# Autodiff generates your exponential family inference code
# +
# Autograd's implementation

HARVARD
INTELLIGENT
PROBABILISTIC
SYSTEMS

HARVARD
School of Engineering
and Applied Sciences

$$p(x \mid \eta) = \exp\left\{\eta \cdot t(x) - \log Z(\eta)\right\}$$

$$p(x \mid \eta) = \exp\{\eta \cdot t(x) - \log Z(\eta)\}$$

Bernoulli:

$$t(x) = x$$

$$\eta = \log p$$

$$p(x \mid \eta) = \exp\left\{\eta \cdot t(x) - \log Z(\eta)\right\}$$

Gaussian:

$$t(x) = \begin{pmatrix} x & x^2 \end{pmatrix}$$

$$\eta = \left( \frac{\mu}{\sigma^2}, -\frac{1}{2\sigma^2} \right)$$

$$p(x \mid \eta) = \exp\left\{\eta \cdot t(x) - \log Z(\eta)\right\}$$

Gamma:

$$t(x) = \begin{pmatrix} \log x & x \end{pmatrix}$$

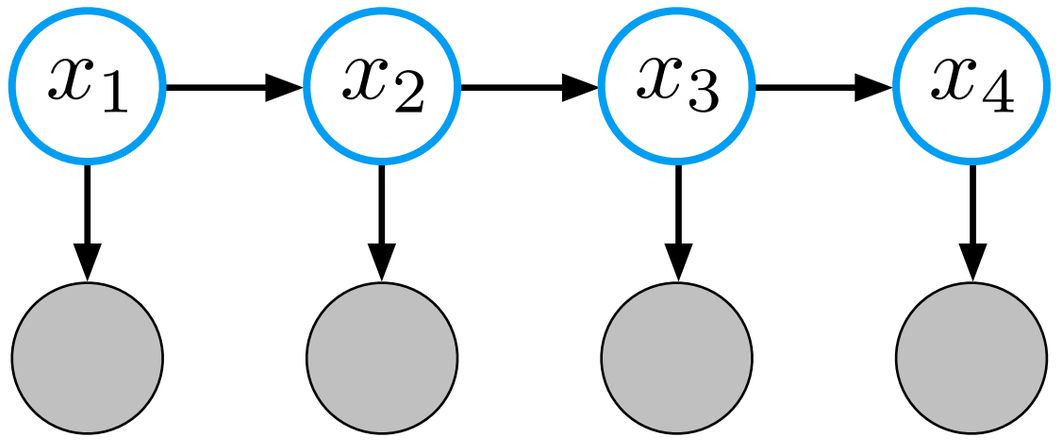
$$\eta = \begin{pmatrix} \alpha - 1 & -\beta \end{pmatrix}$$

$$p(x \mid \eta) = \exp\left\{\eta \cdot t(x) - \log Z(\eta)\right\}$$

$$p(x \mid \eta) = \exp\{\eta \cdot t(x) - \log Z(\eta)\}$$

Hidden Markov models:

$$t(x) = \begin{pmatrix} x_i & x_i x_j^\mathsf{T} \end{pmatrix}$$



$$\nabla \log Z(\eta) = \mathbb{E}[\, t(x) \,]$$

$$\nabla \log Z(\eta) = \mathbb{E}[\, t(x) \,]$$

$$\log Z(\eta) = \qquad \log \int \exp\{\eta \cdot t(x)\} \, \mathrm{d}x$$

$$\nabla \log Z(\eta) = \mathbb{E}[\, t(x) \,]$$

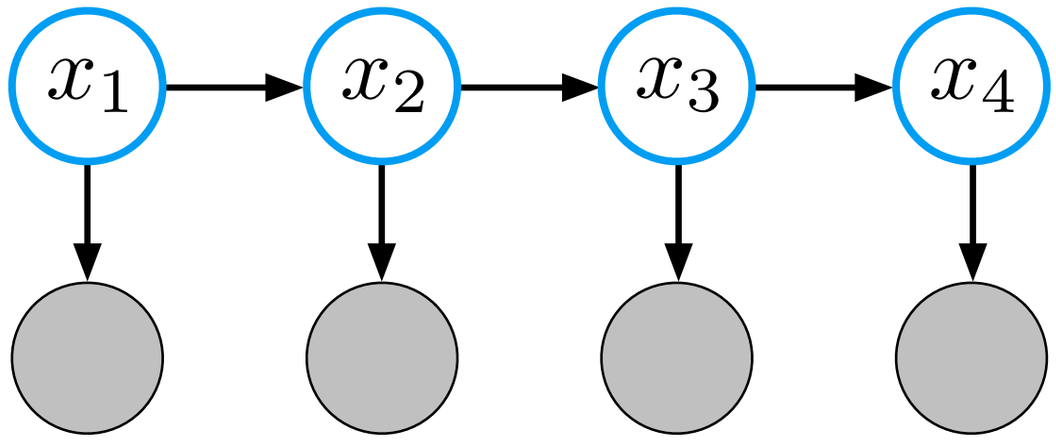
$$\nabla \log Z(\eta) = \nabla_\eta \log \int \exp\{\eta \cdot t(x)\} \, \mathrm{d}x$$

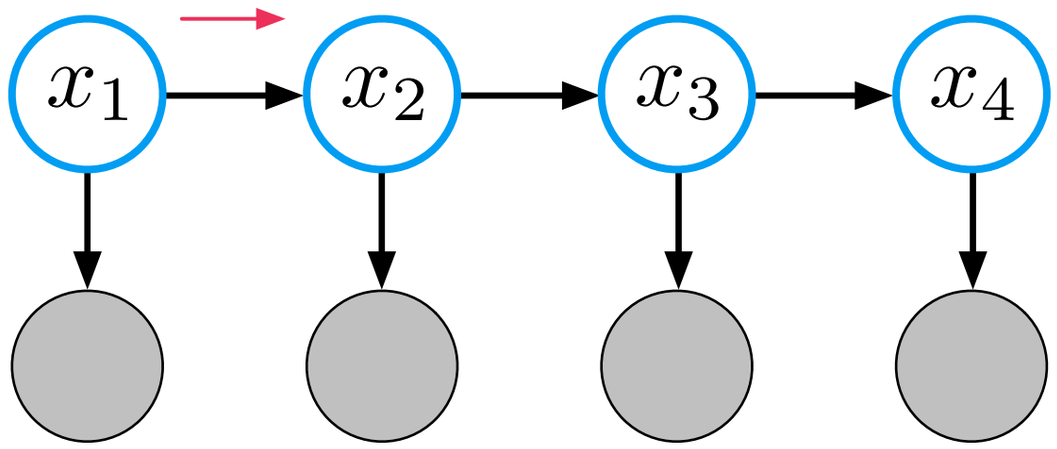
$$\nabla \log Z(\eta) = \mathbb{E}[\, t(x) \,]$$

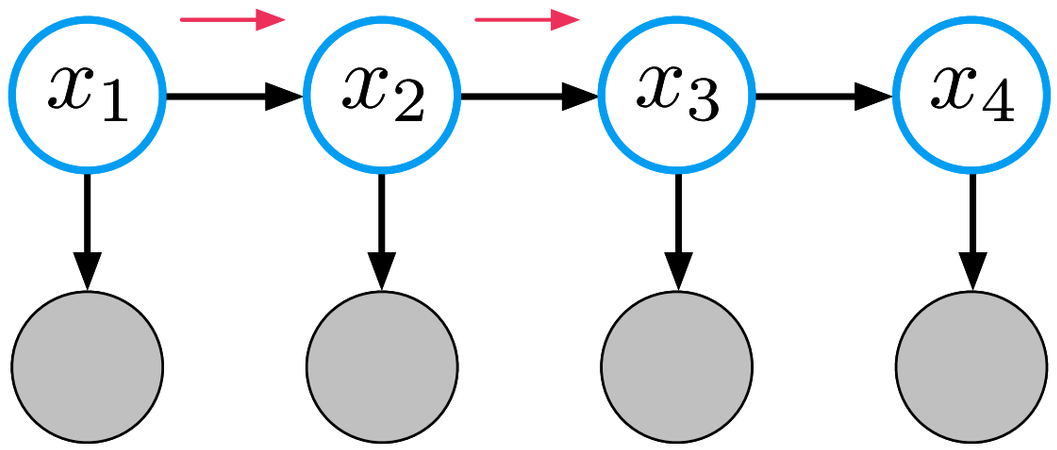
$$\begin{aligned}
\nabla \log Z(\eta) &= \nabla_\eta \log \int \exp\{\eta \cdot t(x)\} \, \mathrm{d}x \\
&= \left( \int \exp\{\eta \cdot t(x)\} \, \mathrm{d}x \right)^{-1} \int \nabla_\eta \exp\{\eta \cdot t(x)\} \, \mathrm{d}x \\
&= \frac{1}{Z} \int \exp\{\eta \cdot t(x)\} \, t(x) \, \mathrm{d}x \\
&= \int p(x \,|\, \eta) \, t(x) \, \mathrm{d}x
\end{aligned}$$

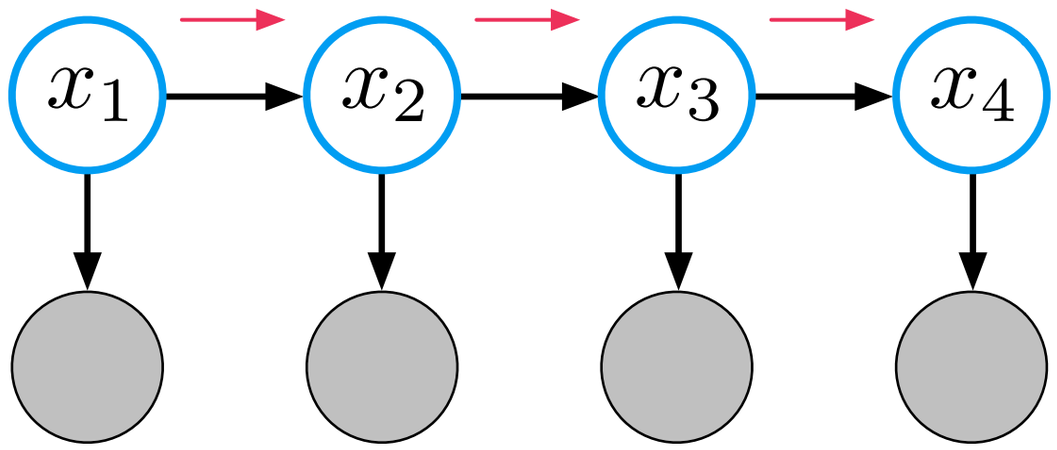
$$\nabla \log Z(\eta) = \mathbb{E}[\, t(x) \,]$$

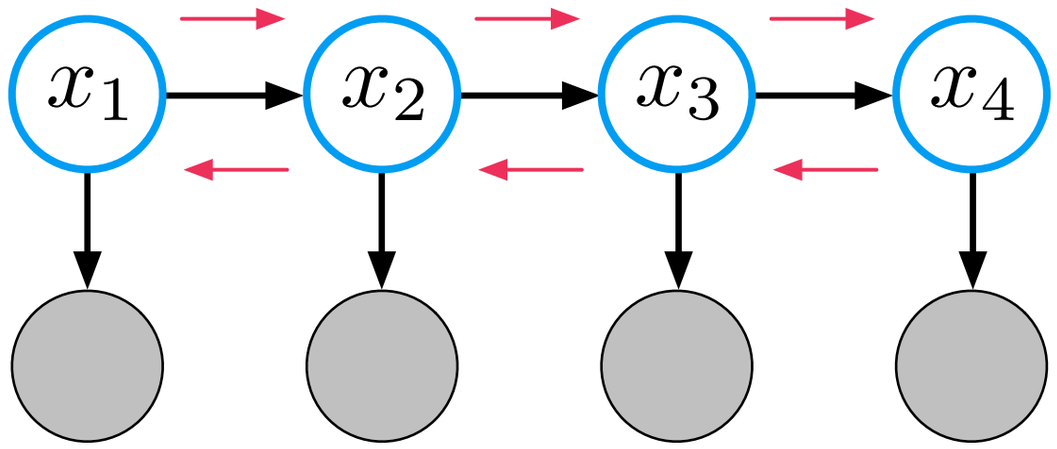
Hey, gradients! We can use autodiff!


$x_1$
$x_2$
$x_3$
$x_4$

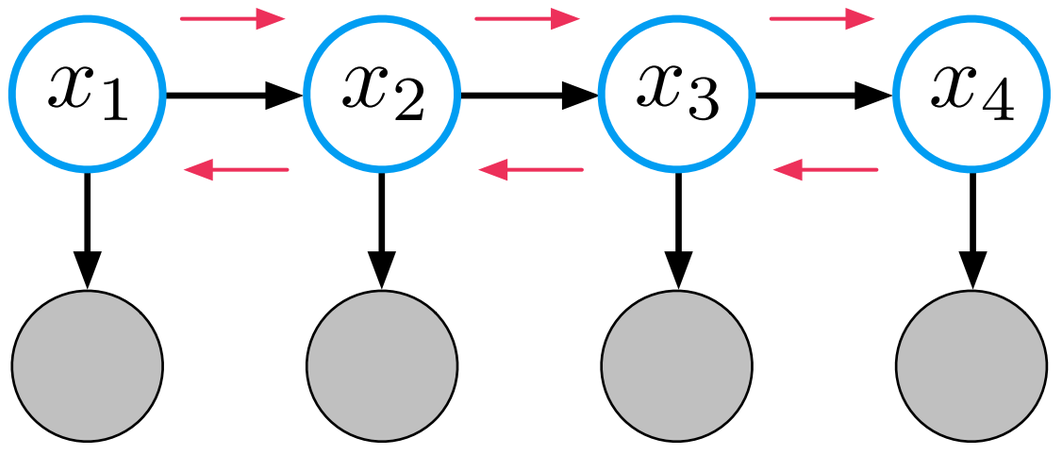

$x_1$
$x_2$
$x_3$
$x_4$

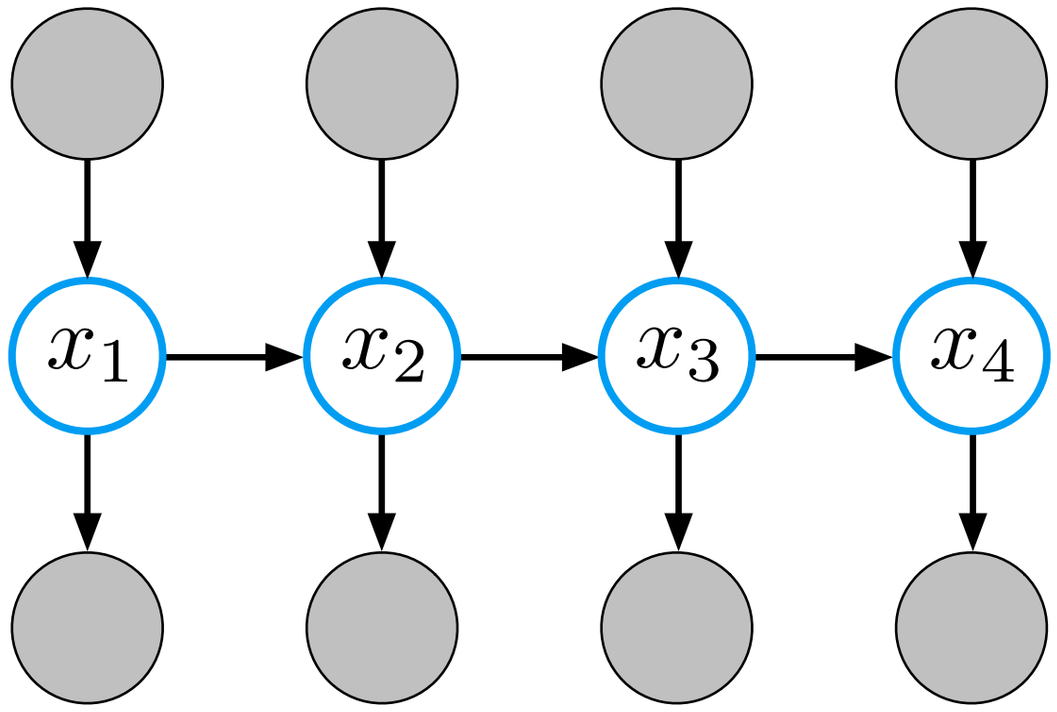

$x_1$
$x_2$
$x_3$
$x_4$


$x_1$
$x_2$
$x_3$
$x_4$


$x_1$
$x_2$
$x_3$
$x_4$

```
import autograd.numpy as np
from autograd.scipy.misc import logsumexp
from autograd import value_and_grad

def log_Z(natural_params, data):
    log_pi, log_A, log_B = natural_params

    log_alpha = log_pi
    for y in data:
        log_alpha = logsumexp(log_alpha[:,None] + log_A, axis=0) + log_B[:,y]

    return logsumexp(log_alpha)

log_likelihood, expected_stats = value_and_grad(log_Z)(natural_params, data)
```


$x_1$
$x_2$
$x_3$
$x_4$


$x_1$
$x_2$
$x_3$
$x_4$

```
import autograd.numpy as np
from autograd.scipy.misc import logsumexp
from autograd import value_and_grad

def log_Z(natural_params, inputs, data):
    log_pi, log_A, log_B = natural_params

    log_alpha = log_pi
    for u, y in zip(inputs, data):
        log_alpha = logsumexp(log_alpha[:,None] + log_A[u], axis=0) + log_B[:,y]

    return logsumexp(log_alpha)

E_step = value_and_grad(log_Z)
log_likelihood, expected_stats = E_step(natural_params, inputs, data)
```

# Kalman smoother

```
def log_Z(natural_params):
    init_params, pair_params, node_params = natural_params

    def unit(J, h, logZ):
        return (J, h), logZ

    def bind(result, step):
        message, lognorm = result
        new_message, term = step(message)
        return new_message, lognorm + term

    steps = interleave(map(condition, node_params), map(predict, pair_params))
    message, lognorm = monad_runner(bind, unit(*init_params), steps)
    lognorm = lognorm + natural_lognorm(message)

    return lognorm

def monad_runner(bind, result, steps):
    for step in steps:
        result = bind(result, step)
    return result

kalman_smoother = grad(log_Z)
```

```python
import autograd.numpy as np
from autograd.scipy.linalg import solve_triangular
from toolz import curry

@curry
def condition(node_param, message):
    J, h = message
    J_node, h_node, log_Z_node = node_param
    return (J + J_node, h + h_node), log_Z_node

@curry
def predict(pair_param, message):
    J, h = message
    J11, J12, J22, log_Z_pair = pair_param
    L = np.linalg.cholesky(-2*(J + J11))
    v = solve_triangular(L, h)
    lognorm = 1./2 * np.dot(v, v) - np.sum(np.log(np.diag(L)))
    h_predict = np.dot(J12.T, solve_triangular(L, v, trans='T'))
    temp = solve_triangular(L, J12)
    J_predict = J22 + 1./2 * np.dot(temp.T, temp)
    return (J_predict, h_predict), lognorm + log_Z_pair
```

Take-aways:

- autodiff can help with writing and testing inference
- code gen for autodiff means code gen for inference
- higher-order autodiff is useful

# Autograd's implementation

[github.com/hips/autograd](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!

```
git checkout $(git rev-list --max-parents=0 HEAD)
```

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

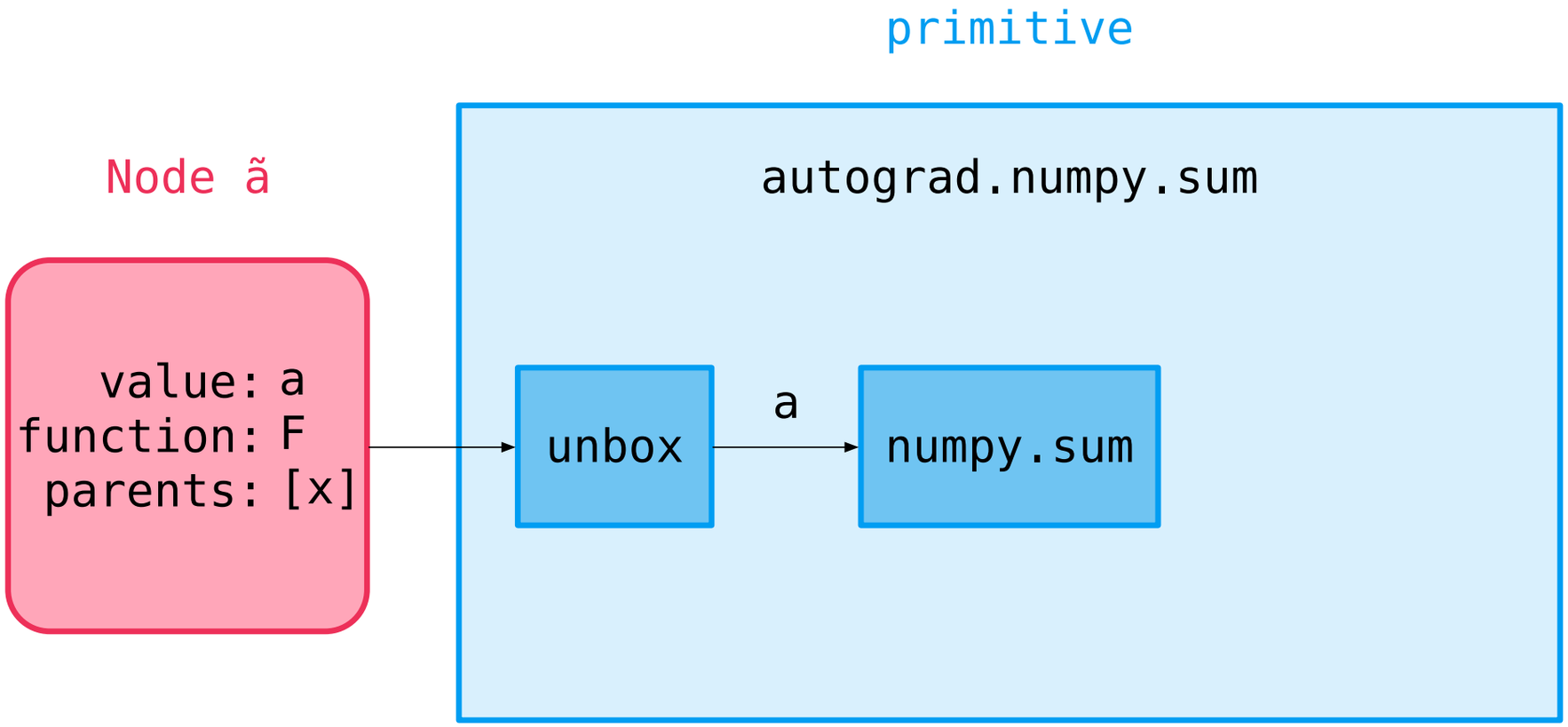
primitive

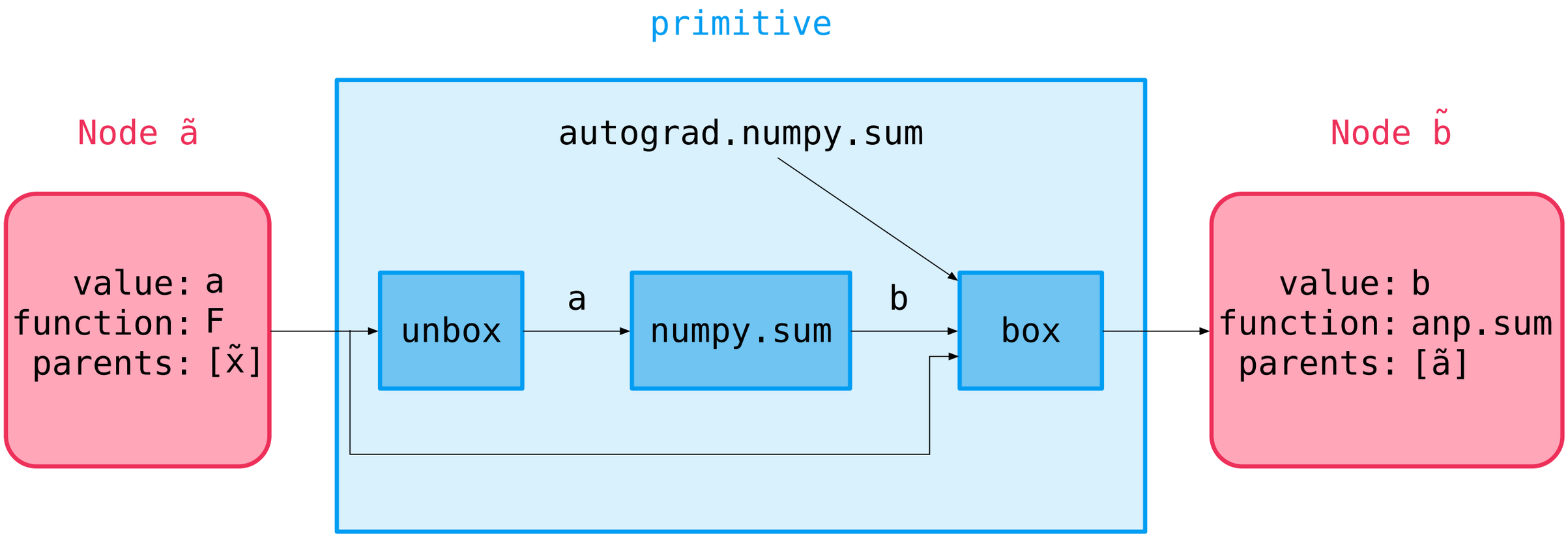
Node ã

value: a
function: F
parents: [x]

autograd.numpy.sum

numpy.sum


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

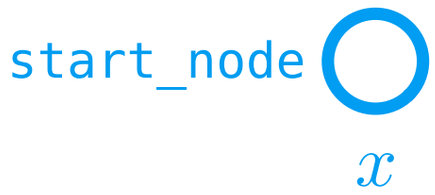

primitive
Node ã
autograd.numpy.sum
Node b̃
value: a
function: F
parents: [x̃]
unbox
a
numpy.sum
b
box
value: b
function: anp.sum
parents: [ã]

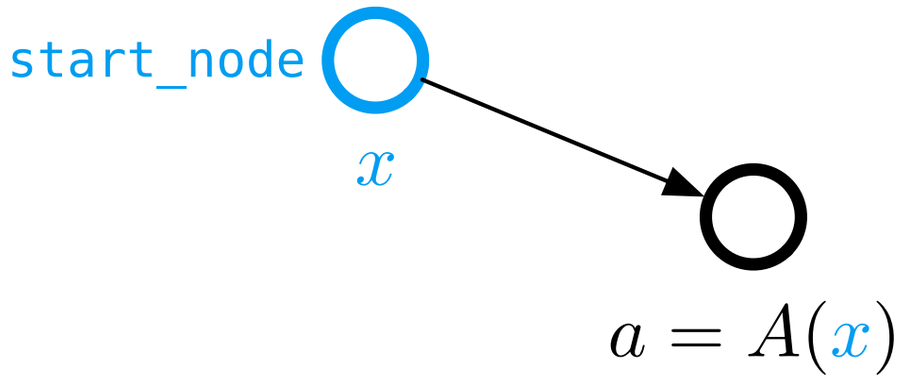
```
class Node(object):
    __slots__ = ['value', 'recipe', 'progenitors', 'vspace']

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

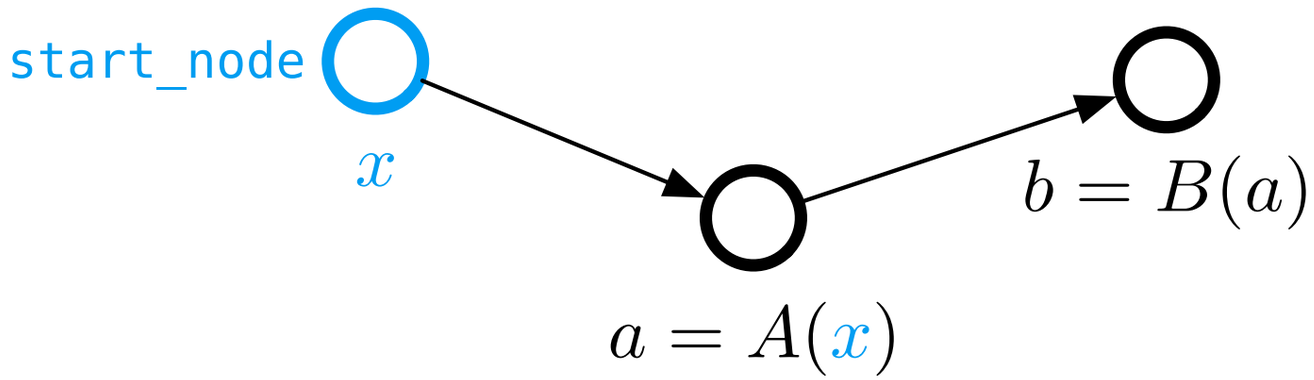
```
class primitive(object):
    def __call__(self, *args, **kwargs):
        argvals = list(args)

        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),                )
```

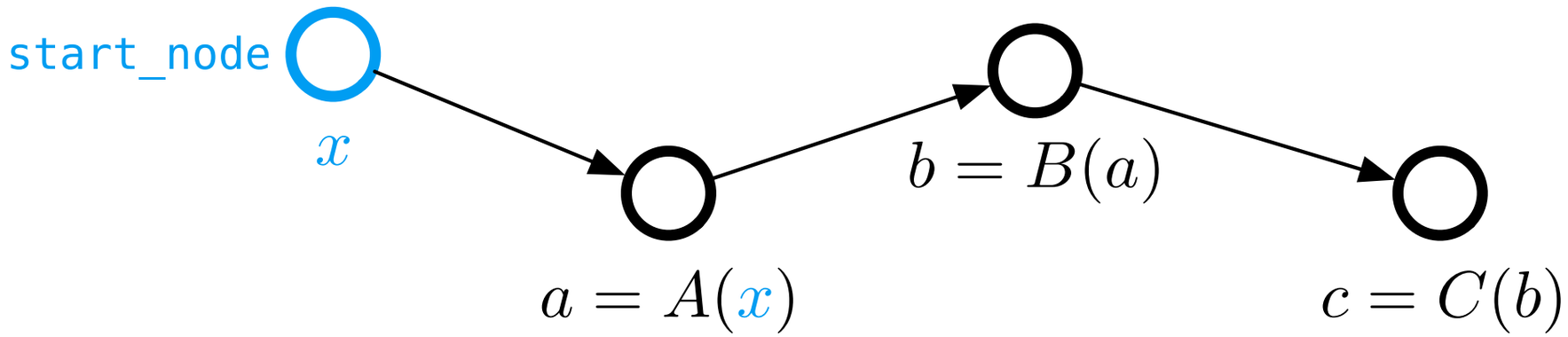
```
class primitive(object):
    def __call__(self, *args, **kwargs):
        argvals = list(args)
        progenitors = set()
        parents = []
        for argnum, arg in enumerate(args):
            if isnode(arg):
                argvals[argnum] = arg.value
                if argnum in self.zero_vjps: continue
                parents.append((argnum, arg))
                progenitors.update(arg.progenitors & active_progenitors)

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

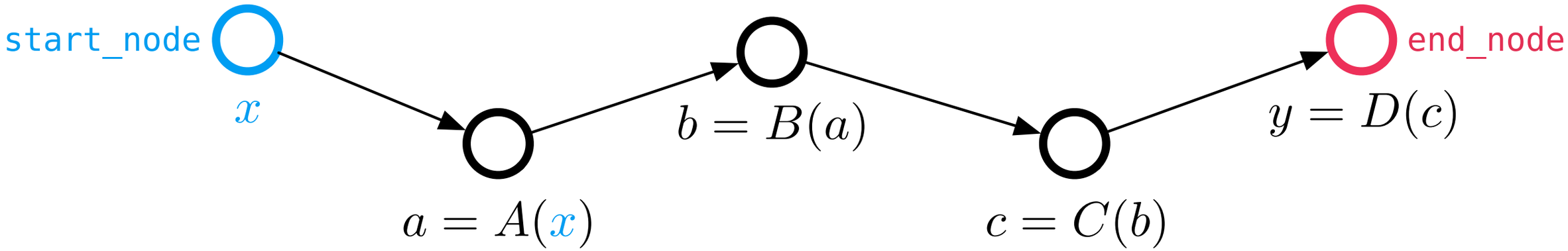
```
def forward_pass(fun, args, kwargs, argnum=0):
    args = list(args)
    start_node = new_progenitor(args[argnum])
    args[argnum] = start_node
    active_progenitors.add(start_node)
    end_node = fun(*args, **kwargs)
    active_progenitors.remove(start_node)
    return start_node, end_node
```

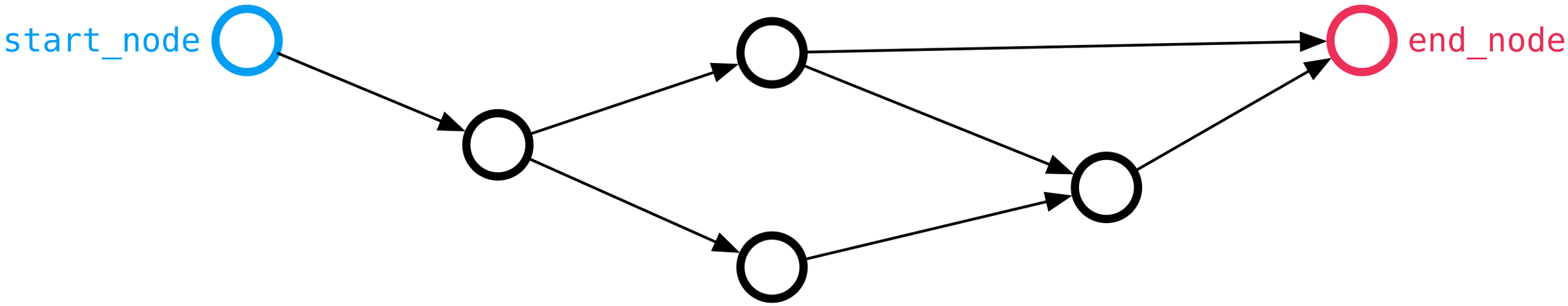

start_node
$x$

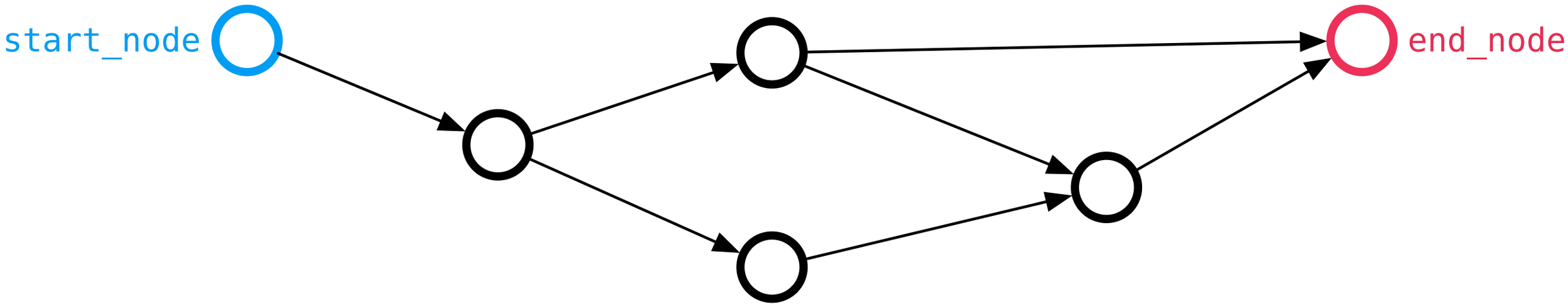

start_node
$x$
$a = A(x)$


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

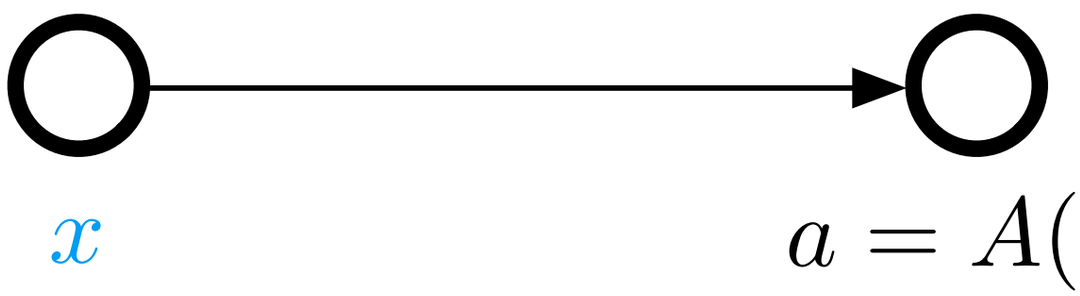

start_node
end_node




No control flow!

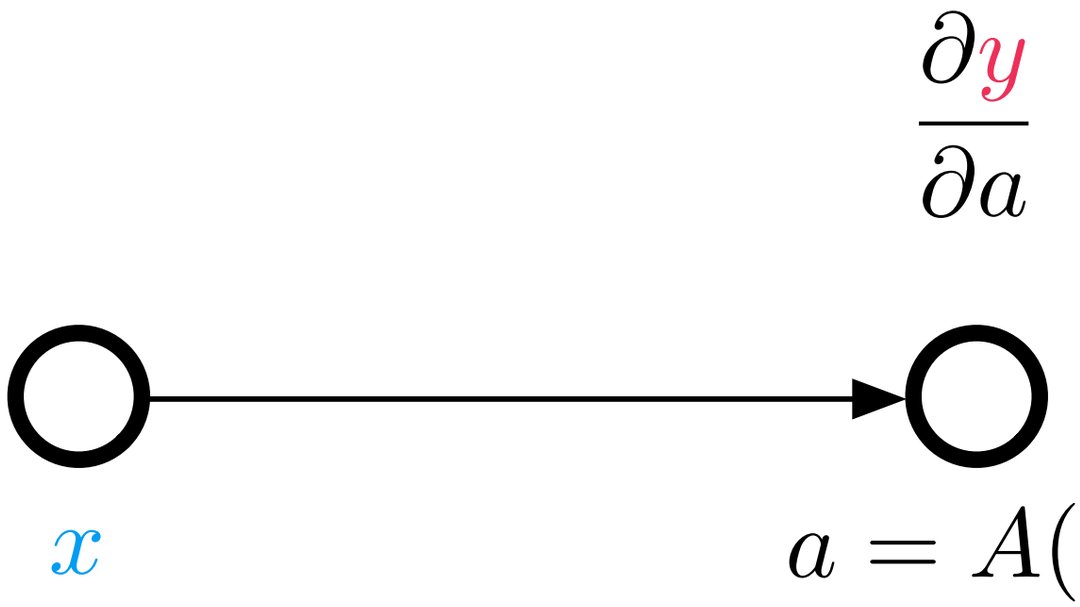
ingredients:

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



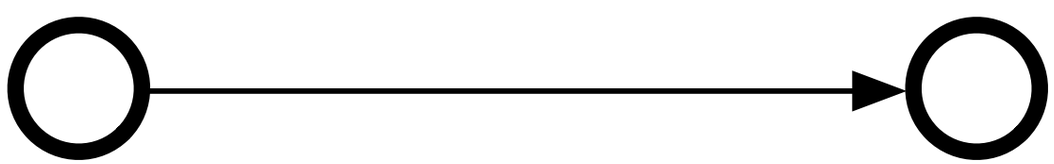
$x$

$a = A(x)$


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

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

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

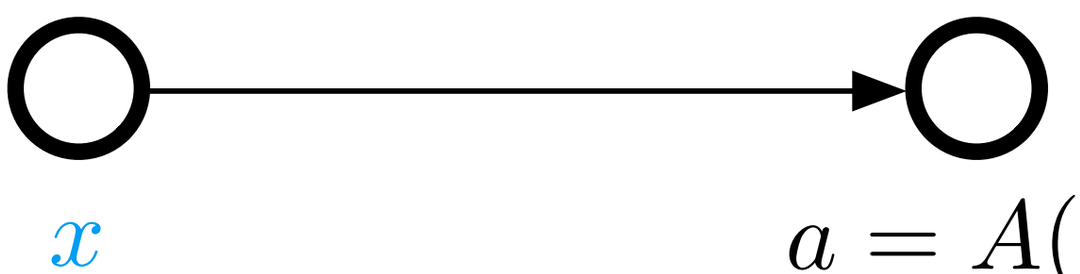


$x$

$$a = A(x)$$

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

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

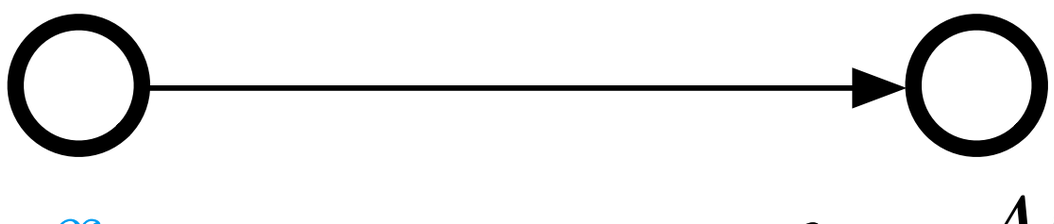


$x$

$$a = A(x)$$

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

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

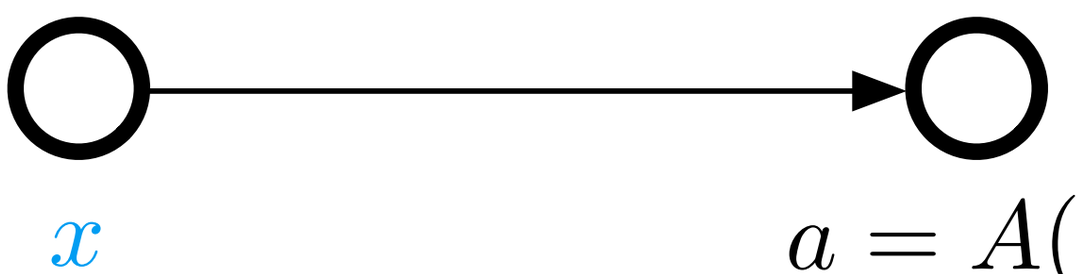


$x$

$$a = A(x)$$

vector-Jacobian product

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



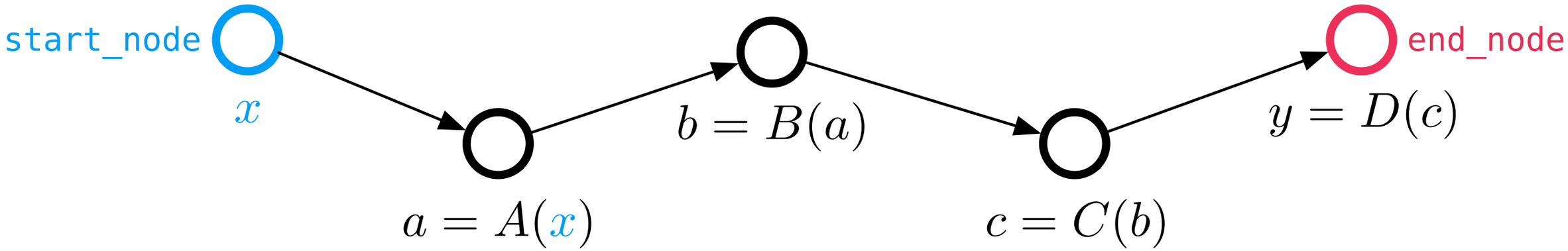
$x$ $\qquad$ $a = A(x)$

```
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)
```

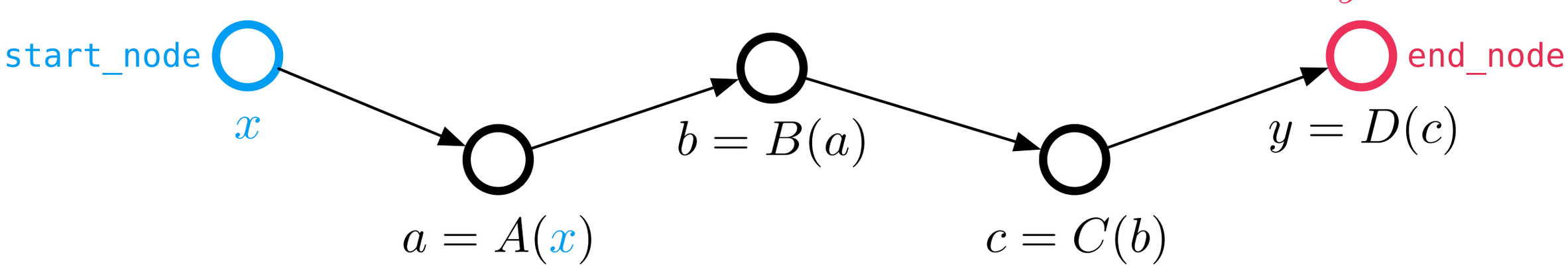
```
def grad_sort(g, ans, vs, gvs, x, axis=-1, kind='quicksort', order=None):
    sort_perm = anp.argsort(x, axis, kind, order)
    return unpermuter(g, sort_perm)
anp.sort.defvjp(grad_sort)
```

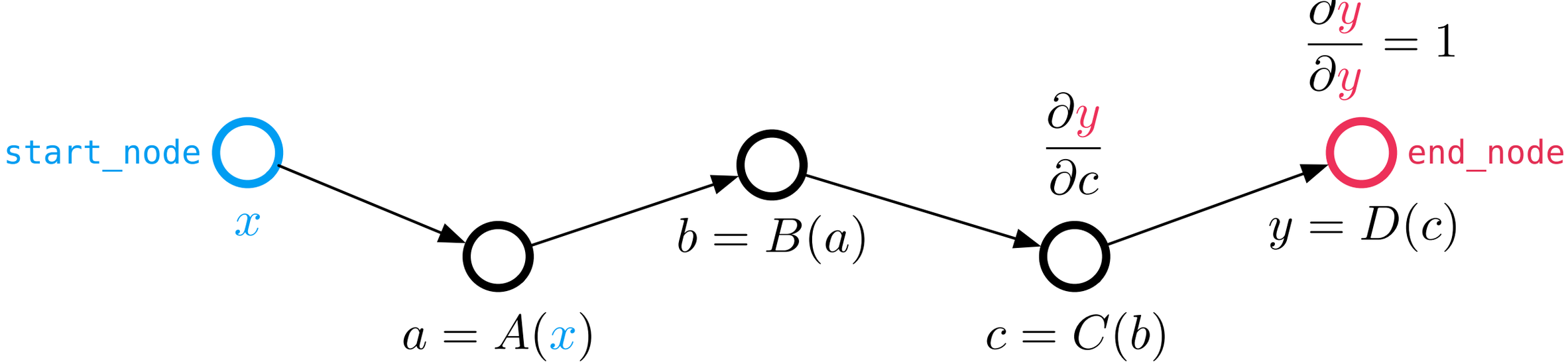
ingredients:

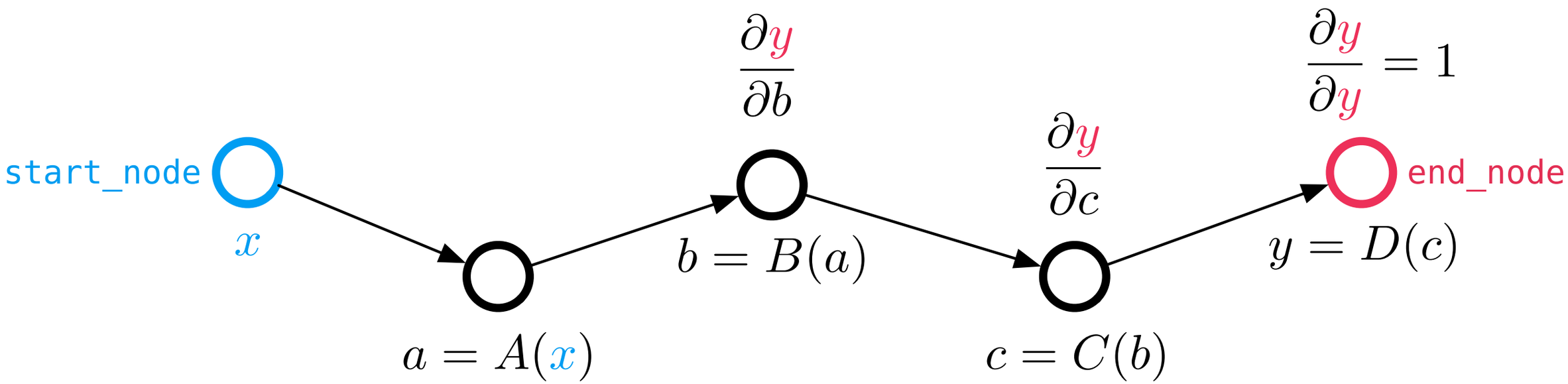
1. tracing composition of primitive functions
2. vector-Jacobian product for each primitive
3. composing VJPs backward

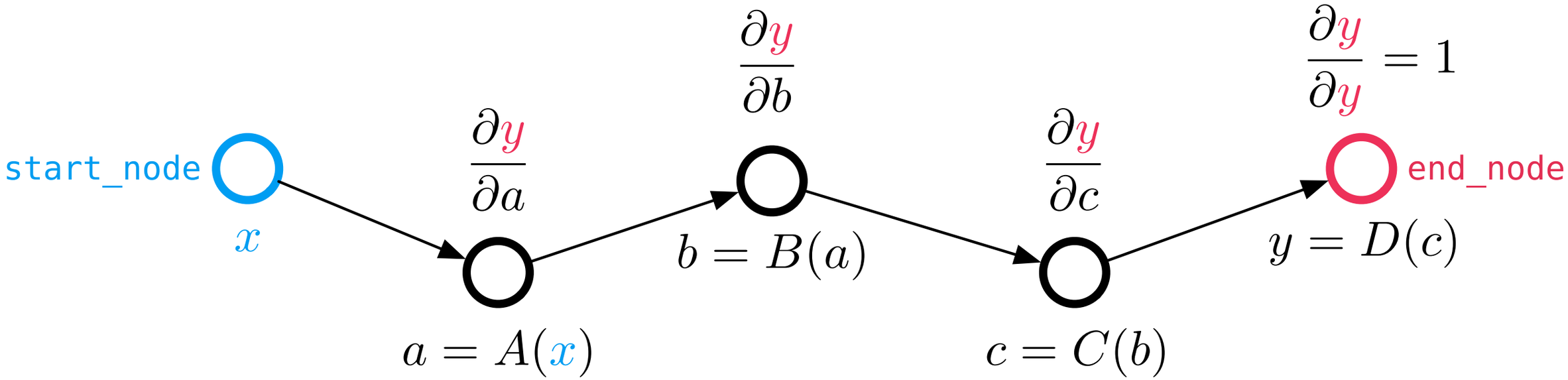

start_node
$x$
$a = A(x)$
$b = B(a)$
$c = C(b)$
$y = D(c)$
end_node

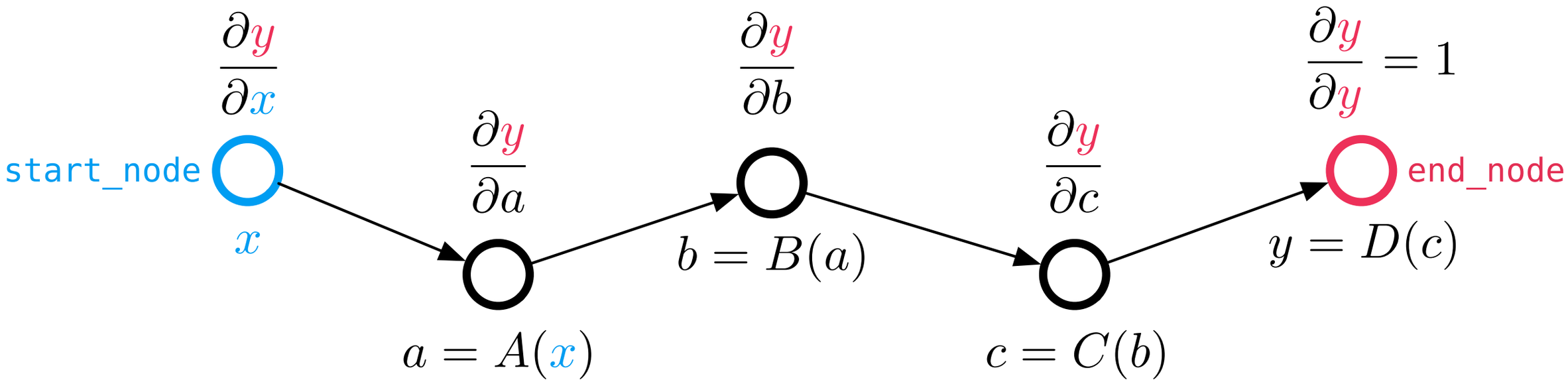
$$\frac{\partial y}{\partial y} = 1$$




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

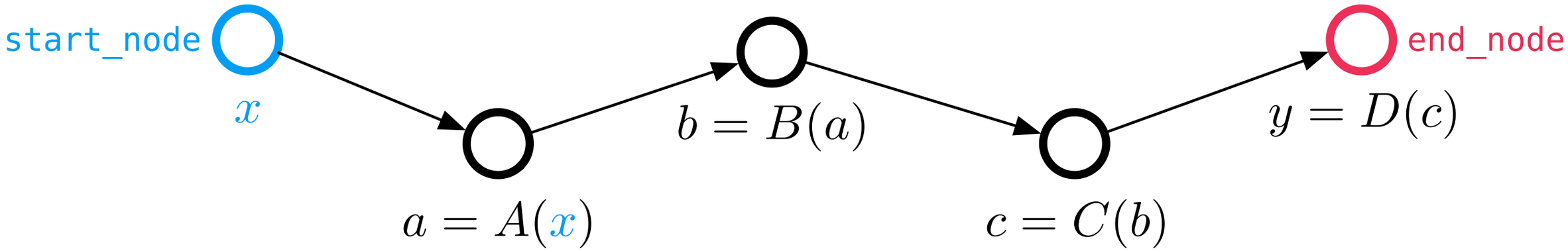

start_node
$x$
$a = A(x)$
$\frac{\partial y}{\partial b}$
$b = B(a)$
$\frac{\partial y}{\partial c}$
$c = C(b)$
$\frac{\partial y}{\partial y} = 1$
end_node
$y = D(c)$

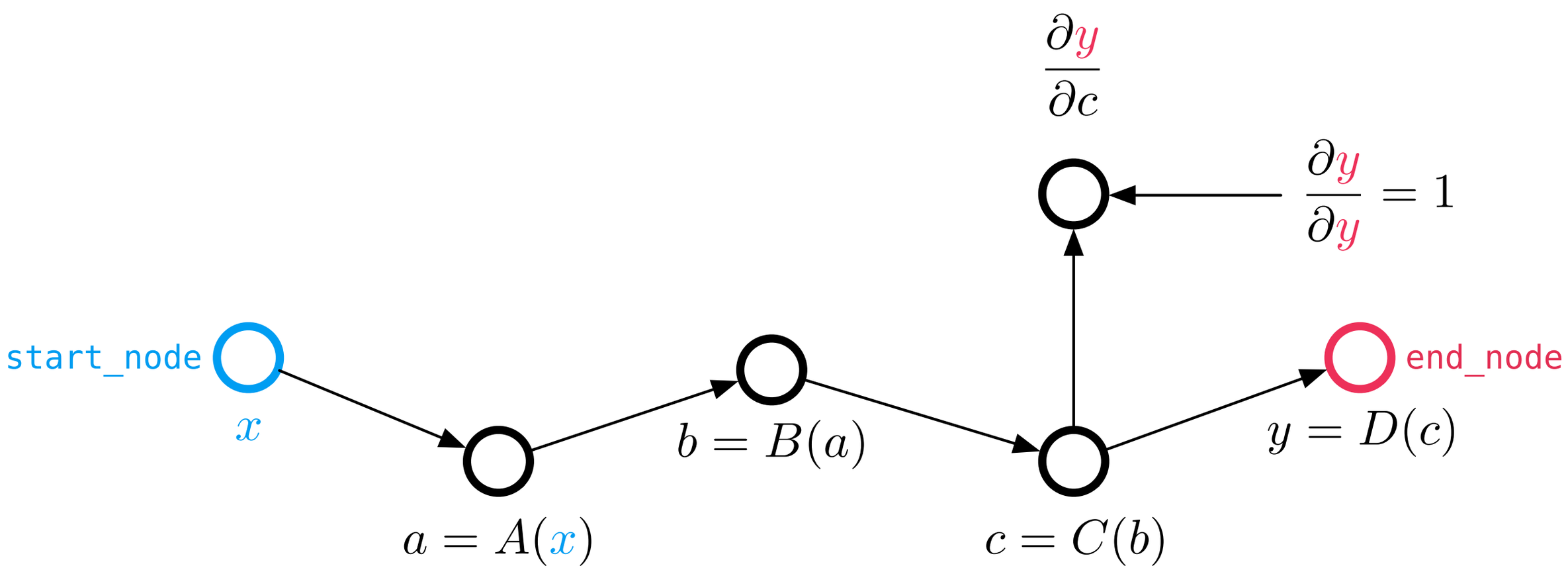

start_node
$x$
$a = A(x)$
$b = B(a)$
$c = C(b)$
$y = D(c)$
end_node
$\frac{\partial y}{\partial a}$
$\frac{\partial y}{\partial b}$
$\frac{\partial y}{\partial c}$
$\frac{\partial y}{\partial y} = 1$

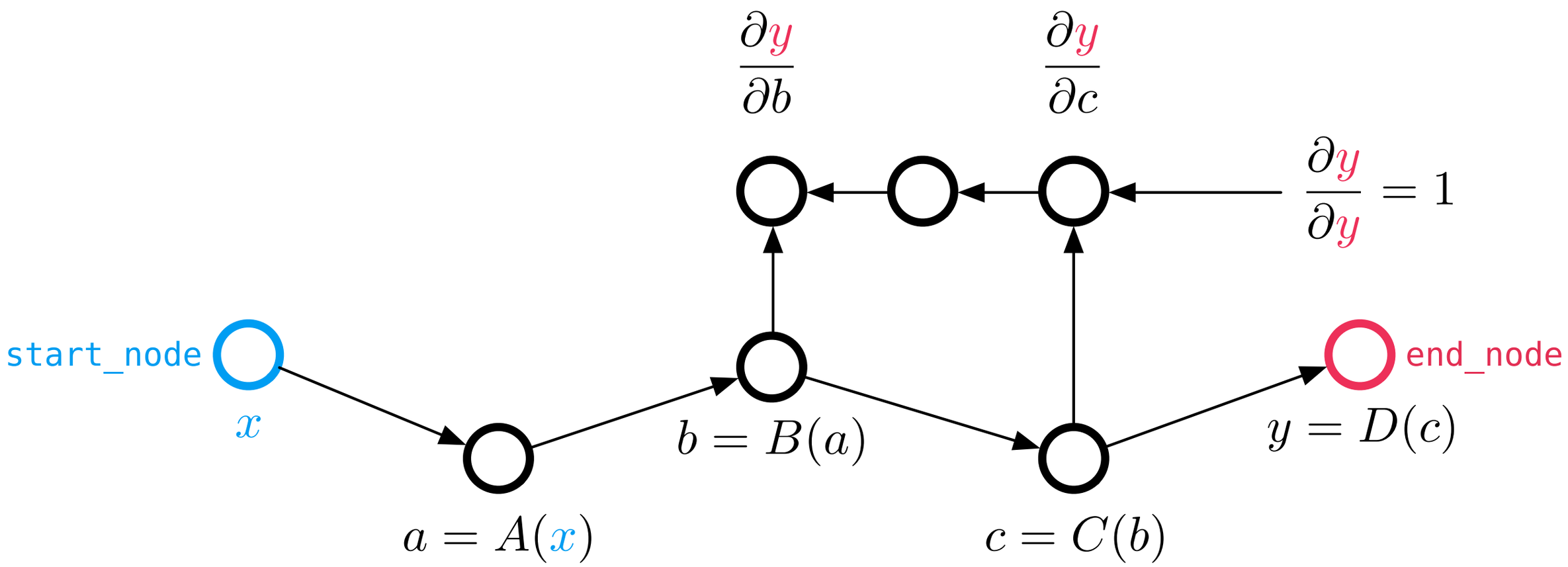

$\frac{\partial y}{\partial x}$
$\frac{\partial y}{\partial a}$
$\frac{\partial y}{\partial b}$
$\frac{\partial y}{\partial c}$
$\frac{\partial y}{\partial y} = 1$
start_node
$x$
$a = A(x)$
$b = B(a)$
$c = C(b)$
$y = D(c)$
end_node

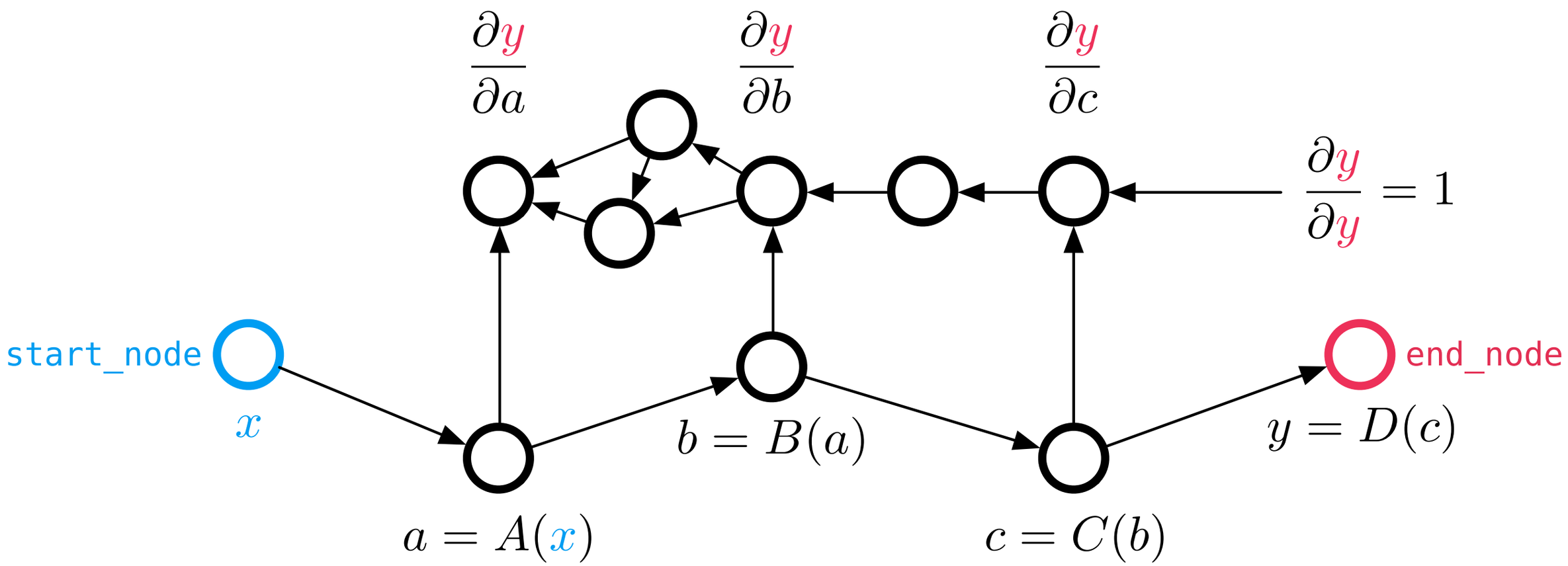
higher-order autodiff just works:
the backward pass can itself be traced

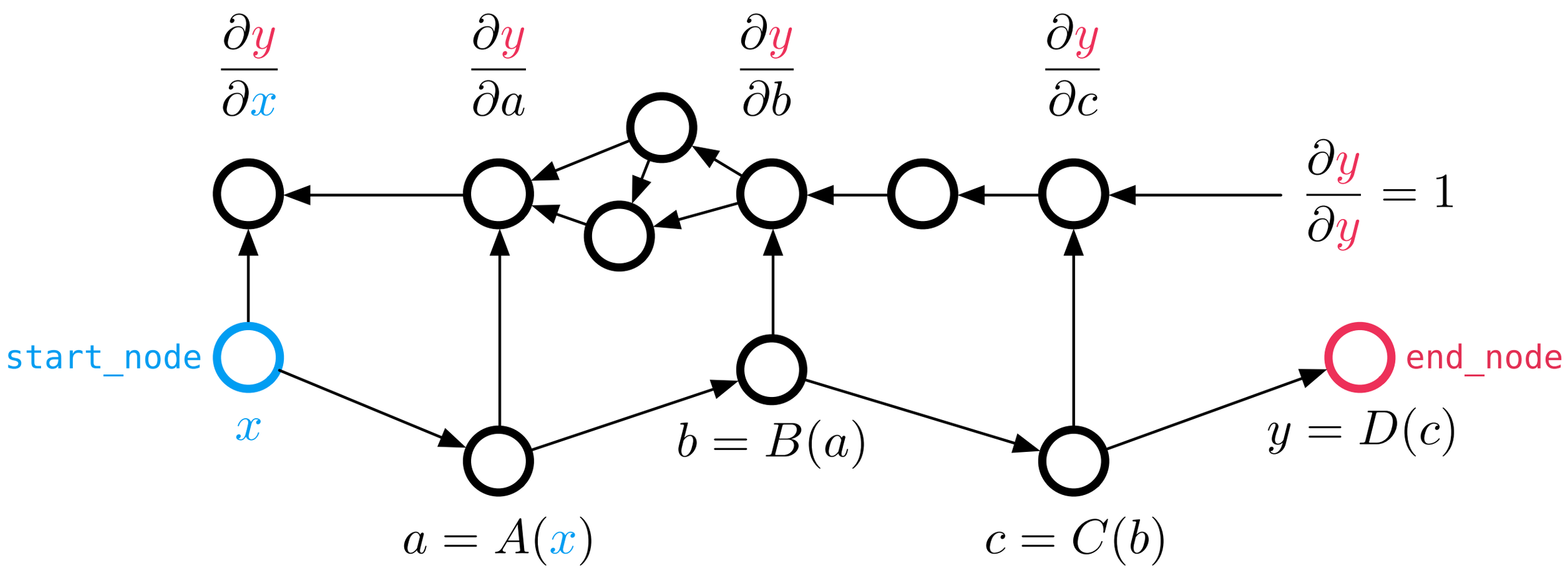
$$\frac{\partial y}{\partial y} = 1$$

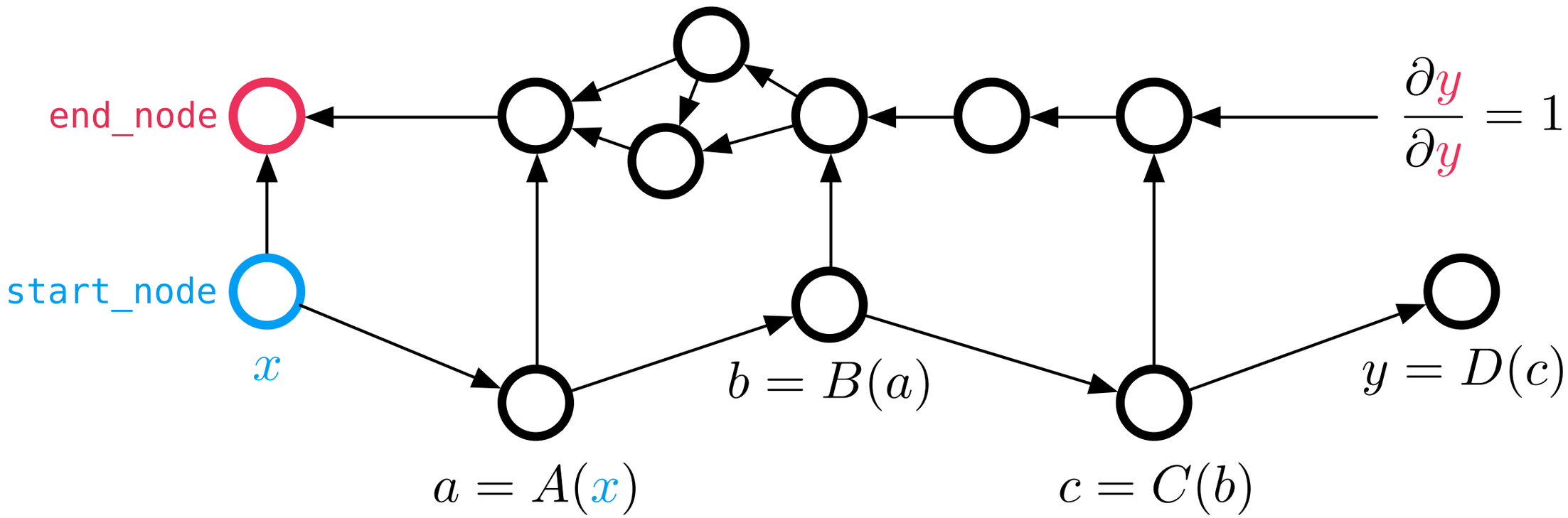



$\frac{\partial y}{\partial c}$
$\frac{\partial y}{\partial y} = 1$
start_node
$x$
$a = A(x)$
$b = B(a)$
$c = C(b)$
$y = D(c)$
end_node


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


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


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


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

```
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
```

```
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
```

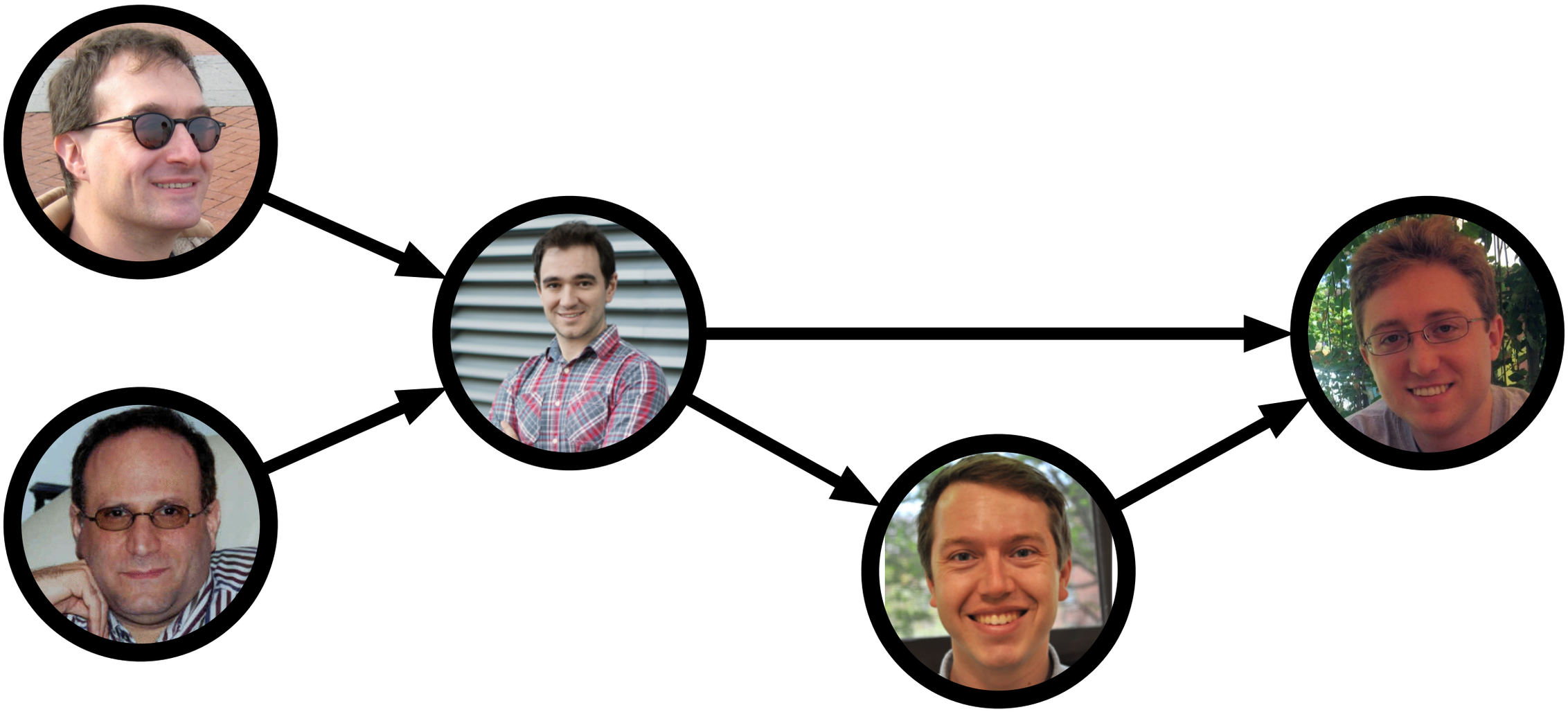
```
def make_vjp(fun, argnum=0):
    def vjp(*args, **kwargs):
        start_node, end_node = forward_pass(fun, args, kwargs, argnum)
        if not isnode(end_node) or start_node not in end_node.progenitors:
            warnings.warn("Output seems independent of input.")
            return lambda g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

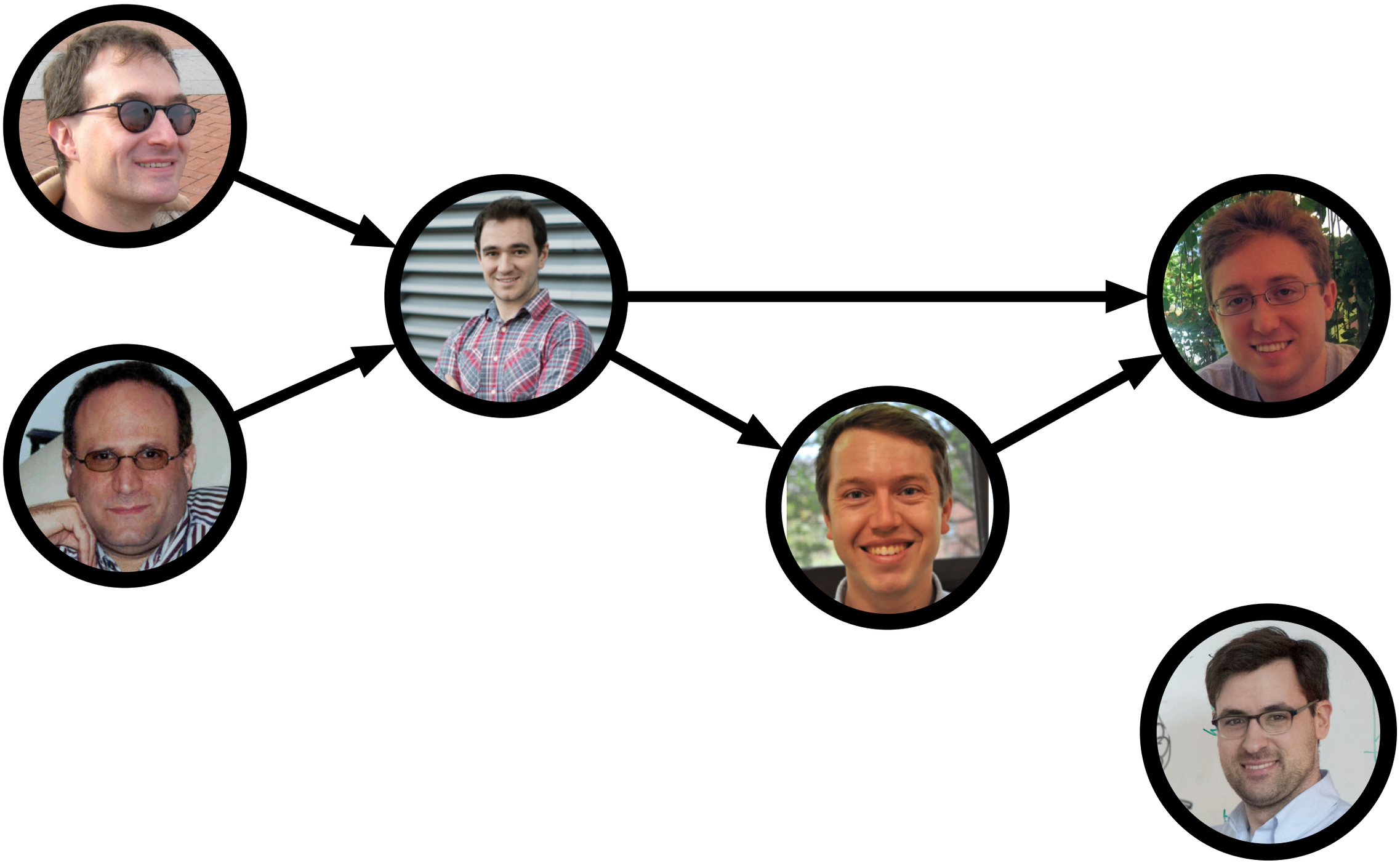
ingredients:

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





# Autodiff generates your exponential family inference code
# +
# Autograd's implementation

HARVARD
INTELLIGENT
PROBABILISTIC
SYSTEMS

HARVARD
School of Engineering
and Applied Sciences