HW 5
50pts
Posted Tuesday, October 28
Due Monday, November 3 at midnight

Longest Non-Decreasing Subsequence

In this homework we take a break from parsers and interpreters to write two Scheme functions that compute a longest non-decreasing subsequence from a list of numbers. E.g.,

```
> (lis-fast '(1 2 3 2 4 1 2))
(1 2 3 4)
```

Note that there might be more than one longest subsequences for a list of numbers. For example, `(1 2 2 4)` and `(1 2 2 2)` are longest non-decreasing subsequences as well.

Part 1: `lis-slow`. In this part, write an inefficient solution. First, write Scheme function `sublists` that computes all i-element subslists of the input list. For example:

```
> (sublists 2 '(1 2 3 4))
((1 2) (1 3) (1 4) (2 3) (2 4) (3 4))
```

Next, write Scheme function `lis-slow` which takes a list of numbers as argument and produces a longest non-decreasing subsequence as a result. It should implement, in Core Scheme (see note at the end of the homework), the following pseudocode:

```
for i := n downto 1, where n is the length of the input list
    for each i-element sublist s of the input list
        if s is a non-decreasing sequence of numbers
            print s and quit
```

You can create a few sample arguments to test your functions. For example, you can save

```
(define list1 '(1 2 3 2 4 1 2))
(define list2 '(2 4 3 1 2 1))
```

and test your functions on those lists more conveniently:

```
> (sublists 5 list2)
((2 4 3 1 2) (2 4 3 1 1) (2 4 3 2 1) (2 4 1 2 1) (2 3 1 2 1) (4 3 1 2 1)
   )
> (lis-slow list1)
(1 2 3 4)
> (lis-slow list2)
(2 4)
>
```

For `lis-slow` you do not need to worry about the case of more than one longest subsequences. As long as the function produces *a* longest non-decreasing subsequence, it is correct.

Part 2: `lis-fast`**.** Now write an efficient, polynomial-time function, `lis-fast`. Feel free to research efficient algorithms, but as a quick reference, here is a standard one. Traverse the list from left to right and for each element save a tuple `(max-length prev-index)`. `max-length` stores the length of the longest subsequence ending with the element at index i. `prev-index` stores the index of the previous element in the longest subsequence ending with the element at i. One invariant is that if (m j) is the tuple at i and (n k) is the tuple at j, then m is n + 1.

Assuming 0-based indices, the following listing shows `list2` and the tuples for `list2`:

```
(   2       4      3      1       2      1   )
((1  -1)  (2  0)  (2  0)  (1  -1)  (2  0)  (2  3))
```

As an example, at index 2 (the number 3), we have tuple (2 0) meaning that the longest subsequence that ends with element 3 is of length 2 and the previous index is 0.

Once we have computed the tuples, the next step is to reconstruct a longest subsequence by first finding the max length in the list of tuples, then reconstructing the full sequence by following the indices backwards. In the above example the max length is 2, and the corresponding sequence is `(2 4)`. **For** `lis-fast` **we expect the "leftmost" longest subsequence as output.**

`lis-fast` is just like `lis-slow` but it should handle much larger lists. An example call is:

```
(lis-fast (append list1 list2))
(1 2 2 2 2 4)
```

Please read this carefully before starting to code:

- Submit your code in file `lis.rkt` in Submitty. We run a command-line R5RS interpreter and we are testing your `sublists`, `lis-slow` and `lis-fast`.
- You can create helper functions in addition to `sublists, lis-slow` and `lis-fast`. The body of each function should be an expression in the purely functional subset of Scheme we covered in class. I call it Core Scheme:

E	$\to$	`const`	constant: list, integer, boolean, etc.
E	$\to$	`id`	identifier expression
E	$\to$	`(if` E E E`)`	if expression
E	$\to$	`(cond (`E E`)` ... `(else` E`))`	cond expression
E	$\to$	`(`E E ... `)`	function call expression
E	$\to$	`(let ((id` E`)` ... `)` E`)`	let or let* expression

- The only built-in Scheme functions and forms you are allowed to use at calls are the ones from class: equality testing (`eq?, equal?`), type predicates (e.g., `null?, pair?, symbol?, integer?, ...`), list operations (`cons, list, append, reverse, length, car, cdr, caddr, ...`), boolean operations (`and, or, not`), arithmetic (e.g., `+, *, ...`) and comparison (e.g., `<, <=, ...`). Importantly, you are *not* allowed to use `set!` or other imperative features and functions, and you are *not* allowed to use `list-ref`, you should code your own version of it.
- Comment your code using the contract style outlined in `Contracts.htm`. 40 points are autograded in Submitty and 10 points are awarded for code quality are comments.