Recall: implementing forward mode using operator overloading

```python
class dfloat:
    val : float
    dval : float

dfloat operator+(dfloat x, dfloat y):
    return dfloat(x.val+y.val, x.dval+y.dval)

dfloat operator-(dfloat x, dfloat y):
    return dfloat(x.val-y.val, x.dval-y.dval)

dfloat operator*(dfloat x, dfloat y):
    return dfloat(x.val*y.val,
                  x.dval*y.val+y.val*x.dval)
```
Recall: implementing forward mode using operator overloading

what is the equivalent for reverse mode?

```python
class dfloat:
    val : float
    dval : float

dfloat operator+(dfloat x, dfloat y):
    return dfloat(x.val+y.val, x.dval+y.dval)

dfloat operator-(dfloat x, dfloat y):
    return dfloat(x.val-y.val, x.dval-y.dval)

dfloat operator*(dfloat x, dfloat y):
    return dfloat(x.val*y.val,
                  x.dval*y.val+y.val*x.dval)
```
One possible way to implement reverse mode

tape = []
**Class** rfloat:
  val = 0
  dval = 0
  tape_id = -1
One possible way to implement reverse mode

tape = []

Class rfloat:
  val = 0
  dval = 0
  tape_id = -1

rfloat operator+(rfloat a, rfloat b):
  r = rfloat()
  r.val = a.val + b.val
  r.tape_id = len(tape)
  tape.append((a, b), '+', r)
  return r
One possible way to implement reverse mode

tape = []

Class rfloat:
    val = 0
dval = 0
tape_id = -1

rfloat operator+(rfloat a, ...):
r = rfloat()
r.val = a.val + b.val
r.tape_id = len(tape)
tape.append((a, b), '+', r)
return r

def reverse(out : rfloat, dout):
    out.dval = dout
    for t in reversed(tape):
        if t.op == '+':
            t.left.dval += t.r.dval
            t.right.dval += t.r.dval
        elif t.op == '-':
            ...
        ...
The “taping” approach records the computational graph and revert it

```python
tape = []
Class rfloat:
    val = 0
dval = 0
tape_id = -1

def reverse(out : rfloat, dout):
    out.dval = dout
    for t in reversed(tape):
        if t.op == ‘+’:
            t.left.dval += t.r.dval
            t.right.dval += t.r.dval
        elif t.op == ‘-’:
            ...
```

the tape is sometimes called “Wengert list”, though it probably wasn’t invented by Wengert
A slight optimization of taping

tape = []

```python
class rfloat:
    val = 0
dval = 0
tape_id = -1

def operator_plus(self, a, b):
    r = rfloat()
    r.val = a.val + b.val
    r.tape_id = len(tape)
    // Store partial derivatives
    // of the plus function
    tape.append([(1, a.tape_id),
                 (1, b.tape_id)], r)
    return r
```

def reverse(out: rfloat, dout):
    out.dval = dout
    for t in reversed(tape):
        for i in t.inputs:
            tape[i[1]].dval += i[0] * t.r.dval

directly store the partial derivatives of the edges in the tape
My taping-based AD library

https://github.com/BachiLi/had
What is the difference between taping and the HW2 approach?

\[
\begin{align*}
\text{Taping} & : & \text{Source-to-source} \\
\text{v.s.} & : & \\
\text{DTf}(x, y, db): & & \\
& & a = g(x, y) \\
& & b = h(a) \\
& & da = DTh(a, db) \\
& & dx, dy = DTg(x, y, da) \\
& & \text{return } dx, dy
\end{align*}
\]
Taping is likely much slower

in my experience, it can be >100x slower in certain cases

rfloat operator+(rfloat a, ...):
  r = rfloat()
  r.val = a.val + b.val
  r.tape_id = len(tape)
  // Store partial derivatives
  // of the plus function
  tape.append([(1, a.tape_id),
               (1, b.tape_id)], r)

return r

these are constants

but taping is sometimes easier to implement,
and it might compile faster

def reverse(out : rfloat, dout):
  out.dval = dout
  for t in reversed(tape):
    for i in t.inputs:
      tape[i[1]].dval += 
        i[0] * t.dval

so we should not need to
emit multiplication instruction here
Taping can easily handle control flows if we don’t care about efficiency

```python
def f(x : rfloat) -> rfloat:
    ...
    f(x - 1)  # can easily handle recursion
    ...
    while ...:  # loops are flatten to a linear tape
        ...
```
Taping can be seen as a "tracing JIT" compilation

used in, e.g., TraceMonkey (JS), PyPy (Python), LuaJIT (Lua)

primal program

```python
def f(x, y):
    ...
    f(x - 1, y)
    ...
    while ...:
        ...
    return ...
```

Taping and source-to-source can both be seen as “staged compilation”

```
def f(x, y):
    ...
    f(x - 1, y)
    ...
    while ...
        ...
    return ...
```

```
def DTf(...):
    ...
```

```
def DTf(...):
    ...
    compile
    mov ...
    add ...
    execute
    DTf(…)
```

Taping (trace)

Execute

Compile

(source to source)

(machine code)

aka partial evaluation, or specialization

https://en.wikipedia.org/wiki/Multi-stage_programming
https://en.wikipedia.org/wiki/Partial_evaluation
https://en.wikipedia.org/wiki/Partial_template_specialization
There is a full spectrum of approaches between taping and source-to-source

how much differentiation is done in compile time
For example, we can compile the tape into machine code

def f(x, y):
    ...
f(x - 1, y)
    ...
while ...:
    ...
return ...

this allows us to:
1. optimize the derivative code
2. reuse the generated tape for different inputs
   (assuming the control flow doesn’t change)
For example, we can **specialize** the source-to-source transformation

```python
def f(x, y):
    ...
    f(x - 1, y)
    ...
    while ...
    ...
    return ...

def DTf(...):
    ...

specialize to:

def DTf_y1(...):
    ...

compile:
```
For example, we can add a while loop primitive for taping:

```
def f(x, y):
    ...
    f(x - 1, y)
    ...
MyWhileLoop(...):
    ...
return ...
```

taping (trace)

this allows us to:
1. optimize the derivative code
2. reuse the generated tape for different inputs (now the control flow can change! but now the user needs to differentiate host code while loop and the AD while loop)

compile

```
mov ...
add ...
```

(machine code)

execute

```
DTf(...)```
Deep learning systems are a hybrid between taping and source-to-source.

(the early PyTorch vs TensorFlow design fight was artificial, they are basically the same)

Nodes of the graph are arrays, not scalars.

individual computations call to manually written C/CUDA code.

```python
x = torch.tensor(...)  
y = 2 * x  
for i in range(10):  
y = torch.sin(y)
```
Deep learning systems are a hybrid between taping and source-to-source

(the early PyTorch vs TensorFlow design fight was artificial, they are basically the same)

\[ x = \text{torch.tensor}(\ldots) \]
\[ y = 2 \times x \]
\[ \text{for } i \text{ in range}(10): \]
\[ y = \text{torch.sin}(y) \]

individual computations still call to manually written C/CUDA code
Take home message:
AD is an (embedded) domain-specific language/compiler
design problem

• Taping vs source-to-source boils down to staged compilation

• Design choices involve:
  • how much compiler optimization you want to apply to the code
    (compile time vs runtime tradeoffs)
  • deep learning systems avoid slow CUDA compilation in every training
  • specialization allows source-to-source to be even faster for certain inputs
  • language features you plan to support (e.g., static vs dynamic languages)
  • how you want your frontends, intermediate representations, backends to be