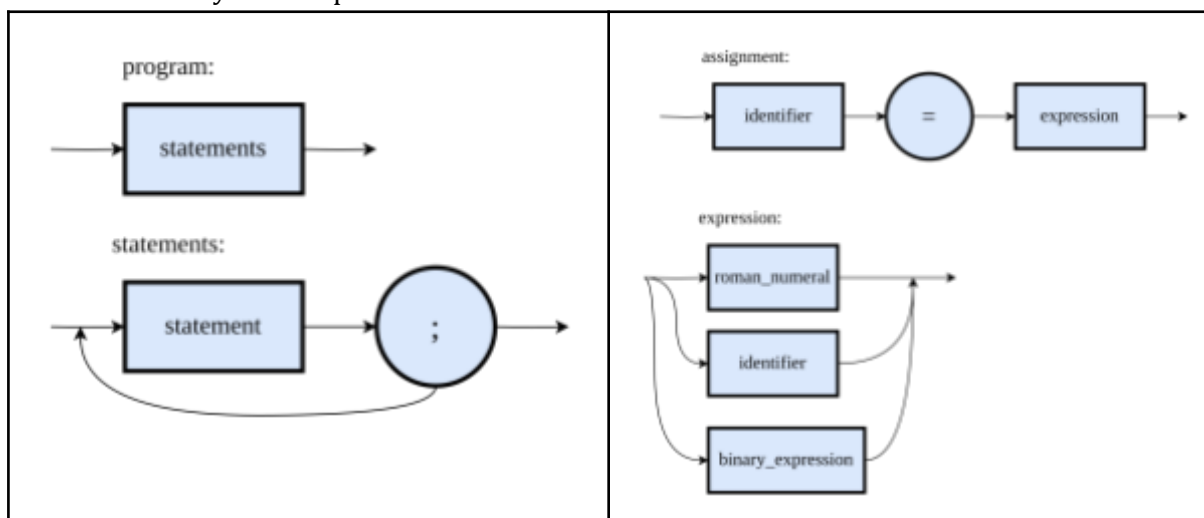
<parameters> ::= <identifier> "," <parameters> |
<identifier> | " "

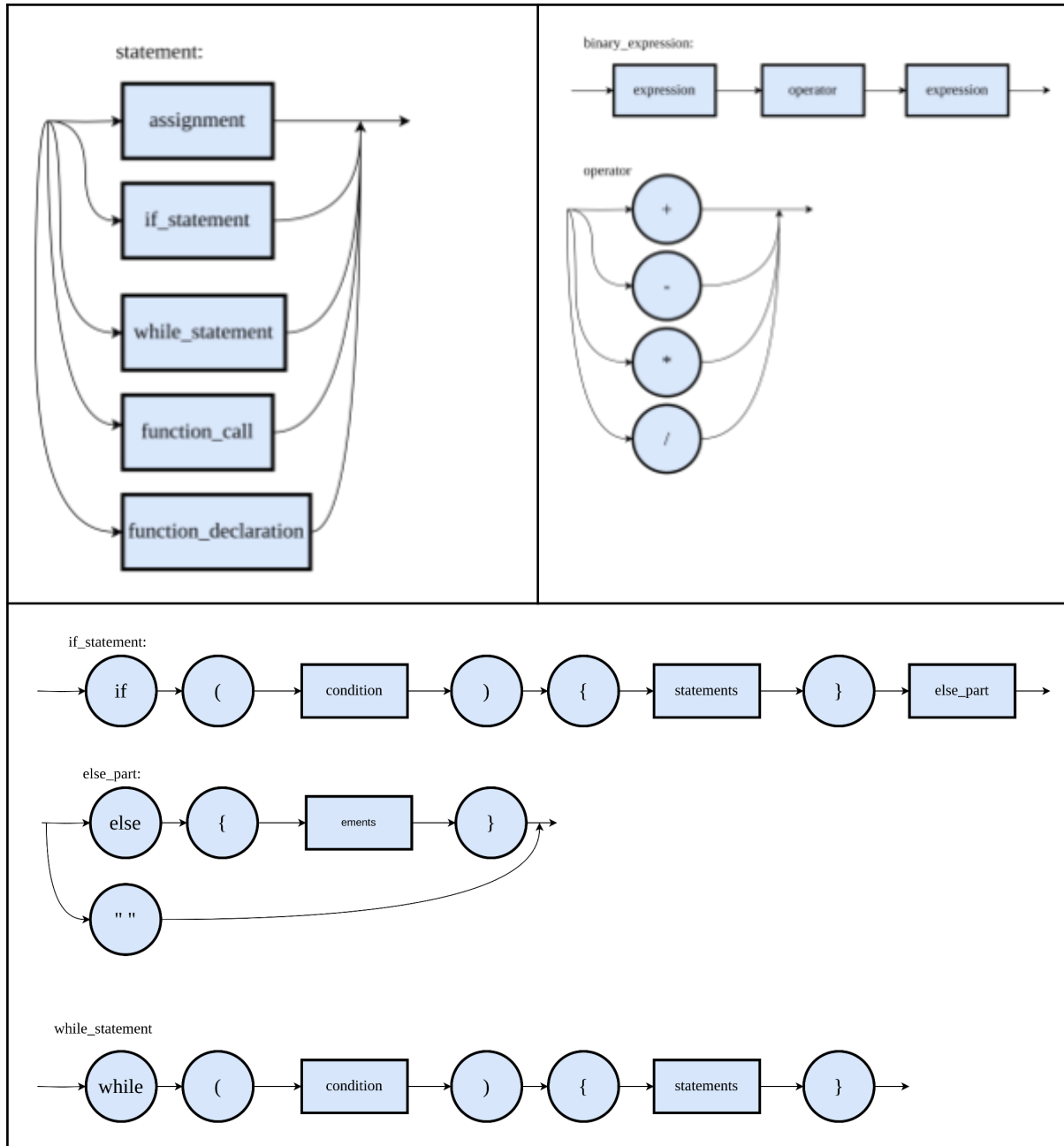
<roman_numeral> ::= <thousands> <hundreds> <tens>
<ones>
<thousands> ::= "M" <thousands> | " "
<hundreds> ::= "CM" | "CD" | "D" <hundreds_rest> |
"C" <hundreds_rest> | " "
<hundreds_rest> ::= "C" <hundreds_rest> | " "
<tens> ::= "XC" | "XL" | "L" <tens_rest> | "X"
<tens_rest> | " "
<tens_rest> ::= "X" <tens_rest> | " "
<ones> ::= "IX" | "IV" | "V" <ones_rest> | "I"
<ones_rest> | " "
<ones_rest> ::= "I" <ones_rest> | " "

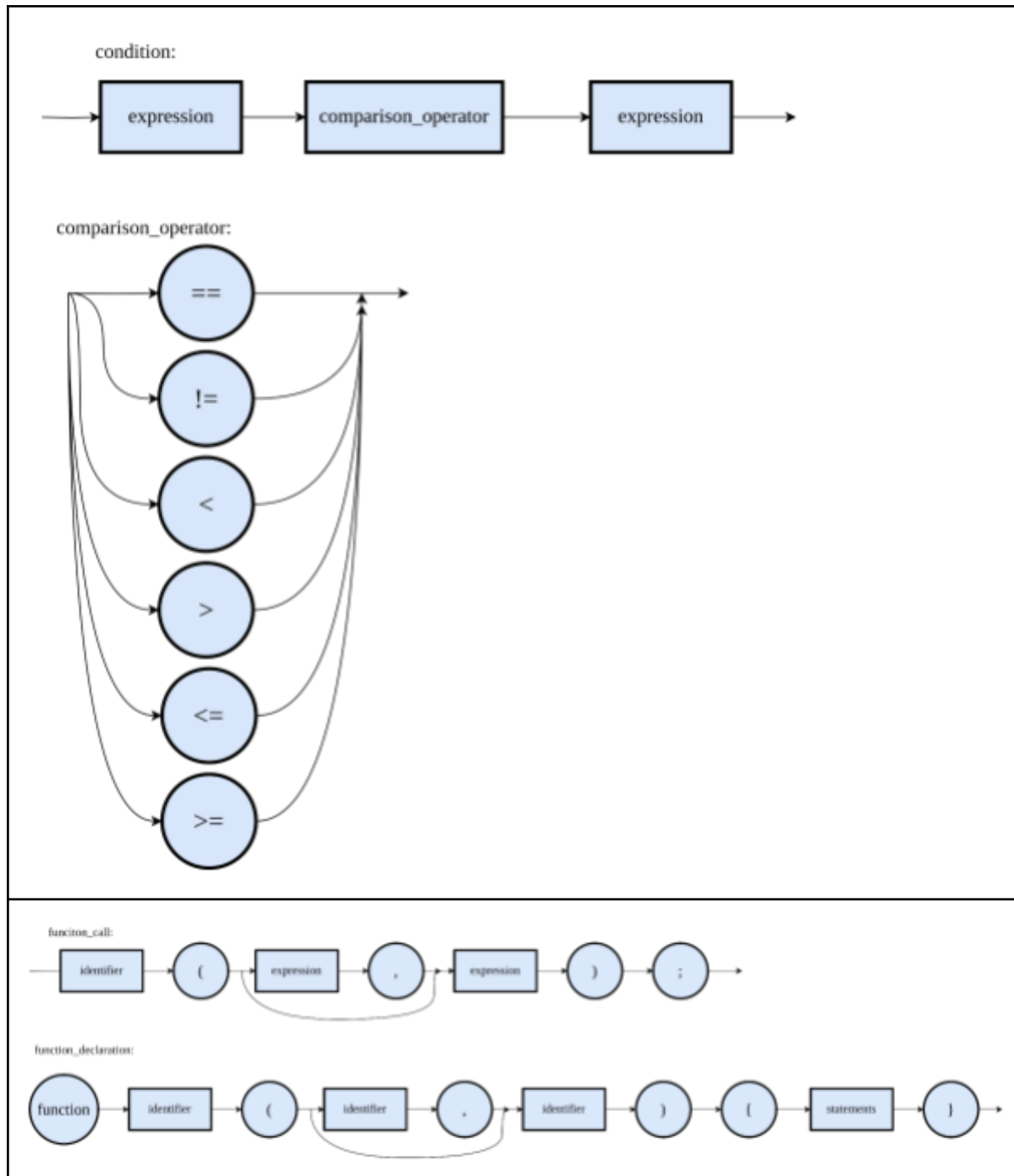
<identifier> ::= <letter> (<letter_or_digit>)*
<letter_or_digit> ::= <letter> | <digit>
<letter> ::= "a" | "b" | "c" | "d" | "e" | "f" |
"g" | "h" | "i" | "j" | "k" | "l" | "m" | "n" | "o"
| "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" |
"x" | "y" | "z" | "A" | "B" | "C" | "D" | "E" | "F"
| "G" | "H" | "I" | "J" | "K" | "L" | "M" | "N" |
"O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W"
| "X" | "Y" | "Z"
<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6"
| "7" | "8" | "9"

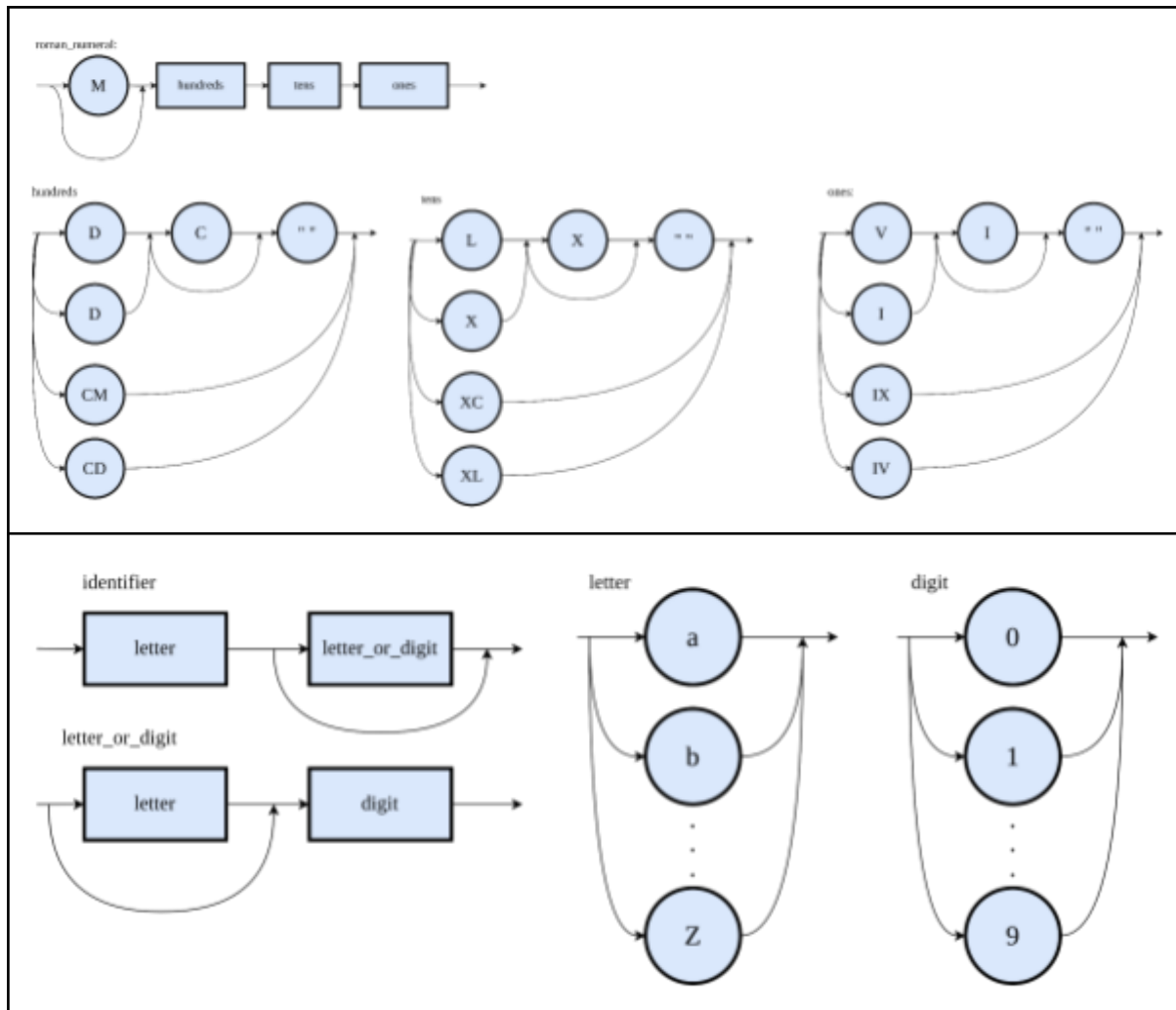
```

b. Syntax Graphs of the Grammars









c. At least 3 derivations

Derivations 1:

`<program> ::= <statements>`

`<statements> ::= <statement> ";"`

`<statement> ";" ::= <assignment> ";"`

`<assignment> ::= <identifier> "=" <expression> ";"`

`<identifier> ::= <letter> (<letter_or_digit>)*`

`<letter> ::= a`

`(<letter_or_digit>)* ::= empty (e)`

```

<expression> ::= <binary_expression> ";"
<binary_expression> ::= <expression> <operator> <expression> ";"
<expression> ::= <roman_numeral>
<roman_numeral> ::= <tens>
<tens> ::= X
<operator> ::= "+"
<expression> ";" ::= <roman_numeral> ";"
<roman_numeral> ::= <ones> ";"
<ones> ";" ::= V ";"
Final program: a = X + V;

```

Derivations 2:

```

<program> ::= <statements>
<statements> ::= <statement> ";"
<statement> ::= <if_statement>
<if_statement> ::= "if" "(" <condition> ")" "{" <statements> "}"
<else_part>
<condition> ::= <expression> <comparison_operator> <expression> ";"
expression ::= <roman_numeral>
<roman_numeral> ::= <tens> ";"
<tens> ::= X
<comparison_operator> ::= ">"
expression ::= <roman_numeral>
<roman_numeral> ::= <ones> ";"
<ones> ::= V
<statements> ::= <statement> ";"

```

```

<statement> ";" ::= <assignment> ";"
<assignment> ::= <identifier> "=" <expression> ";"
<identifier> ::= <letter> (<letter_or_digit>)*
<letter> ::= a
(<letter_or_digit>)* ::= empty (e)
<expression> ::= <binary_expression> ";"
<binary_expression> ::= <expression> <operator> <expression> ";"
<expression> ::= <roman_numeral>
<roman_numeral> ::= <tens>
<tens> ::= X
<operator> ::= "+"
<expression> ";" ::= <roman_numeral> ";"
<roman_numeral> ::= <ones> ";"
<ones> ::= V ";"
<else_part> ::= " "
Final program: if (X > V) { a = X + V; }

```

Derivation 3:

```

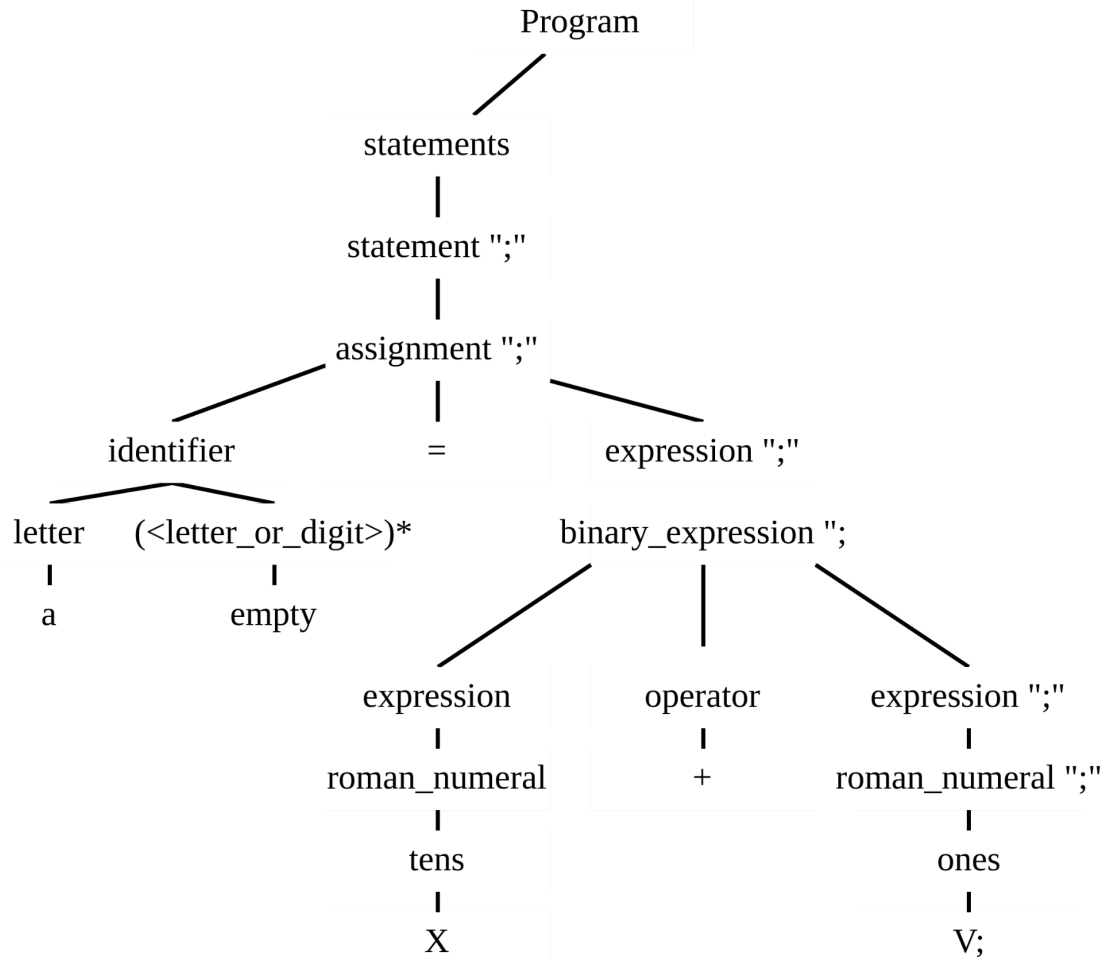
<program> ::= <statements>
<statements> ::= <statement> ";"
<statement> ::= <while_statement>
<while_statement> ::= "if" "(" <condition> ")" "{" <statements> "}"
<condition> ::= <expression> <comparison_operator> <expression> ";"
expression ::= <roman_numeral>
<roman_numeral> ::= <ones> ";"

```

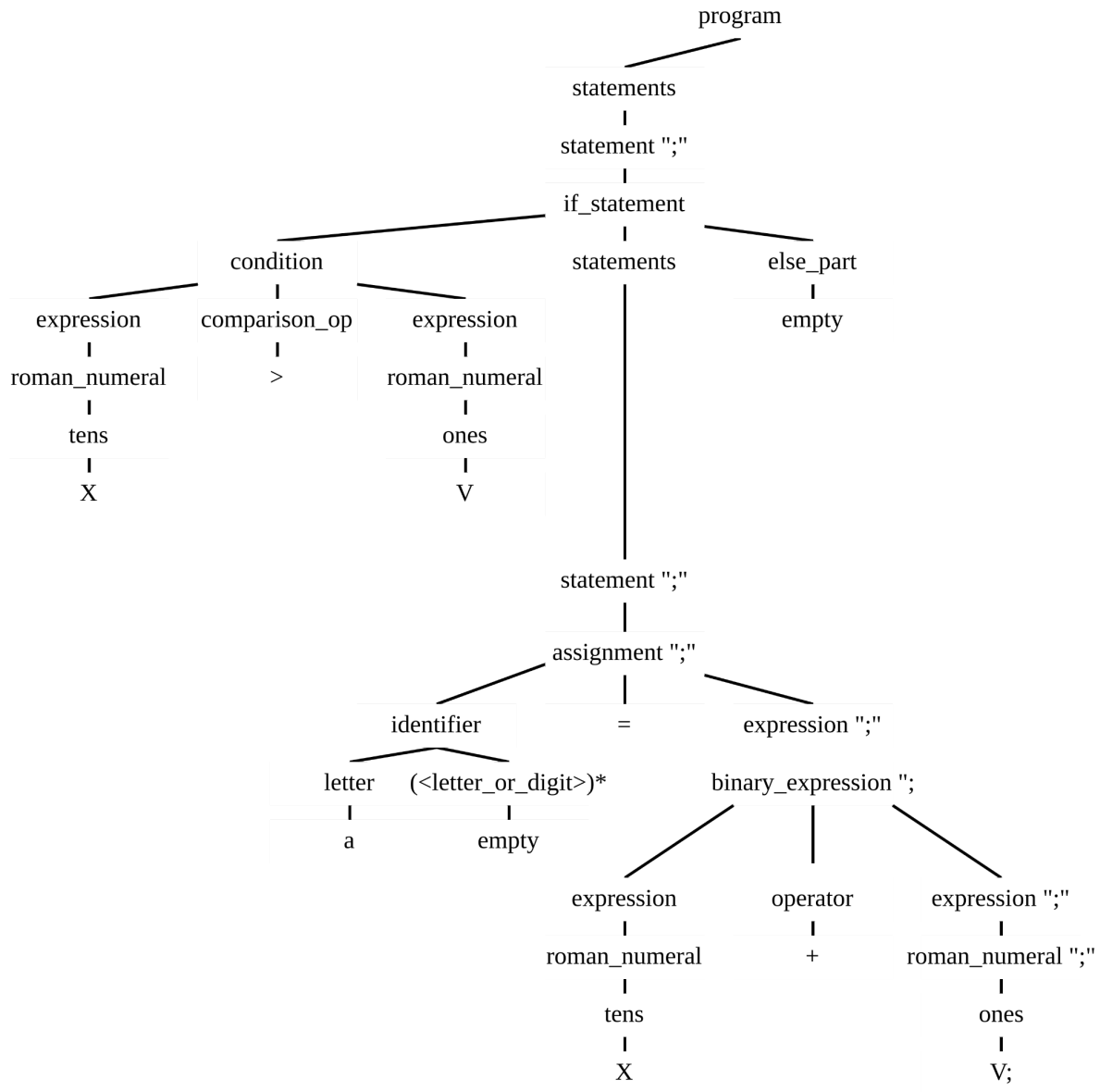
```
<ones> ::= I <ones_rest>
<ones_rest> ::= I <ones_rest>
<ones_rest> ::= empty
<comparison_operator> ::= ">"
expression ::= <roman_numeral>
<roman_numeral> ::= <tens> ";"
<tens> ::= X
<statements> ::= <statement> ";"
<statement> ";" ::= <assignment> ";"
<assignment> ::= <identifier> "=" <expression> ";"
<identifier> ::= <letter> (<letter_or_digit>)*
<letter> ::= b
(<letter_or_digit>)* ::= empty (e)
<expression> ::= <binary_expression> ";"
<binary_expression> ::= <expression> <operator> <expression> ";"
<expression> ::= <roman_numeral>
<roman_numeral> ::= <tens>
<tens> ::= X
<operator> ::= "-"
<expression> ";" ::= <roman_numeral> ";"
<roman_numeral> ::= <ones> ";"
<ones> ::= I <ones_rest>
<ones_rest> ::= I <ones_rest>
<ones_rest> ::= empty
```


d. Parse trees in lieu with the derivations

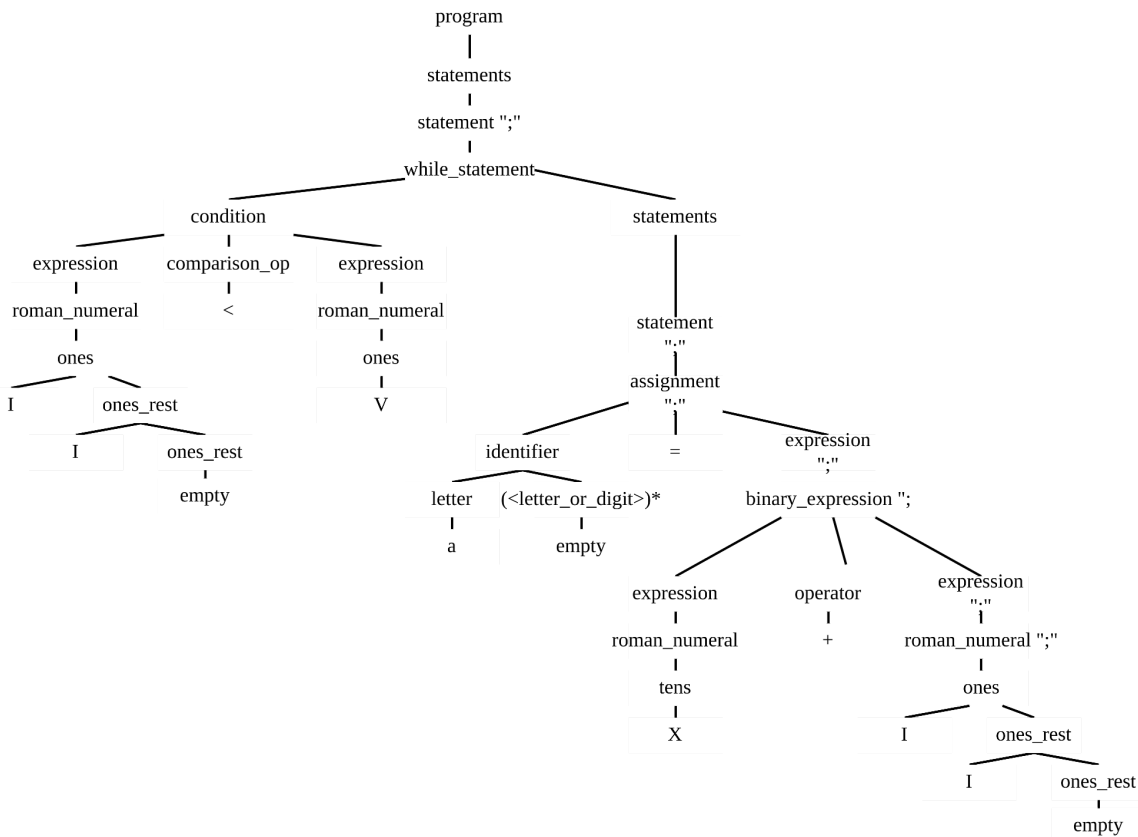
Parse Tree 1:



Parse Tree 2:



Parse Tree 3:



B. Semantics

a. Informal Semantics (*in lieu with each BNF*)

1. Program and Statements:

<program>: A program is a series of statements, terminated by a semicolon. Each statement performs a specific action, such as variable assignment, control flow, or function interaction.

<statements>: A collection of one or more valid statements. Statements are sequentially executed.

2. Statement Types:

<assignment>: Assigns the result of an expression (e.g., a Roman numeral or computation) to an identifier.

<if_statement>: Executes a block of statements if a condition evaluates to true; optionally executes an else block if false.

<while_statement>: Repeatedly executes a block of statements while a condition remains true.

<function_call>: Executes a previously declared function with the given arguments.

<function_declaration>: Defines a function with a name, parameters, and a body of statements.

3. Expressions:

<expression>: Evaluates to a Roman numeral, an identifier's value, or the result of a binary operation.

<binary_expression>: Combines two expressions using an arithmetic operator.

<operator>: Defines the valid arithmetic operations (addition, subtraction, multiplication, division).

4. Conditions:

<condition>: Evaluates the relationship between two expressions using a comparison operator.

<comparison_operator>: Defines equality, inequality, and relational comparisons.

5. Functions:

<function_call>: A reference to execute a named function using arguments.

<function_declaration>: Declares reusable code logic under a named function with optional parameters.

6. Roman Numerals:

<roman_numeral>: Represents the composition of a valid Roman numeral based on thousands, hundreds, tens, and ones.

Sub-rules (<thousands>, <hundreds>, <tens>, <ones>): Define valid Roman numeral symbols and their repetition.

7. Identifiers:

<identifier>: Represents variables or function names, starting with a letter and followed by letters or digits.

1. Basic Syntax

The program:

- Uses a statement-oriented structure where each action ends with a semicolon (;).
- Functions are defined using the function keyword, followed by parameters and a body enclosed in {}.
- Roman numerals are defined hierarchically (M, CM, etc.) for simplicity and reusability.

2. Control Structures

- If Statements: Provides conditional execution. If the condition is true, the if block executes; otherwise, the else block executes (if present).
- While Loops: Implements iteration based on a condition. The loop runs while the condition is true.

3. Data Types

- Roman Numerals: Represented as strings (e.g., X, IV, MMXVII).
- Identifiers: Used for variables and functions. These hold values like Roman numerals or integers.
- Expressions: Include arithmetic or logical combinations of values.
- Conditions: Boolean expressions that evaluate to true or false.

4. Subprograms

- Function Declaration: Defines reusable blocks of code. Functions are invoked using their name and optional arguments.
- Function Call: Executes the logic of a declared function with specific inputs.

b. Formal Semantics (*in lieu with each BNF, whichever is applicable*)

■ Operational Semantics

Given `<roman_numeral>`, the operational semantics specify:

- Start at the last character of the string.
- For each Roman numeral character, look up its value using a mapping (e.g., 'I' \rightarrow 1).
- Accumulate the result while deciding whether to add or subtract based on the numeral's position relative to its neighbor.
 - Evaluate `<condition>`. If true, execute the `<statements>` in `{}`. Otherwise, execute `<else_part>` if defined.
- **`<while_statement>`:**
 - Repeatedly evaluate `<condition>`. If true, execute the `<statements>` in `{}` until the condition evaluates to false.

Control Flow(`<if_statement>` and `<while_statement>`):

- **`<if_statement>`:**
 - Evaluate `<condition>`. If true, execute the `<statements>` in `{}`. Otherwise, execute `<else_part>` if defined.
- **`<while_statement>`:**
 - Repeatedly evaluate `<condition>`. If true, execute the `<statements>` in `{}` until the condition evaluates to false.

• Denotational Semantics

State and Environment:

- **State:** A function mapping variables (e.g., identifiers) to their current values.
- **Environment:** A mapping from identifiers to their definitions
- **Semantics for Roman Numerals (`<expression>`):**

- Denotation of a Roman numeral string `roman` is a function `FFF`

$$F(roman) = i = 0 \sum^n - 1 value(roman[i]) \pm adjustments$$

- Axiomatic Semantics
- **Hoare Triples:** $\{P\} S \{Q\}$
 - **P:** Precondition (what must be true before executing `S`).
 - **S:** Statement.
 - **Q:** Postcondition (what must be true after executing `S`).
- **For Control Flow:**
 - While loop: $\{P \wedge C\} while(C) S \{P \wedge \neg C\}$
 - **P:** Invariant true before and after each iteration.
 - **C:** Loop condition.

Evaluation

The four programming paradigms in this project, Imperative, Object-Oriented, Logic, and Functional or Common Lisp, take distinct approaches to solving Roman numeral conversion problems. Each paradigm displays strengths and weaknesses in terms of readability, efficiency, and expressiveness.

1. Imperative Paradigm (C):
 - It is efficient because it has direct memory manipulation and is a widely used and well-understood paradigm, however, this can lead to complex and error-prone code, especially in terms of larger problems, it is also less expensive and harder to reason about compared to other paradigms.
2. Object-Oriented Paradigm (Java):
 - The OOP paradigm encapsulates data and behavior into objects, this prompts reusability and modularity, providing a clear structure for organizing code. Its weakness is that it introduces overhead due to object creation and method calls requiring careful design to avoid over-engineering.
3. Logic Paradigm (Prolog):
 - Prolog is a declarative style that allows concise and great solutions making it well suited for problems that involve pattern matching and search, but this is inefficient in large-scale problems due to backtracking because it requires a different mindset and learning curve.
4. Functional Paradigm (Common Lisp):
 - Common Lisp encourages pure functions and avoids side effects which leads to a more readable and testable code, this supports higher-order functions and lazy evaluation for concise solutions. Its weakness lies in it being less intuitive for programmers to be accustomed to imperative or object-oriented style as it requires more advanced programming techniques and understanding.

Concluding Statements

This project successfully demonstrated the application of four distinct programming paradigms to a common problem which is the Roman Numeral Conversion. While each paradigm offers unique solutions in terms of advantages and disadvantages, the choice of the paradigm depends on factors like the problem domain, preference, and lastly requirements.

Understanding the strengths and weaknesses of each paradigm can help significantly in terms of informed decisions about which approach is the best suited for a particular task, in the context of the Roman Numeral Conversion the choice of paradigm may not have any significant impact on the performance, however, it can influence the readability, maintainability and overall elegance of the solution.

In the future this project could explore more complex Roman numerical conversions like handling larger numbers or more intricate rules, in addition, a comparative analysis of the performance and memory usage of the different implementations could provide some valuable insights.

Bibliographic Sources

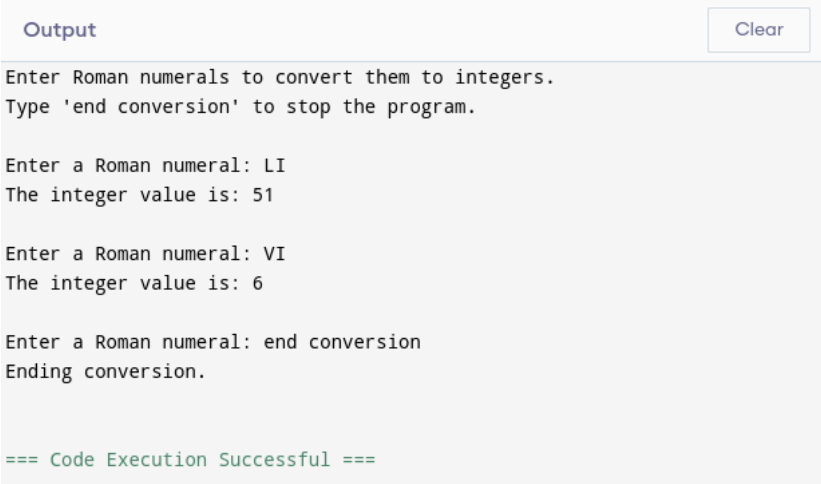
[1]<https://leetcode.com/problems/roman-to-integer/description/>

[2]<https://medium.com/@AlexanderObregon/solving-the-roman-to-integer-problem-on-leetcode-c-solutions-walkthrough-682dd9e29452>

Appendices

A. Screenshots

1. Imperative - C



```
Output Clear
Enter Roman numerals to convert them to integers.
Type 'end conversion' to stop the program.

Enter a Roman numeral: LI
The integer value is: 51

Enter a Roman numeral: VI
The integer value is: 6

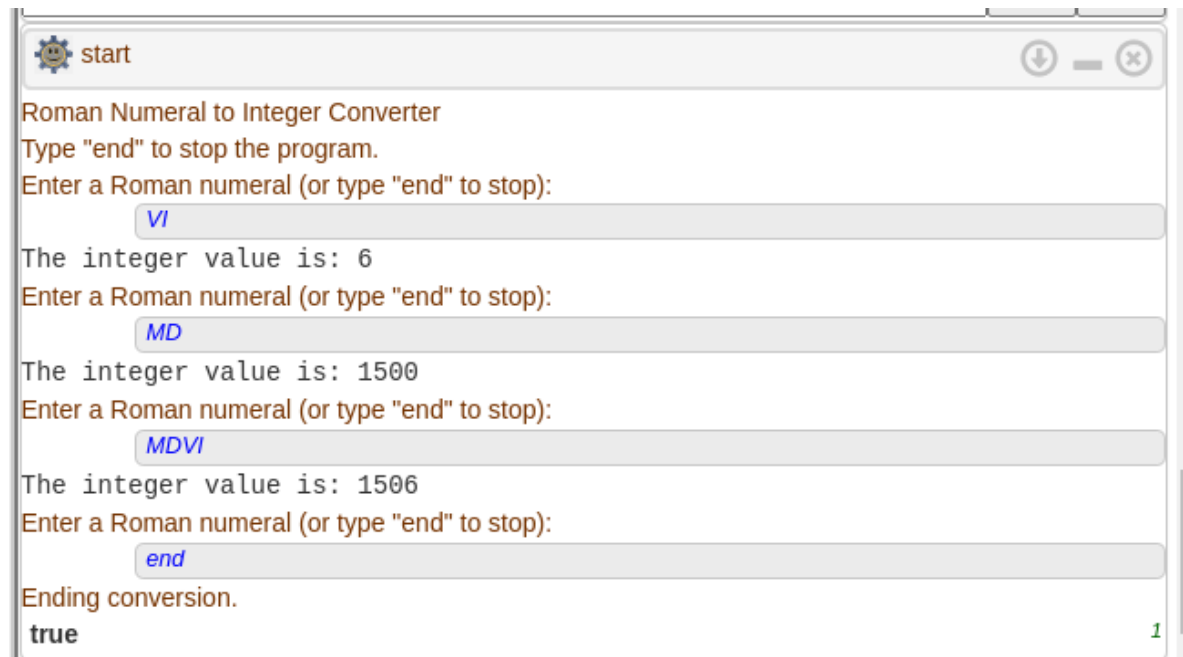
Enter a Roman numeral: end conversion
Ending conversion.

=== Code Execution Successful ===
```

2. Object-Oriented - Java

```
Enter Roman numerals to convert them to integers.  
Enter Roman numerals to convert them to integers.  
Type 'end conversion' to stop the program.  
  
Enter a Roman numeral: MD  
The integer value is: 1500  
  
Enter a Roman numeral: MDII  
The integer value is: 1502  
  
Enter a Roman numeral: end conversion  
Ending conversion.  
  
...Program finished with exit code 0  
Press ENTER to exit console.
```

3. Logic - Prolog



4. Functional - Common Lisp

```
Terminal

Roman Numeral to Integer Converter
Type 'end' to stop the program.
Enter a Roman numeral (or type 'end' to stop): VI
The integer value is: 6
Enter a Roman numeral (or type 'end' to stop): MD
The integer value is: 1500
Enter a Roman numeral (or type 'end' to stop): MDVI
The integer value is: 1506
Enter a Roman numeral (or type 'end' to stop): |
```

B. Photo documentation

The screenshot shows a Google Meet window with a presentation titled "Programiz Online Java Compiler". The code being presented is a Java program for converting Roman numerals to integers. The code is as follows:

```
44 System.out.println("Enter Roman numerals to convert them  
45 to integers.");  
46 System.out.println("Type 'end conversion' to stop the  
47 program.");  
48 while (true) {  
49     System.out.print("\nEnter a Roman numeral: ");  
50     String input = scanner.nextLine().trim();  
51  
52     // Check if the user wants to end the conversion  
53     if (input.equalsIgnoreCase("end conversion")) {  
54         System.out.println("Ending conversion.");  
55         break;  
56     }  
57  
58     // Convert Roman numeral to integer  
59     int result = convertRomanToInt(input);  
60  
61     // Check for invalid input  
62     if (result == 0) {
```

The output of the program is displayed on the right side of the compiler interface:

```
Enter Roman numerals to convert them to integers.  
Enter Roman numerals to convert them to integers.  
Type 'end conversion' to stop the program.  
Enter a Roman numeral:
```

The Google Meet interface shows several participants in a grid view on the right, including Imroz Mae S. Khan, Via Nicole A. Preciado, Ismael Sulpicio Zarate, Febron Jr. B. Sedoriosa, Ella Decapio, and Cherry Jimenez. The bottom of the screen shows the time as 12:01 AM and the URL as ceo-hdho-www.