211lab04_S11

Lambda Functions

Accumulators

======= Due to the unexpected closure of Rice and the subsequent disruption of our lecture schedule the following topics will be covered in a future lab ==========

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:

- `(listify empty) = empty`.
- `(listify (list (make-open) (make-close)) = (list empty)`.
- `(listify (list (make-open) 'a (make-close)) = (list '(a))`
- `(listify (list (make-open) 'a 'b (make-close)) = (list '(a b))`
- `(listify (list (make-open) (make-open) 'a (make-close) 'b (make-close)) = (list '((a) b))`

You will probably want to define the following predicate

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

in your program. As a rough guideline follow the parse.ss\ program that we wrote in class on Monday, Feb 2. 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 *primitive 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.