Introduction

In this problem set we'll use closures and state to implement a limited form of object-oriented programming. For our purposes, here are the relevant properties of an object:

- It is a value that contains state (i.e. fields or properties) and procedures (methods) that operates on this state.
- The methods can access the properties, but nothing else can.
- The object's users can only interact with it through specifically exposed methods.

We won't worry about inheritance.

A simplistic example of almost-objects

Here is something that looks almost like an object, and demonstrates the technique of closures and state: a simple increasing counter.

```
(define (make-counter)
  (let ([counter 0])
    (lambda ()
      (let ([result counter])
        (set! counter (+ counter 1))
        result))))

> (define a (make-counter))
> (a)
0
> (a)
 1
> (a)
 2
> (define b (make-counter))
> (b)
0
```

`make-counter` first defines a variable `counter`, set to 0. Then it defines an unnamed procedure (the `lambda` expression) which actually does the counting, by operating on the variable. Then it returns this procedure.

After `make-counter` returns this procedure, nothing else has access to the `counter` variable. We can only access this piece of private state using the procedure returned to us. And each invocation of `make-counter` produces a fresh counter procedure with its own hidden state. Note how `b` and `a` operate on totally different `counter` variables. And note how changes to `a`'s counter variable persist between invocations of `a`.

Closer to the real thing

Above, we saw how to make a procedure that operates on its own hidden state. But an object has *several* procedures, all operating on *the same* hidden state. Below is a more complete implementation:

```
(define (make-counter-object initial-count step-size)
  (let ([counter initial-count])
    (define (count-method)
      (let ([result counter])
        (set! counter (+ counter step-size))
        result))
    (define (set-method x)
      (set! counter x))
    (lambda (msg)
      (cond [(eq? msg 'count) count-method]
            [(eq? msg 'set) set-method]))))
```

This is more complicated along two axes. First, `make-counter-object`, which can be viewed as a constructor, now takes two arguments: an initial count, and the step size between counts. Second, instead of just defining a single hidden variable, and returning a procedure which operates on it, the constructor defines a hidden variable *and two procedures*: `count-method` and `set-method`. What is actually returned is a method dispatching procedure. The method dispatcher simply returns one of the methods based on its argument, `msg`.

To actually call methods, we have a helper procedure:

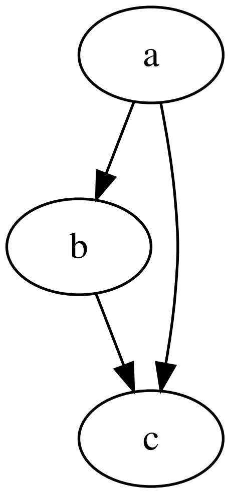
```
(define (send-message obj msg . msg-args)
  (let ([method (obj msg)])
    (apply method msg-args)))
```

This passes message to an object, and calls the returned method. Now we can do this:

```
> (define a (make-counter-object 0 1))
> (send-message a 'count)
0
> (send-message a 'count)
1
> (send-message a 'count)
2
> (send-message a 'set 5)
> (send-message a 'count)
5
> (send-message a 'count)
6
```

Directed Graphs and Graph Traversal

Directed graphs are a fundamental data structure in Computer Science. A directed graph is a set of objects, called *nodes*, which carry references to other nodes. These references are called *edges*. We draw graphs using circles and arrows:



In the above picture, node `a` has edges to `b` and `c`, and node `b` has an edge to `c`. If a node `n` has an edge to a node `v`, we will say `v` is a *child* of `n`. In the above example `b` and `c` are children of `a`, and `c` is a child of `b`.

Also, for our purposes, the children of a node have a particular order. There is a first child, a second child, etc. Each node has a *list* of children.

Some examples of graphs are:

- The Web. Every page is a node, and every link is an edge.
- All the memory in a computer program. Every object is a node, and every reference is an edge.
- Every tree.
- And, in fact, every list.

Graph traversal algorithms are ways of systematically visiting every node in a graph. "Visiting" can mean different things–a web crawler is a kind of graph traversal, and it "visits" web pages by downloading them, for example. We will just return a list of nodes, in the order we visited them.

There are two basic graph traversal algorithms: depth-first search and breadth-first search. Both of these procedures take a single starting point as an argument, and eventually visit all nodes that are reachable by following edges from the starting node.

Here is some pseudocode for breadth-first search:

```
function BFS(start-node):
    Let Q be an empty FIFO queue;
    enqueue(x,Q);
    while Q is not empty do
        let x = dequeue(Q);
        if x is not marked as visited then
            mark(x);
            forall c in children(x) do
                enqueue(c,Q)
            end
        end
    end
```

and for depth first search:

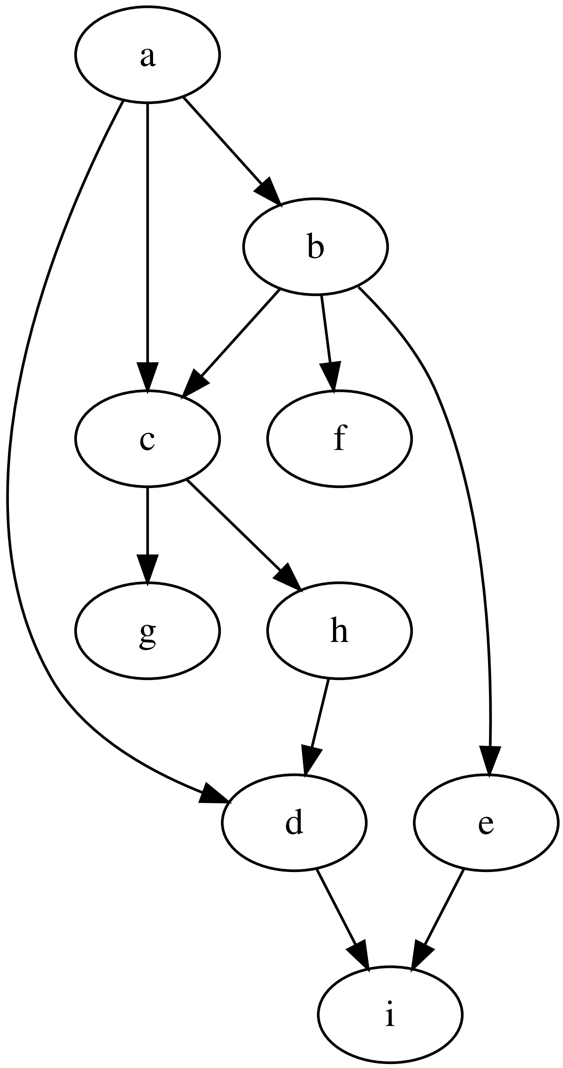
```
function DFS(start-node):
    MARK(start-node);
    forall c in CHILDREN(start-node) do
        DFS(c);
    end
```

The above DFS code looks very different from the BFS code, but below is another version of DFS that looks almost identical. The only difference is you use a LIFO queue instead of a FIFO queue, and enqueue the children in reverse order. “LIFO queue” is just a silly way of saying “stack.”

```
function DFS(start-node):
    Let Q be an empty LIFO queue.;
    enqueue(x,Q);
    while Q is not empty do
        let x = dequeue(Q);
        if x is not marked then
            mark(x);
            forall c in children-in-reverse-order(x) do
                enqeue(c,Q);
            end
        end
    end
```

Example

Consider the following graph. Assume that each node's children are in alphabetical order:



Below is a table of the order in which DFS and BFS traverse the nodes, as well as what happens if you forget to reverse the children for DFS:

DFS	BFS	LIFO without reversed children
A	A	A
B	B	C
C	C	H
G	D	D
H	E	I
D	F	G
I	G	B
F	H	F
E	I	E

Assignment

You are given an implementation of a node class in the procedure `make-node`, and an implementation of the search loop described above in `graph-search`. You are also given a procedure `build-graph` which takes an adjacency list description of a graph as an input, and creates node objects with the relevant structure. To understand the adjacency list, read the comments, and look at the examples in `example-graphs.rkt` and the provided images. The `graph-search` procedure takes a `container-constructor` as an argument, and `build-graph` takes a `node-constructor` as an argument.

You are also given `search-wrapper`, which takes an adjacency list, builds a node objects for all its entries, and does a graph search from the first node. And finally, `dfs` and `bfs` procedures, defined in terms of the constructors you will implement.

LIFO queue

Implement a LIFO queue (that is, a stack) object, with the following methods:

- `enqueue!`
- `dequeue!`
- `empty?`

Remember, a list makes a nice stack.

FIFO queue

Implement a FIFO queue, called `make-fifo` with the same methods as above. Hint: you can implement a FIFO with two stacks.

DFS node wrapper

Implement a `make-dfs-node` constructor which wraps a regular `node`, but which gives the result of `'get-children` in reverse order.