CS403: Programming Languages

Notes on Stacks

It is very easy to implement the classic data structures in Scheme. Stacks are especially easy. Let's define a *stack* constructor:

```
(define (Stack)
    (define store nil)

    (define (dispatch msg)
        (cond
            ((eq? msg 'push) push)
            ((eq? msg 'pop) pop)
            ((eq? msg 'empty?) empty?)
            (else (error "stack message not understood: " msg))
            )
        )
    (define (push x)
        (set! store (cons x store))
        )
    (define (pop) ; user is responsible ensuring stack is non empty
        (define tmp (car store))
        (set! store (cdr store))
        tmp
        )
    (define (empty?)
        (eq? store nil)
        )

    dispatch
    )
```

Note that since we add to the front of the store and remove from the front, we get LIFO behavior. Also note how clean the implementation is and that it is perfectly generic. We can push any kind of element onto the stack and even mix kinds of elements. Here is a sample interaction with a *stack* object:

```
(define s (Stack))
((s 'push) 3)
((s 'push) 5)
((s 'push) 4)
((s 'pop))              ; should yield 4
((s 'empty))            ; should return #f
```

Using Scam's object system, a *stack* class becomes even simpler:

```
(define (Stack)
    (define store nil)

    (define (push x)
        (set! store (cons x store))
        )
    (define (pop) ; user is responsible ensuring stack is non empty
        (define tmp (car store))
        (set! store (cdr store))
        tmp
        )
    (define (empty?)
        (eq? store nil)
        )

    this
    )
```

Scam obviates the need for a dispatch function; the interaction with this stack is exactly the same as with the Scheme dispatch version.