For now assume all values are scalars.

We will “backpopagate” the assignments the reverse order.

red $\partial \ell / \partial u = 1$

$\partial \ell / \partial u = 1$

red $\partial \ell / \partial z = (\partial \ell / \partial u)\ (\partial h / \partial z)$ (this uses the value of z)

$\partial \ell / \partial u = 1$

$\partial \ell / \partial z = (\partial \ell / \partial u)\ (\partial h / \partial z)$

red $\partial \ell / \partial y = (\partial \ell / \partial z)\ (\partial g / \partial y)$ (this uses the value of y and x)

$\partial \ell / \partial u = 1$

$\partial \ell / \partial z = (\partial \ell / \partial u)\ (\partial h / \partial z)$

$\partial \ell / \partial y = (\partial \ell / \partial z)\ (\partial g / \partial y)$

red $\partial \ell / \partial x =$??? Oops, we need to add up multiple occurrences.

We let red x.grad be an attribute (as in Python) of node red x.

We will initialize red x.grad to zero.

During backpropagation we will accumulate contributions to red $\partial \ell / \partial x$ into red x.grad.

$z.\text{grad} = y.\text{grad} = x.\text{grad} = 0$

$u.\text{grad} = 1$

Loop Invariant: For any variable w whose definition has not yet been processed we have that $w.\text{grad}$ is $\partial \ell / \partial w$ as de

$z.\text{grad} = y.\text{grad} = x.\text{grad} = 0$

$u.\text{grad} = 1$

Loop Invariant: For any variable w whose definition has not yet been processed we have that $w.\text{grad}$ is $\partial \ell / \partial w$ as de

$z.\text{grad} \mathrel{+}= u.\text{grad} * \partial h / \partial z$

$z.\text{grad} = y.\text{grad} = x.\text{grad} = 0$

$u.\text{grad} = 1$

Loop Invariant: For any variable w whose definition has not yet been processed we have that $w.\text{grad}$ is $\partial\ell/\partial w$ as d

$z.\text{grad} \mathrel{+}= u.\text{grad} * \partial h/\partial z$

$y.\text{grad} \mathrel{+}= z.\text{grad} * \partial g/\partial y$

$x.\text{grad} \mathrel{+}= z.\text{grad} * \partial g/\partial x$

$z.\text{grad} = y.\text{grad} = x.\text{grad} = 0$

$u.\text{grad} = 1$

$z.\text{grad} \mathrel{+}= u.\text{grad} * \partial h/\partial z$

$y.\text{grad} \mathrel{+}= z.\text{grad} * \partial g/\partial y$

$x.\text{grad} \mathrel{+}= z.\text{grad} * \partial g/\partial x$

$x.\text{grad} \mathrel{+}= y.\text{grad} * \partial f/\partial x$

```
class F(CompNode):

    def __init__(self, x):
        CompNodes.append(self)
        self.x = x

    def forward(self):
        self.value = f(self.x.value)

    def backward(self):
        self.x.addgrad(self.grad * ∇_x f(x))        #needs x.value
```

The following Python code constructs the computation graph of a multi-layer perceptron (NLP) with one hidden la

```
class Affine(CompNode):

    def __init__(self,Phi,x):
        CompNodes.append(self)
        self.x = x
        self.Phi = Phi

    def forward(self):
        self.value = (np.matmul(self.x.value,
                                self.Phi.w.value)
                      + self.Phi.b.value)
```

```
def backward(self):

    self.x.addgrad(
       np.matmul(self.grad,
                 self.Phi.w.value.transpose()))

    self.Phi.b.addgrad(self.grad)

    self.Phi.w.addgrad(self.x.value[:,:,np.newaxis]
                       * self.grad[:,np.newaxis,:])
```

```
class Parameter:

    def __init__(self,value):
        Parameters.append(self)
        self.value = value

    def addgrad(self, delta):
        #sums over the minibatch
        self.grad += np.sum(delta, axis = 0)

    def SGD(self):
        self.value -= learning_rate*self.grad
```

```
y = Affine([W,B],x)

 forward:
  y.value[b,j] = x.value[b,i]W.value[i,j]
  y.value[b,j] += B.value[j]

 backward:
    x.grad[b,i] += y.grad[b,j]W.value[i,j]
    W.grad[i,j] += y.grad[b,j]x.value[i]
    B.grad[j] += y.grad[b,j]
```

```
class Affine(CompNode):

    def __init__(self,Phi,x):
        CompNodes.append(self)
        self.x = x
        self.Phi = Phi

    def forward(self):
        # y.value[b,j] = x.value[b,i]W.value[i,j]
        # y.value[b,j] += B.value[j]
        self.value = (np.matmul(self.x.value,
                                self.Phi.W.value)
                      + self.Phi.B.value)
```

```
def backward(self):

    self.x.addgrad(
       # x.grad[b,i] += y.grad[b,j]W.value[i,j]
       np.matmul(self.grad,
                 self.Phi.W.value.transpose()))

    # B.grad[j] += y.grad[b,j]
    self.Phi.B.addgrad(self.grad)

    # W.grad[i,j] += y.grad[b,j]x.value[b,i]
    self.Phi.W.addgrad(self.x.value[:,:,np.newaxis]
                       * self.grad[:,np.newaxis,:])
```

```
class Affine(CompNode):
    ...
    def backward(self):
        ...
        # W.grad[i,j] += y.grad[b,j]x.value[b,i]
        self.Phi.W.addgrad(self.grad[:,np.newaxis,:]
                           *self.x.value[:,:,np.newaxis])

class Parameter:
    ...
    def addgrad(self, delta):
        self.grad += np.sum(delta, axis = 0)
```