“Difference Between Dates” Case Study, Part I

Problem

Write and test a Scheme program to compute how many days are spanned by two given days. The program will include a function called `day-span` that returns the number of days between and including its two inputs. The inputs to `day-span` represent dates in 1994 (a non-leap year); each input is a two-element list whose first element is a month name and whose second element is a legal date in the month. Assume that the first input date is earlier than the second.

In solving this problem, it helps to remember the short poem:

> Thirty days hath September,
> April, June, and November.
> All the rest have thirty-one,
> Excepting February alone,
> Which has four and twenty-four
> Till leap year gives it one day more.
> (Anonymous)

The table below lists some sample calls to `day-span` and the desired return values.

expression	*desired return value*
`(day-span '(january 3) '(january 9))`	7
`(day-span '(january 30) '(february 2))`	4
`(day-span '(june 7) '(august 25))`	80
`(day-span '(january 1) '(december 31))`	365

Preparation

The reader should have experience with defining his or her own Scheme functions and constants, with the use of conditional expressions in Scheme (`if` and `cond`), and with the use of lists and functions for constructing and accessing parts of them.

Exercises

Analysis 1. Determine the number of days between January 20 and February 5.

Analysis 2. Determine the number of days between January 20 and December 5.

Reflection 3. What was different, if anything, about the way you determined the answer to exercise 1 as compared to exercise 2? If you used two different approaches to find the two answers, explain the circumstances under which you would use each approach.

Analysis 4. Give two days that span an interval of 20 days.

Analysis 5. Give two days that span an interval of 200 days.

Reflection 6. What was different, if anything, about the way you determined the answer to exercise 4 as compared to exercise 5? If you used two different approaches to find the two answers, explain the circumstances under which you would use each approach.

A design based on a procedure for solving the problem by hand

How do programmers design problem solutions?

Figuring out how to solve a problem by hand usually provides ideas for the design of Scheme functions to solve the problem.

How can the difference between two dates be computed by hand?

Working through the examples in the problem statement by hand suggests three different situations: dates in the same month, dates in consecutive months, and dates in non-consecutive months.

Dates in the same month

> To compute the interval between two dates in the same month, we merely subtract the date in the first month from the date in the second month and add 1. We add 1 because both the days are to be included in the returned value. Thus the interval from January 3 through January 9 spans
> (9–3)+1 = 7 days.

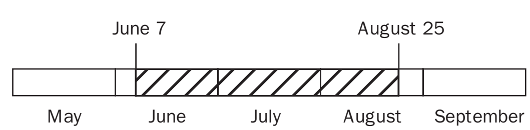
Dates in consecutive months

> To compute the interval between two dates in consecutive months, like January 30 and February 2, we determine the number of days remaining in the first month and add that to the date in the second month. Thus the interval from January 30 to February 2 spans January 30 and 31 (two days) plus February 1 and 2 (two more days).

Dates in non-consecutive months

> To compute the interval between dates in non-consecutive months, we can add three quantities: the number of days in all the months between the two given months, the days remaining in the first month, and the date in the second month. The interval from June 7 and August 25 includes 31 days for all of July, 24 days from June 7 to June 30, and 25 days in August, a total of 80 days.

It is often helpful to draw a diagram that represents a computation, since a picture can be worth a thousand words. For this problem, one might draw months as adjacent rectangles and indicate a given date by a line through the corresponding rectangle. Then the interval between the lines for two given dates can be shaded to indicate the dates spanned by the two dates. For instance, a diagram representing the interval spanned by June 7 and August 25 would be the following:



Stop and help (

Draw diagrams that represent the intervals between June 7 and June 25, between June 7 and July 25, and between January 30 and December 5.

How is the algorithm represented in pseudocode?

A commonly-used technique for designing a computer program is first to outline a solution at a high level, then to *decompose* it, that is, rewrite the solution step by step, adding more detail at each step. It is useful to give descriptive names to data as soon as possible, so we do this in our first step, calling the dates earlier-date and later-date.

```
return the number of days
    between earlier-date and later-date.
```

Our initial description is in semi-English, semi-Scheme. This intermediate representation between a programming language and a natural language is called *pseudocode*. We then add a level of decomposition, again in pseudocode, that reflects the three situations mentioned above:

```
if the months for the two dates are the same,
    then …
otherwise if the months for the two dates are con-
    secutive, then …
otherwise …
```

Each "…" in the pseudocode represents the computation for the corresponding situation. We choose not to write the details down yet to avoid getting bogged down in detail.

How are the three situations represented in Scheme?

The pseudocode solution combining these three situations can be represented as a cond in Scheme. The tests to see what situation exists and the computations for the three situations can be represented as functions. Choosing good names for the functions helps to manage details, and produces Scheme code that is almost as understandable as the pseudocode algorithm:

```
; Return the number of days spanned by earlier-date and
; later-date. Earlier-date and later-date both represent
; dates in 1994, with earlier-date being the earlier of
; the two.

(define (day-span earlier-date later-date)
    (cond
        ((same-month? earlier-date later-date)
         (same-month-span earlier-date later-date) )
        ((consecutive-months? earlier-date later-date)
         (consec-months-span earlier-date later-date) )
        (else
         (general-day-span earlier-date later-date) ) ) )
```

The code includes a comment describing the purpose of the function and the assumptions made about the inputs. Same-month? and consecutive-months? are *predicate* functions, as indicated by the question mark at the end of their names; they return true or false, and thus can be used as the first half of a cond's condition-value pair. The three …-span functions will return the number of days between the two dates in the more restricted situations just described.

Stop and help ⟨ *Write the comments for each of the five functions* same-month?, consecutive-months?, same-month-span, consec-months-span, *and* general-day-span.

Here we have used the five functions before even thinking about *how* to write them. We have merely specified *what* they do, which is really all that's needed in order to use the functions. Decomposition typically involves repeating the following steps:

a. determining a value that is to be computed,

b. naming a function that computes it and deciding on its arguments,

c. using the function, and finally

d. designing the body of the function.

What is left to do?

We started by thinking about the problem as being made of various cases. This allowed us to begin by providing an outline of the problem. We decomposed that high-level outline using pseudocode, until we were able to write a Scheme function. This process allowed us to make headway on the problem without getting caught up in the details of each piece. We will reapply some of these techniques as we continue to work on the problem.

The new problem is to design the five functions. Ideally, each new problem is easier to solve than the original problem. We call this "dividing and conquering".

Stop and consider ⟨ *Has breaking one task into five subtasks complicated the solution or simplified it?*

Stop and predict ⟨ *Think about the functions yet to be designed. Which of them seems hardest to design? Which seems easiest? How are they similar? Does the solution you worked out by hand help you in figuring out how they might be coded?*

How is same-month? coded in Scheme?

We start with same-month?, since it will probably be easiest to code. Recall from the problem statement that a date is a two-element list. The first element is the month name and the second element is a date number within the month. Thus two dates are in the same month if they have the same first element.

Stop and predict ⟨ *Write the* same-month? *function.*

Why use specially-defined access functions to access components of a date?

One might devise a solution that accesses the months by using function car directly. The code will be clearer, however, if *accessing* functions are used to refer to the two components of a date. ("Car" is not the most descriptive name for a function.) Accessing functions *name* the corresponding pieces of the list, making it easier to understand what is being accessed in a given section of code. Thus we will provide a function called month-name to retrieve the month name from the date, and another called date-in-month to retrieve the date in the month.

Accessing functions also localize the details of how a particular piece of data—in this case, a date—is represented. This makes it easier to reuse code, by making the code that calls the accessing function less dependent on the Scheme representation of the data. For example, if some future assignment requires the use of dates in a slightly different format, the only functions that will have to change will be the month-name and date-in-month functions; everything else should work unchanged.

The same-month? function is then coded as follows:

```
; Return true if date1 and date2 are dates in the
; same month, and false otherwise.
; Date1 and date2 both represent dates in 1994.

(define (same-month? date1 date2)
   (equal? (month-name date1) (month-name date2)) )
```

The dates don't need to be in a particular relationship for same-month? to work, so we use date1 and date2 as names for its arguments rather than earlier-date and later-date.

How is consecutive-months? coded?

Next we work on consecutive-months?. One way to code this is as an eleven-way test:

```
; Return true if date1 is in the month that
; immediately precedes the month date2 is in,
; and false otherwise.

(define (consecutive-months? date1 date2)
   (or
      (and (equal? (month-name date1) 'january)
         (equal? (month-name date2) 'february))
      (and (equal? (month-name date1) 'february)
         (equal? (month-name date2) 'march)
         ... ) ) )
```

Stop and help ⁅ *How many lines of code does this function have?*

Stop and consider ⁅ *What do you think of this function? Explain.*

What's wrong with this code?

This code is much too complicated—too "brute force". What's needed instead is something hat handles months in a general way rather than check each case separately. The function merely should check if the second date is in the month immediately after the first; this shouldn't require much work.

How can consecutive-months? be coded more simply?

Another way to say that months are consecutive is to say that the second month is "one month after" the first. With month numbers instead of month names, we could merely check that the second month number is equal to the first plus 1. This suggests a simplifying intermediate step: translating a month name to a month number.

Stop and predict (*Why is this simpler?*

A month-number function that returned the number corresponding to a month name—1 for January, 2 for February, and so on—could be used in a much shorter consecutive-months? as follows:

```
; Return true if date1 is in the month that
; immediately precedes the month date2 is in,
; and false otherwise.

(define (consecutive-months? date1 date2)
    (=
        (month-number (month-name date2))
        (+ 1 (month-number (month-name date1))) ) )
```

The month-number function can be coded as a twelve-clause cond:

```
(define (month-number month)
    (cond
        ((equal? month 'january) 1)
        ((equal? month 'february) 2)
            … ) )
```

An additional advantage of this design is that the month-number function is more likely to be useful in other applications involving dates.

How can this code be tested?

The code to check for consecutive months is relatively complicated compared to same-month?. It makes sense to type it in and test it before going on, both to make sure we haven't made any design errors and to give ourselves a bit of encouragement for having completed part of the problem. (We should probably test same-month? at the same time.) This piecewise testing of code is called *incremental development*.

Testing consecutive-months? requires testing month-number as well: we need to verify month-number returns the correct value for each month, and we must test that consecutive-months? uses the result correctly. Errors to test for include *logic errors* that result from confusion on the programmer's part, as well as simple typing errors.

The only way to ensure that no errors appear in the large cond appearing in month-number is to provide arguments for which each condition succeed at least once. Thus at least twelve calls to month-number are necessary.

Once month-number has been tested, we turn to consecutive-months? and same-month?. Fewer tests are necessary for these functions; for instance, we get as much information from the call

```
(consecutive-months? '(march 1) '(april 13))
```

as from the call

```
(consecutive-months? '(april 1) '(may 13))
```

Stop and help ⟮ *Why does one of the two calls just given provide no more information about the correctness of* consecutive-months? *than the other call?*

We will need to test negative as well as positive results, that is, with dates that aren't in the same or consecutive months as well as those that are. Most people overlook such tests (preferring to think positively?). Experienced programmers also test functions with *extreme values* as well as *typical values*, since programmers often make mistakes handling the boundaries of a set of data. The definition of "extreme value" depends on the situation. Two obvious extreme values for months are January and December, so we should be sure to test same-month? and consecutive-months? on dates in those months.

Stop and help ⟮ *Produce a comprehensive set of function calls to test the code just written, and explain the evidence that each call provides for the purpose of convincing someone that the code works correctly.*

How is same-month-span **coded?**

We continue the design, going on to same-month-span. The pseudocode for this was given at the beginning of the narrative. Here's the code:

```
(define (same-month-span earlier-date later-date)
   (+ 1
      (- (date-in-month date2)
         (date-in-month date1) ) ) )
```

Stop and help ⟮ *Design test cases for the function* same-month-span. *Categorize them as extreme or typical.*

How is consec-months-span **coded?**

Next comes consec-months-span. This requires more decomposition. What's needed, according to the procedure for computing the result by hand, is a way to find how many days remain in the first month. This is the number of days in the month, minus the date of the month, plus 1.

A function that returns the number of days in a given month can use a twelve-way test, similar to the code in month-number. We'll call this function days-in-month. *Recycling* the month-number code in this way saves time. In general, it is also good to design functions whose purpose is similar to have similar code as well; the correctness of one of the functions provides some evidence of the correctness of the other.

Stop and help ⟮ *Write* days-in-month.

Days-in-month is used in days-remaining as follows:

```
; Return the number of days remaining in the month
; of the given date, including the current day.

(define (days-remaining date)
   (+ 1 (- (days-in-month (month-name date))
           (date-in-month date) )) )
```

Following the previously designed pseudocode, we code consec-months-span as follows:

```
; Return the difference in days between earlier-date
; and later-date, which represent dates in consecutive
; months of 1994.

(define (consec-months-span earlier-date later-date)
    (+ (days-remaining earlier-date)
        (date-in-month later-date) ) )
```

How can consec-months-span **be tested?**

Again we apply the technique of incremental development, and stop to test the code just designed. The arithmetic done in consec-months-span and days-remaining provides the opportunity for *off-by-one* errors, where the answer differs from the correct result by 1. All the functions should be tested individually; this more easily exposes bugs.

Extreme cases for dates in consecutive months would be days as close together as possible, i.e. the last day in one month and the first day in the next, and dates as far apart as possible.

Stop and help (

Devise a set of test data for the functions just designed, and use it to test the functions.

We have designed almost all of the program. Is there any way we can test everything we've written so far, including the main function day-span? One way to do this is to write a *stub function* for the final remaining function, general-day-span, that is, a function that does only a small part of its intended task. Coding of stub functions is another technique for incremental development. At this point, we could just have the function return the atom not-ready-yet.

Stop and help (

Write a stub general-day-span *function. Then test the entire* day-span *function, making sure that all the conditions in the* cond *statement are tested.*

How is general-day-span **coded?**

On to general-day-span. Figuring this by hand, we determined the number of days in all the months between the two given months, added that to the days remaining in the first month, and added that sum to the date in the second month. We have already figured out everything except how to determine the number of days in all the months between the two given months.

To compute the number of days in all the months between two given months, we think about new techniques we have learned. But nothing seems to provide any better solution than a very long and complicated cond to handle all pairs of non-consecutive months. It would take a long time merely to type, much less test sufficiently to convince ourselves of its correctness.

We have reached an impasse. We need either a better algorithm or some more powerful Scheme tools.

What went wrong, and what can be done about it?

This is a good time to step back and look at our overall approach to the problem.

In retrospect, we see that the impasse arose because computing the number of days between non-consecutive months is not significantly simpler than the original problem. Functions learned so far in Scheme are not powerful enough to allow implementation of the procedure for computing the answer by hand. (Functions introduced later in the course will provide other ways to do this.)

Programmers occasionally need to back up and try to find better problem solutions. Usually, however, some of the code they generate before reaching an impasse can be recycled in the new solution. Our partially designed program is listed in Appendix A for reference.

Exercises

Reflection

7. The `same-month?` function was designed first because it seemed easiest. When you are confronted with a choice among tasks, do you choose to attack the easiest or the hardest task first? Explain.

Reflection

8. In what ways have you used a diagram to help you understand a concept? How did the diagram help?

Reflection

9. What do you do when you reach an impasse when trying to solve a problem?

Reflection

10. Generally, one should use a noun to name a function. Why?

Modification, reflection

11. Rewrite the functions designed so far without using accessing functions. Then explain why the rewritten code might be more difficult to understand than the code designed in the narrative.

Debugging

12. Change a single symbol in the code in Appendix A, and get a fellow programmer to test the code and figure out what you changed.

Reflection

13. What aspects of the code make this task difficult for your programmer friend?

A better solution, based on transforming the dates to a simpler form

What's a better way to view the computation of the difference between dates?

Almost all the functions in the first attempt at a solution worked with dates in their original format. One way to generate an alternative solution is to convert inputs to a different format before working with them, in the same way we did in designing the consecutive-months? function. Here we seek to convert the dates to something easier to manipulate, like integers, and then write functions to work with dates in the new format. (Recall that we did the same thing to design the consecutive-months? function.)

Stop and predict *List at least two ways to represent a date as an integer.*

A function that transformed a month-day pair into an integer could then be used as follows:

```
; Return the number of days spanned by earlier-date
; and later-date. Earlier-date and later-date both
; represent dates in 1994, with earlier-date being
; the earlier of the two.

(define (day-span earlier-date later-date)
   (+ 1
      (- (transformed later-date)
         (transformed earlier-date) ) ) )
```

How can a date be represented as an integer?

The date-in-month is already an integer. Perhaps if we represent the month name as an integer also, we can put the two together using some arithmetic to produce an integer representing the entire date.

One integer representation for the month is its month number (1, 2, …, 12). Unfortunately, since there is no pattern to the order of the months and their lengths, this does us little good. Another possibility is to convert the month name to a number representing how many days are in the month. For example, January would become 31, February would become 28, and so on. This transformation, however, is worse, because we lose critical information: How could we tell the transformed date of (january 2) from the transformed date of (december 2)?

Looking back at the problem statement, we find information that will help us in our conversion. The problem statement says that both dates are in the year 1994. We can use the fact that every date is some number of days from January 1, 1994 to solve the problem by converting each date into an integer day-in-1994! Thus January 1 would be represented as 1, January 31 as 31, February 1 as 32, December 31 as 365, and so forth.

To design the conversion function, we again work out a procedure by hand. The day of the year for a date, i.e. the number of days from January 1 to the date, is the "date-in-the-month" of the given date, plus the number of days in all the preceding months. The latter quantity is best computed using a table:

month	*days in month*	*total days in preceding months*
January	31	0
February	28	0+31=31
March	31	31+28=59
April	30	59+31=90
May	31	90+30=120
June	30	120+31=151
July	31	151+30=181
August	31	181+31=212
September	30	212+31=243
October	31	243+30=273
November	30	273+31=304
December	31	304+30=334

Here is another opportunity to recycle code designed for the first solution, namely the month-number and days-in-month functions. Here, we want a function that, given the name of a month, returns the total number of days in all preceding months. Transforming the old functions into the desired function is done merely by substituting values in the third column of the table above for values in the second column.

Stop and help (*Write a* days-preceding *function that, given a month as input, returns the number of days from January 1 to the first day of that month. Test it to verify that it works correctly.*

How is days-preceding used?

We now use days-preceding as just described to code a function day-of-year:

```
; Return the number of days from January 1
; to the given date, inclusive.
; Date represents a date in 1994.

(define (day-of-year date)
   (+ (days-preceding (month-name date))
      (date-in-month date)))
```

Day-span has already been designed. We merely substitute day-of-year for the "transformed" function. The result is a complete program that computes the days between two dates; the code appears in Appendix B.

How may the program best be tested and debugged?

Test data designed for the previous version will be useful here as well. The functions should be tested individually, using extreme cases as well as typical cases. Extreme cases here are consecutive dates (in the same month or in different months), dates as far apart as possible, dates in January or December, and dates at the beginning and the end of a month.

Exercises

Debugging 14. Change a single symbol in the code in Appendix B, and get a fellow programmer to test the code and figure out what you changed.

Reflection 15. What aspects of the code make this task more difficult than the task of inserting a bug in the Appendix A code?

Modification 16. Suppose day-span had already been coded so as not to use the days-preceding function. Provide a version of days-preceding that produces its result by doing some arithmetic on results from day-span and days-in-month.

Application 17. Draw a diagram that represents the day-span computation described in this section of the narrative.

Application 18. Fill in the blanks in the function below, designed to help *check* if days-preceding is returning the correct value.

```
(define (dp-correct? month)
    (if (equal? month 'january)
        (= 0 (days-preceding month))
        (=
            (days-in-month month)
            (- (days-preceding _____ )
                (days-preceding _____ ) ) ) ) ))
```

Analysis 19. If the inputs to day-span are legal dates, what are the possible values for days-preceding's input? Explain briefly.

Outline of design and development questions

A design based on a procedure for solving the problem by hand

How do programmers design problem solutions?

How can the difference between two dates be computed by hand?

- How is the algorithm represented in pseudocode?
- How are the three situations represented in Scheme?
- What is left to do?

How is `same-month?` coded in Scheme?

- Why use specially-defined access functions to access components of a date?

How is `consecutive-months?` coded?

- What's wrong with this code?
- How can `consecutive-months?` be coded more simply?
- How can this code be tested?

How is `same-month-span` coded?

How is `consec-months-span` coded?

How can `consec-months-span` be tested?

How is `general-day-span` coded?

- What went wrong, and what can be done about it?

A better solution, based on transforming the dates to a simpler form

What's a better way to view the computation of the difference between dates?

How can a date be represented as an integer?

How is `days-preceding` used?

How may the program best be tested and debugged?

Exercises

Analysis 20. Provide calls to each version of `day-span` that return the value 100, or explain why one of the versions cannot return such a value.

Analysis 21. What is the largest value that each version of `day-span` can return?

Analysis 22. Provide a call to either version of `day-span` that returns the value 0. Hint: the arguments will not satisfy the claim made in the comment.

Analysis 23. List all ways to provide *illegal* arguments to `day-span`.

Analysis 24. For which of the types of arguments you described in the previous exercise does each version of `day-span` crash?

Modification 25. The current `day-span` works for dates given in standard U.S. format, where the month name is followed by the date-in-month. Rewrite the program so that it works with dates given in standard European format, where the date-in-month precedes the month name.

Reflection 26. What programming technique employed in this case study simplified the modification of the previous exercise?

Reflection 27. What bugs do you think that a programmer is most likely to encounter in the final `day-span` program?

Application 28. The twelve months of the Islamic year have alternately 30 and 29 days; thus a non-leap year has 354 days. Write a Scheme function that's given a two-element list representing a month and a day on the Islamic calendar and returns the day of the year, an integer between 1 and 354.

Reflection, analysis 29. What makes this problem easier or harder than computing the day of year using our calendar?

Application 30. Write a function `valid-date?` that returns true when given as argument a legal date in 1994, and returns false otherwise.

Application 31. Write a function `precedes?` that, given two legal dates as arguments, returns true when the first date precedes the second and returns false when the two dates are the same or the second date precedes the first.

Modification 32. Extend the `day-span` function to take arguments whose third elements specify a year in the 20th century.

Application 33. Write two solutions for the problem of computing the difference between two *distance measurements*. A distance measurement is a two-element list whose first element is an integer number of feet and whose second element is an integer number of inches. Thus the list `(5 2)` represents the distance 5' 2" (5 feet, 2 inches). The difference between two distances will also be represented by a distance measurement; thus the difference between `(5 2)` and `(6 8)` is `(1 6)`, and the difference between `(5 8)` and `(6 2)` is `(0 6)`.

Model one of your solutions on the program in Appendix A and the other on the program in Appendix B.

Analysis 34. Explain why the approach that led to the program in Appendix A works with distance measurements.

Modification 35. A better way to code the `month-number` function is to represent the months as *data*—a list—rather than in the *program* code of a `cond`. In particular, the result of the `member` function, used with the list `(january february ... november december)`, can be used to determine a month number without any `cond` or `if` at all. Recode the `month-number` function to use `member` in this way.

Reflection 36. Compare the techniques you use to organize an essay to those we used to design a program. What techniques seem useful for both activities? What techniques seem appropriate only for organizing an essay, or only for programming? Explain.

Appendix A
Partially designed attempt to compute the difference between dates

```
; Return the number of days spanned by earlier-date and later-date.
; Earlier-date and later-date both represent dates in 1994,
; with earlier-date being the earlier of the two.
; Note: general-day-span is not implemented.

(define (day-span earlier-date later-date)
   (cond
      ((same-month? earlier-date later-date)
       (same-month-span earlier-date later-date) )
      ((consecutive-months? earlier-date later-date)
       (consec-months-span earlier-date later-date) )
      (else
       (general-day-span earlier-date later-date) ) ) )

; Access functions for the components of a date.

(define (month-name date) (car date))
(define (date-in-month date) (cadr date))

; Return true if date1 and date2 are dates in the same month, and
; false otherwise. Date1 and date2 both represent dates in 1994.

(define (same-month? date1 date2)
   (equal? (month-name date1) (month-name date2)) )

; Return the number of the month with the given name.

(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) ) )

; Return true if date1 is in the month that immediately precedes
; the month date2 is in, and false otherwise.
; Date1 and date2 both represent dates in 1994.

(define (co1nsecutive-months? date1 date2)
   (=
      (month-number (month-name date2))
      (+ 1 (month-number (month-name date1))) ) )
```

```
; Return the difference in days between earlier-date and
; later-date, which both represent dates in the same month of 1994.

(define (same-month-span earlier-date later-date)
   (+ 1
      (- (date-in-month later-date)
      (date-in-month earlier-date) ) ) )

; Return the number of days in the month named month.

(define (days-in-month month)
   (cond
      ((equal? month 'january) 31)
      ((equal? month 'february) 28)
      ((equal? month 'march) 31)
      ((equal? month 'april) 30)
      ((equal? month 'may) 31)
      ((equal? month 'june) 30)
      ((equal? month 'july) 31)
      ((equal? month 'august) 31)
      ((equal? month 'september) 30)
      ((equal? month 'october) 31)
      ((equal? month 'november) 30)
      ((equal? month 'december) 31) ) )

; Return the number of days remaining in the month of the given
; date, including the current day. Date represents a date in 1994.

(define (days-remaining date)
   (+ 1
      (- (days-in-month (month-name date))
      (date-in-month date) ) ) )

; Return the difference in days between earlier-date and
; later-date, which represent dates in consecutive months of 1994.

(define (consec-months-span earlier-date later-date)
   (+ (days-remaining earlier-date) (date-in-month later-date)) )
```

Appendix B
The complete program to compute the difference between two dates

```
; Access functions for the components of a date.

(define (month-name date) (car date))
(define (date-in-month date) (cadr date))

; Return the number of days from January 1 to the first day
; of the month named month.

(define (days-preceding month)
   (cond
      ((equal? month 'january) 0)
      ((equal? month 'february) 31)
      ((equal? month 'march) 59)
      ((equal? month 'april) 90)
      ((equal? month 'may) 120)
      ((equal? month 'june) 151)
      ((equal? month 'july) 181)
      ((equal? month 'august) 212)
      ((equal? month 'september) 243)
      ((equal? month 'october) 273)
      ((equal? month 'november) 304)
      ((equal? month 'december) 334) ) )

; Return the number of days from January 1 to the given date,
; inclusive. Date represents a date in 1994.

(define (day-of-year date)
   (+ (days-preceding (month-name date)) (date-in-month date)) )

; Return the number of days spanned by earlier-date and later-date.
; Earlier-date and later-date both represent dates in 1994,
; with earlier-date being the earlier of the two.

(define (day-span earlier-date later-date)
   (+ 1 (- (day-of-year later-date) (day-of-year earlier-date))) )
```

“Difference Between Dates” Case Study, Part II
****

Preparation

The reader should have been introduced to recursion.

A recursive solution

How can recursion be used to implement the day-span function?

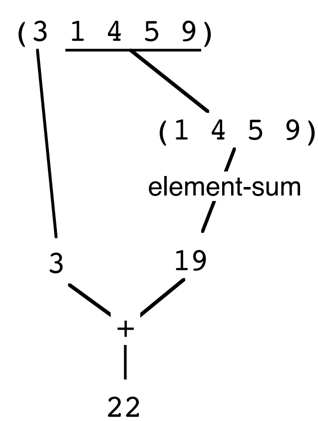
An alternative solution to the day-span function may be designed using recursion, a function's call of itself. Recall the dead end encountered in the original design: given the start and end of a month range, we were unable to determine how many days the months in that range contain.

How can recursion be used to add values in a list?

One defines a recursive computation in terms of a simpler yet similar computation. This allows the computation to be repeated and its results accumulated. For example, recursion for list processing typically involves regarding a list as the combination of its first element and the list of remaining elements; thus a function that returns the sum of all the numbers in a list would be coded as

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

A diagram that represents the computation of the sum of all the elements in the list (3 1 4 5 9) is



Any recursion involves a *base case*, the situation(s)—there may be more than one—that represents the simplest possible computation. A base case for a recursion involving a list is typically the empty list, as in the element-sum function.

How can recursion be used without a list?

Values to be added don't necessarily have to be already collected in a list. One may use recursion to find the sum of all integers in a given range, say from 22 to 500. The approach here is to break the range down into a value—we'll call it v—combined with a smaller range, find the sum of all the integers in the smaller range, then add it to v. The simplest possible range of integers to add is the empty one, and that provides the base case. Here are two functions that find the sum of integers in a range whose endpoints are called first and last.

```
(define (range-sum1 first last)
   (if (> first last) 0
      (+ first (range-sum1 (+ first 1) last)) ) )

(define (range-sum2 first last)
   (if (> first last) 0
      (+ last (range-sum2 first (- last 1))) ) )
```

Stop and consider (*Which of the two* range-sum *functions do you prefer? Why?*

The two functions break down the range of integers in different ways. In range-sum1, the range of integers is broken down into its first element and the range of remaining integers; in range-sum2, the range is broken down into the range of integers from the first up to but not including the last, and the last integer itself. For example, the range of integers between 22 and 500 would be broken down in one of two ways:

22 and 500 is …

> 22 combined with the range of integers between 23 and 500, inclusive; or
>
> the range of integers between 22 and 499, inclusive, combined with 500.

How can range-sum be tailored to compute the number of days in a range of months?

A range of months isn't much different from a range of integers. Instead of the integers in the range, the month lengths for months in the range are to be added. We modify one of the range-sum functions as follows:

```
(define (day-sum first-month last-month)
   (if (> first-month last-month) 0
      (+ cays-in-first-month
         (day-sum (+ first-month 1) last-month)) ) )
```

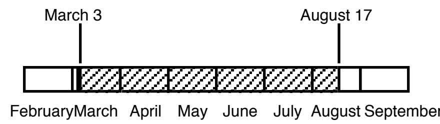
What extra code is needed?

An expression computing the number of days in first-month must still be designed. Reviewing the code in part I of the case study, we notice the days-in-month function; it seems to provide exactly what's needed, except that its argument is a month name rather than the integer representing that month. What's needed is the inverse of the month-number function that, given a month name, returns its numeric index—but such a function will be easy to provide. We'll call it name-of. Substituting the appropriate calls to these functions gives

```
(define (day-sum first-month last-month)
   (if (> first-month last-month) 0
      (+ (days-in-month (name-of first-month))
         (day-sum (+ first-month 1) last-month)) ) )
```

Stop and help (*Write the* name-of *function.*

The last step is to determine how to use day-sum. What was missing from the dead-end code was the general-day-span function, which was to handle the case where the two argument dates were neither in the same month nor in consecutive months, for example, March 3 and August 17. In part 1 of this case study, we represented this in diagram form:



The day-sum function just designed allows the computation of the number of the days of the complete months shaded in the diagram. The days in the partial months at the ends of the shaded region can be counted as in consec-months-span. Required in addition is code to find the first complete month and the last complete month in the range; we'll call these functions next-month-number and prev-month-number.

Stop and help (*Write the* next-month-number *function.*

Stop and consider (*Why should the function be named* next-month-number *instead of* next-month *or* next-month-name*?*

The following function results. The complete program appears in Appendix C.

```
(define (general-day-span earlier-date later-date)
    (+
        (days-remaining earlier-date)
        (day-sum
            (next-month-number earlier-date)
            (prev-month-number later-date) )
        (date-in-month later-date) ) )
```

Outline of design and development questions

A recursive solution

How can recursion be used to implement the day-span function?

- How can recursion be used to add values in a list?
- How can recursion be used without a list?
- How can range-sum be tailored to compute the number of days in a range of months?
- What extra code is needed?

Exercises

Application 1. Write a function called total-days that, given a list of month names, returns the total number of days in those months.

Application 2. The following function is intended to compute the difference between its argument dates recursively, reducing the day span between the argument dates by exactly one day at each recursive call. Supply the code it's missing, along with any auxiliary functions that are necessary.

```
(define (day-span earlier-date later-date)
   (if (equal? earlier-date later-date) 1
      (+ 1 (day-span _____ _____ ) ) ) )
```

Analysis 3. Are the first two cases in day-span still necessary? That is, can day-span now be coded merely as a call to general-day-span? Explain why or why not.

Analysis 4. What happens if the dates provided to general-day-span are out of order, that is, earlier-date is later than later-date?

Debugging 5. Your programming partner, in a late-night coding session, changes a line in the Scheme code, with the result that day-span's return values are now too large:

call to day-span	*returned value*
`(day-span '(january 1) '(december 31))`	*396*
`(day-span '(february 1) '(december 31))`	*362*
`(day-span '(january 1) '(november 30))`	*365*

What line could your partner have changed to produce this behavior?

Application 6. Write a function average that, given a list of numbers, returns the average of the values in the list.

Application 7. Write a function standard-deviation that, given a list of numbers, returns their standard deviation. If the numbers in the list are x_1, x_2, …, x_n and m is their average, then the standard deviation of the numbers is

$$\sqrt{\frac{(x_1-m)^2+(x_2-m)^2+\ldots+(x_n-m)^2}{n}}$$

Analysis 8. Consider a function element-product that, given a list of numbers, returns their product. Thus

```
(element-product '(2 3 5))
```

should return 30. What would be the base case for element-product, and what value should be returned in this case?

Application 9. Code the days-preceding function as a single call to day-sum.

Reflection 10. What's your "mental image" of recursion? That is, when you picture a recursive computation in your head, what does it look like?

Modification 11. Modify one of the range-sum functions so that it returns the sum only of the integers in the given range that satisfy the predicate good?.

Appendix C
Recursive computation of the difference between dates

```
; Return the number of days spanned by earlier-date and later-date.
; Earlier-date and later-date both represent dates in 1994,
; with earlier-date being the earlier of the two.

(define (day-span earlier-date later-date)
   (cond
      ((same-month? earlier-date later-date)
       (same-month-span earlier-date later-date) )
      ((consecutive-months? earlier-date later-date)
       (consec-months-span earlier-date later-date) )
      (else
       (general-day-span earlier-date later-date) ) ) )

; Access functions for the components of a date.
(define (month-name date) (car date))
(define (date-in-month date) (cadr date))

; Return true if date1 and date2 are dates in the same month, and
; false otherwise. Date1 and date2 both represent dates in 1994.

(define (same-month? date1 date2)
   (equal? (month-name date1) (month-name date2)) )

; Return the number of the month with the given name.

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

; Return true if date1 is in the month that immediately precedes
; the month date2 is in, and false otherwise.
; Date1 and date2 both represent dates in 1994.

(define (consecutive-months? date1 date2)
   (=
      (month-number (month-name date2))
      (+ 1 (month-number (month-name date1))) ) )
```

```
; Return the difference in days between earlier-date and
; later-date, which both represent dates in the same month of 1994.

(define (same-month-span earlier-date later-date)
   (+ 1
      (- (date-in-month later-date)
         (date-in-month earlier-date) ) ) )

; Return the number of days in the month named month-name.

(define (days-in-month month-name)
   (cond
      ((equal? month-name 'january) 31)
      ((equal? month-name 'february) 28)
      ((equal? month-name 'march) 31)
      ((equal? month-name 'april) 30)
      ((equal? month-name 'may) 31)
      ((equal? month-name 'june) 30)
      ((equal? month-name 'july) 31)
      ((equal? month-name 'august) 31)
      ((equal? month-name 'september) 30)
      ((equal? month-name 'october) 31)
      ((equal? month-name 'november) 30)
      ((equal? month-name 'december) 31) ) )

; Return the number of days remaining in the month of the given
; date, including the current day. Date represents a date in 1994.

(define (days-remaining date)
   (+ 1
      (- (days-in-month (month-name date))
         (date-in-month date) ) ) )

; Return the difference in days between earlier-date and
; later-date, which represent dates in consecutive months of 1994.

(define (consec-months-span earlier-date later-date)
   (+ (days-remaining earlier-date) (date-in-month later-date)) )

; Return the name of the month with the given number.
; 1 means January, 2 means February, and so on.

(define (name-of month-number)
   (list-ref
      '(january february march april may june
        july august september october november december)
      (- month-number 1) ) )

; Return the sum of days in the months represented by the range
;    first-month ... last-month.
; first-month and last-month are integers; 1 represents January,
; 2 February, and so on.

(define (day-sum first-month last-month)
   (if (> first-month last-month) 0
      (+ (days-in-month (name-of first-month))
         (day-sum (+ first-month 1) last-month) ) ) )
```

```
; Return the number of the month that immediately precedes the
; month of the given date.  1 represents January, 2 February, and
; so on.

(define (prev-month-number date)
   (- (month-number (month-name date)) 1) )

; Return the number of the month that immediately follows the month
; of the given date.  1 represents January, 2 February, and so on.

(define (next-month-number date)
   (+ (month-number (month-name date)) 1) )

; Return the difference in days between earlier-date and
; later-date, which represent dates neither in the same month
; nor in consecutive months.

(define (general-day-span earlier-date later-date)
   (+
      (days-remaining earlier-date)
      (day-sum
         (next-month-number earlier-date)
         (prev-month-number later-date) )
      (date-in-month later-date) ) )
```