In this assignment we will explore a few of the “object patterns” from PLAI, again using the Stacker.

The programs will be rewritten to conform to the SMoL (which is intentionally low on features, since it represents a common linguistic abstraction). For instance, we replace symbols with strings, `case` with `cond`. We will also get rid of some of the methods to keep the code simpler.

Basic Objects

This program is a simple version of the basic object pattern:

```
(deffun (o-state-2 init)
  (lambda (m)
    (cond
      [(eq? m "inc")
       (lambda () (set! init (+ init 1)))]
      [(eq? m "get")
       (lambda () init)]
      [else
       (error "undefined method")])))

(defvar o (o-state-2 5))
((o "inc"))
((o "get"))
```

Run this program through to completion.

1. Provide a screenshot of your final configuration.
2. You should see four closures. Using the addresses in the above configuration:
    a. Which of these correspond to *objects*, and why?
    b. Which of these correspond to *classes*, and why?

Static Members

This program represents the `static` pattern, where `counter` is the static member:

```
(defvar o-static-1
  (let ([counter 0])
    (lambda (amount)
      (set! counter (+ 1 counter))
      (lambda (m)
        (cond
          [(eq? m "inc")
           (lambda () (set! amount (+ amount 1)))]
          [(eq? m "count")
           counter]
          [else
           (error "undefined member")])))))

(defvar o-1 (o-static-1 5))
(defvar o-2 (o-static-1 7))
((o-2 "inc"))
(o-2 "count")
```

Run this program through to completion.

1. Provide a screenshot of your final configuration.
2. How many objects are created? Where are they located in your configuration (using the addresses in the above screenshot)?
3. Trace through the configuration to show why `counter` behaves like a `static`.

Dynamic Dispatch

The following program represents the “dynamic dispatch” pattern.

```
(defvar mt
  (let ([self "dummy"])
    (set! self
          (lambda (m)
            (cond
              [(eq? m "sum")
               (lambda () 0)]
              [else
               (error "mt")]))))
    self))

(deffun (node v l r)
  (let ([self "dummy"])
    (set! self
          (lambda (m)
            (cond
              [(equal? m "sum")
               (lambda () (+ v
                             ((l "sum"))
                             ((r "sum"))))]
              [else
               (error "node")]))))
    self))

(defvar a-tree
  (node 10
        (node 5 mt mt)
        mt))

((a-tree "sum"))
```

We have simplified the program slightly: `mt` is a singleton. Because it has no parameters, we have also gotten rid of the constructor and focused on the core object representation.

1. Run this program and provide a screenshot of the configuration at the point where the `a-tree` object has been initialized and before the method dispatch begins.
2. In the above configuration, point out which values represent `mt` and `node` *objects*. Do any values here correspond to *classes*? If so, which one(s) and why?
3. Dynamic dispatch depends on objects having a consistent representation (this is the heart of "object polymorphism": you can treat them interchangeably, and the differences in behavior are hidden inside the objects). In what way do these objects have a consistent representation?