Roman Numerals

Roy Ratcliffe[1,*]

Abstract

This text explores the conversion between Roman numerals and Hindu-Arabic numbers through a logical and computational framework, emphasizing the significance of Roman numerical legacy within the context of Western civilization. It delves into the structure of Roman numerals, detailing their components and the principles governing their formation, particularly using a constraint-logic programming approach in Prolog. The recursive grammar and backtracking logic employed facilitate the generation of multiple representations for the same numerical value, showcasing the inherent flexibility within Roman numeral representation. Additionally, the implementation demonstrates the absence of a numeral for zero, aligning with historical Roman numeral conventions. The succinctness of the representation is prioritized, emphasizing the importance of larger factors in rendering numbers. This exploration highlights both the strengths and weaknesses of the logical implementation while providing insights into the historical and mathematical context of Roman numeral systems.

Keywords:
Prolog

Friends, Romans, countrymen, lend me your ears—
Marc Antony from William Shake-speare's (likely Edward de Vere's, hyphenation intentional) Julius Caesar.

Apparently, men think about the Roman Empire[2] regularly, a significant proportion of us at least once a week. I confess to be among them. The United Kingdom is full of Roman ruins; it is hard to escape their legacy, especially in the north.

0.1. Carpe Diem

With the Romans in mind, how easily can software logic convert between Roman numerals and Arabic numbers? Strictly speaking, what we traditionally call Arabic numbers should be called Hindu-Arabic numbers since the Arabic mathematicians took the decimal system from the Hindu mathematicians. Call it "decimal" for short.

Roman to decimal and back. Forwards and backwards matters. The solution must give Roman for decimal and decimal for Roman. Complete symmetry is required.

0.2. Veni, Vidi, Vici

Fundamentally, the Roman "number" represents a numerical sum using 9, 5, 4 and 1 as the principal factors. $I = 1$, $V = 5$ and $X = 10$. Prefix V and X with I to represent 4 and 9, respectively. The same pattern recurs for $L = 50$, $C = 100$, $D = 500$ and $M = 1000$.

In definite-clause grammar terms [1] this formula becomes the following logical phrase.

```
roman_numeral(1000) --> "M".
roman_numeral(900)  --> "CM".
roman_numeral(500)  --> "D".
roman_numeral(400)  --> "CD".
roman_numeral(100)  --> "C".
roman_numeral(90)   --> "XC".
roman_numeral(50)   --> "L".
roman_numeral(40)   --> "XL".
roman_numeral(10)   --> "X".
roman_numeral(9)    --> "IX".
roman_numeral(5)    --> "V".
roman_numeral(4)    --> "IV".
roman_numeral(1)    --> "I".
```

Here, I ignore the upper-lower-case issue. Roman numerals have upper case only, although frequently the numerals can have either case, albeit consistently all upper or all lower.

These are the factors. Composite numerals comprise a sequence of these sub-phrases. They need to add up to some numerical value. Enter constraint-logic programming for finite domains or CLP_{FD} [2] for symbolically representing logical relations between numbers.

```
roman_numerals(Number) -->
    { Number #> 0,
      Number #= Number0 + Number1
    },
    roman_numeral(Number0),
    roman_numerals(Number1).
roman_numerals(0) --> [].
```

[*]Corresponding author
Email address: roy@ratcliffe.me (Roy Ratcliffe)
[1]See more hackery at GitHub.
[2]Napoleon and Hitler also

This is a recursive phrase. For every *Number* > 0, there is a sum $Number = Number_0 + Number_1$ where $Number_0$ is a Roman numeral, *and* $Number_1$ is the sum of subsequent Roman numerals. Finally, no Roman numeral `[]` corresponds to 0. The initial clause applies arithmetic constraints. The terms do not need to bind to integers initially. Hence, the constraints appear first at the head of the predicate.

"Quod erat demonstrandum!"

0.2.1. How does it work?

Prolog searches the problem space using the given constraints. The sum of numbers must always sum to a positive result. The sum $\sigma > 0$ can never be zero or below. Romans did not grasp zero or negatives, apparently. Perhaps they did grasp the abstractions but found in them little practical value; Romans were pragmatic people after all.

The backtracking logic has an interesting side effect. It finds **all** the possible Roman representations of a number. Using the simple Roman combinatorial logic, one number has multiple alternative representations in Roman numerals.

Take 5 for example.

```
?- phrase(roman_numerals(5), A), string_codes(B, A).
A = [86],
B = "V" ;
A = `IVI`,
B = "IVI" ;
A = `IIV`,
B = "IIV" ;
A = `IIIII`,
B = "IIIII".
```

There are four alternative representations:

1. V
2. $IV + I = V$
3. $I + IV = V$
4. $I + I + I + I + I = V$

Clause order matters for the `roman_numeral//1` predicate; big numbers must come first so that the solution finds larger factors before smaller ones. Romans being Roman, the "correct" representation is the shortest possible representation, or put another way: the form with the largest possible factors. Hence big factors take priority over smaller ones.

It helps, therefore, to wrap the grammar using a cut (`!`). The following terminates the search for a solution when it finds the first one; the first solution being the sum with the largest factors.

```
roman_number(Roman, Number) :-
    phrase(roman_numerals(Number), Roman),
    !.
```

What's Roman for 9999?

```
?- roman_number(A, 9999).
A = `MMMMMMMMMCMXCIX`.

?- roman_number(`MMMMMMMMMCMXCIX`, B).
B = 9999.
```

The Roman term, *A* in the previous query, unifies with a list of character codes hence the backticks.

0.3. Strengths and Weaknesses

The logical implementation has a somewhat surprising outcome: Roman numerals have alternative renderings.

The representation for 0 is **logically** blank. That makes total sense, come to think about it. Subtly, the unification does not *fail.* It succeeds with nothing instead. This implies that the Romans could represent zero by writing nothing.

```
?- phrase(roman_numerals(0), A).
A = [].
```

Prolog makes the implementation pretty simple, but there are some subtleties: clause order has semantic significance; green cutting likewise.

References

[1] M. Triska, Prolog dcg primer, accessed: 2024-12-19 (2024).
URL https://www.metalevel.at/prolog/dcg

[2] M. Triska, Constraint logic programming over finite domains (clp(fd)), accessed: 2024-12-19 (2025).
URL https://github.com/triska/clpfd