211lab05_S11

Generative Recursion

Students should use DrScheme, following the design recipe, on the exercises at their own pace, while labbies wander among the students, answering questions, bringing the more important ones to the lab's attention.

Generative Recursion Design Recipe Review

In class, we presented a new more general methodology for developing Scheme functions called *generative recursion*.
To develop a generative recursive, we use a generalized design recipe that incorporates the following changes:

- When making function examples, we need even more examples, as they help us to find the algorithm we will use.
- When making a template, we don't follow the structure of the data. Instead, we ask ourselves six questions:
    1. What is (are) the trivial case(s)?
    2. How do we solve the trivial case(s)?
    3. For the non-trivial case(s), how many subproblems do we use?
    4. How do we generate these subproblems?
    5. Can we combine the results of these subproblems to solve the given problem?
    6. How do we combine the results of these subproblems?
- We use the answers to the preceding questions to fill in the following very general template:

```
(define (f _args_)
  (cond [(_trivial_? _args)) (_solve-trivial_ _args_)]
        [else (_combine_ (f (_generate-subproblem-1_ _args_))
                         ...
                         (f (_generate-subproblem-n_ _args_)))]))
```

- We add a step to reason about why our algorithm terminates, since this is no longer provided by simply following the structure of the data.

Converting a token stream to a list input stream

The Scheme reader reads list *inputs* rather than symbols or characters where a list input is defined by the following set of mutually recursive data definitions:

An *input* is either:

- any built-in Scheme value that is not a list (`empty`? or =cons?=)
- a *general list*.

A *general list* is either:

- `empty` ,
- `(cons i agl)` where `i` is an *input* and `agl` is a *general list*.

Note that the second clause above could be written as the following two clauses:

- `(cons p l)` where p is a primitive Scheme value that is not a list, or
- `(cons el l)` where `el` and `l` are general lists.

Your task is to write the core function `listify` used in an idealized Scheme reader stream. `listify` converts a list of *tokens*, where a *token* is defined below, to a list of *inputs* as defined above.

A *token* is either

- any built-in Scheme value (including symbol, boolean, number, string, `empty`, *etc.*; or
- (make-open), or
- (make-close)
  given the structure definitions

```
(define-struct open ())
(define-struct close ())
```

The open and close parenthesis objects (not the symbols `'(` and `')`) clearly play a critical role in token streams because they delimit general lists.

Examples:

```
(check-expect (listify empty) empty)
(check-error (listify '(a))  "listify: no open token found!")
(check-expect (listify (list (make-open) (make-close))) (list empty))
(check-expect (listify (list (make-open) 'a (make-close))) (list '(a)))
(check-expect (listify (list (make-open) 'a 'b (make-close))) (list '(a b)))
(check-expect (listify (list (make-open) (make-open) 'a (make-close) 'b (make-close))) (list '((a) b)))
```

Note the error style that is used: the error message starts with the name of the function where the error occurred. This is VERY useful when you have many functions that are calling each other and you have to figure out where an error occurred!

Methodology:

1. Always start with the structural function template for the data structure that you are processing -- your code must fundamentally match the data you are working on.
    a. In this case, with generative recursion, we can modify our standard function template on a list, wtih the notion that the list of tokens has 4 possibilities: empty, open, close and everything else. Create a template based on that "virtual" structural breakdown of the data.
    b. Are all the cases valid here? For instance, can a valid list of tokens start with a "close"? Write your listify function in terms of what is legal and what is not, leaving "..." where the list is legal. We'll come back and fill that part in.
2. Next, consider how else the data breaks down. Once again, in this generative recursion example, we can think of how the list of tokens breaks down into "phrases" that begin with "open" and end with "close".
    a. The processing the list of tokens thus breaks down into processing a phrase, which ends with a "close" and the rest of the list of tokens, which is the rest of the token list after the matching "close" that ends the phrase.
    b. Write helper functions (let's call them "listify-phrase" and "listify-rest" for consistency's sake) to process these pieces of the token list.
3. Write listify in terms of these helper function -- don't worry yet about how they work, just assume that they do (maybe you should write their contract and purposes now?).
    a. Since we've broken the problem down in terms of these two pieces, the "phrase" and the "rest of the tokens", the job now is to combine those results.
4. Now write the helpers.
    a. listify-rest: Once again, break the function down into cases corresponding to the possible token types.
        i. This function must find the matching "close" and return the list after that point.
        ii. Are there illegal tokens you could encounter?
        iii. What happens if you encounter an "open"?
    b. listify-phrase: Once again, break the function down into cases corresponding to the possible token types.
        i. This must create a gen-list and assumes that the last token it needs to process is a "close".
        ii. Are there illegal tokens you could encounter?
        iii. What should you return if you encounter a "close"?
        iv. What should you do if your encounter an "open"?
        v. What about something else, e.g. a symbol, number, whatever? What about an empty list?

Here's some test cases to use:

```
(check-error (listify-rest empty) "listify-rest: encountered empty list!")
(check-expect (listify-rest (list (make-close))) empty)
(check-expect (listify-rest (list 1 (make-close))) empty)
(check-expect (listify-rest (list 1 (make-close) 2)) '(2))
(check-expect (listify-rest (list 1 (make-open) 2 3 (make-close) 4 (make-close) 5 6)) '(5 6))

(check-error (listify-phrase empty)  "listify-phrase: empty list encountered!")
(check-error (listify-phrase (list (make-open) (make-close)))  "listify-phrase: empty list encountered!")
(check-expect (listify-phrase (list (make-open) (make-close) (make-close))) '(()))
(check-error (listify-phrase (list (make-open) 'a (make-close))) "listify-phrase: empty list encountered!")
(check-expect (listify-phrase (list (make-open) 'a 'b (make-close) (make-close))) '((a b)))
(check-expect (listify-phrase (list 'x (make-open) 'a 'b (make-close) 'y (make-close))) '(x (a b) y))
(check-expect (listify-phrase (list 'x (make-open) (make-open) 'a (make-close) 'b (make-close) 'y (make-
close))) '(x ((a) b) y))
```

You might find it useful to define the following predicate, though it is possible to not need it:

```
(define (input? t) (and (not (open? t)) (not (close? t))))
```

in your program. As a rough guideline follow this parse.ss program. Note that finding the first input is more involved than finding the first line. You will probably want to explicitly check for errors in the input because they correspond to natural tests on the form of the input.

Insertion Sort vs. Quicksort

In class, we've described two different ways of sorting numbers, insertion sort and quicksort.

```
;; insertion sort
(define (isort alon)
  (cond [(empty? alon) empty]
        [(cons? alon)  (insert (first alon) (isort (rest alon)))]))

(define (insert new sortedlon)
  (cond [(empty? sortedlon) (list new)]
        [else (if (< new (first sortedlon))
                  (cons new sortedlon)
                  (cons (first sortedlon) (insert new (rest sortedlon))))]))

;; quicksort (adapted to functional lists)
(define (qsort alon)
  (cond [(empty? alon) empty]
        [(cons? alon)
         (local [(define pivot (first alon))
                 (define other (rest alon))
                 (define lesser (filter (lambda (n) (<= n pivot)) other))
                 (define greater (filter (lambda (n) (> n pivot)) other))]
           (append (qsort lesser) (list pivot) (qsort greater)))]))
```

So we have two fundamentally different approaches to sorting a list (and there are lots of others, too). It seems unsurprising that each might behave differently. Can we observe this difference? Can we provide explanations? We'll do some timing experiments, outline the theoretical analysis, and see if the two are consistent.

If your programs only sort small lists a few times, it doesn't matter; any sort that is easy to write and woks correctly is fine. However, for longer lists, the difference really is huge. In the real world, sorting is an operation often done on very long lists (repeatedly).

In DrScheme, there is an expression `(time` *`expr`*`)` that can be used to time how long it takes the computer to evaluate something. For example,

```
(time (isort big-list))
```

determines how long it takes to sort `big-list` using the function `isort` and prints the timing results along with the evaluation result. Since we're only interested in the time, we can avoid seeing the long sorted list by writing

```
(time (empty? (isort big-list)))
```

The `time` operation prints three timing figures, each in milliseconds. The first is how much time elapsed while the computer was working on this computation, which is exactly what we're interested in.

Timing Exercises

Partner with someone else in lab to split the following work.

1. We need some data to use for our timing experiments. Write a function `up: nat -> (list-of nat)` which constructs a list of naturals starting with `0` in ascending order or retrieve it from a previous lab. For example, `(up 7)` returns `(list 0 1 2 3 4 5 6)`. You can probably write it faster using `build-list` that you can retrieve it. Similarly, write a function `down: nat -> (list-of nat)` that constructs a list of naturals ending in `0` in descending order. For example, `(nums-down 7)` returns `(list 6 5 4 3 2 1 0)`.
2. Now write a function

   ```
   rands
   ```

   that constructs a list of random numbers in the range from `0` to `32767`. To create a single random number in the range from `0` to *`n`*, compute `(random` *`n+1`*`)`. For larger numbers in the range `[0, 32768)`, try:

   ```
   (define max-rand 32768)
   (random max-rand)         ; random is built-in
   ```

Note: `random` is a function in the Scheme library, but it is not a mathematical function, since the output is not determined by the input.

3. Make sure you know how to use `time` by using it to time `isort` on a list of 200 elements. Run the exact same expression again a couple times. Note that the time varies some. We typically deal with such variation by taking the average of several runs. The variation is caused by a number of low-level hardware phenomena that are beyond the scope of this course (see ELEC 220 and COMP 221).
4. Use the `time` operation to fill in the following chart.

		input size	input size	input size
		200	1000	5000
	up			
isort	down			
	rand			
	up			
qsort	down			
	rand			

When you are done, compare your results with those generated by other pairs in the lab.

COMP 280 introduces concepts of how these algorithms behave in general.
The punchline is that we can say that both insertion sort and quicksort on lists are *O(n*2)* **in the worst case. *i.e.*, for a list of *n* elements, they each take about *c*** *n**2* evaluation steps to sort the list, for some constant *c*. Furthermore, in the "average" case, Quicksort does better: *O(n log n)*. A well-coded formulation of quicksort using arrays almost never exhibits worst-case behavior. COMP 280, and later COMP 482, show precisely what big *O* notation means these statements mean and how to derive them.

Ackermann's Function Example

If you have time, here is the definition of a strange numerical function on natural numbers,
called Ackermann's function:

```
A(m,n) = n+1                if m=0
       = A(m-1,1)           if m>0 and n=0
       = A(m-1,A(m,n-1))    if m>1 and n>0
```

Note that this definition is not structurally recursive. In fact, it *cannot be defined* in a structurally recursive manner. In technical jargon, the function is not *pri mitive recursive*. See COMP 481 for what that means.

Here's the equivalent Scheme code (assuming `m` nad `n` are natural numbers:

```
(define (ack m n)
  (cond [(= m 0) (add1 n)]
        [(= n 0) (ack (sub1 m) 1)]
        [else (ack (sub1 m) (ack m (sub1 n)))]))
```

Ackermann's function exercises

1. Try `ack` on some small inputs. Warning: try *very, very small* inputs for `m`, like 0, 1, 2, or 3, because `(ack 4 1)` takes a very, very long time to compute. You can compute `(ack 4 1)` using DrScheme if you infer a simple non-recursive formula for `(ack 3 n)` and add a clause containing this short-cut for computing `(ack 3 n)` to the Scheme code for `ack`. Do not try to compute `(ack 4 2)`. The answer is more than 19,000 digits long.
2. Prove why `ack` always terminates for natural numbers. Hint: you need to use "double-induction".

Ackermann's function is not useful in constructing real software applications, but it is an important mathematical defintion because it is provably not primitive recursive and it plays an important role in the complexity analysis of some algorithms (notably *union-find*). In COMP 482 you will learn about *union -find* and its complexity analysis.

Notes on Tail Recursion

When a program makes a function call, the code has to remember the location from where the call was made in order to return to it after the function finishes its work. The calling program also needs to know where to retrieve the result of the computation performed by the function. Behind the scene, the compiler sets up a stack of "call frames." Each call frame keeps track of the return address, the arguments for the function call and the result of the function's computation. Creating a call frame, pushing it on the call stack and popping it off the call stack are relatively costly operations in execution time. In addition, each call frame takes space in the portion of memory allocated for program control.

When a recursive function creates a deep call stack, it is costly both in time and space. Really deep recursion can exhaust the space available for program control (which is usually a fixed partition of memory). Fortunately, there is a special form of recursion called tail recursion that can be converted into an iterative loop by smart compilers, eliminating the creation of multiples call frames and thus greatly improving performance. A recursive function is said to be tail recursive if all of the recursive calls in the function body occur in tail positions. A position in the function body is a tail position if it is the last operation performed in any execution of the body that reaches it. In other words, after returning from a recursive call in tail position there is no pending operation waiting to be performed.

Consider the following function that counts the number of elements in a list.

```
;; how-many1: list of X -> number
;; (how-many lox) returns the number of elements in lox.
;; examples
(check-expect (how-many1 empty) 0)
(check-expect (how-many1 '(a)) 1)
(check-expect (how-many1 '(a b c)) 3)
;; code
(define (how-many1 lox)
  (cond
    [(empty? lox) 0]
    [(cons? lox) (+ (how-many1 (rest lox)) 1)]))
```

The function how-many1 is not tail recursive. Why?

After the recursive call (how-many1 (rest lox)), the calling function still needs to perform an addition before returning the result.

Now consider the following function.

```
;; how-many-acc: list-of-X number -> number
;; (how-many-acc lox a) returns a if lox is empty otherwise it returns
;; a + the number of elements in (rest lox).
;; examples
(check-expect (how-many-acc empty 5) 5)
(check-expect (how-many-acc '(a) 0) 1)
(check-expect (how-many-acc '(a b c) 0) 3)
;; code
(define (how-many-acc lox acc)
  (cond
    [(empty? lox) acc]
    [(cons? lox) (how-many-acc (rest lox) (+ acc 1))]))
```

The function how-many-acc is tail recursive. Why?

The function how-many-acc returns immediately to the caller after making its recursive call (how-many-acc (rest lox) (+ acc 1)), without having to perform any additional computation. The function how-many-acc updates its argument acc (by adding 1 to it) efore passing it on to the recursive call. By the time the recursive call reaches the end of the list, marked by empty, it simply returns the argument acc. The function how-many-acc uses its acc parameter to "accumulate" the result of its computation. As such the parameter acc is called an accumulator.

We now can make use of how-many-acc to write a highly efficient function to compute the number of elements in a list.

```
;; how-many2: list-of-X -> number
;; (how-many2 lox) returns the number of elements in lox.
;; examples
(check-expect (how-many2 empty) 0)
(check-expect (how-many2 '(a)) 1)
(check-expect (how-many2 '(a b c)) 3)
;; code
(define (how-many2 lox)
  (how-many-acc lox 0))
```

Note that how-many2 is not tail recursive. It simply calls a (helper) recursive function to do the job.

Aside: Many proofs of theorems in Mathematics are carried out by first proving a series of lemmas, which are put together to prove the theorem. Think of the helper functions as "lemmas" for the calling function, the "theorem."

Exercises

1. Write a function called prod-nums that takes a list of numbers and returns the product of the number in the list, using a tail recursive helper function.

What should the function return on an empty list?

2. Revise the preceding definition to terminate immediately if it encounters a 0 in the list of numbers being multiplied.

3. Write a function called last-elt that takes in a list of X and returns the last element in the list, using a tail recursive helper function. For an empty list, call (error 'last-elt "applied to empty list").

4. Write a function called find-min that takes a list of numbers and returns the smallest number in the list, using a tail recursive helper. For an empty list, call (error 'last-elt "applied to empty list").

5. Write a function called make-palindrome that takes a list and returns a list consisting of the input list and its mirror around the last element, using a (non tail-recursive) helper with an accumulator. For example, (make-palindrome '(a b c)) returns '(a b c b a).