"Difference Between Dates" Case Study (Higher-Order Procedures Version) © 1996 M. J. Clancy and M. C. Linn

Preparation

The reader should have experience with the use of every, keep, and accumulate.

Yet another approach to computing the difference between dates

What are the drawbacks of the first solution?

The earlier solution to the difference-between-dates problem can be difficult to debug because there is no easy way to check the correctness of the procedure that returns the number of days preceding a given month other than by repeating the computation by which we determined it in the first place:

```
(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) ) )
```

One addition error, for instance, might invalidate the result values for all subsequent months.

Stop and consider ➨ *The* month-number *and* days-in-month *procedures are almost identical to the* days-preceding *procedure. Why are they easier to verify as correct?*

We would like to replace this procedure by a computation that reflects the procedure we used to compute the value by hand, which applies *the same* process to all the months. This would significantly reduce the chance of an accidental error in the computation for a single month. The computation should begin with the days-in-month values, since most people can quickly verify that these values are correct.

How can higher-order procedures help?

This process of adding up values for a *collection* of months suggests using higher-order procedures, which take a sentence or word as an argument and apply a procedure to all of its elements in a single operation.

For instance, the adding up of days in the various months suggests accumulation with the accumulate procedure. Given an argument representing the days in a sequence of months, we could use accumulate with + to compute the number of days in all the months. Thus one might find the number of days preceding June by accumulating the procedure + over the list of days in the months January through May, i.e. (3128313031).

We might then replace the cond in days-preceding by the following code:

```
(cond
   ((equal? month 'january)
      (accumulate + '()))
   ((equal? month 'february)
      (accumulate + '(31)))
   ((equal? month 'march)
      (accumulate + '(31 28)))
   ((equal? month 'april)
      (accumulate + '(31 28 31)))
      ... )
```

This solution is not much of an improvement. It requires lots of code, which is likely to contain at least a typing error if not worse.

How might a twelve-way test be replaced by a more general computation?

A procedure to *create* the list over which to accumulate the sum would be better. We'll call this procedure preceding-months-lengths. Then a twelve-way cond would not be necessary; a simple expression like

```
(accumulate +
   (preceding-months-lengths month) )
```

would suffice. For example, given the month April, preceding-months-lengths would return the month lengths of the preceding months, (31 28 31). In general, preceding-months-lengths would return the following:

month	*returned result*
january	( )
february	(31)
march	(31 28)
april	(31 28 31)
may	(31 28 31 30)
june	(31 28 31 30 31)
july	(31 28 31 30 31 30)
august	(31 28 31 30 31 30 31)
september	(31 28 31 30 31 30 31 31)
october	(31 28 31 30 31 30 31 31 30)
november	(31 28 31 30 31 30 31 31 30 31)
december	(31 28 31 30 31 30 31 31 30 31 30)

How is preceding-months-lengths designed?

Design of preceding-months-lengths will involve both reasoning backward from what preceding-months-lengths should return, and reasoning from what we already have. For instance, we can easily construct the sentence
(january february … december) of all the months for preceding-months-lengths to work with. Two approaches for using this sentence are the following:

- From the sentence containing all the month names, produce the sentence containing only the names of months that precede the given month. From that smaller sentence, produce the sentence containing the lengths in days of those months.
- From the sentence containing all the month names, produce the sentence containing the lengths in days of those months, i.e. (31 28 31 … 30 31). From that sentence, produce the sentence containing the lengths in days of the relevant months, i.e. those that precede the given month.

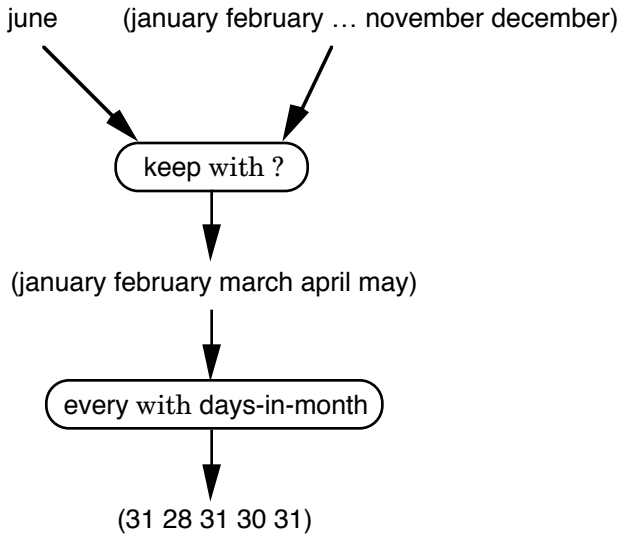
Stop and predict ➨ *Explain which higher-order procedures will be needed to produce each of the sentences just described, and how they will be used.*

Which higher-order procedures are used in preceding-months-lengths?

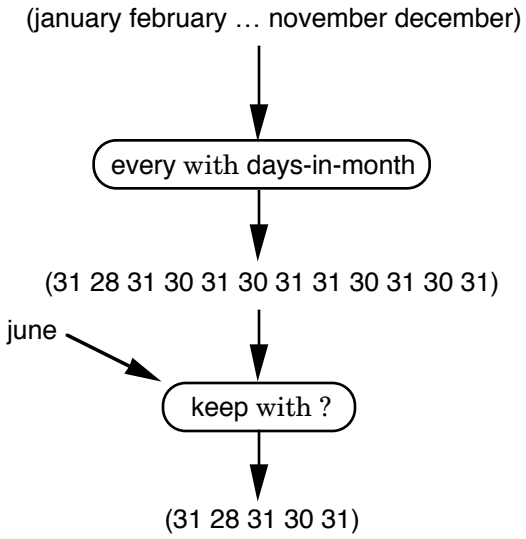
Each of the approaches just described involves translation of a sentence of month names to a sentence of month lengths, along with shrinking a sentence of information (names or lengths) of all months to a sentence of information only for relevant months. The translation is done with every, using the days-in-month procedure as argument, and the shrinking with keep. We now must determine which order to apply these higher-order procedures: every first, or keep first.

Is keep applied to the result of every, or vice-versa?

An excellent aid for choosing between the two approaches is a *data-flow diagram*, in which the successive transformations of sentences are accompanied with the procedures that perform those transformations. The two diagrams representing the (every … (keep …)) option and the (keep … (every …)) option for the months preceding June appear below.



the (every … (keep …)) option:
applying keep first, then every



the (keep … (every …)) option:
applying every first, then keep

The (keep … (every …)) option would apply keep to the sentence (312831303130 3131 30 31 30 31). The problem here is that keep tests each word in its argument sentence *in isolation* from all the other words in the sentence. It will not be able, for instance, to return all words in the sentence up to or after a given position, since such a test would require information about the *relation* of words in the sentence (i.e. where they appear relative to one another). Thus keep will not be appropriate for shrinking the sentence of month lengths.

Stop and consider ➨ *Explain in your own words why the* (keep … (every …)) *option won't work.*

We move to the (every … (keep …)) option, applying keep before the month names are removed. Here's the rearranged code:

```
(define (preceding-months-lengths month)
   (every
      days-in-month
      (keep
         ____________________________
         '(january february … december) ) ) )
```

What procedure is used with keep to shrink the list of months?

So, which months are to be kept? It helps to create another table that shows the result returned by keep:

month	*returned result from* keep
january	()
february	(january)
march	(january february)
april	(january february march)
…	

The months to be retained are those that precede the month specified. Thus a procedure that determines if one month precedes another is needed.

In part I of this case study, we defined a procedure consecutive-months? that determined if two months were adjacent. Design of this procedure applied the technique of converting the arguments to values that were easy to compare.

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

A similar approach can be used here:

```
; Return true if month1 precedes month2.
(define (month-precedes? month1 month2)
   (<
      (month-number month1)
      (month-number month2) ) )
```

(The "dead end" solution in the first version has yielded a procedure we could use again. It was not such a dead end after all!)

How is month-precedes? adapted for use with keep?

There is a problem here. Month-precedes? takes two arguments, while keep's procedure argument takes only one argument of its own. What we really need is a version of month-precedes? with the second month *held constant*. It would fit into the code we've already designed as follows:

```
; Return a sentence of days in months
; preceding the given month.

(define (preceding-months-lengths month)
   (every days-in-month
      (keep
         procedure that applies month-precedes?
         with the second month held constant
         '(january february … december) ) ) )
```

The lambda special form provides a way to construct the desired procedure. Used inside preceding-months-lengths, the procedure

```
(lambda (month2)
   (month-precedes? month2 month) )
```

is exactly what is needed.

Stop and consider ➨ *What error message will be produced if a procedure* earlier-than-month? *is defined as given above, outside the* preceding-months-lengths *procedure?*

The higher-order procedure version of the days-spanned-by procedure appears in Appendix D.

Stop and help ➨ *Devise test data and test the program in Appendix D.*

How can higher-order procedures enable progress on the dead-end approach?

An approach like the one just used can help us complete the code we gave up in part I. Recall the approach there of separating the computation into three situations:

1. two dates in the same month (handled successfully);
2. two dates in consecutive months (also handled successfully);
3. two dates in months further apart (not handled).

We had devised a procedure for handling the third case by hand, namely computing the sum of three quantities:

> the number of days remaining in the month of the first given date;
> the number of days in all the months between the two given dates; and
> the date-in-month of the second date.

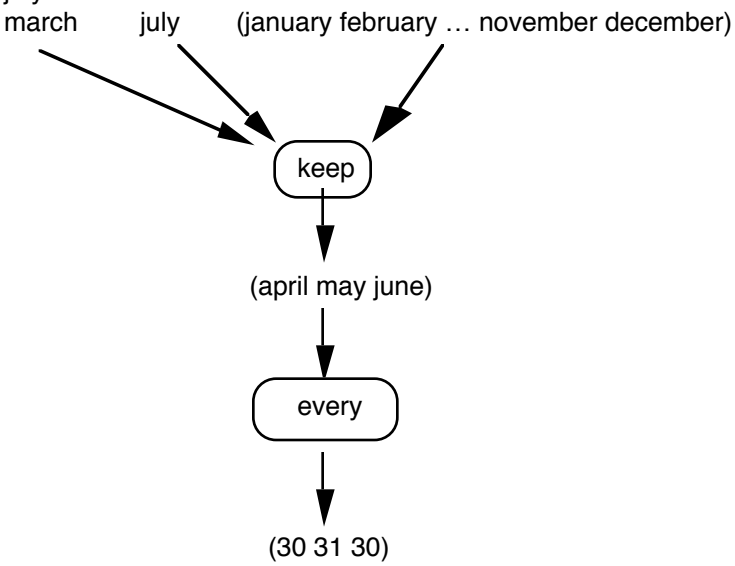
Previously, we had been unable to determine the number of days in all the months between the two given dates in a reasonable way. Higher-order procedures provide the tools for doing this.

We just designed a procedure that, given a month, returns a sentence of the days in the preceding months. This procedure can be used as the basis for another procedure that, given two months, returns a sentence of the days in all the months in between. We'll call it between-months-lengths, to reflect the similarity with preceding-months-lengths:

```
; Return a sentence of days in months between the two
; given months (not including those months).

(define (between-months-lengths earlier-month later-month)
   (every days-in-month
      (keep
         (lambda (month)
            (and
               (month-precedes? earlier-month month)
               (month-precedes? month later-month) ) )
         '(january february … december) ) ) )
```

A data-flow diagram appears below that displays how this procedure works given the arguments march and july.



Between-months-length can then be used to code general-span:

```
(define (general-day-span earlier-date later-date)
   (+
      (days-remaining earlier-date)
      (accumulate
         +
         (between-month-lengths
            (month-name earlier-date)
            (month-name later-date)) )
      (date-in-month later-date) ) )
```

The resulting code appears in Appendix E.

Appendix D—A version of day-span that uses higher-order procedures

```
;; Access procedures for the components of a date.
(define (month-name date) (first date))
(define (date-in-month date) (first (butfirst date)))

(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 the number of days in the month named month.
(define (days-in-month month)
    (item
        (month-number month)
        '(31 28 31 30 31 30 31 31 30 31 30 31) ) )

;; Return true when month1 precedes month2, and false otherwise.
(define (month-precedes? month1 month2)
    (< (month-number month1) (month-number month2)) )

;; Return a sentence containing the lengths of months that precede
;; the given month.
(define (preceding-months-lengths month)
    (every
        days-in-month
        (keep
            (lambda (month2) (month-precedes? month2 month))
            '(january february march april may june
              july august september october november december) ) ) )

;; Return the number of days from January 1 to the first day
;; of the month named month.
(define (days-preceding month)
    (accumulate + (preceding-months-lengths month)) )

;; 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 difference in days between 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))) )
```

Appendix E—A modified version of the "dead-end" code from Appendix A

```
;; Return the difference in days between earlier-date and later-date.
;; Earlier-date and later-date both represent dates in a non-leap year,
;; 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 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 a non-leap year.
(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 the difference in days between earlier-date and later-date,
;; which both represent dates in the same month of a non-leap year.
(define (same-month-span earlier-date later-date)
    (+ 1
        (- (date-in-month later-date) (date-in-month earlier-date)) ) )

;; 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 a non-leap year.
(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 represent dates in consecutive months of a non-leap year.
(define (consec-months-span earlier-date later-date)
    (+ (days-remaining earlier-date) (date-in-month later-date)))
```

```
;; Return the number of days in the month named month.
(define (days-in-month month)
    (item
        (month-number month)
        '(31 28 31 30 31 30 31 31 30 31 30 31) ) )

;; Return the number of days remaining in the month of the given date,
;; including the current day. date represents a date in a non-leap year.
(define (days-remaining date)
    (+ 1 (- (days-in-month (month-name date)) (date-in-month date))) )

;; Return true when month1 precedes month2, and false otherwise.
(define (month-precedes? month1 month2)
    (< (month-number month1) (month-number month2)) )

;; Return a sentence of lengths of months between the two given months
;; (not including those months).
(define (between-month-lengths earlier-month later-month)
    (every
        days-in-month
        (keep
            (lambda (month)
                (and
                    (month-precedes? earlier-month month)
                    (month-precedes? month later-month) ) )
            '(january february march april may june
                july august september october november december) ) ) )

;; Return the difference in days between earlier-date and later-date,
;; which represent dates in consecutive months of a non-leap year.
;; This is just the number of days remaining in the earlier month
;; plus the date in month of the later month plus the number of days
;; in all the months in between.
(define (general-day-span earlier-date later-date)
    (+
        (days-remaining earlier-date)
        (accumulate
            +
            (between-month-lengths
                (month-name earlier-date)
                (month-name later-date)) )
        (date-in-month later-date) ) )
```