“Roman Numerals” Case Study
© 1996 M. J. Clancy and M. C. Linn

Background

Values less than 3999 in the Roman numeral system are written using seven “digits” whose decimal equivalents are given in the table below. (There are additional conventions for writing values larger than 3999, which we will not discuss here.)

Roman digit	*decimal equivalent*
M	*1000*
D	*500*
C	*100*
L	*50*
X	*10*
V	*5*
I	*1*

Roman numerals are composed using the following rules.

Digit order: Roman digits are written in nonascending order, and digit values are added to produce the represented value, except for prefixes as mentioned below.

Number of occurrences: No more than three occurrences of M, C, X, or I may appear consecutively, and no more than one D, L, or V may appear at all.

Prefixes: The digits C, X, and I can be used as prefixes, and their value is then subtracted from rather than added to the value being accumulated, as follows:

- One C may prefix an M or a D to represent 900 or 400; the prefixed M is written after the other M’s, as in MMMCM. The digits following the M or D represent a value no more than 99.
- One X may prefix a C or an L to represent 90 or 40; the prefixed C is written after the other C’s. The digits following the C or L represent a value no more than 9.
- One I may prefix an X or a V to represent 9 or 4. The prefixed digit must appear at the end of the numeral.

Some examples:

decimal value	*Roman numeral representation*
1987	*MCMLXXXVII*
1999	*MCMXCIX*
339	*CCCXXXIX*

Problem

Write and test a function `decimal-value` that, given a list representing a legal Roman numeral, returns the corresponding decimal value. Each element of the argument list will be a single Roman digit, for example, `(M C M X C I X)`. Given this list as argument, your `decimal-value` function should return 1999.

Preparation

The reader should have already been introduced to recursion using lists, and should be familiar with part 2 of the “Difference Between Dates” case study.

Exercises

Analysis 1. What is the decimal value of each of the following Roman numeral values: MCMLII, MMCDV, CXCIX?

Analysis 2. How is each of the following decimal values written using Roman numerals: 1988, 1000, 1349?

Analysis 3. For each of the following, determine its value if it's a legal Roman numeral, or describe which of the rules listed in the problem statement that it violates: XMICVC, XLIX, IIX, XIXIV.

Analysis 4. How many Roman digits can the argument list contain? (I.e. what's the longest Roman numeral whose value is 3999 or less?)

Preliminary planning

What's a good first step toward a solution?

The function to be written for this problem takes a list of Roman digits as its argument and returns a number, the value of the corresponding Roman numeral. How are we to do this?

We first note that recursion will be useful. In part II of the "Difference Between Dates" case study, recursion provided a way to add the days in months in a given range, no matter how many months the range included. Here, a list of Roman digits must somehow be turned into a number. The number of Roman digits contained in the list isn't known, so some method for repeating a computation until the end of the list is appropriate. Such repetition in Scheme is done recursively.

Recursion involves one or more base cases and one or more recursive calls, so one way to proceed would be to plunge in directly and look for the base cases and recursion. For example, most recursive list processing functions involve the empty list as a base case.

Another approach would be to try to apply solutions we've already devised. We haven't seen anything quite like Roman numeral translator, so this approach would involve breaking the solution down into smaller pieces and then recognizing one of the pieces as something we've seen before. An advantage of this second approach is that it's likely to waste less time. Recall that in part I of the "Difference Between Dates" we encountered a dead end, then were forced to rethink our solution. A dead end is a possibility with any problem. Looking for existing code that that we can modify for a solution thus is a safer approach to design.

What programming patterns may be applicable here?

The problem statement says that "digit values are added to produce the represented value, except for prefixes". Adding numbers in a list was done in part II of “Difference Between Dates”:

```
(define (element-sum number-list)
   (if (null? number-list) 0
      (+ (car number-list)
         (element-sum (cdr number-list)) ) ) )
```

This is an instance of an *accumulating recursion* that combines elements of a list. Similar code that accumulates values of Roman digits will be useful here.

Another component of the solution will be *translation*. We can’t work with Roman digits directly, at least in a convenient way. In part I of “Difference Between Dates”, we translated dates to numbers in order to subtract them more easily. Here, it makes sense to do something similar: translate Roman digits to numbers in order to add them more easily.

What kind of translation provides a good model for translating Roman digits?

In “Difference Between Dates”, we translated dates to numbers and month names to numbers. The month-to-number translation seems more relevant here since month names and Roman digits are both symbols; here it is.

```
(define (month-number month)
   (cond
      ((equal? month 'january) 1)
      ((equal? month 'february) 2)
      ((equal? month 'march) 3)
      ((equal? month 'april) 4)
      ((equal? month 'may) 5)
      ((equal? month 'june) 6)
      ((equal? month 'july) 7)
      ((equal? month 'august) 8)
      ((equal? month 'september) 9)
      ((equal? month 'october) 10)
      ((equal? month 'november) 11)
      ((equal? month 'december) 12) ) )
```

The code to translate a Roman digit to its corresponding decimal value is similar:

```
(define (decimal-digit-value roman-digit)
   (cond
      ((equal? roman-digit 'm) 1000)
      ((equal? roman-digit 'd)  500)
      ((equal? roman-digit 'c)  100)
      ((equal? roman-digit 'l)   50)
      ((equal? roman-digit 'x)   10)
      ((equal? roman-digit 'v)    5)
      ((equal? roman-digit 'i)    1) ) )
```

Ordinarily one might add a `cond` case to return a value for the default case—here, something that’s not a legal Roman digit—but the problem statement has specified that only legal Roman numerals need be considered.

How is the entire sequence of Roman digits translated?

Extending the translation to an entire list of Roman digits involves including it in code that returns the result of applying a function to every element of a list* :

```
(define (funct-applied-to-all L)
    (if (null? L) '( )
        (cons
            (funct (car L))
            (funct-applied-to-all (cdr L)) ) ) )
```

Applying this template produces a function that translates all the Roman digits in a list to their decimal digit values, for example as follows:

list	*translated list*
(M C M X C I X)	(1000 100 1000 10 100 1 10)

Here's the code.

```
(define (digit-values roman-digit-list)
    (if (null? roman-digit-list) '( )
        (cons
            (decimal-digit-value (car roman-digit-list))
            (digit-values (cdr roman-digit-list)) ) ) )
```

How are the translation and accumulation steps combined?

The decimal-value function will involve *both* translating and accumulating; thus we'll have to combine the digit-values and element-sum functions somehow. Straightforward function composition along the lines of

```
(define (decimal-value roman-digit-list)
    (element-sum
        (digit-values roman-digit-list) ) )
```

won't completely solve the problem, since we need somehow to deal with prefixes. However, the code just designed does work for *some* Roman numerals and, with luck, we'll only need to make a few changes to make it work in general.

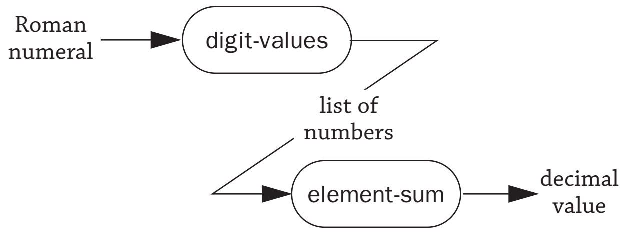
Stop and help ➨ *For which Roman numerals does the* decimal-value *function above correctly return the corresponding decimal value?*

How can the process be represented in a diagram?

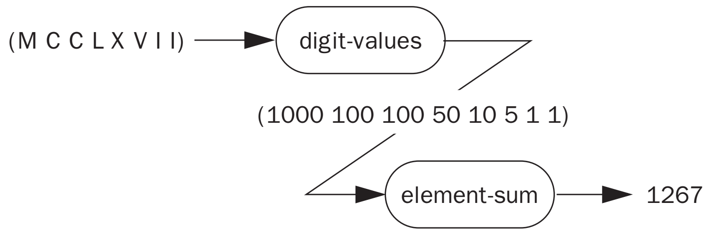
A *data flow diagram* helps to keep track of the steps in a computation. In a data flow diagram, functions are represented as boxes and their arguments and returned results are repre-

* This is often called a *mapping* function, since it "maps" another function onto each element of the argument list. The term "map" arises in mathematical terminology.

sented as labeled arrows. The diagram for the Roman numeral evaluation appears below.



It means that first the digit-values function is applied to the Roman numeral, and then the element-sum function is applied to the list of numbers that results. Here's an example.



Exercises

Analysis 5. When the decimal-value function just designed is given a Roman numeral with a prefix, is the value it returns too high or too low? Briefly explain.

Analysis 6. What's the value of (decimal-value '( ))?

Analysis 7. Sometimes inexperienced programmers get the order of function applications confused. Describe the error message from calling a decimal-value function coded as

```
(define (decimal-value roman-digit-list)
   (digit-values
      (element-sum roman-digit-list) ) )
```

Analysis 8. What error message results from supplying an argument to decimal-value that contains something that's not a Roman digit?

Modification 9. Suppose that Scheme provided a function called position that, given a list and an item, returned the position of the item in the list. (Position would be the inverse of the list-ref function.) Rewrite the decimal-digit-value as a call to position.

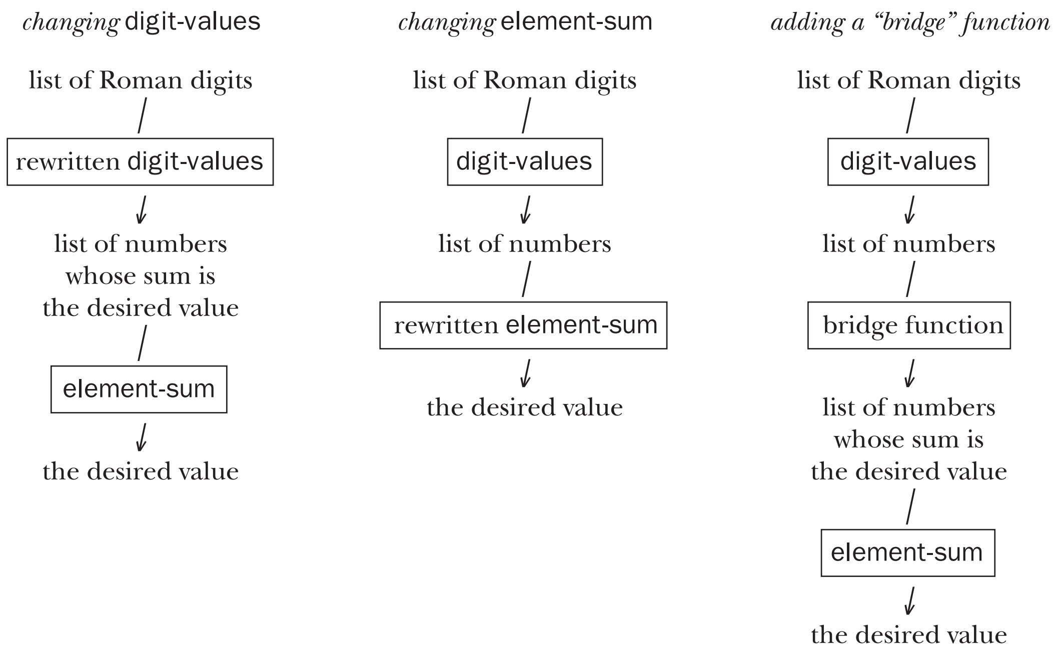
Modification 10. Rewrite decimal-value to return the value 0 if its argument contains something that's not a Roman digit.

Analysis 11. Explain why digit-values appeared before element-sum in the data flow diagram, but appeared after element-sum in the Scheme code for the decimal-value function.

Extending the partial solution to work for all Roman numerals

Which function must be extended to complete the program?

Our solution so far has two components, the digit-values function that translates a list of Roman digits to a list of numbers and the element-sum function that sums the numbers in the list. Extending this partial solution to a set of functions that works for Roman numerals that have prefixes requires either that one of the two parts we already have be modified or that a third component—a "bridge" function that fits between the two—be added. In diagram form, our choices are as follows:



Which design seems easiest?

We note that element-sum is less complicated than digit-values; modifying element-sum leaves us with two moderately complex functions, while modifying digit-values results in an even more complicated function. Thus we decide to leave digit-values alone, and focus on code that, given a list of numbers representing values of individual Roman digits, either sums the values appropriately or converts the list to a collection of values that need merely be added to produce the desired result.

How should prefixes be handled?

The problem statement describes how Roman digits may be used as prefixes. It makes sense here to replace the Roman digits in that description by their numeric values, to get some insight about how the list of numbers must be transformed.

- 100 may prefix 1000 or 500 to represent 900 or 400; the prefixed 1000 is written after the other 1000's, as in (1000 1000 1000 100 1000). The numbers following the (prefixed) 1000 or 500 represent a value no more than 99.

- 10 may prefix 100 or 50 to represent 90 or 40; the prefixed 100 is written after the other 100's. The numbers following the (prefixed) 100 or 50 represent a value no more than 9.
- 1 may prefix 10 or 5 to represent 9 or 4. The prefixed 10 or 5 must appear at the end of the list of numbers.

The examples from the problem statement would then appear as follows:

$$\texttt{(1000 }\underline{\texttt{100 1000}}\texttt{ 50 10 10 10 5 1 1)}$$

represents 900

$$\texttt{(1000 }\underbrace{\texttt{100 1000}}_{900}\ \underbrace{\texttt{10 100}}_{90}\ \underbrace{\texttt{1 10}}_{9}\texttt{)}$$

$$\texttt{(100 100 100 10 10 10 }\underbrace{\texttt{1 10}}_{9}\texttt{)}$$

What design is suggested by the examples?

Those examples indicate what a bridge function should do: replace each sequence of two numbers that represent a prefix and the prefixed digit by the corresponding single number. This number is the result of subtracting the value of the prefix from the value of the prefixed digit. We'll call the bridge function prefix-values-removed since it will return a list all prefix values combined with the values they prefix.

How can recursion be used to combine a prefix with the prefixed value?

The need to process the entire list of numbers in this way again suggests the need for recursion. We start with a general framework for a recursive function:

```
(define (prefix-values-removed number-list)
    (cond
        (base-case-1 expr-to-return-1)
        (base-case-2 expr-to-return-2)
            ...
        (recursion-case-1
         expr1-involving-prefix-value-removed)
        (recursion-case-2
         expr2-involving-prefix-value-removed)
            ... ) )
```

Prefix-values-removed will also return a list; thus it is likely that the expressions representing the return values for the recursion cases will involve some list-building function such as cons. (The digit-values function already designed contains a call to cons.)

What are the base cases?

The base cases in prefix-values-removed are the situations where an answer can be immediately returned. It's usually pretty easy to identify these—typical situations involve small lists—so we often design them first before considering the recursive calls. Here's a table.

small list	*result that* prefix-values-removed *should return*
empty list	empty list
list of length 1 (which therefore contains no prefixes)	that list
a list of length 2 in which the first element isn't less than the second	that list

It often happens that some of the base cases aren't really necessary for the function to work, but it doesn't hurt to include them.

Another situation in which we can immediately return an answer involves a list with no prefixes, that is, a list whose values are arranged in decreasing order. *Detecting* this situation, however, is likely to be almost as difficult as writing the remainder of prefix-values-removed, so we temporarily reject it as a base case.

Substituting our base cases into the framework for prefix-values-removed, we have the following:

```
(define (prefix-values-removed number-list)
    (cond
        ((null? number-list) '( ))
        ((null? (cdr number-list)) number-list)
        ((and (null? (cddr number-list))
                (>= (car number-list)
                    (cadr number-list) ) )
            number-list)
        (recursion-case-1
         expr1-involving-prefix-value-removed)
        (recursion-case-2
         expr2-involving-prefix-value-removed)
            ... ) )
```

How can the value returned by a recursive call be used?

Designing the recursive calls involves considering applications of the function to smaller lists, assuming those calls will work correctly, and then figuring out how to use their results. Some programmers find it difficult to "believe in" the recursive call, since the function being called hasn't yet been written. However, if there is a complete specification for what the function should do—as we have for prefix-values-removed—one can use it in the same way as programmers use builtin functions without knowing how they are coded.

A typical smaller list is one that consists of all but one of the given list's elements, say, all but the first. In prefix-values-removed, that would be

```
(prefix-values-removed (cdr number-list) )
```

How would the value returned by this expression be useful? Let's look at some examples (again, those from the problem statement).

`number-list`	`(prefix-values-removed (cdr number-list) )`	*value* prefix-values-removed *should return*
`(1000 100 1000 50 10 10 10 5 1 1)`	`(900 50 10 10 10 5 1 1)`	`(1000 900 50 10 10 10 5 1 1)`
`(1000 100 1000 10 100 1 10)`	`(900 90 9)`	`(1000 900 90 9)`
`(100 100 100 10 10 10 1 10)`	`(100 100 10 10 10 9)`	`(100 100 100 10 10 10 9)`

In these examples, all that's necessary is to cons the first number in number-list onto the list returned by the recursive call to prefix-values-removed and return that.

These examples, however, didn't cover all the situations. None of them *started* with a prefix. We must also consider lists of numbers representing Roman numerals such as XLIV (44), IX (9), and CMLXXVI (976); these appear in the table below.

`number-list`	`(prefix-values-removed (cdr number-list) )`	*value* prefix-values-removed *should return*
`(10 50 1 5)`	`(50 4)`	`(40 4)`
`(1 10)`	`(10)`	`(9)`
`(100 1000 50 10 10 5 1)`	`(1000 50 10 10 5 1)`	`(900 50 10 10 5 1)`

Here, the correct answer results from replacing the first element in the list returned from the recursive call by that element minus (car number-list). Replacement of the first value in a list L by a new value is done as follows:

```
(cons new-value (cdr L))
```

Thus the code below constructs the desired list.

```
(cons
   (- (car (prefix-values-removed (cdr number-list)))
      (car number-list) )
   (cdr (prefix-values-removed (cdr number-list))) )
```

It is somewhat clumsy and inefficient to call prefix-values-removed twice here. We'll worry about fixing this once we get a correctly working function completed.

What code results?

The code above is used when the Roman numeral starts with a prefix. Returning to the almost-complete framework for prefix-values-removed produces the following:

```
(define (prefix-values-removed number-list)
    (cond
        ((null? number-list) '( ))
        ((null? (cdr number-list)) number-list)
        ((and (null? (cddr number-list))
              (>= (car number-list) (cadr number-list)) )
            number-list)
        (first-number-isn't-a-prefix
         (cons
            (car number-list)
            (prefix-values-removed (cdr number-list)) ) )
        (first-number-is-a-prefix
            (cons
                (- (car (prefix-values-removed (cdr num-
ber-list)))
                    (car number-list) )
                (cdr (prefix-values-removed (cdr number-
list))) ) ) ) )
```

Deciding whether or not the first number represents a prefix is easy. If it's less than the second number, it's a prefix, otherwise not. The complete program appears in Appendix A.

Exercises

Application 12. Write a function decreasing? that, given a list of numbers as argument, returns #t if the numbers are in decreasing order and #f otherwise. What should decreasing? return for an empty list?

Application 13. Write a function called successive-diffs that, given a list of numbers as argument, returns the list that results from subtracting the second from the first of each pair of consecutive numbers. For example, (successive-diffs '(19 5 10 3)) should return the list (14 –5 7).

Reflection 14. When designing a recursive function, do you design its base cases before its recursive calls or vice versa? Why?

Reflection 15. We chose not to change digit-values and element-sum at all, designing a completely separate function that could be used with them. Would you have done that? Why or why not?

Reflection 16. We noted that two moderately complicated functions are likely to be easier to understand than one simple function and one very complicated function. Do you agree? Why or why not?

Testing the program

How should the program be tested?

Testing should proceed as in the "Difference Between Dates" case study. Where possible, each function should be tested by itself before being tested in combination with the other functions. At worst, a function should only be tested in combination with code already believed to be correct.

Designing test values can be done in two ways. One is called *black-box testing*. This approach views the program as a "black box" that it's not possible to see inside. Black-box test data are devised directly from the problem description, without reference to the code. The other approach is called *glass-box testing*. Glass-box test data are devised by examining the code and noting opportunities for error, for example, off-by-one errors or infinite recursion. Both approaches yield "typical" test values—well within the problem specification—and "extreme" values, near some boundary of the specification or of some structure in the program.

Where possible, the test cases devised using the glass-box approach should be *exhaustive*; that is, they should cause *every line* of the function in question to be tested. That means, for instance, that one should test `decimal-digit-value` with each possible Roman digit. Fortunately there is an easy way to do this. The `digit-values` function calls `decimal-digit-value` for each element of its argument list. An exhaustive test of `decimal-digit-value` will thus result from calling `digit-values` with a list that contains all the possible Roman digits.

How should extreme cases be determined?

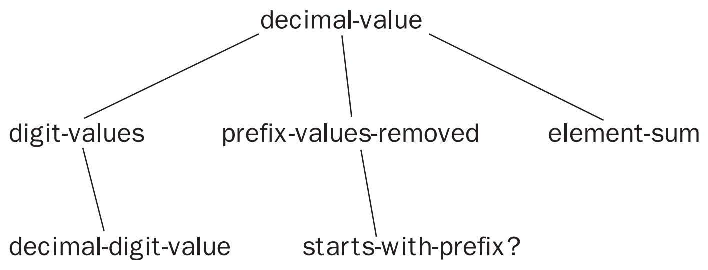
Some candidates for extreme cases are lists of length 0 and 1 and lists as long as possible. The complexity of the Roman numeral system, however, requires more energy in devising test values. Our approach is to look for numeric quantities associated with the problem, and to choose test values that maximize or minimize those quantities. Two such quantities are the number and position of prefixes:

- as many prefixes as possible;
- as few as possible;
- a value that starts with a prefix;
- a value that ends with a prefixed number;
- a value with consecutive pairs of numbers that each contains a prefix.

Stop and predict ➨ *Design test values in the categories just listed.*

In what sequence should the functions in the program be tested?

A good sequence to test the functions in a completely designed program is *bottom-up*. The "bottom" refers to a function's position in a *call tree*, a diagram that shows how the functions call one another. The top of the tree is the main function; it has branches leading to functions that it calls, they have branches for functions they call, and so on. Here's a call tree for the code in Appendix A.



Bottom-up testing would start with the bottom functions and move up, making sure before testing a given function that everything below it in the tree has been tested.

Stop and consider ➨ *Why might bottom-up testing be better than top-down testing?*

We start by testing digit-values and decimal-digit-value as described above. Our input is in boldface.

```
: (digit-values '(m d c l x v i))
(1000 500 100 50 10 5 1)

: (digit-values '(m))
(1000)

: (digit-values '())
()
```

Element-sum is a good candidate for the next thing to test:

```
: (element-sum '())
0

: (element-sum '(1))
1

: (element-sum '(1 3 5 8))
17
```

Next comes the prefix code.

```
: (starts-with-prefix? '(1 10))
#t

: (starts-with-prefix? '(10 1))
#f

: (prefix-values-removed
    '(1000 100 1000 50 10 10 10 5 1 1))
(1000 900 50 10 10 10 5 1 1)

: (prefix-values-removed
    '(1000 100 1000 10 100 1 10))
(1000 900 90 9)

: (prefix-values-removed
    '(100 100 100 10 10 10 1 10))
(100 100 100 10 10 10 9)

: (prefix-values-removed '(10 50 1 5))
(40 4)
```

```
: (prefix-values-removed '(1 10))
(9)

: (prefix-values-removed '(100 1000 50 10 10 5
1))
(900 50 10 10 5 1)
```

Stop and help ➨ *Classify the test values used for* prefix-values-removed *into the categories for extreme cases described above.*

And finally the main function.

```
: (decimal-value '(m c c l x v i i))
1267

: (decimal-value '(m c m l x x x v i i))
1987

: (decimal-value '(m c m x c i x))
1999

: (decimal-value '(c c c x x x i x))
339

: (decimal-value '(x l i v))
44

: (decimal-value '(i x))
9

: (decimal-value '(i))
1

: (decimal-value '(x))
10

: (decimal-value '(c m l x x v i))
976
```

Exercises

Analysis 17. Either verify that all the `cond` clauses in the program have been exercised by the testing described in this section, or find a `cond` clause that was not exercised.

Debugging 18. Suppose your programming partner accidentally changed one of the `cdr`s in `prefix-values-removed` to a `cddr`. Design the smallest collection of test calls necessary to identify which `cdr` was changed.

Reflection 19. Explain how you convince yourself that your code is correct. How much test data do you need?

Reflection 20. How would the steps you would take to convince yourself of the correctness of someone else's code differ from those you would take to convince yourself that your own code is correct?

Application 21. Find out how to define a set of test calls to `decimal-value` in your Scheme environment so that you would not have to type them again if you fixed a bug in the program and wished to retest it.

Reflection 22. If your tests exposed a bug in your program, would you continue to test or try to fix the bug immediately? Briefly explain.

Improving the program

How can the prefix-values-removed **function be improved?**

Do we stop now that we believe that the program works? No. Our next step is to look for ways to improve the code—to make it more understandable or easier to show correct.

A good place to start is in prefix-values-removed, the most complicated function. Back when this function was designed, we noted the possibility that there might be more base cases than necessary. There are base cases for an empty list, a list of length 1, and a list of length 2. The last of these cases may be superfluous, since a recursive call may reduce it to one of the other two base cases.

Which base cases can be removed from prefix-values-removed**?**

Let's see what happens if we omit checking for a list of length 2. A number list that would be intercepted by that test is (10 1). Without the test for a list of length 2, this list instead is handled by the next test since it doesn't start with a prefix, and the expression

```
(cons 10 (prefix-values-removed '(1)))
```

is evaluated. The argument to recursive call contains only one element, so it is handled by the second base case, and the list (1) is returned. Back in the original call, we return the result of (cons 10 '(1)), namely (10 1).

How can the recursive calls be simplified?

That's progress. Less code (generally) means less complexity. Moving on to the recursive calls, we notice that prefix-value-removed is called twice to get the same answer. This can be shortened in two ways: using an auxiliary function to construct the answer that would be called with (prefix-values-removed (cdr number-list)) as an argument; or using a let.

However, a better way to fix this involves noticing that prefixes are currently handled by constructing an incorrect list with the prefixed element, then taking it apart and putting it back together correctly. Perhaps there is a way to construct the list correctly in the first place.

We examine an example: (100 500 10 1 5), the numeric form of CDXIV (414 in decimal). The desired result is (400 10 4), derived from substituting 400 for the 100 and 500 and substituting 4 for the 1 and 5. The 400 is found by subtracting the first number in the list from the second. What's needed is a recursive call that returns (10 4); the expression

```
(cons
   (- (cadr number-list) (car number-list))
   (prefix-values-removed ___ ) )
```

would then return the desired list. The correct argument for the recursion is a list with the prefix-prefixed element pair removed. That's (cddr number-list).

The rewritten prefix-values-removed function is shown below.

```
(define (prefix-values-removed number-list)
    (cond
        ((null? number-list) '( ))
        ((null? (cdr number-list)) number-list)
        ((not (starts-with-prefix? number-list))
         (cons
            (car number-list)
            (prefix-values-removed (cdr number-list)) ) )
        ((starts-with-prefix? number-list)
         (cons
            (- (cadr number-list) (car number-list))
            (prefix-values-removed (cddr number-list)) )
) ) )
```

How can reorganization simplify the program?

One final improvement is to replace prefix-values-removed and element-sum by a function that, instead of building a list and summing its elements in two separate steps, combines those two operations. All that's necessary is to replace calls to cons by calls to +, and to replace the lists returned in the base cases by numbers. We'll call the new function roman-sum; it appears in Appendix B. We test it on the same test data as before, successfully.

Exercises

Modification 23. Modify the code in Appendix B so that it returns 0 if the argument list contains an illegal Roman digit.

Debugging 24. Your programming partner, in an all-night programming session, accidentally deletes one of the cond clauses in roman-sum, with the result that the function only works for Roman numerals that end with a prefixed element such as (I V) or (M M X L). Which clause is missing?

Analysis 25. The empty list isn't really a legal Roman numeral. Suppose we removed the test for an empty list from the prefix-values-removed function in Appendix A and retained the test for a two-element list. Would the modified prefix-values-removed function work correctly? Explain why or why not.

Reflection 26. At what point are you satisfied with a solution to a programming problem? How many "rough drafts" of your code do you typically need?

Analysis 27. Which of the cond clauses in roman-sum are not exercised for calls with arguments that don't contain any prefixes?

Analysis 28. Describe all arguments, even those representing illegal Roman numerals, for which roman-sum returns the value 5.

Outline of design and development questions

Preliminary planning

What's a good first step toward a solution?

What programming patterns may be applicable here?

What kind of translation provides a good model for translating Roman digits?

How is the entire sequence of Roman digits translated?

How are the translation and accumulation steps combined?

How can the process be represented in a diagram?

Extending the partial solution to work for all Roman numerals

Which function must be extended to complete the program?

Which design seems easiest?

How should prefixes be handled?

- What design is suggested by the examples?
- How can recursion be used to combine a prefix with the prefixed value?
- What are the base cases?
- How can the value returned by a recursive call be used?

What code results?

Testing the program

How should the program be tested?

How should extreme cases be determined?

In what sequence should the functions in the program be tested?

Improving the program

How can the `prefix-values-removed` function be improved?

- Which base cases can be removed from `prefix-values-removed`?
- How can the recursive calls be simplified?

How can reorganization simplify the program?

Exercises

Modification 29. Modify each version of the code to handle two more Roman digits: T, meaning 10000, and F, meaning 5000. M can prefix T to represent 9000 or F to represent 4000. No more than three occurrences of T may appear consecutively, and no more than one occurrence of F may appear at all.

Analysis 30. What is the largest value that can be represented using T and F and the other Roman digits? What is the longest Roman numeral that can be written with these Roman digits?

Application 31. Write a function roman-value that, given a positive integer, returns the corresponding Roman numeral represented as a list as described in this case study.

Application 32. Write a function legal-roman? that returns #t if its argument is a list representing a legal Roman numeral.

Application 33. Write a function roman-plus that, given two Roman numerals, returns the Roman numeral that represents their sum.

Modification 34. Rewrite the function digit-values from Appendix A so that in the list it returns, prefixes are translated to negative values. For example, (digit-values '(X L I V)) should return (–10 50 –1 5). Then modify the decimal-value function accordingly to make use of the modified digit-values.

Application, modification 35. Write a function grouped that, given a Roman numeral list, returns the result of translating each prefix and prefixed digit pair into a two-element list. For example, (grouped '(M X C I V)) should return (M (X C) (I V)). Then modify the decimal-value function accordingly to call grouped.

Modification 36. Write a function roman-sum-helper that is called by a modified roman-sum as follows:

```
(define (roman-sum number-list)
   (roman-sum-helper
      (car number-list)
      (cdr number-list) ) )
```

Roman-sum-helper's two arguments are the most recently seen numeric Roman digit value and the list of remaining numeric Roman digit values. Thus the following recursive calls should result from evaluating the expression (roman-sum '(10 50 1 5)):

```
(roman-sum-helper 10 '(50 1 5))
(roman-sum-helper 50 '(1 5))
```

```
(roman-sum-helper 1 '(5))
(roman-sum-helper 5 '( ))
```

Debugging 37. Your programming partner accidentally changes a word of the roman-sum function. Some calls to the modified function return the following results:

call to roman-sum	*result*
`(roman-sum '(10 50 1 5))`	99
`(roman-sum '(1 10))`	19
`(roman-sum '(10 1))`	11
`(roman-sum '(5 1 1 1))`	8

Which word could have been changed?

Appendix A
First version of the program

```
; Return the decimal value of the Roman numeral whose digits are
; contained in roman-digit-list.
; Roman-digit-list is assumed to contain only Roman digits.
; Sample call: (decimal-value '(x i v)), which should return 14.

(define (decimal-value roman-digit-list)
   (element-sum
      (prefix-values-removed
         (digit-values roman-digit-list) ) ) )

; Return a list containing the decimal values of the Roman digits
; in roman-digit-list.
; Roman-digit-list is assumed to contain only Roman digits.
; Sample call: (digit-values '(x i v)), which should return (10 1 5).

(define (digit-values roman-digit-list)
   (if (null? roman-digit-list) '( )
      (cons
         (decimal-digit-value (car roman-digit-list))
         (digit-values (cdr roman-digit-list)) ) ) )

; Return the decimal value of the given Roman digit.

(define (decimal-digit-value roman-digit)
   (cond
      ((equal? roman-digit 'm) 1000)
      ((equal? roman-digit 'd)  500)
      ((equal? roman-digit 'c)  100)
      ((equal? roman-digit 'l)   50)
      ((equal? roman-digit 'x)   10)
      ((equal? roman-digit 'v)    5)
      ((equal? roman-digit 'i)    1) ) )
```

```
; Return the result of removing prefixes from number-list.
; Number-list is assumed to contain only positive numbers.
; A prefix is a number that is less than its successor in the list.
; The prefix and its successor are replaced by the difference between
; the successor value and the prefix.
; Sample call: (prefix-values-removed '(10 1 5)), which should return
; (10 4).

(define (prefix-values-removed number-list)
   (cond
      ((null? number-list) '( ))
      ((null? (cdr number-list)) number-list)
      ((and (null? (cddr number-list)) ; length = 2?
            (>= (car number-list) (cadr number-list)) )
         number-list)
      ((not (starts-with-prefix? number-list))
       (cons
         (car number-list)
         (prefix-values-removed (cdr number-list)) ) )
      ((starts-with-prefix? number-list)
       (cons
         (- (car (prefix-values-removed (cdr number-list)))
            (car number-list) )
         (cdr (prefix-values-removed (cdr number-list))) ) ) ) )

; Return true if the number-list starts with a prefix, i.e. a number
;.that's less than the second value in the list.
; Number-list is assumed to be of length at least 2 and to contain
; only positive numbers.

(define (starts-with-prefix? number-list)
   (< (car number-list) (cadr number-list)) )

; Return the sum of the values in number-list.
; Number-list is assumed to contain only positive numbers.

(define (element-sum number-list)
   (if (null? number-list) 0
      (+ (car number-list)
         (element-sum (cdr number-list)) ) ) )
```

Appendix B
Rewritten functions

```
; Return the decimal value of the Roman numeral whose digits are
; contained in roman-digit-list.
; Roman-digit-list is assumed to contain only Roman digits.
; Sample call: (decimal-value '(x i v)), which should return 14.

(define (decimal-value roman-digit-list)
   (roman-sum
      (digit-values roman-digit-list) ) )

; Return the decimal value of a Roman numeral. The decimal equivalents
; of its Roman digits are contained in number-list.
; Sample call: (roman-sum '(10 1 5)), which should return 14.

(define (roman-sum number-list)
   (cond
      ((null? number-list) 0)
      ((null? (cdr number-list)) (car number-list))
      ((not (starts-with-prefix? number-list))
       (+
         (car number-list)
         (roman-sum (cdr number-list)) ) )
      ((starts-with-prefix? number-list)
       (+
         (- (cadr number-list) (car number-list))
         (roman-sum (cddr number-list)) ) ) ) )
```