"Difference Between Dates" Case Study

Problem

Write and test a Scheme program to compute how many days are spanned by two given days. The program will include a procedure called day-span that returns the number of days between and including its two arguments. The arguments to day-span represent dates in 2002 (a non-leap year); each argument is a two-word sentence whose first word is a month name and whose second word is a legal date in the month. Assume that the first argument 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

Exercises

1. (Analysis) Determine the number of days between January 20 and February 5.
2. (Analysis) Determine the number of days between January 20 and December 5.
3. (Reflection) 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.
4. (Analysis) Give two days that span an interval of 20 days.
5. (Analysis) Give two days that span an interval of 200 days.
6. (Reflection) 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.

Preparation

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

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 procedures 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 determine the number of days in all the months between the two given months, add that to the days remaining in the first month, and add that sum to 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, rewriting a 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
consecutive, 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 procedures. Choosing good names for the procedures 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 2002, 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-day-span earlier-date later-date) )
      (else
         (general-days-spanned earlier-date later-date) ) ) )
```

The code includes a comment describing the purpose of the procedure and the assumptions made about the arguments. Same-month? and consecutive-months? are *predicate* procedures, 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 procedures 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 procedures* same-month?, consecutive-months?*,* same-month-span*,* consec-months-day-span*, and* general-day-span.

Here we have used the five procedures 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 procedures. Decomposition typically involves repeating the following steps:

a. determining a value that is to be computed,

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

c. using the procedure, and finally

d. designing the body of the procedure.

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 procedure. 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 procedures. 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 procedures 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-word sentence. The first word is the month and the second word is a day within the month. Thus two dates have the same month if they have the same first word.

Stop and predict ➨ *Write the* same-month? *procedure.*

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

One might devise a solution that accesses the months by using procedure first directly. The code will be clearer, however, if *accessing* procedures are used to refer to the two components of a date. Accessing procedures *name* the corresponding pieces of the sentence, making it easier to understand what is being accessed in a given section of code. Thus we will provide a procedure 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 procedures 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 procedure 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 procedures that will have to change will be the month-name and date-in-month procedures; everything else should work unchanged.

The same-month? procedure 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 2002.

(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 to name 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 procedure have?*

Stop and consider ➨ *What do you think of this procedure? 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 procedure 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 procedure 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 procedure 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 procedure 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 procedures; 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 procedures 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 procedure 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 procedure* 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 are left in the first month. This is the number of days in the month, minus the date of the month, plus 1.

A procedure 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 procedure days-in-month. *Recycling* the month-number code in this way saves time. In general, it is also good to design procedures whose purpose is similar to have similar code as well; the correctness of one of the procedures 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 2002.

(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 procedures 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 procedures just designed, and use it to test the procedures.*

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

Stop and help ➨ *Write a stub* general-day-span procedure. *Then test the entire* day-span procedure*, 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. Procedures learned so far in Scheme are not powerful enough to allow implementation of the procedure for computing the answer by hand. (Procedures 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

7. (Reflection) The same-month? procedure 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.

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

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

10. (Reflection) Generally, one should use a noun to name a procedure. Why?

11. (Modification, reflection) Rewrite the procedures designed so far without using accessing procedures. Then explain why the rewritten code might be more difficult to understand than the code designed in the narrative.

12. (Debugging) 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.

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

14. (Modification) Fill in the blank to complete the following alternative definition of the days-in-month procedure:

```
(define (days-in-month month)
   (item
      (month-number month)
      _____________________________ ) )
```

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 procedures in the first attempt at a solution worked with dates in their original format. One way to generate an alternative solution is to convert arguments to a different format before working with them, in the same way we did in designing the consecutive-months? procedure. Here we seek to convert the dates to something easier to manipulate, like integers, and then write procedures to work with dates in the new format.

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

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

```
; Return the difference in days between earlier-date and
; later-date. Earlier-date and later-date both represent dates
; in 2002, 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 merely concatenate the two with the word procedure 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 2002. We can use the fact that every date is some number of days from January 1, 2002 to solve the problem by converting each date into an integer day-in-2002! 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 procedure, 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 months preceding the date. 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 procedures. Here, we want a procedure that, given the name of a month, returns the total number of days in all preceding months. Transforming the old procedures into the desired procedure 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 *procedure that, given a month as argument, 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 procedure day-in-year:

```
; Return the number of days from January 1
; to the given date, inclusive.
; Date represents a date in 2002.

(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 "transform" procedure. 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 procedures 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

15. (Debugging) 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.

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

17. (Modification) Fill in the blank to complete the following alternative definition of the days-preceding procedure:

```
(define (days-preceding month)
   (item
      (month-number month)
      _____________________________ ) )
```

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

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

20. (Application) Fill in the blanks in the procedure 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 _____ )))))
```

21. (Analysis) If the arguments to day-span are legal dates, what are the possible values for days-preceding's argument? 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 procedures 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

22. (Analysis) 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.
23. (Analysis) What is the largest value that each version of day-span can return?
24. (Analysis) Provide a call to either version of days-spanned that returns the value 0. Hint: the arguments will not satisfy the claim made in the comment.
25. (Analysis) List all ways to provide *illegal* arguments to day-span.
26. (Analysis) For which of the types of arguments you described in the previous exercise does each version of day-span crash?
27. (Modification) 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.
28. (Reflection) What programming technique employed in this case study simplified the modification of the previous exercise?
29. (Reflection) What bugs do you think that a programmer is most likely to encounter in the final day-span program?
30. (Application) The twelve months of the Islamic year have alternately 30 and 29 days; thus a non-leap year has 354 days. Write a Scheme procedure that's given a two-word sentence representing a month and a day on the Islamic calendar and returns the day of the year, an integer between 1 and 354.
31. (Reflection, analysis) What makes this problem easier or harder than computing the day of year using our calendar?
32. (Application) Write a procedure valid-date? that returns true when given as argument a legal date in 2002, and returns false otherwise.
33. (Application) Write a procedure 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.
34. (Modification) Extend the day-span procedure to take arguments whose third words specify a year in the 20th century.

35. (Application) Write two solutions for the problem of computing the difference between two *distance measurements*. A distance measurement is a two-word sentence whose first word is an integer number of feet and whose second word is an integer number of inches. Thus the sentence (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.

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

37. (Reflection) 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 difference in days between earlier-date and later-date.
; earlier-date and later-date both represent dates in 2002, 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 procedures for the components of a date.

(define (month-name date) (first date))
(define (date-in-month date) (first (butfirst date)))

; Return true if date1 and date2 are dates in the same month, and
; false otherwise. Date1 and date2 both represent dates in 2002.

(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 2002.

(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 2002.

(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 2002.

(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 2002.

(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 procedures for the components of a date.

(define (month-name date) (first date))
(define (date-in-month date) (first (butfirst 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 2002.
(define (day-of-year date)
    (+ (days-preceding (month-name date)) (date-in-month date)) )

; Return the difference in days between earlier-date and later-date.
; Earlier-date and later-date both represent dates in 2002,
; 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))))
```