```
import numpy as np
import operator as op
import itertools as it
from functools import partial
from core import primitive, getval, untake

P = primitive

# ----- Operator gradients -----
I = lambda x : x # Identity operator
op.neg = P(op.neg, lambda ans, x    : [op.neg])
op.add = P(op.add, lambda ans, x, y : unbroadcast(ans, x, y, [I, I]))
op.mul = P(op.mul, lambda ans, x, y : unbroadcast(ans, x, y, [lambda g : y * g, lambda g : x * g]))

op.sub = P(op.sub, lambda ans, x, y : unbroadcast(ans, x, y, [I, op.neg]))
op.div = P(op.div, lambda ans, x, y : unbroadcast(ans, x, y, [lambda g : g / y, lambda g : - g * x /
y**2]))
op.pow = P(op.pow, lambda ans, x, y : unbroadcast(ans, x, y, [lambda g : g * y * x ** (y - 1),

                                                              lambda g : g * np.log(x) * x ** y]))

isarray = lambda x : isinstance(getval(x), np.ndarray)
isfloat = lambda x : isinstance(getval(x), float)

def unbroadcast(ans, x, y, funs):
    return [unbroadcast_fun(ans, x, funs[0]),
            unbroadcast_fun(ans, y, funs[1])]

def unbroadcast_fun(ans, x, fun):
    if isfloat(x) and isarray(ans):
        return lambda g : np.sum(fun(g))
    elif isarray(x):
        shape = x.shape
        def new_fun(g):
            result = fun(g)
            while result.ndim > len(shape):
                result = np.sum(result, axis=0)
            for axis, size in enumerate(shape):
                if size is 1:
                    result = np.sum(result, axis, keepdims=True)
            return result
        return new_fun
    else:
        return fun

# ----- Numpy gradients -----

np.abs    = P(np.abs,    lambda ans, x : [lambda g : np.sign(x) * g])
np.exp    = P(np.exp,    lambda ans, x : [lambda g : ans * g])
np.log    = P(np.log,    lambda ans, x : [lambda g : g / x])
np.sin    = P(np.sin,    lambda ans, x : [lambda g : g * np.cos(x)])
np.cos    = P(np.cos,    lambda ans, x : [lambda g : - g * np.sin(x)])
np.tan    = P(np.tan,    lambda ans, x : [lambda g : g / np.cos(x) **2])
np.sinh   = P(np.sinh,   lambda ans, x : [lambda g : g * np.cosh(x)])
np.cosh   = P(np.cosh,   lambda ans, x : [lambda g : g * np.sinh(x)])
np.tanh   = P(np.tanh,   lambda ans, x : [lambda g : g / np.cosh(x) **2])
```

```
53    np.square = P(np.square, lambda ans, x : [lambda g : g * 2 * x])
54    np.sign   = P(np.sign,   lambda ans, x : [lambda g : 0.0])
55    np.full   = P(np.full,   lambda ans, shape, fill_value : [None, lambda g :  np.sum(g)])
56    np.reshape     = P(np.reshape,     lambda ans, x, shape, order=None : [lambda g : np.reshape(g, x.shape,
      order=order)])
57    np.ravel       = P(np.ravel,       lambda ans, x,        order=None : [lambda g : np.reshape(g, x.shape,
      order=order)])
58    np.expand_dims = P(np.expand_dims, lambda ans, x, axis              : [lambda g : np.squeeze(g, axis)])

59    np.squeeze     = P(np.squeeze,     lambda ans, x, axis              : [lambda g : np.repeat(g,
      x.shape[axis], axis)])
60    np.repeat      = P(np.repeat,      lambda ans, x, shape, axis       : [lambda g : np.sum(g, axis,
      keepdims=True)])
61    np.transpose   = P(np.transpose,   lambda ans, x                    : [lambda g : np.transpose(g)])

62    np.split       = P(np.split,       lambda ans, A, idxs, axis=0      : [lambda g : np.concatenate(g,
      axis=axis)])
63
64    def make_grad_np_sum(ans, x, axis=None, keepdims=False):
65        if not isarray(x):
66            return [I]
67        shape = x.shape
68        if axis is None:
69            return [lambda g : np.full(shape, g)]
70        else:
71            if keepdims:
72                return [lambda g : np.repeat(g, shape[axis], axis)]
73            else:
74                return [lambda g : np.repeat(np.expand_dims(g, axis),
75                                             shape[axis], axis)]
76    np.sum = P(np.sum, make_grad_np_sum)
77
78    def make_grad_np_mean(ans, x, axis=None, keepdims=False):
79        if not isarray(x):
80            return [I]
81        shape = x.shape
82        if axis is None:
83            return [lambda g : np.full(shape, g) / np.prod(shape)]
84        else:
85            if keepdims:
86                return [lambda g : np.repeat(g, shape[axis], axis) / shape[axis]]
87            else:
88                return [lambda g : np.repeat(np.expand_dims(g, axis),
89                                             shape[axis], axis) / shape[axis]]
90    np.mean = P(np.mean, make_grad_np_mean)
91
92    def make_grad_np_max(ans, x):
93        def gradfun(g):
94            idxs = np.argmax(getval(x))
95            return untake(g, np.unravel_index(idxs, x.shape))
96        return [gradfun]
97    np.max = P(np.max, make_grad_np_max)
98
99    def make_grad_np_dot(ans, A, B):
100       def grad_np_dot_A(g):
101           if B.ndim is 2:
102               return np.dot(g, B.T)
103           elif A.ndim is 2:
104               return np.outer(g, B)
105           else:
106               return g * B
107       def grad_np_dot_B(g):
108           if A.ndim is 2:
109               return np.dot(A.T, g)
110           elif B.ndim is 2:
111               return np.outer(A, g)
112           else:
113               return g * A
114       return [grad_np_dot_A, grad_np_dot_B]
115   np.dot = P(np.dot, make_grad_np_dot)
116
117   def make_grad_np_concatenate(ans, arr_list, axis=0):
118       def grad_np_concatenate(g):
119           idxs = np.cumsum([a.shape[axis] for a in getval(arr_list)[:-1]])
120           return np.split(g, idxs, axis=axis)
121       return [grad_np_concatenate]
```

```
122     np.concatenate = P(np.concatenate, make_grad_np_concatenate)
123
124     # ----- Special list constructor -----
125
126     class ArgnumGrad(object):
127         def __init__(self, fun_with_argnum):
128             self.fun = fun_with_argnum
129         def __getitem__(self, argnum):
130             return partial(self.fun, argnum)
131
132     def kylist(*args):
133         return list(args)
134     kylist = primitive(kylist, lambda ans, *args : ArgnumGrad(lambda argnum, g : g[argnum]))

135
136     # Wrap the concatenation function to automatically wrap the list into a kylist.
137     unwrapped_np_concatenate = np.concatenate
138     def concatwrapper(*args, **kwargs):
139         args = (kylist(*(args[0])),) + args[1:]
140         return unwrapped_np_concatenate(*args, **kwargs)
141     np.concatenate = concatwrapper
```