```
from __future__ import absolute_import
from autograd.core import primitive, Node, getval, zeros_like, cast
from six.moves import zip
import six


class TupleNode(Node):
    __slots__ = []
    def __getitem__(self, idx):
        return tuple_take(self, idx)
    def __len__(self):
        return len(self.value)

    @staticmethod
    def zeros_like(value):
        return tuple([zeros_like(item) for item in getval(value)])

    @staticmethod
    def sum_outgrads(outgrads):
        return primitive_sum_tuples(*outgrads)

Node.type_mappings[tuple] = TupleNode

@primitive
def primitive_sum_tuples(*tuples):
    return tuple([primitive_sum(elements) for elements in zip(*tuples)])
primitive_sum_tuples.gradmaker = lambda *args : lambda g : g

@primitive
def tuple_take(A, idx):
    return A[idx]
def make_grad_tuple_take(ans, A, idx):
    return lambda g : tuple_untake(g, idx, A)
tuple_take.defgrad(make_grad_tuple_take)

@primitive
def tuple_untake(x, idx, template):
    result = list(zeros_like(template))
    result[idx] = x
    return tuple(result)
tuple_untake.defgrad(lambda ans, x, idx, template : lambda g : tuple_take(g, idx))
tuple_untake.defgrad_is_zero(argnums=(1, 2))


class ListNode(Node):
    __slots__ = []
    def __getitem__(self, idx):
        return list_take(self, idx)
    def __len__(self):
        return len(self.value)

    @staticmethod
    def zeros_like(value):
        return [zeros_like(item) for item in getval(value)]

    @staticmethod
    def sum_outgrads(outgrads):
        return primitive_sum_lists(*outgrads)

    @staticmethod
    def cast(value, example):
        return cast(value, cast_to_list)

def cast_to_list(x):
    return list(x)

Node.type_mappings[list] = ListNode

@primitive
def primitive_sum_lists(*lists):
    return [primitive_sum(elements) for elements in zip(*lists)]
primitive_sum_lists.gradmaker = lambda *args : lambda g : g

@primitive
def list_take(A, idx):
    return A[idx]
def make_grad_list_take(ans, A, idx):
    return lambda g : list_untake(g, idx, A)
list_take.defgrad(make_grad_list_take)

@primitive
def list_untake(x, idx, template):
        result = list(zeros_like(template))
        result[idx] = x
        return result
list_untake.defgrad(lambda ans, x, idx, template : lambda g : list_take(g, idx))
list_untake.defgrad_is_zero(argnums=(1, 2))


class DictNode(Node):
        __slots__ = []
        def __getitem__(self, idx):
                return dict_take(self, idx)
        def __len__(self):
                return len(self.value)
        def __iter__(self):
                return self.value.__iter__()

        @staticmethod
        def zeros_like(self):
                return {k : zeros_like(v) for k, v in six.iteritems(getval(self))}

        @staticmethod
        def sum_outgrads(outgrads):
                return primitive_sum_dicts(*outgrads)

        @staticmethod
        def cast(value, example):
                return cast(value, cast_to_dict)

def cast_to_dict(x):
        return dict(x)

Node.type_mappings[dict] = DictNode

@primitive
def primitive_sum_dicts(*dicts):
        """Takes a list of dicts having identical keys.
             Returns a new dict whose values are the sum over all input dicts."""
        # assert set(dicts[0]) == set(dicts[0]).intersection(*dicts)
        keys = dicts[0]
        return {k : primitive_sum([dict[k] for dict in dicts]) for k in keys}
primitive_sum_dicts.gradmaker = lambda *args : lambda g : g
```

```
@primitive
def dict_take(A, idx):
    return A[idx]
def make_grad_dict_take(ans, A, idx):
    return lambda g : dict_untake(g, idx, A)
dict_take.defgrad(make_grad_dict_take)

@primitive
def dict_untake(x, idx, template):
    result = dict(zeros_like(template))
    result[idx] = x
    return result
dict_untake.defgrad(lambda ans, x, idx, template : lambda g : dict_take(g, idx))
dict_untake.defgrad_is_zero(argnums=(1, 2))

primitive_summers = {
    list: primitive_sum_lists,
    tuple: primitive_sum_tuples,
    dict: primitive_sum_dicts,
}

def primitive_sum(container):
    thetype = type(container[0])
    if thetype in primitive_summers:
        return primitive_summers[thetype](*container)
    return sum(container[1:], container[0])
```