Sometimes Structural Recursion Isn't Enough

CS 5010 Program Design Paradigms “Bootcamp” Lesson 8.1
Module Introduction

• Sometimes problems don't fit neatly into the pattern of recursion on the sub-pieces of the data.

• In this module, we'll see some examples of problems like this, and introduce a new design strategy, *generative recursion*, to handle this.

• Generative recursion and invariants together provide a powerful combination.
Generative Recursion is more powerful than data decomposition

• Functions written using data decomposition are guaranteed to halt with an answer, because they follow the shape of the data, but generative recursion allows you to write functions that don't always halt.

• So every time we write a function using generative recursion, we need to provide a termination argument that explains why the function really does halt
  – or else warn the user that it may not halt.
  – easiest way to make a termination argument is by supplying a halting measure.
Learning Objectives

• At the end of this lesson, the student should be able to
  – identify two common algorithms that do not fit into the pattern of data decomposition
  – explain why they can't be made to fit the pattern
An example: decode

(define-struct diffexp (exp1 exp2))

;; A DiffExp is either
;; -- a Number
;; -- (make-diffexp DiffExp DiffExp)

Here is the data definition for diffexps. These are a simple representation of difference expressions, much like the arithmetic expressions we considered in some of the earlier problem sets.
Examples of diffexps

(make-diffexp 3 5)
(make-diffexp 2 (make-diffexp 3 5))
(make-diffexp
  (make-diffexp 2 4)
  (make-diffexp 3 5))

Writing out diff-exps is tedious at best.
Not very human-friendly...

• How about using more Scheme-like notation, eg:

(- 3 5)
(- 2 (- 3 5))
(- (- 2 4) (- 3 5))
Task: convert from human-friendly notation to diffexps.

• Info analysis:
  – what's the input?
  – answer: S-expressions containing numbers and symbols
Data Definitions

;; An Atom is one of
;; -- a Number
;; -- a Symbol

;; An SexpOfAtom is either
;; -- an Atom
;; -- a ListofSexpOfAtom

;; A ListofSexpOfAtom is either
;; -- empty
;; -- (cons SexpOfAtom ListofSexpOfAtom)
Templates

(define (sexp-fn sexp)
  (cond
    [(atom? sexp) (... Sexp ...)]
    [else (... (los-fn sexp) ...) ]))

(define (los-fn los)
  (cond
    [(empty? los) ...]
    [else (... (sexp-fn (first los))
       (los-fn (rest los)))]))

And the templates to go with it.
Contract and Examples

decode : SexpOfAtom -> DiffExp

(- 3 5) => (make-diffexp 3 5)
(- 2 (- 3 5)) => (make-diffexp

    2

    (make-diffexp 3 5))

(- (- 2 4) (- 3 5)) => (make-diffexp

    (make-diffexp 2 4)

    (make-diffexp 3 5))
Umm, but not every SexpOfAtom corresponds to a diffexp

(- 3) does not correspond to any diffexp
(+ 3 5) does not correspond to any diffexp
(- (+ 3 5) 5) does not correspond to any diffexp
((1)) does not correspond to any diffexp
(( - 2 3) (- 1 0)) does not correspond to any diffexp
(- 3 5 7) does not correspond to any diffexp

But here are some other inputs that are legal according to our contract. None of these is the human-friendly representation of any diff-exp.
A Better Contract

;; A Maybe<X> is one of
;; -- false
;; -- X

;; (define (maybe-x-fn mx)
;;   (cond
;;     [(false? mx) ...]
;;     [else (... mx)]))

decode
  : SexpOfAtom -> Maybe<DiffExp>
Function Definition (1)

;;; decode : SexpOfAtom -> Maybe<DiffExp>

;;; Algorithm: if the sexp looks like a diffexp at the top level, 
;;; recur, otherwise return false. If either recursion fails, return 
;;; false. If both recursions succeed, return the diffexp.

(define (decode sexp)
  (cond
    [(number? sexp) sexp]
    [(looks-like-diffexp? sexp)
      (local
        ((define operand1 (decode (second sexp)))
         (define operand2 (decode (third sexp))))
        (if (and (succeeded? operand1)
                 (succeeded? operand2))
            (make-diffexp operand1 operand2)
            false))]
    [else false]))

Now we can write the function definition.
Function Definition (2)

;; looks-like-diffexp? : SexpOfAtom -> Boolean
;; WHERE: sexp is not a number.
;; RETURNS: true iff the top level of the sexp looks like
;; a diffexp.
;; At the top level, a representation of a
;; diffexp must be either a number or a list of
;; exactly 3 elements, beginning with the symbol -
;; STRATEGY: function composition
(define (looks-like-diffexp? sexp)
  (and
   (list? sexp)
   ;; at this point we know that
   ;; sexp is a list
   (= (length sexp) 3)
   (equal? (first sexp) '-))))

In this function definition, we add an invariant (the WHERE clause) to record the assumption that our input is not merely an SexpOfAtom, but is rather an SexpOfAtom that is not a number. We know this assumption is true, because looks-like-diffexp? is only called after number? fails.
Function Definition (3)

;;; succeeded? : Maybe<X> -> Boolean
;;; RETURNS: Is the argument an X?
;;; strategy: data decomposition on Maybe<X>
(define (succeeded? mx)
  (cond
    [(false? mx) false]
    [else true]))

How can this function be simplified?
But wait: what's the strategy?

;; decode : SexpOfAtom -> Maybe<DiffExp>

;; Algorithm: if the sexp looks like a diffexp at the top level, recur, otherwise return false. If either recursion fails, return false. If both recursions succeed, return the diffexp.

(define (decode sexp)
  (cond
    [(number? sexp) sexp]
    [(looks-like-diffexp? sexp)
     (local
      ((define operand1 (decode (second sexp)))
       (define operand2 (decode (third sexp))))
      (if (and (succeeded? operand1)
               (succeeded? operand2))
        (make-diffexp operand1 operand2)
        false))]
    [else false]))

But what strategy is this? It doesn’t fit the template for SexpOfAtom. It’s not even close.
Something new happened here

• We recurred on the subpieces, **but**
  – we didn't use the structural predicates
  – we didn't recur on all of the subpieces

• This is not data decomposition
Another example: merge-sort

• Let's turn to a different example: merge sort, which you should know from your undergraduate data structures or algorithms course.

• Divide the list in half, sort each half, and then merge two sorted lists.
merge

;; merge : SortedList SortedList -> SortedList
;; merges its two arguments
;; strategy: data decomposition on lst1, lst2: SortedList
;; (see book)
(define (merge lst1 lst2)
  (cond
   [(empty? lst1) lst2]
   [(empty? lst2) lst1]
   [else
    (if (< (first lst1) (first lst2))
     (cons (first lst1)
           (merge (rest lst1) lst2))
     (cons (first lst2)
           (merge lst1 (rest lst2))))]))

To define merge-sort, we start with merge, which merges two sorted lists of numbers. This is one of the few places where we really need structural decomposition on two lists at once. The textbook contains more information on this topic.

If the lists are of length n, this function takes time proportional to n. We say that the time is O(n).
merge-sort

;; merge-sort : ListOfNumber -> SortedList
(define (merge-sort lon)
  (cond
   [(empty? lon) lon]
   [(empty? (rest lon)) lon]
   [else
     (local
       ((define evens (even-elements lon))
        (define odds  (odd-elements lon)))
       (merge
        (merge-sort evens)
        (merge-sort odds)))]))

Now we can write merge-sort. merge-sort takes its input and divides it into two approximately equal-sized pieces. Depending on the data structures we use, this can be done in different ways. We are using lists, so the easiest way is to take every other element of the list, so the list (10 20 30 40 50) would be split into (10 30 50) and (20 40).

We sort each of the pieces, and then merge the sorted results.
Running time for merge sort

• Splitting the list in this way takes time proportional to the length \( n \) of the list. The call to merge likewise takes time proportional to \( n \). We say this time is \( O(n) \).

• If \( T(n) \) is the time to sort a list of length \( n \), then \( T(n) \) is equal to the time \( 2 \times T(n/2) \) that it takes to sort the two sublists, plus the time \( O(n) \) of splitting the list and merging the two results:

• So the overall time is

\[
T(n) = 2 \times T(n/2) + O(n)
\]

• When you take algorithms, you will learn that all this implies that \( T(n) = O(n \log n) \). This is better than an insertion sort, which takes \( O(n^2) \).
Something new happened here

• Merge-sort did something very different: it recurs on two things, neither of which is \( \text{rest lon} \).

• We recurred on
  – \( \text{even-elements lon} \)
  – \( \text{odd-elements lon} \)

• Neither of these is a sublist of \( \text{lst} \)
  – We didn't follow the data definition!
Summary

• You should now be able to
  – identify two common algorithms that do not fit into the pattern of data decomposition
  – explain why they can't be made to fit the pattern