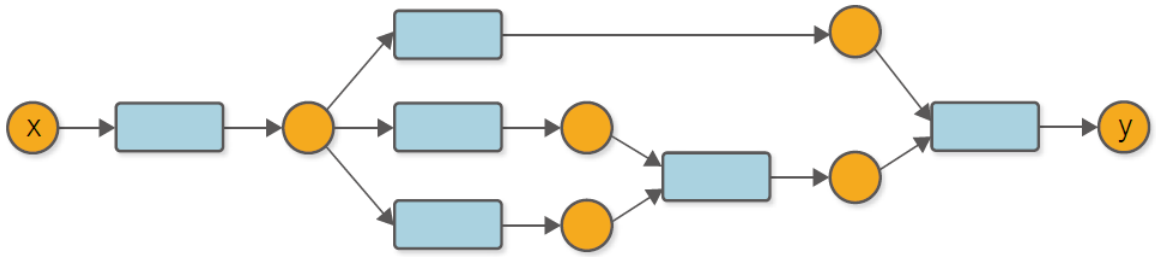
제2 고지 : 자연스러운 코드로

STEP 15 : 복잡한 계산 그래프(이론 편)

그림 15-1 일직선 계산 그래프

현재까지는 위의 그림 처럼 한 줄로 늘어선 계산 그래프를 다뤘다. 하지만 변수와 함수가 꼭 그렇게 구성된다는 보장은 없다.
예를들어, 아래와 같이 동일한 변수를 반복사용하거나, 여러 변수를 입력으로 받는 등 복잡한 계산 그래프를 구성할 수 있다.
그러나 **현재의 Dezero는 복잡한 계산 그래프의 역전파를 제대로 구현해 낼 수 없다.**

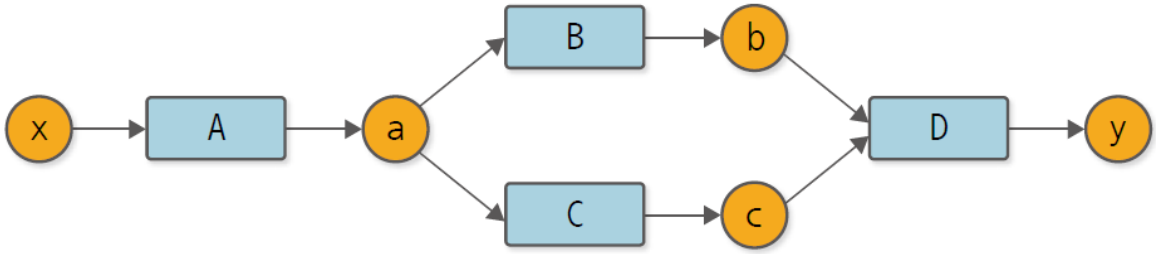
그림 15-2 더 복잡하게 연결된 계산 그래프 예



15.1 역전파의 올바른 순서

어떤 문제가 있는지 우선 아래의 그래프를 살펴보면,

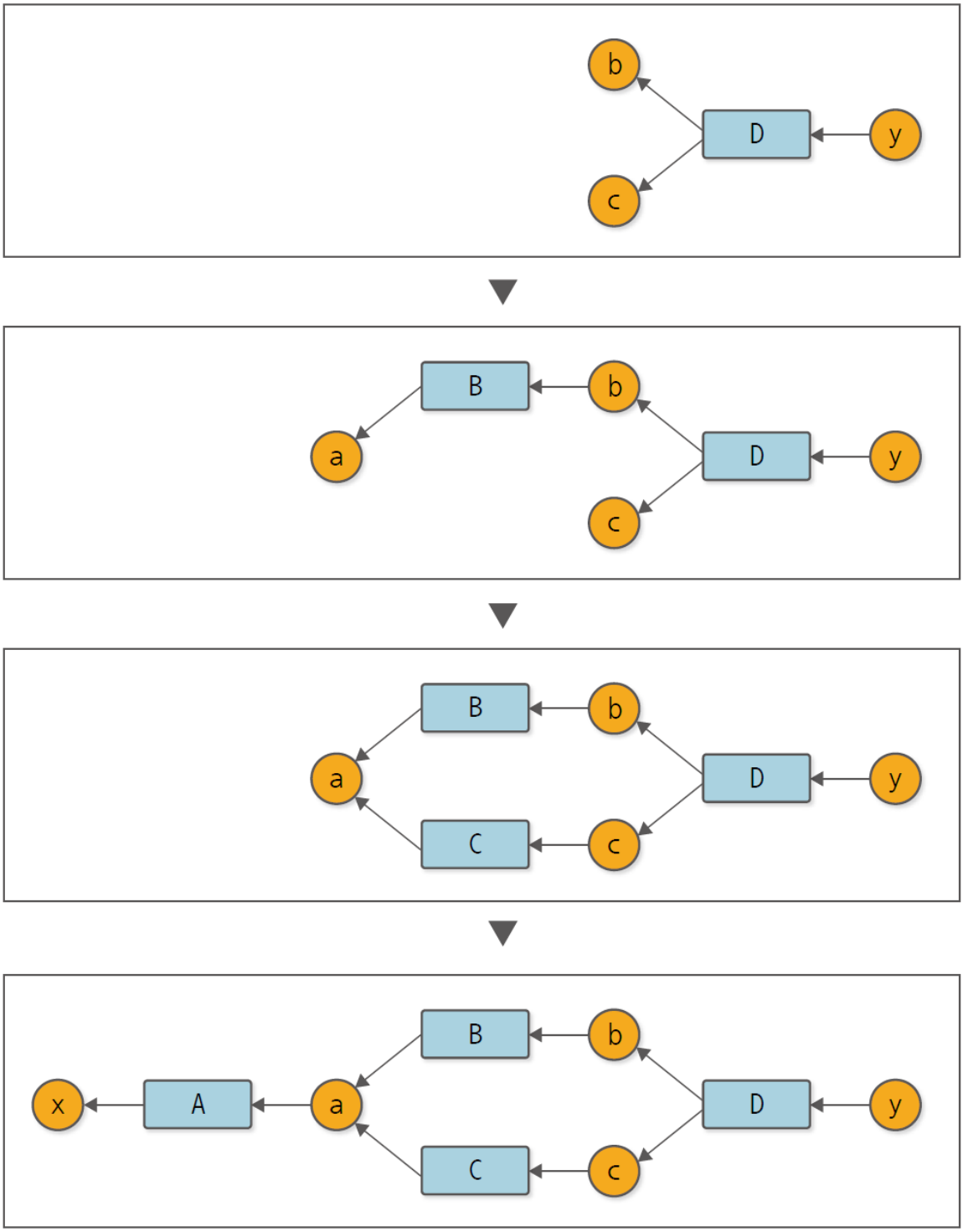
그림 15-3 중간에 분기했다가 다시 합류하는 계산 그래프



이전 스텝에서 살펴봤듯이, **동일한 변수를 반복사용**하면 역전파때는 출력쪽에서 전파되는 미분값을 **합**해서 전파해야한다. 즉, `b` 와 `c` 의 `grad` 가 둘 다 전파되야 `a` 로 미분값을 전파 할수 있다.

조금 더 구체적으로 살펴보면, 역전파의 순서는 다음과 같아야한다.

그림 15-4 올바른 역전파의 순서



함수의 관점에서 본다면, `D,B,C,A` 또는 `D,C,B,A` 순서로 전파되야 한다.(`B` , `C` 의 순서는 상관없다.) 여기서 핵심은 **`B` , `C` 의 역전파를 모두 끝내고 나서 `A` 를 전파해야 한다는 것이다.**

15.2 현재의 Dezero

현재의 Dezero 구현을 보면, 처리할 함수의 후보를 `func` 스택에 순차적으로 담고 (`func.append(x.creator)`), 꺼낸다(`func.pop()`)

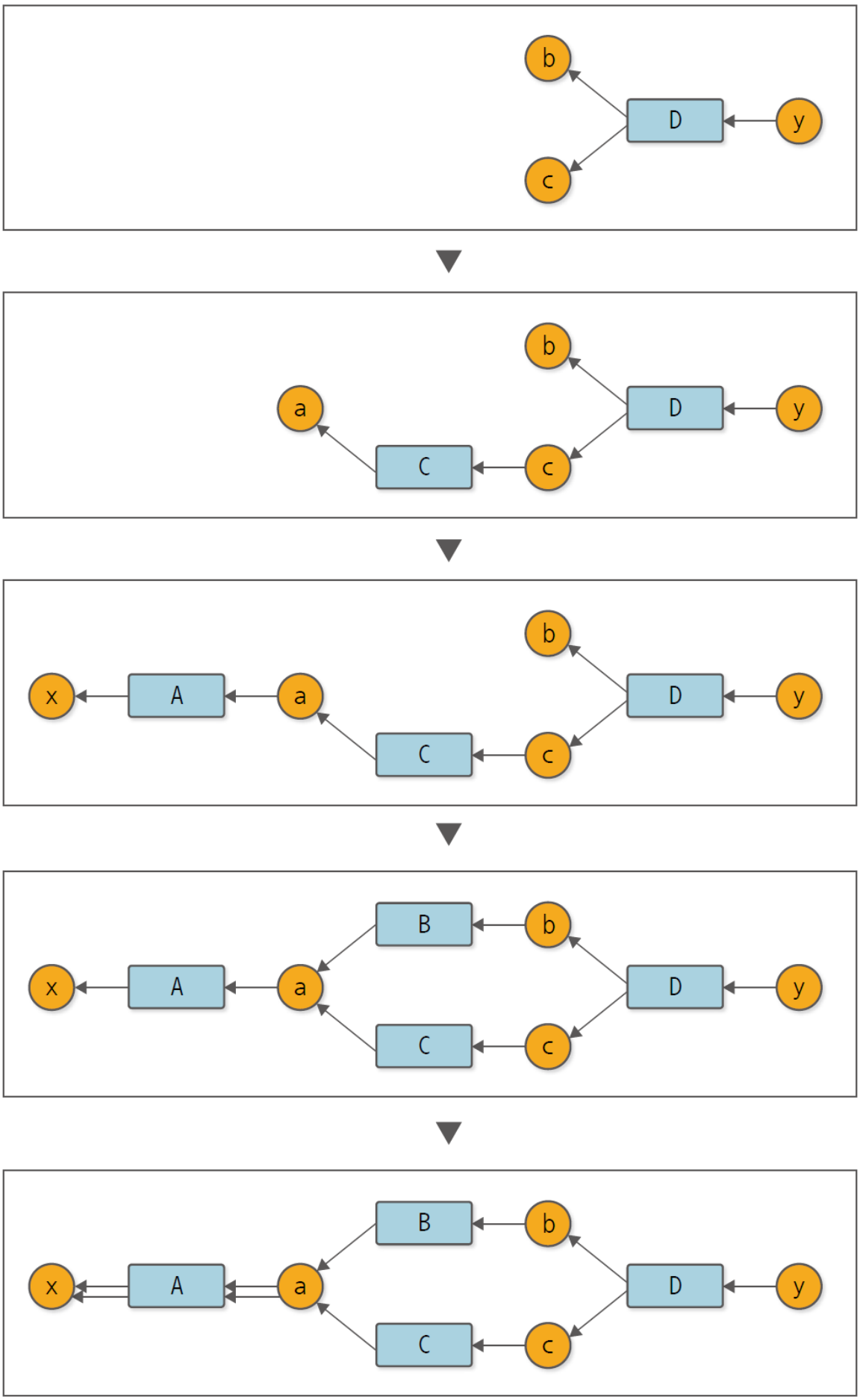
```
class Variable:
    def __init__(self, data: np.ndarray) -> None:
        if data is not None:
            if not isinstance(data, np.ndarray):
                raise TypeError(f"{type(data)}은(는) 지원하지 않습니다.")
```

```
        self.data = data
        self.grad = None  # gradient
        self.creator = None  # creator

    def set_creator(self, func) -> None:
        self.creator = func

    def backward(self):
        """
        자동 역전파 (반복)
        """
        if self.grad is None:
            self.grad = np.ones_like(self.data)
        funcs = [self.creator]
        while funcs:
            ######################
            # NOTE:
            f = funcs.pop()
            ######################
            gys = [output.grad for output in f.outputs] # 1. 순전파의 결과가 **여러개의 출력인 경우**를 처리
            gxs = f.backward(*gys) # 2. 역전파 기준 **여러 개의 입력(=순전파의 여러 개 출력)** 을 처리.
            if not isinstance(gxs,tuple): # 3. 역전파 **결과값이 하나인 경우(=역전파의 출력이 1개인 경우)** 튜플로 변환.
                gxs = (gxs,)
            for x,gx in zip(f.inputs,gxs): # 4. **역전파 결과가 여러개의 출력인 경우** 각각 대응
                #  첫 grad를 설정시에는 `그대로` 출력하고,
                if x.grad is None :
                    x.grad = gx
                # 다음 미분은 기존 미분 값에 `더해준다.`
                else :
                    x.grad +=gx

                if x.creator is not None:
                    ######################
                    # NOTE:
                    funcs.append(x.creator)  # 하나 앞의 함수를 리스트에 추가한다
                    ######################
```

이 과정을 살펴보면 아래의 그림과 같이, B 와 C 모두 역전파가 완료되지 않았음에도 불구하고, 당장의 연결된 함수만을 고려하여 전파하여 D,C,A,B,A 처리된다.

또한, A 의 역전파가 두번 일어 난다.

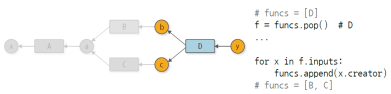
그림 15-5 현재의 DeZero에서의 역전파 흐름



조금 더 구체적으로 살펴보면,

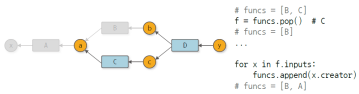
1. `func` 에 `D` 가 추가된 상태 (`[D]`) 로 시작하여, 그 다음 `D` 의 입력변수 `B,C` 가 `funcs` 에 담기게 된다. (`[B,C]`)
2. `func` 에서 `C` 가 꺼내지고, `C` 와 연결된 `A` 가 담기게 된다. (`[B,A]`)
3. 그 다음 순서는 `C` 여야 올바른 역전파가 이뤄지지만, 현재의 Dezero는 `func` 에서 `A` 가 꺼내진다. (`[B]`) (**문제 : 올바른 역전파 순서를 유지하지 못한다**)
4. `func` 에서 `B` 를 꺼내고, 연결된 `A` 를 담는다. (`[A]`)
5. `func` 에서 `A` 를 꺼내고, 마무리 한다. (**문제 : 역전파가 두번 이뤄진다**)

그림 15-6 함수 D의 역전파(오른쪽은 funcs 리스트를 처리하는 코드)



```
# funcs = [D]
f = funcs.pop()  # D
...
for x in f.inputs:
    funcs.append(x.creator)
# funcs = [B, C]
```

그림 15-7 함수 C의 역전파와 대응 코드



```
# funcs = [B, C]
f = funcs.pop()  # C
# funcs = [B]
...
for x in f.inputs:
    funcs.append(x.creator)
# funcs = [B, A]
```

..

15.3 함수 우선순위

이 문제를 해결 하기 위해서는 **함수의 우선순위**를 할당 할 수 있어야한다. 주목해야 할것은 **순전파시 함수가 변수를 만들어내고 있다는 것이다.**
즉, 아래의 그림과 같이, **부모-자식 관계**를 띄고 있는 것을 활용하여 **세대** 를 나누어 **세대수가 큰 순서대로** 꺼내어 계산하면 된다.

그림 15-8 순전파 때의 함수와 변수 '세대'