5. Consider a Racket procedure \texttt{(make-stack name)} that takes a symbol name and returns a Racket procedure that implements a stack object with local storage including a list representing a push down stack - a last-in, first-out (LIFO) data structure, which can process the following commands:

- **name** -- returns the name of the stack.
- **size** -- returns the number of elements in the stack.
- **empty?** -- return \texttt{#t} if the stack is empty and \texttt{#f} otherwise.
- **push element** - adds the given element to the top of the stack. Returns an error if element is absent.
- **pop** -- returns the most recent element pushed on the stack. Removes that element from the stack. Returns an error if the stack is empty.

Examples of using \texttt{(make-stack name)}:

```racket
> (define s1 (make-stack 'stack1))
> (s1 'empty?)
#t
> (s1 'size)
0
> (s1 'push 5)
5
> (s1 'empty?)
#f
> (s1 'push 7)
7
> (s1 'push 9)
9
> (s1 'size)
3
> (s1 'pop)
9
> (s1 'size)
2
> (define s2 (make-stack 'stack2))
> (s1 'name)
'stack1
> (s2 'name)
'stack2
> (s2 'size)
0
> (s2 'pop)
'Error:stack-empty
> (s2 'push)
'Error:usage:push_element
```
5. (a) (10 points) 
Write a Racket procedure to implement (make-stack name).

```racket
(define (make-stack name)
  (let ((stack '())
         (size 0))
    (lambda (cmd . args)
      (case cmd
        ((name) name)
        ((empty?)
         (null? stack))
        ((push)
         (if (null? args)
             'Error: usage: push_element
             (begin
               (set! stack (cons (first args) stack))
               (set! size (+ size 1))
               (first args))))
        ((size) size)
        ((pop)
         (if (null? stack)
             'Error: stack-empty
             (let ((result (car stack)))
               (set! stack (cdr stack))
               (set! size (- size 1))
               (result))))
      )))
)
```

5. (b) (2 points) What is the run time complexity of the following stack operations, using big-O notation?

- empty?
- size
- pop
- push