Implementing autograd

Slides by Matthew Johnson

Autograd's implementation

github.com/hips/autograd

Dougal Maclaurin, David Duvenaud, Matt Johnson

- differentiates native Python code
- handles most of Numpy + Scipy
- loops, branching, recursion, closures
- arrays, tuples, lists, dicts...
- derivatives of derivatives
- a one-function API!

autodiff implementation options

A. direct specification of computation graph
B. source code inspection
C. **monitoring function execution**

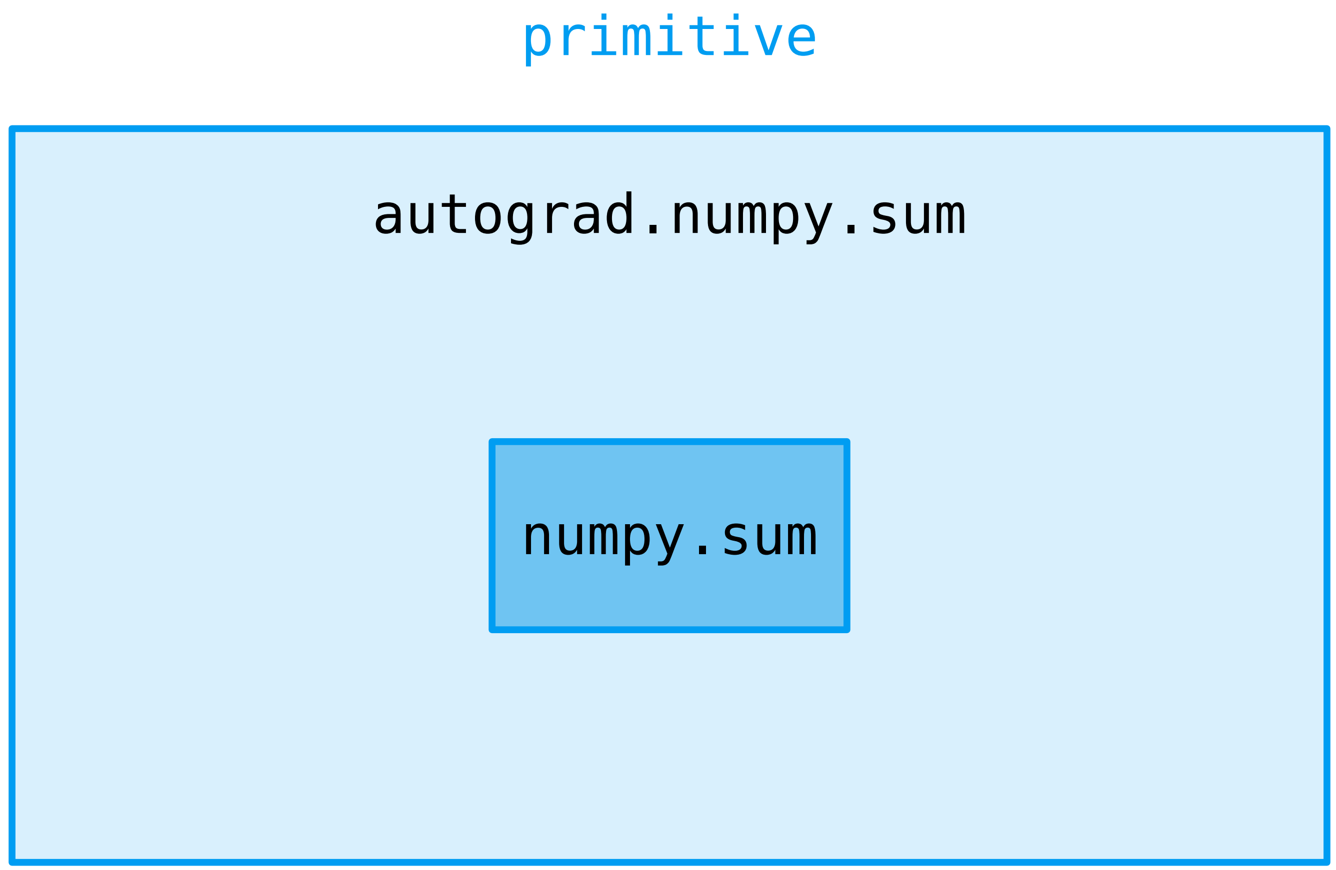
ingredients:

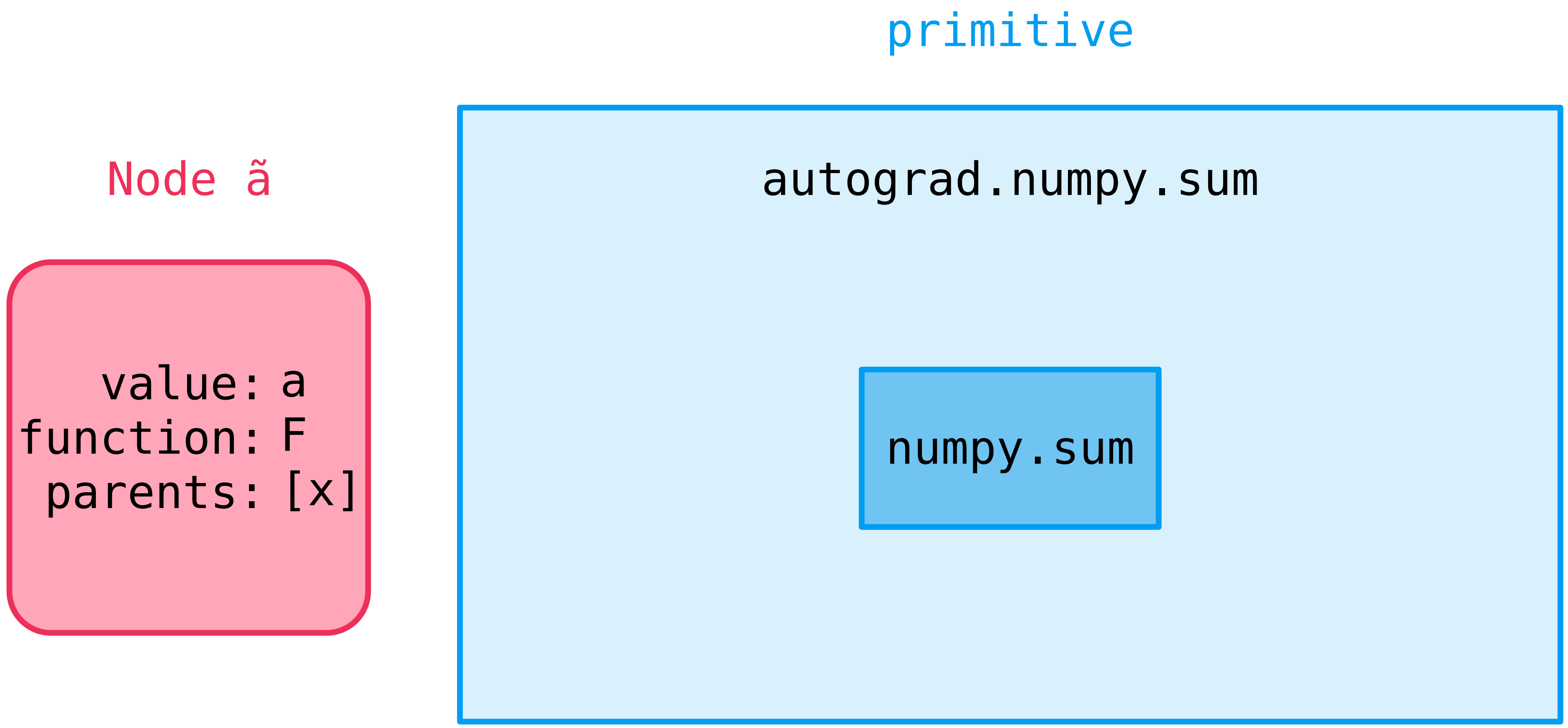
1. tracing composition of primitive functions
2. vector-Jacobian product for each primitive
3. composing VJPs backward

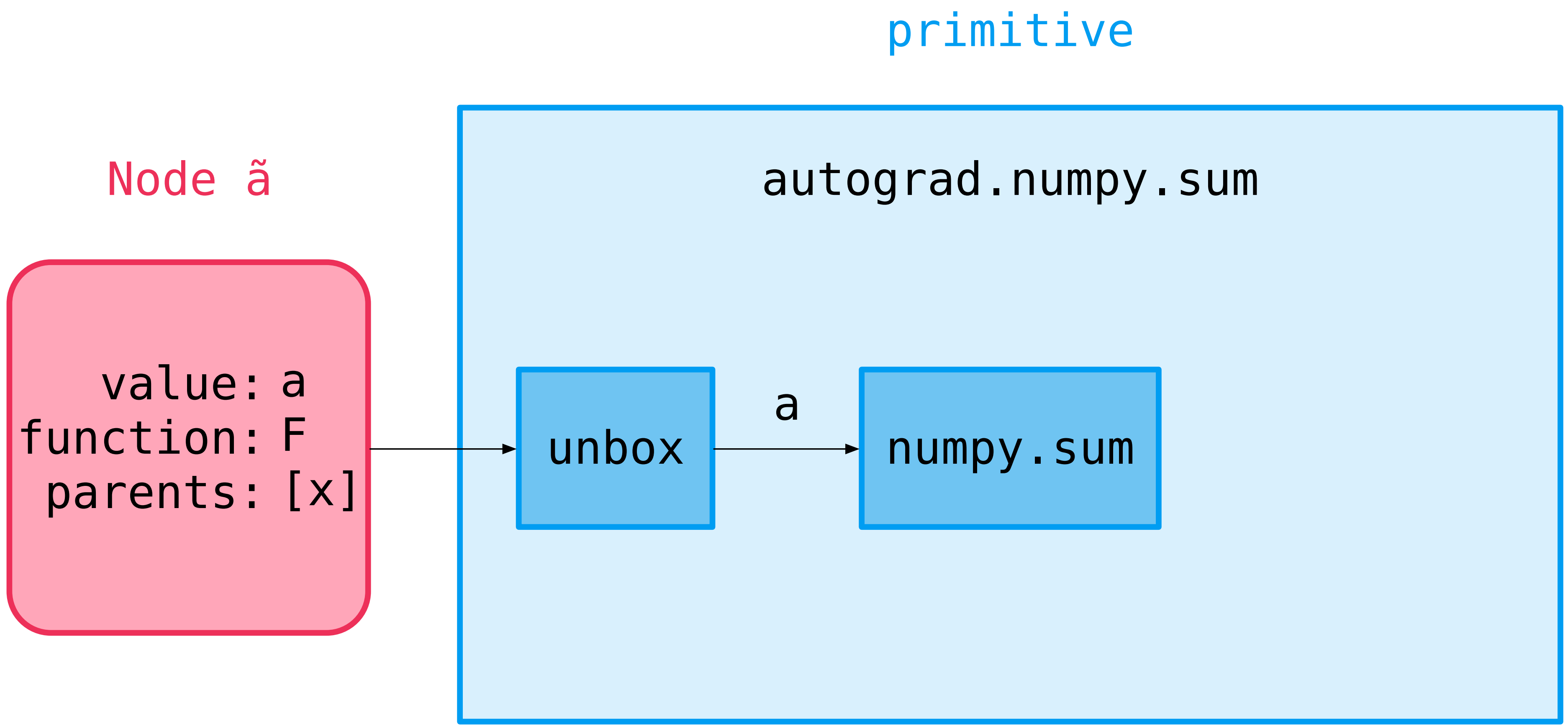
ingredients:

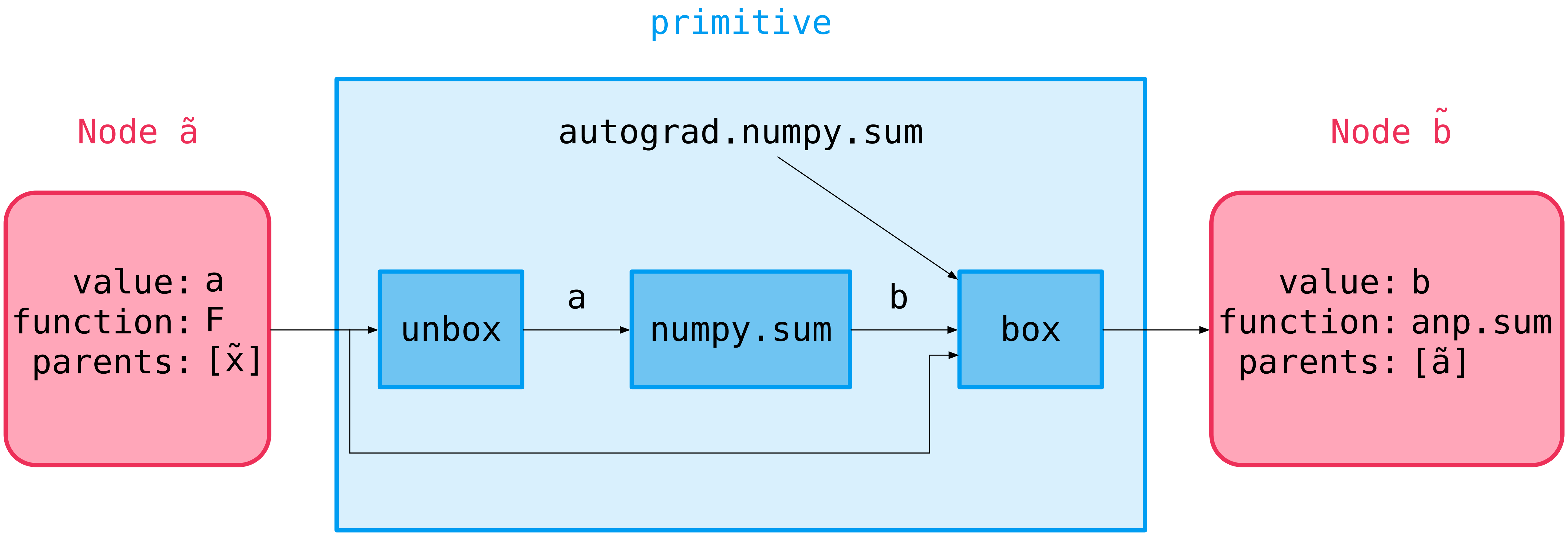
1. tracing composition of primitive functions
2. vector-Jacobian product for each primitive
3. composing VJPs backward

`numpy.sum`


primitive
autograd.numpy.sum
numpy.sum


primitive
Node ã
value: a
function: F
parents: [x]
autograd.numpy.sum
numpy.sum

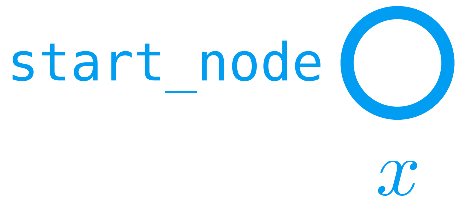

primitive
Node ã
value: a
function: F
parents: [x]
autograd.numpy.sum
unbox
a
numpy.sum

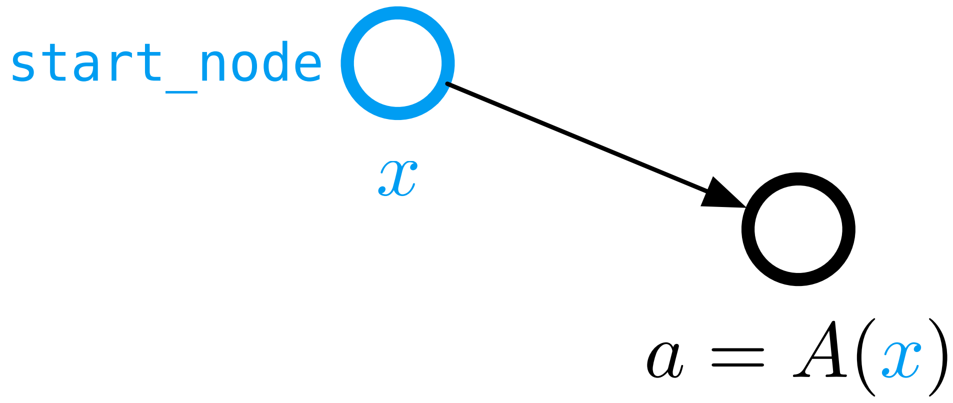

primitive
Node ã
autograd.numpy.sum
Node b̃
value: a
function: F
parents: [x̃]
unbox
a
numpy.sum
b
box
value: b
function: anp.sum
parents: [ã]

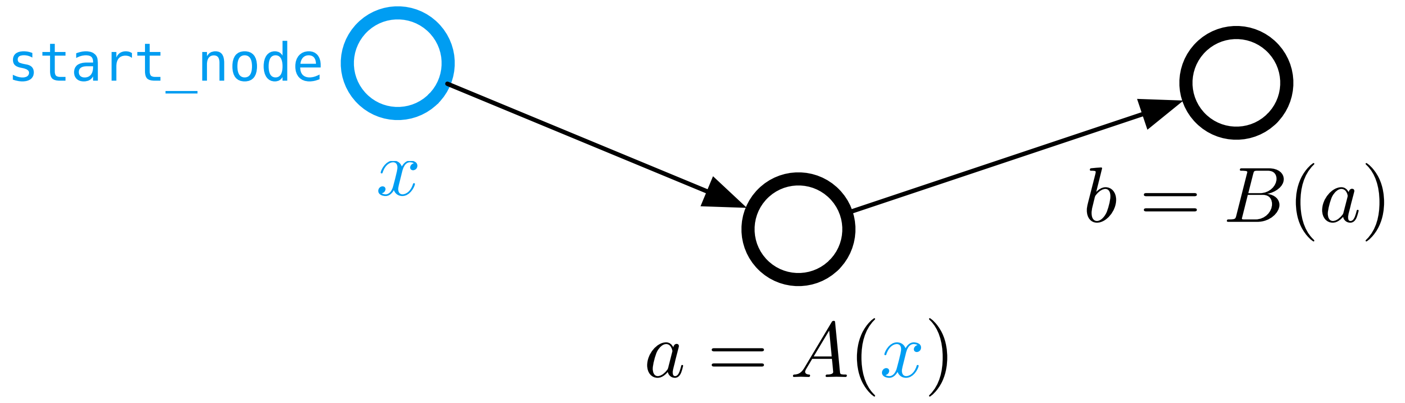
```python
class Node(object):
    __slots__ = ['value', 'recipe', 'progenitors', 'vspace']

    def __init__(self, value, recipe, progenitors):
        self.value = value
        self.recipe = recipe
        self.progenitors = progenitors
        self.vspace = vspace(value)
```

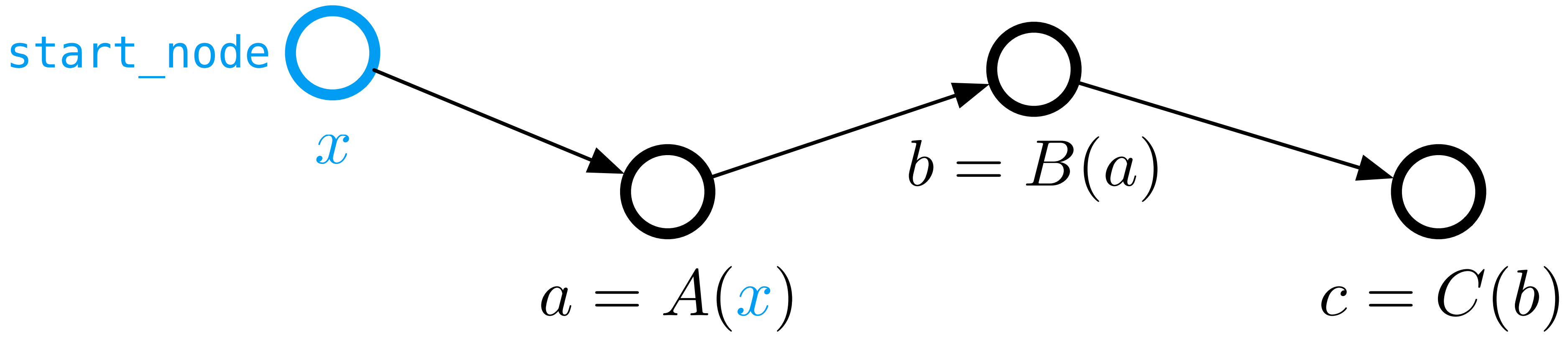
```python
class primitive(object):
    def __call__(self, *args, **kwargs):
        argvals = list(args)

        parents = []
        for argnum, arg in enumerate(args):
            if isnode(arg):
                argvals[argnum] = arg.value
                if argnum in self.zero_vjps: continue
                parents.append((argnum, arg))


        result_value = self.fun(*argvals, **kwargs)
        return new_node(result_value, (self, args, kwargs, parents),          )
```

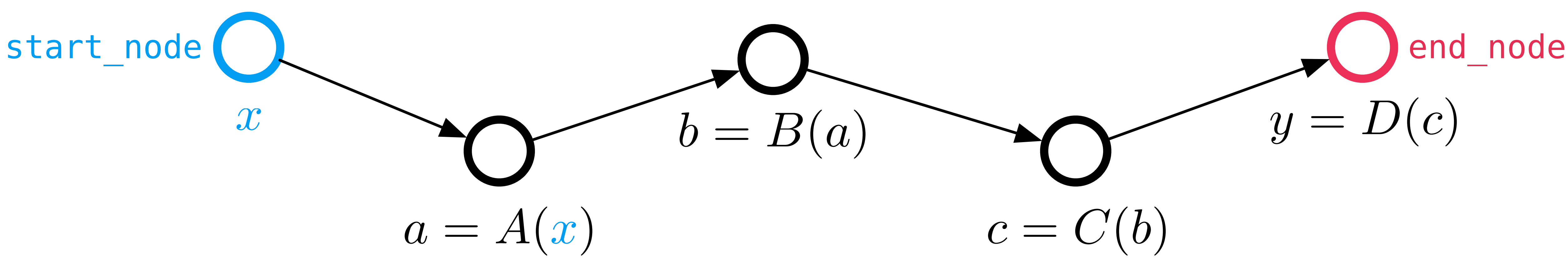
```python
def forward_pass(fun, args, kwargs, argnum=0):
    args = list(args)
    start_node = new_progenitor(args[argnum])
    args[argnum] = start_node
    active_progenitors.add(start_node)
    end_node = fun(*args, **kwargs)
    active_progenitors.remove(start_node)
    return start_node, end_node
```

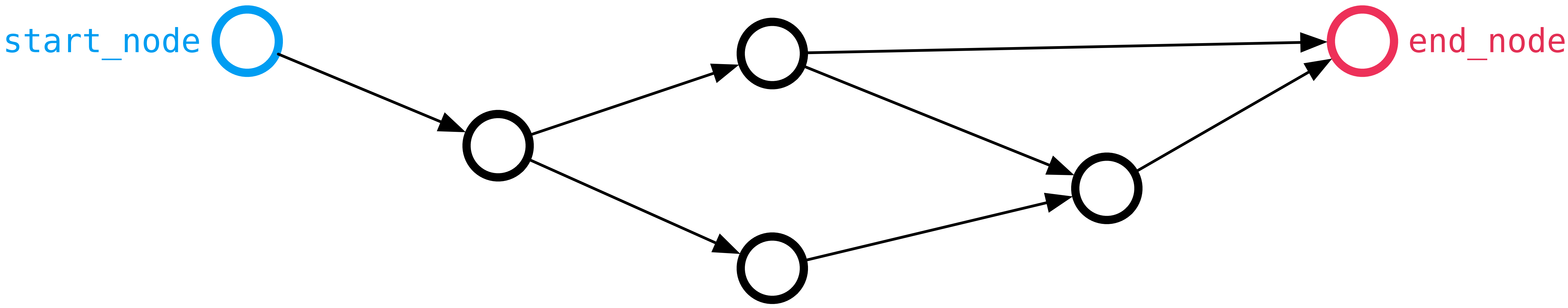

start_node
x


start_node
x
$a = A(x)$


start_node
x
$a = A(x)$
$b = B(a)$


start_node
x
$a = A(x)$
$b = B(a)$
$c = C(b)$

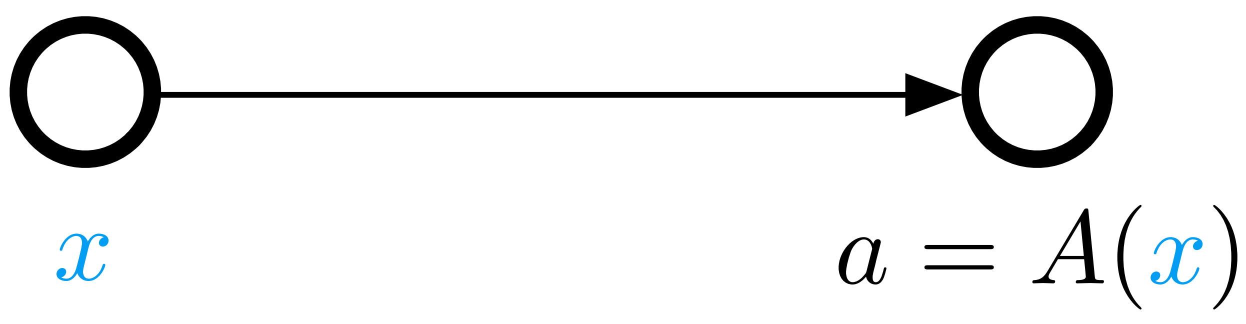

start_node
x
$a = A(x)$
$b = B(a)$
$c = C(b)$
$y = D(c)$
end_node

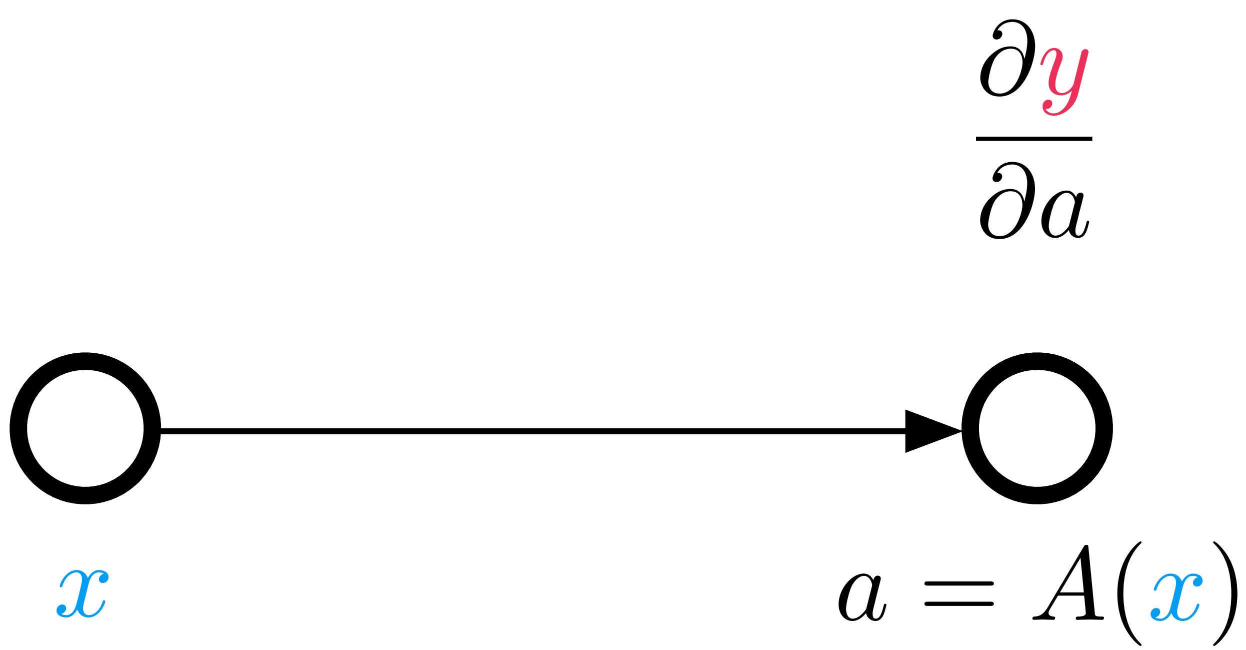


No control flow!

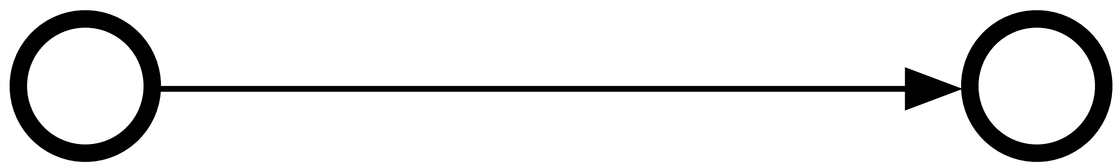
ingredients:

1. tracing composition of primitive functions
2. vector-Jacobian product for each primitive
3. composing VJPs backward


x
$a = A(x)$

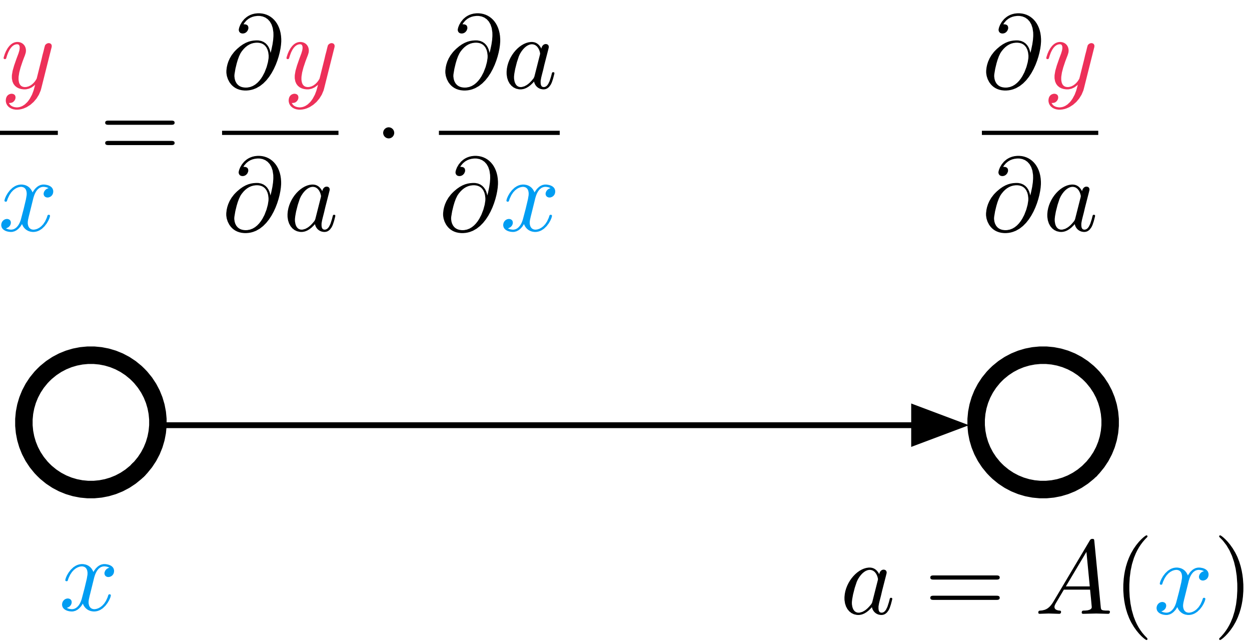

$\frac{\partial y}{\partial a}$
x
$a = A(x)$

$$\frac{\partial y}{\partial x} = ? \qquad \frac{\partial y}{\partial a}$$



$$x \qquad a = A(x)$$

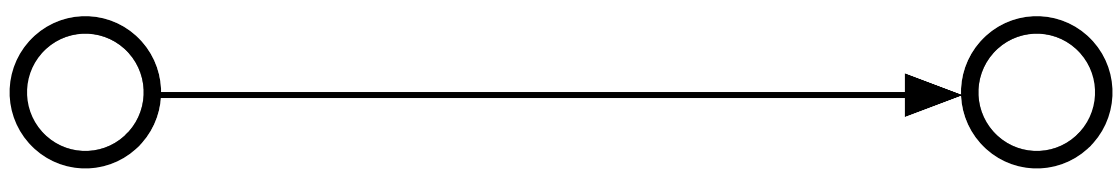
$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial a} \cdot \frac{\partial a}{\partial x}$$



vector-Jacobian product

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial a}\boxed{\cdot\, A'(x)}$$

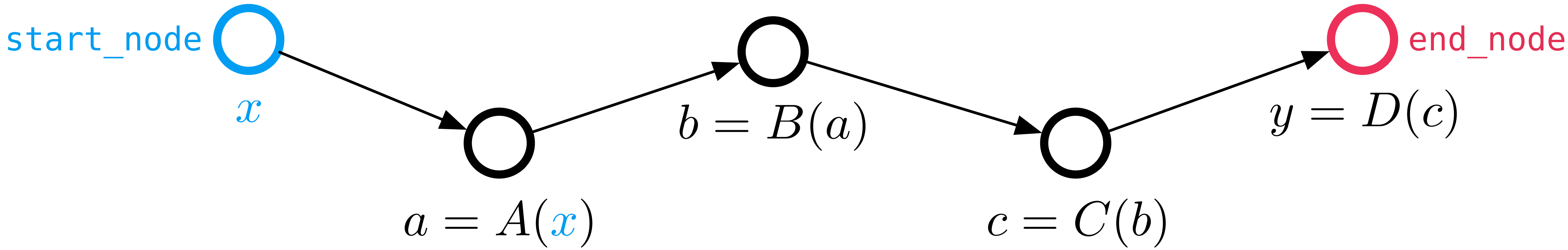
$$\frac{\partial y}{\partial a}$$

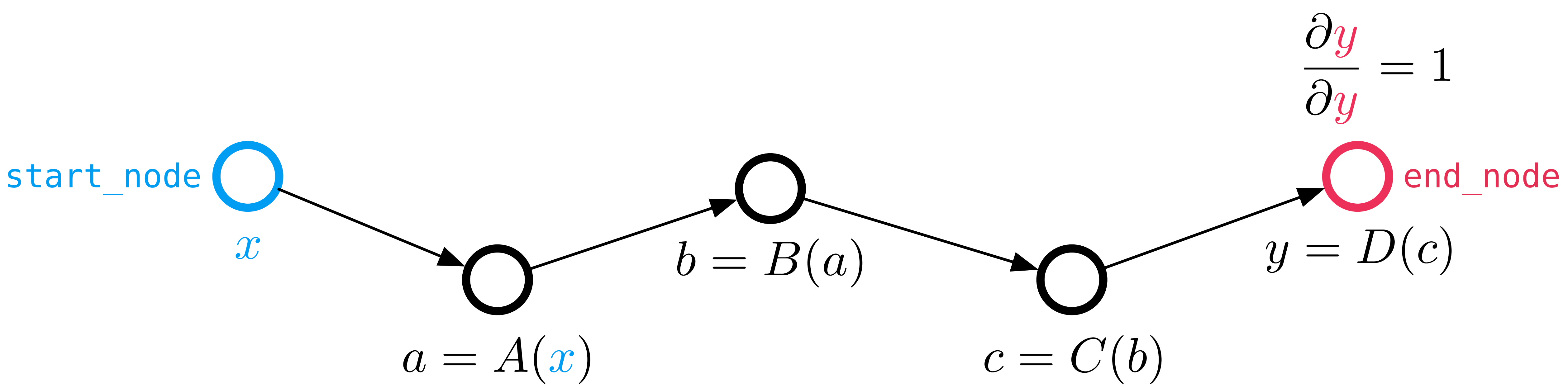


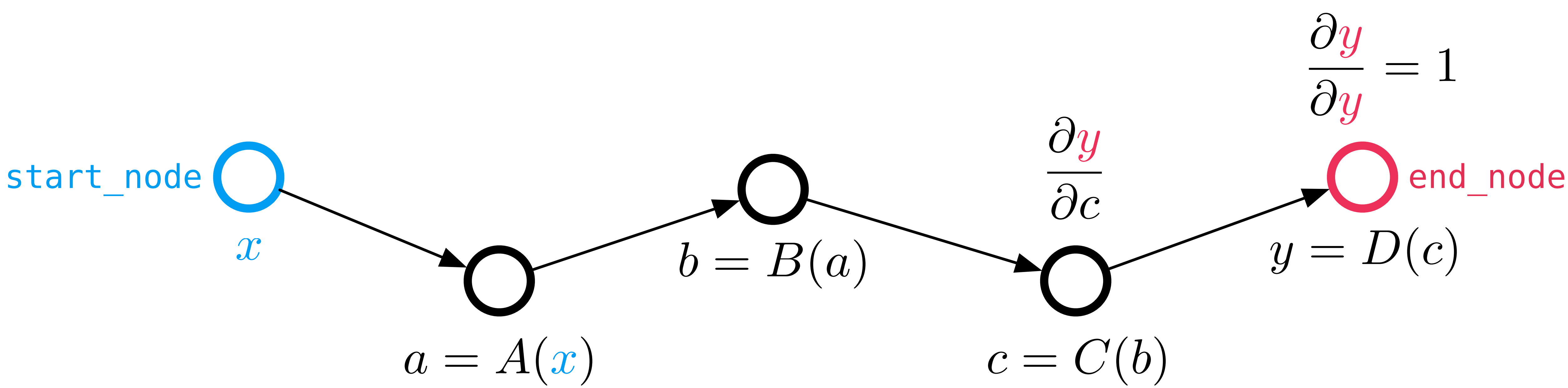
```python
anp.sinh.defvjp(lambda g, ans, vs, gvs, x : g * anp.cosh(x))
anp.cosh.defvjp(lambda g, ans, vs, gvs, x : g * anp.sinh(x))
anp.tanh.defvjp(lambda g, ans, vs, gvs, x : g / anp.cosh(x) **2)

anp.cross.defvjp(lambda g, ans, vs, gvs, a, b, axisa=-1, axisb=-1, axisc=-1, axis=None :
                anp.cross(b, g, axisb, axisc, axisa, axis), argnum=0)

def grad_sort(g, ans, vs, gvs, x, axis=-1, kind='quicksort', order=None):
    sort_perm = anp.argsort(x, axis, kind, order)
    return unpermuter(g, sort_perm)
anp.sort.defvjp(grad_sort)
```

ingredients:

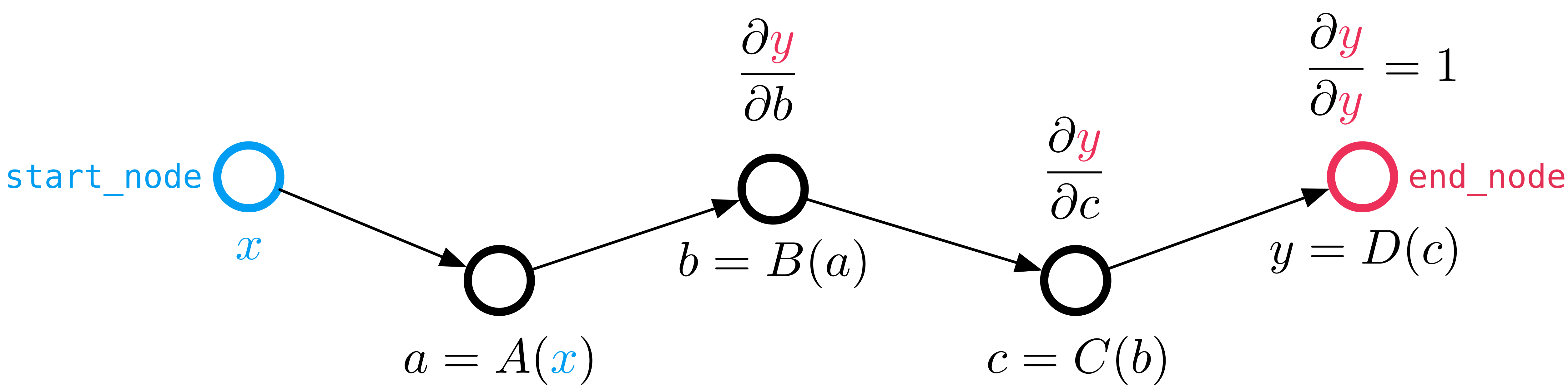
1. tracing composition of primitive functions
2. vector-Jacobian product for each primitive
3. composing VJPs backward

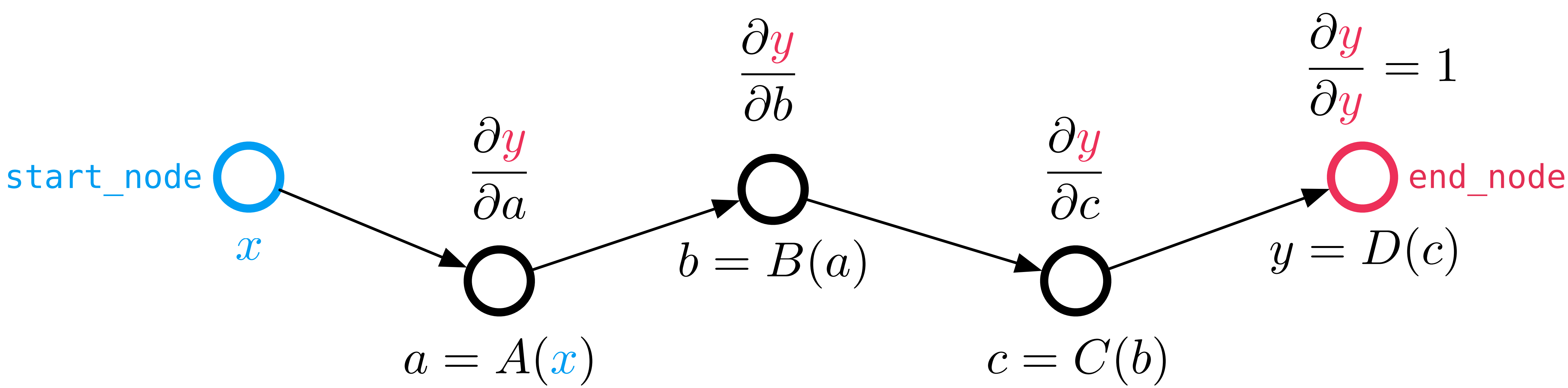

start_node
end_node
x
$a = A(x)$
$b = B(a)$
$c = C(b)$
$y = D(c)$


$\frac{\partial y}{\partial y} = 1$
start_node
end_node
x
$a = A(x)$
$b = B(a)$
$c = C(b)$
$y = D(c)$

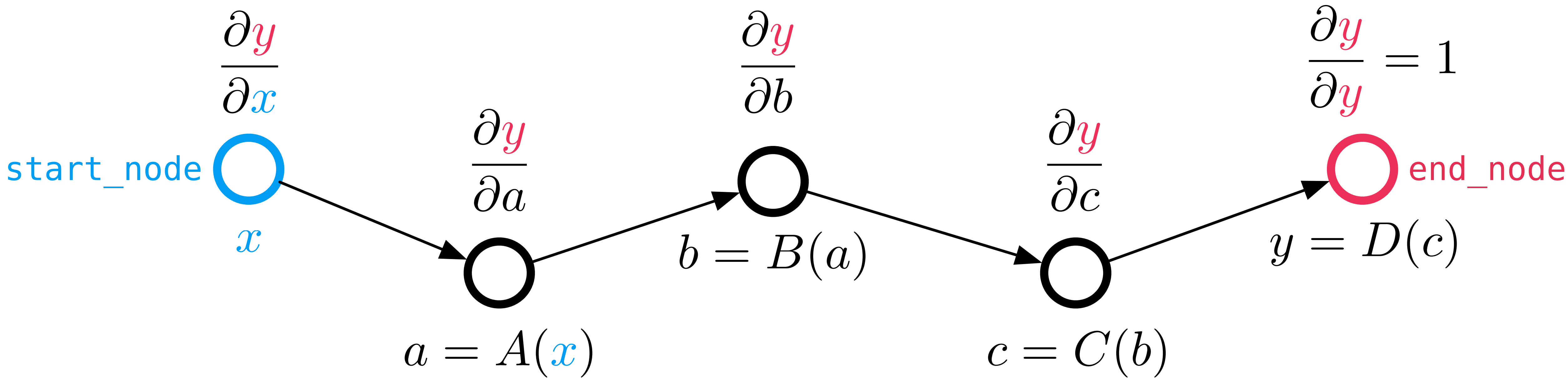


start_node
x
$a = A(x)$
$b = B(a)$
$\frac{\partial y}{\partial c}$
$c = C(b)$
$\frac{\partial y}{\partial y} = 1$
end_node
$y = D(c)$


start_node
x
$a = A(x)$
$\frac{\partial y}{\partial b}$
$b = B(a)$
$\frac{\partial y}{\partial c}$
$c = C(b)$
$\frac{\partial y}{\partial y} = 1$
end_node
$y = D(c)$

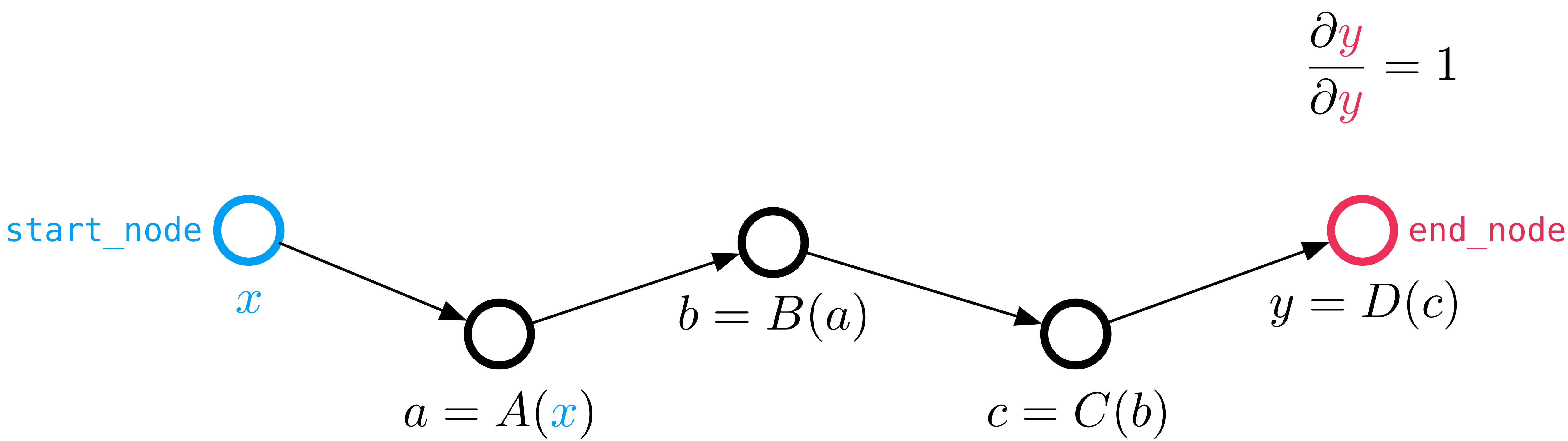


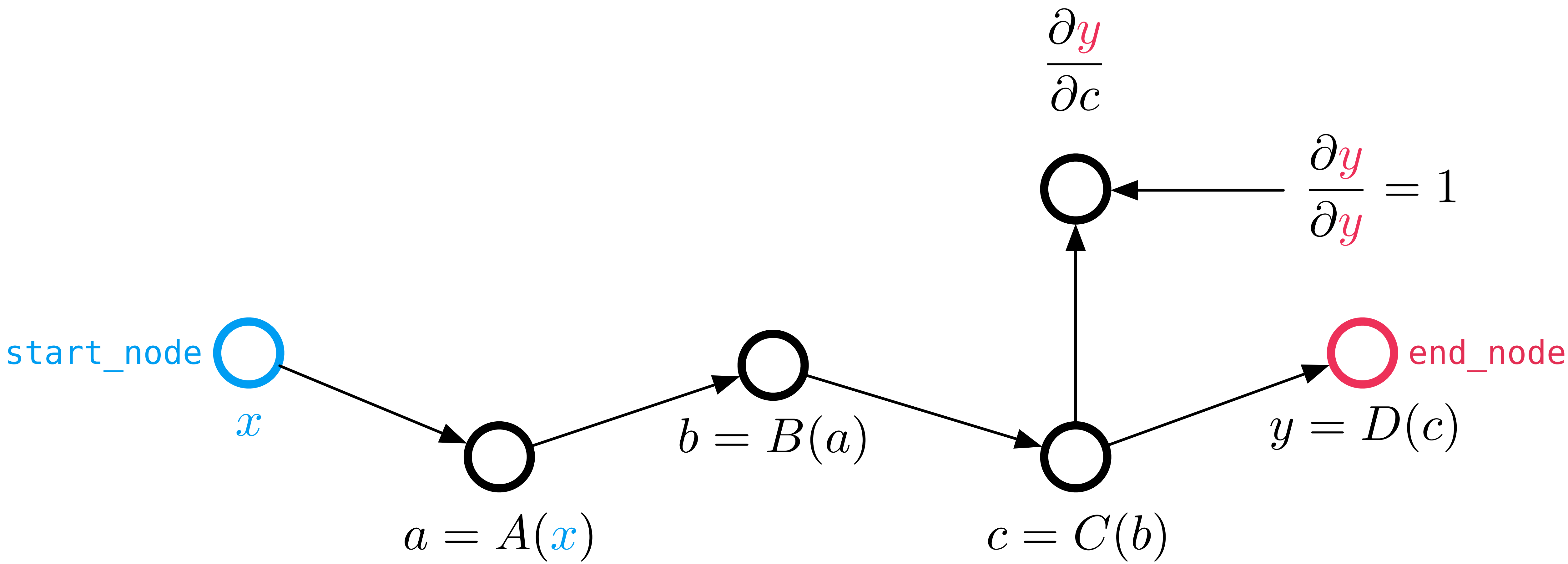
start_node
x
$\frac{\partial y}{\partial a}$
$a = A(x)$
$\frac{\partial y}{\partial b}$
$b = B(a)$
$\frac{\partial y}{\partial c}$
$c = C(b)$
$\frac{\partial y}{\partial y} = 1$
$y = D(c)$
end_node

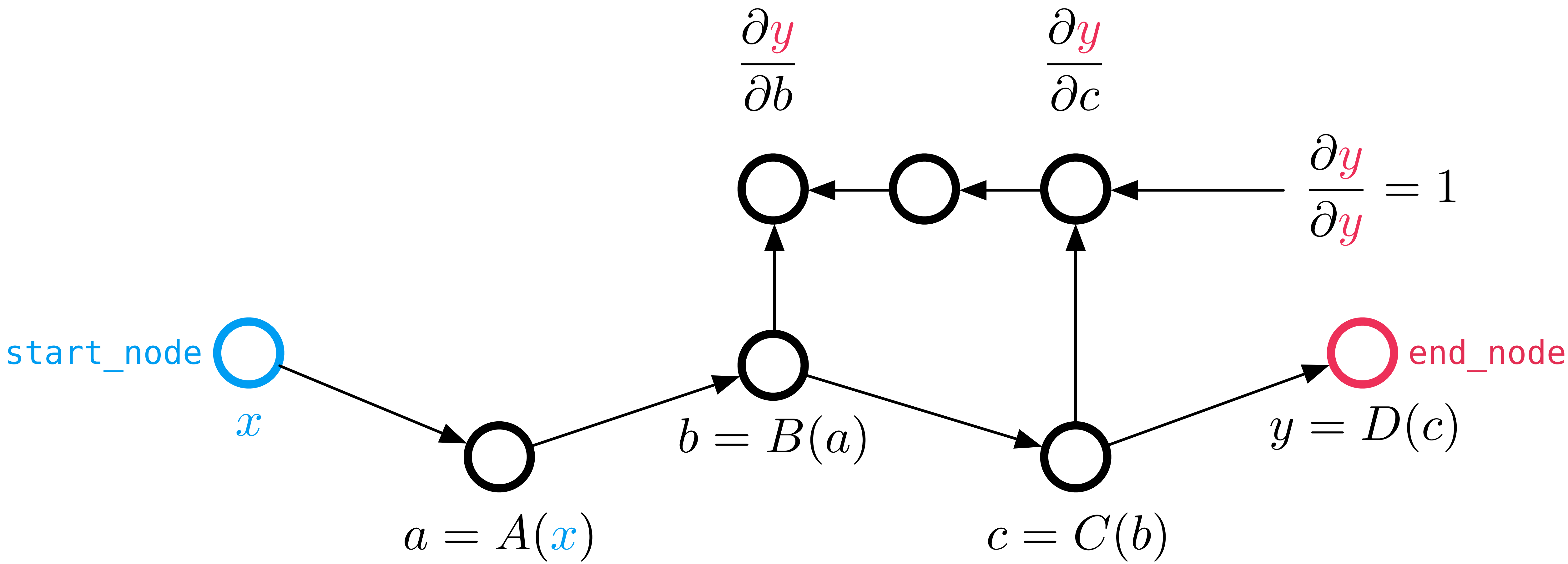


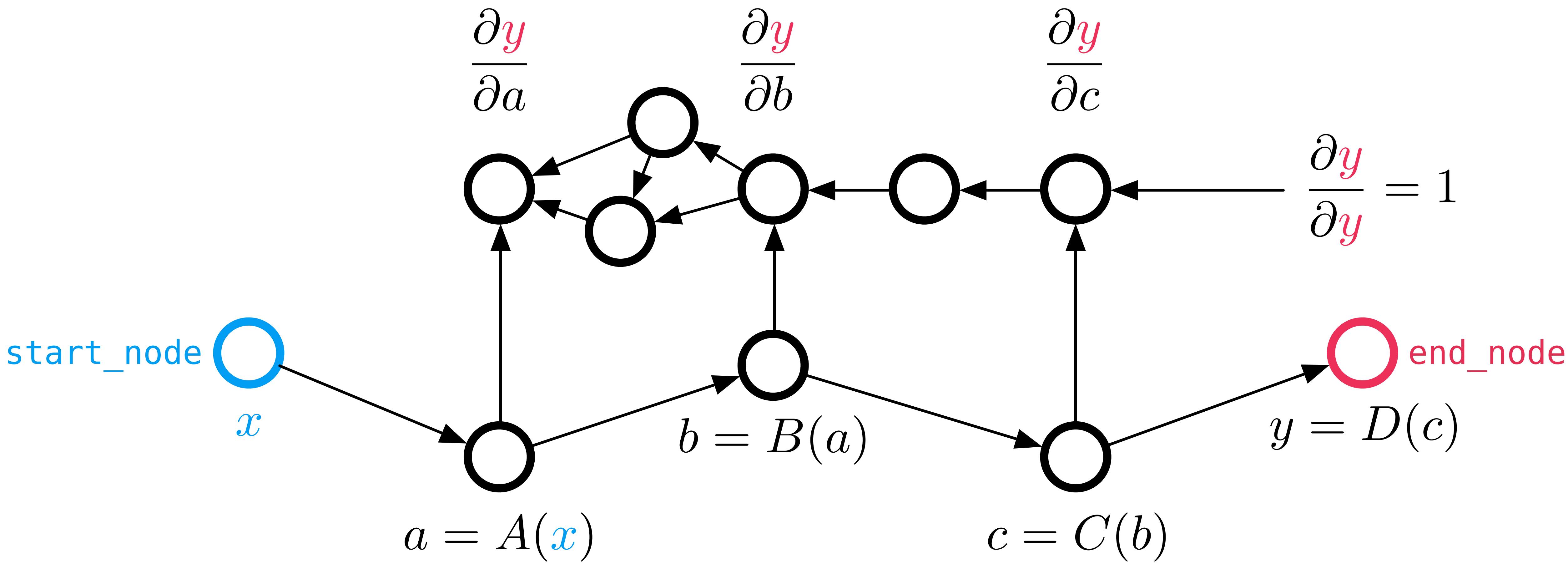
start_node
end_node
$\frac{\partial y}{\partial x}$
x
$\frac{\partial y}{\partial a}$
$a = A(x)$
$\frac{\partial y}{\partial b}$
$b = B(a)$
$\frac{\partial y}{\partial c}$
$c = C(b)$
$\frac{\partial y}{\partial y} = 1$
$y = D(c)$

higher-order autodiff just works:
the backward pass can itself be traced

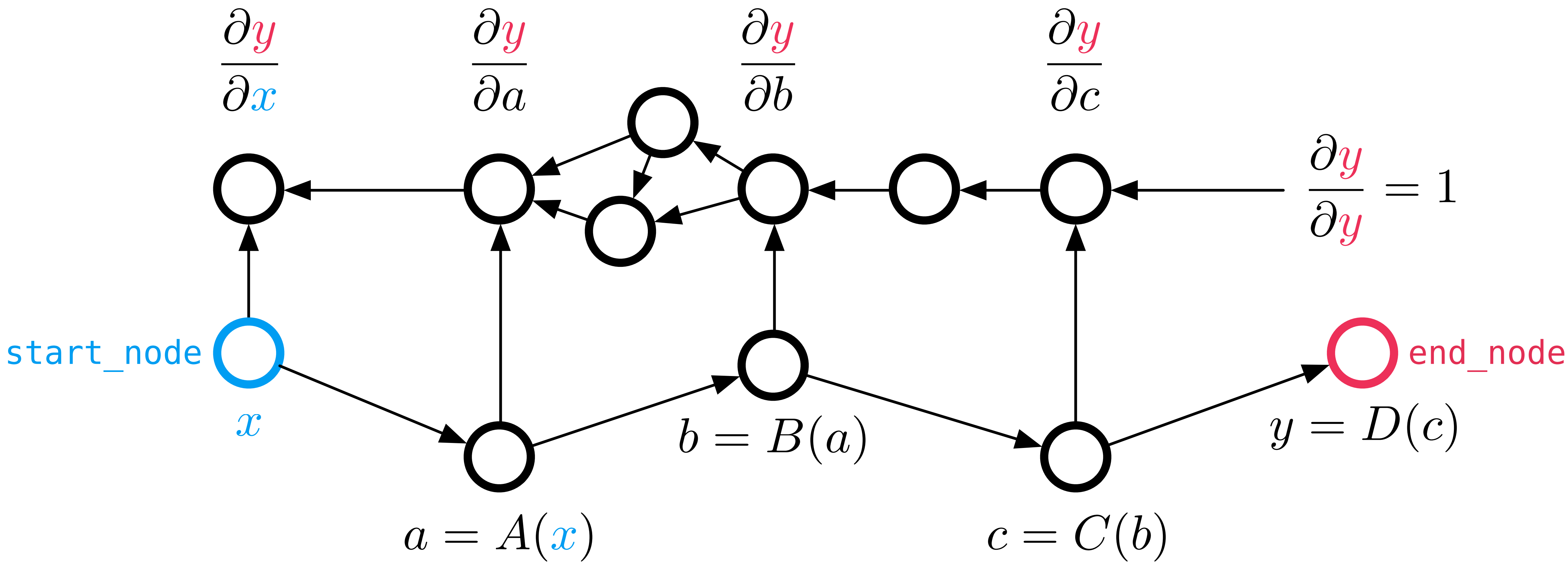

$\frac{\partial y}{\partial y} = 1$
start_node
x
$a = A(x)$
$b = B(a)$
$c = C(b)$
$y = D(c)$
end_node


start_node
x
$a = A(x)$
$b = B(a)$
$c = C(b)$
$y = D(c)$
end_node
$\frac{\partial y}{\partial c}$
$\frac{\partial y}{\partial y} = 1$

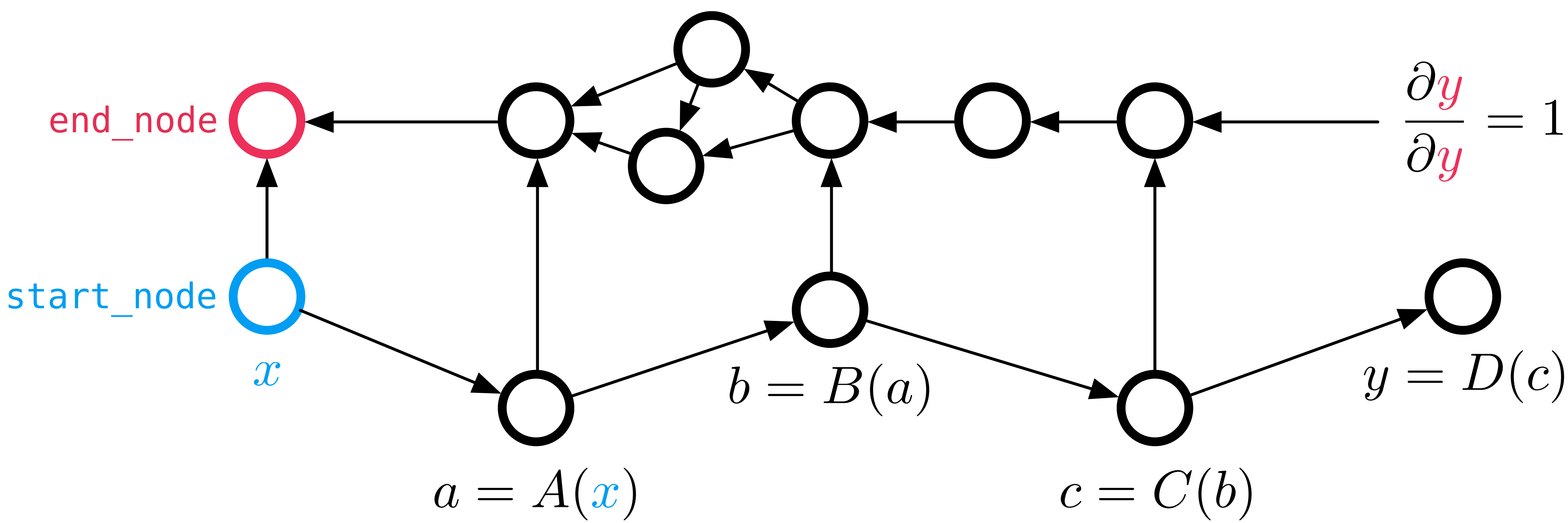

start_node
x
$a = A(x)$
$b = B(a)$
$c = C(b)$
$y = D(c)$
end_node
$\frac{\partial y}{\partial b}$
$\frac{\partial y}{\partial c}$
$\frac{\partial y}{\partial y} = 1$



$\frac{\partial y}{\partial a}$
$\frac{\partial y}{\partial b}$
$\frac{\partial y}{\partial c}$
$\frac{\partial y}{\partial y} = 1$
start_node
x
end_node
$b = B(a)$
$y = D(c)$
$a = A(x)$
$c = C(b)$



$\frac{\partial y}{\partial x}$
$\frac{\partial y}{\partial a}$
$\frac{\partial y}{\partial b}$
$\frac{\partial y}{\partial c}$
$\frac{\partial y}{\partial y} = 1$
start_node
end_node
x
$a = A(x)$
$b = B(a)$
$c = C(b)$
$y = D(c)$


end_node
start_node
x
$a = A(x)$
$b = B(a)$
$c = C(b)$
$y = D(c)$
$\frac{\partial y}{\partial y} = 1$

```python
def backward_pass(g, end_node, start_node):
    outgrads = defaultdict(list)
    outgrads[end_node] = [g]
    assert_vspace_match(outgrads[end_node][0], end_node.vspace, None)
    for node in toposort(end_node, start_node):
        if node not in outgrads: continue
        cur_outgrad = vsum(node.vspace, *outgrads[node])
        function, args, kwargs, parents = node.recipe
        for argnum, parent in parents:
            outgrad = function.vjp(argnum, cur_outgrad, node,
                                   parent.vspace, node.vspace, args, kwargs)
            outgrads[parent].append(outgrad)
            assert_vspace_match(outgrad, parent.vspace, function)
    return cur_outgrad
```

```python
def grad(fun, argnum=0):
    @attach_name_and_doc(fun, argnum, 'Gradient')
    def gradfun(*args,**kwargs):
        vjp, _ = make_vjp(fun, argnum)(*args, **kwargs)
        return vjp(1.0)

    return gradfun
```

```python
def make_vjp(fun, argnum=0):
    def vjp(*args, **kwargs):
        start_node, end_node = forward_pass(fun, args, kwargs, argnum)
        if not isnode(end_node) or start_node not in end_node.progenitors:
            warnings.warn("Output seems independent of input.")
            return lambda g : start_node.vspace.zeros(), end_node
        return lambda g : backward_pass(g, end_node, start_node), end_node
    return vjp
```

ingredients:

1. tracing composition of primitive functions
   Node, primitive, forward_pass
2. vector-Jacobian product for each primitive
   defvjp
3. composing VJPs backward
   backward_pass, make_vjp, grad

what’s the point? easy to extend!

- develop autograd!
- forward mode
- log joint densities from sampler programs