Sometimes Even Accumulators Aren't Enough

CS 5010 Program Design Paradigms "Bootcamp"

Lesson 6.1

An example: decode

```
(define-struct diffexp (exp1 exp2))

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

Examples of diffexps

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

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?
  - 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)))]))
```

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 anything**
`(+ 3 5)`	**does not correspond to anything**
`(- (+ 3 5) 5)`	**does not correspond to anything**
`((1))`	**does not correspond to anything**
`((- 2 3) (- 1 0))`	**does not correspond to anything**
`(- 3 5 7)`	**does not correspond to anything**

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>
```

Code (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]))
```

Code (2)

```
;; looks-like-diffexp? : SexpOfAtom -> Boolean
;; WHERE: sexp is not a number.
;; does the sexp look like a diffexp at the top level?
;; Algorithm: 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: domain knowledge
(define (looks-like-diffexp? sexp)
  (and
   (not (symbol? sexp))
   ;; at this point we know that sexp must be a list
   (= (length sexp) 3)
   (equal? (first sexp) '-)))
```

Code (3)

```
;; succeeded? : Maybe<X> -> Boolean
;; Is the argument false or an X?
;; strategy: Struct Decomp on Maybe<X>
(define (succeeded? mx)
  (cond
    [(false? mx) false]
    [else true]))
```

But wait: what's the strategy?

```
;; decode : Sexp<Atom> -> 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 (or
             (false? operand1)
             (false? operand2))
         false
         (make-diffexp operand1 operand2)))]
    [else false]))
```

That didn't look like the SD template!

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 structural decomposition

Another example: merge-sort

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

merge

```
;; merge : SortedList SortedList -> SortedList
;; merges its two arguments
;; strategy: structural decomposition on both arguments
   (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))))]))
```

Time = O(n)

merge-sort

```
;; merge-sort : ListOf<Number> -> SortedList
(define (merge-sort lon)
  (cond
    [(empty? lon) lon]
    [(empty? (rest lon)) lon]
    [else
      (local
       ((define evens (even-elements lon))
        (define odds  (even-elements (rest lon))))
       (merge
        (merge-sort evens)
        (merge-sort odds)))]))
```

```
wishlist: even-elements
collects elements 0, 2, 4,
etc. of its argument.
```

- $T(n) = 2T(n/2)+O(n)$
- solve to get
  $T(n) = O(n \log n)$

Something new happened here

- instead of recurring on **`(rest lon)`** , we recurred on
  - **`(even-elements lon)`**
  - **`(even-elements (rest lon))`**
- Neither of these is a sublist of lst
  - We didn't follow the data definition!

Introducing General Recursion

CS 5010 Program Design Paradigms
“Bootcamp”

Lesson 6.2

General Recursion

Goals of this lesson

- Introduce a new design strategy: general recursion
  - "non-structural" recursion
  - divide-and-conquer
- An Example
- Pattern
- New deliverable: *termination argument*

General Recursion

- How to solve the problem:
  - If it's easy, solve it immediately
  - If it's hard
    - find one or more easier problems whose solutions will help you find the solution to the original problem.
    - then combine the solutions to get the solution to your original problem
  - How to find the "easier" problems?
    - Probably need ingenuity
    - *But you must document how each new problem is easier than original.*

Pattern for General Recursion

New required piece of the recipe

```
;; solve : Problem -> Solution
;; purpose statement
;; TERMINATION ARGUMENT: explain how new-problem1 and new-
   problem2 are easier than the-problem.
(define (solve the-problem)
  (cond
    [(trivial1? the-problem) (trivial-solution1 the-problem)]
    [(trivial2? the-problem) (trivial-solution2 the-problem)]
    [(difficult? the-problem)
     (local
       ((define new-problem1 (divider1 the-problem))
        (define new-problem2 (divider2 the-problem)))
       (combine-solutions
        (solve new-problem1)
        (solve new-problem2)))]))
```

No recipe for this; you've just got to be a little clever

Key: problem-specific insight about divider1, divider2, and combine-solutions

This pattern is more flexible than a template

- How many trivial cases are there?
- How many subproblems do you divide your original into?
  - Could be just 1
  - Could be a whole list

Pattern for one subproblem

```
;; solve : Problem -> Solution
;; purpose statement...
;; examples ...
;; TERMINATION ARGUMENT: explain how new-problem is
;; easier than problem
(define (solve problem)
  (cond
    [(trivial1? problem) (trivial-solution1 problem)]
    [(trivial2? problem) (trivial-solution2 problem)]
    [else
     (local
       ((define new-problem (modify-problem problem)))
       (modify-solution (solve new-problem)))]))
```

Pattern for a list of subproblems

```
;; solve : Problem -> Solution
;; purpose statement...
;; examples ...
;; TERMINATION ARGUMENT: explain how each of the problems in
;; new-problems is easier than problem
(define (solve problem)
  (cond
    [(trivial1? problem) (trivial-solution1 problem)]
    [(trivial2? problem) (trivial-solution2 problem)]
    [(difficult? problem)
     (local
       ((define new-problems (generate-subproblems problem)))
       (combine-list-of-solutions
        (map solve new-problems)))]))
```

Termination Argument

- New required piece of the function header.
- Explains how each of the subproblems are easier than the original
  - You get to say what "easier" means
- But how do you explain this?
- Usually this takes the form of a *halting measure*.

Halting Measure

- A quantity that can't be less than zero
- Examples:
  - the size of an s-expression
    - lots of ways to define this
  - the length of a list
- Must guarantee that it decreases at each recursive call in your function.

Halting Measure for **decode**

- the size of an sexp is always non-negative
- **`(second sexp)`** and **`(third sexp)`** each have smaller size than **`sexp`**.
- So **`(size sexp)`** is a halting measure for **`decode`**.

Halting Measure for **`merge-sort`**

- **`(length lst)`** is always non-negative
- At each recursive call, **`(length lst)`** ≥ 2
- If **`(length lst)`** ≥ 2, then

  `(length (even-elements lst))` and

  `(length (even-elements (rest lst)))`

  are both *strictly less* than **`(length lst).`**
- So **`(length lst)`** is a halting measure for **`merge-sort`**.

General Recursion vs. Structural Decomposition

- Structural decomposition is a special case: it's a standard recipe for finding subproblems that are guaranteed to be easier.
  - A field is always smaller than the structure it's contained in.
- For general recursion, must always explain in what way the new problems are easier.
- Use structural decomposition (+ accumulators) when you can, general recursion when you need to.
- Always use the simplest tool that works!

In the definition of function **`f`** :

- **`(... (f (rest lst)) ...)`** is structural
- **`(f (... (rest lst)) ...)`** is general

Summary

- We've introduced *general recursion*.
- Solve the problem by combining solutions to easier subproblems.
- Must give a *termination argument* that shows why each subproblem is easier.
- Structural decomposition is a special case where the data def guarantees the subproblem is easier.
- Always use the simplest tool that works!

Graph Reachability – Part 1

CS 5010 Program Design Paradigms
"Bootcamp"

Lesson 6.3

Goals of this lesson

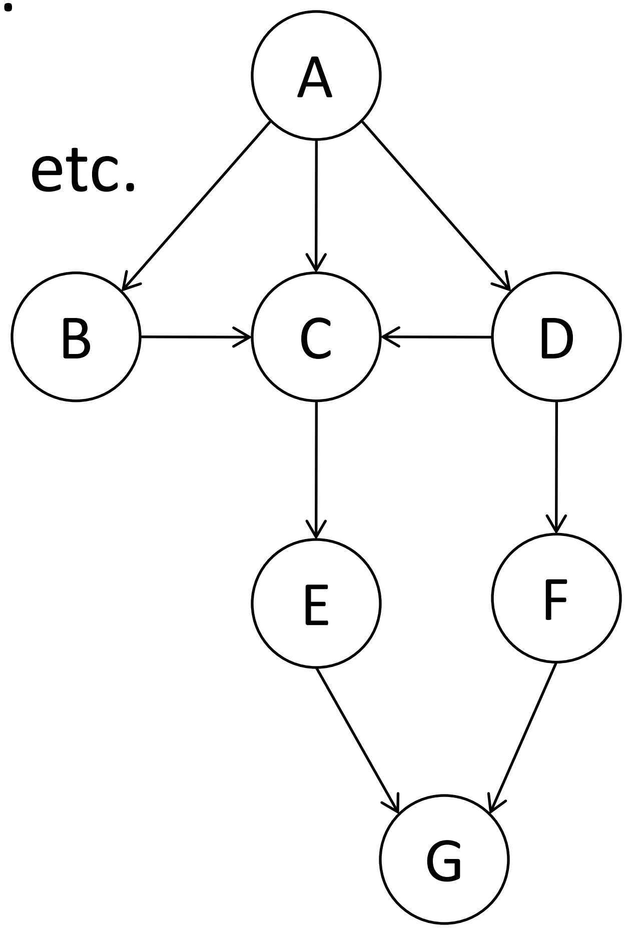
- Learn about an important application of general recursion
- Learn about reachability in a graph
  - think about alternative data representations
- Limits of termination arguments

What's a graph?

nodes: A, B, C, etc.

edges:

(A,B), (A,C),(A,D), etc.



reachability:

is there a path?

from	to	
A	E	Yes
D	G	Yes
C	G	Yes
E	D	No
C	F	No

paths:

(A,C,E)

(B,C,E,G)

(A,D,C,E)

(A)

How to represent a graph as data?

- What information needs to be represented?
  - nodes
  - edges
- What operations do we need to support?
  - **`node=? : Node Node -> Boolean`**
  - **`successors`**

    `: Node Graph -> ListOf<Node>`

List-of-Edges Representation

```
;; A Node is a Symbol

(define-struct edge (from to))
;; An Edge is a (make-edge Node Node)

;; A Graph is a ListOf<Edge>
```

Graph Reachability via General recursion

- Livecoding: 06-3-graph-reachability.rkt

What if my function doesn't always halt?

You must deliver a termination argument for each function that uses general recursion. This has one of two forms:

1. The function produces a solution for all problems because *[show halting function]* is always non-negative and gets smaller at every recursive call
2. The function does not terminate on some input problems, example: *[put here a specific example that does not terminate]*.

Which of these applies to our example?

Summary

- We've applied General Recursion to an important problem: graph reachability
- We considered the functions we needed to write on graphs in order to choose our representation(s).
- We used general recursion with a *list* of subproblems
- We used list abstractions to make our program easier to write