ピュアなふりをする グラフ（差分リスト）

Jin SANO

背景 | 差分リスト

Imntal のデータ構造を扱いたい

差分リストは

1. 連結が O(1) で，他の操作はリストと同等（より強力）
2. Imntal で用いられるデータ構造の中では一番単純な部類

差分リストが扱えることは最低限の条件

背景 | 純粋な差分リスト

純粋に（無理やり）差分リストを表現する手法はある

- 高階関数を用いる手法
- 2本のリストを用いたキューを用いる手法

これらはこれらで面白いと思うが
グラフを扱う前段階としてはちょっと不十分（な気がする）

- グラフを効率的に扱うには残念ながら破壊的更新が必要
  という結論に達した

背景 | 提案手法の概要

グラフ（差分リスト）を破壊的に更新する

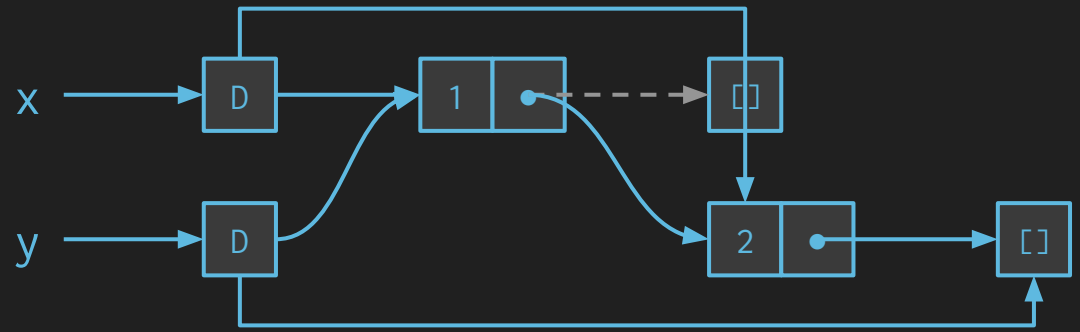
ただし必要に応じて戻すことで純粋に「見せる」

- ユーザをだます

```
x = 1::[]
y = x ++ 2::[]
print x
print y
```

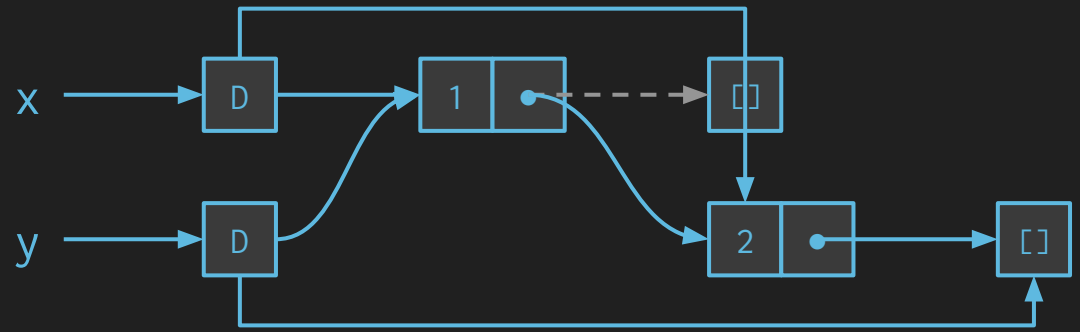
差分リストを生成し，変数 x に束縛する

```
x = 1::[]
y = x ++ 2::[]
print x
print y
```



x を 2::[] と連結し，新たな差分リストを生成して y に束縛
このとき，x の末尾を破壊的に更新することで O(1) で連結できる

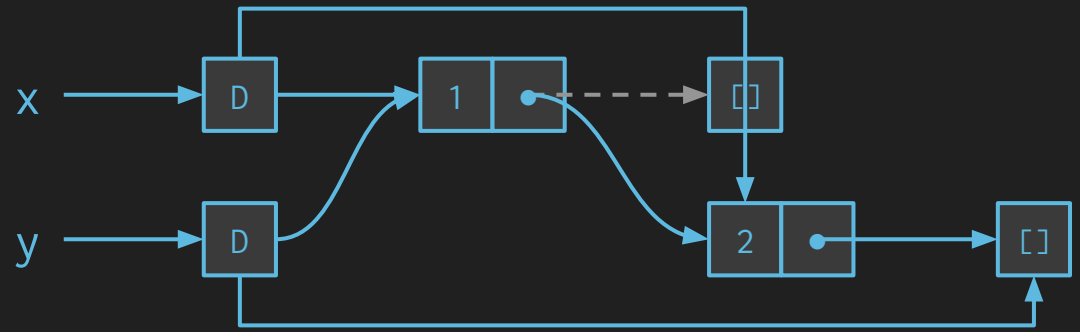
```
x = 1::[]
y = x ++ 2::[]
print x
print y
```



x が欲しくなった

x が参照する差分リストが持つ末尾のアドレスより前までを評価

```
x = 1::[]
y = x ++ 2::[]
print x
print y
```



y が欲しくなった

y が参照する差分リストが持つ末尾のアドレスより前までを評価

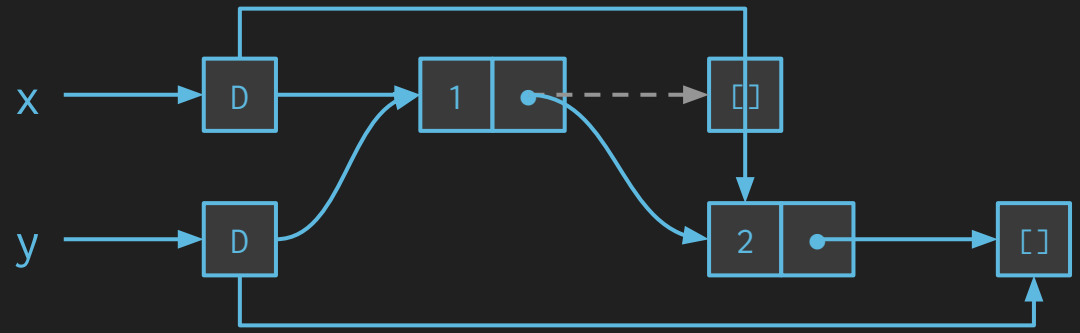
例題 1 | 結論

末尾へのポインタも持っているので，
末尾を破壊的更新しても，末尾かの判定は可能

- 更新したものを元に戻す必要はない？
  （そんなことはない）

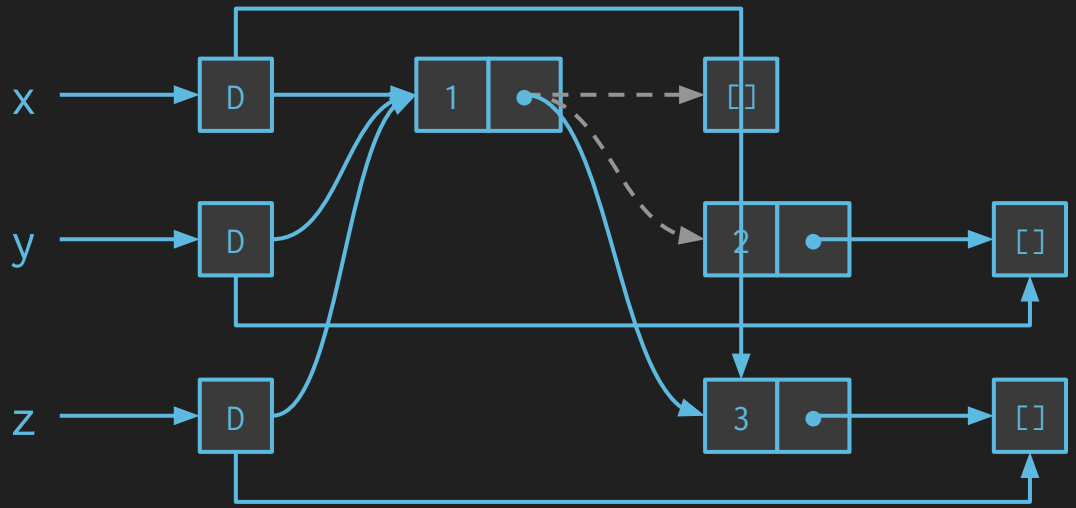
1. 例題 1
2. 例題 2
3. 例題 3
4. 提案手法〉履歴管理木を用いたアルゴリズム
5. 提案手法の改良すべき点と懸念
6. まとめ

```
x = 1::[]
y = x ++ 2::[]
z = x ++ 3::[]
print y
print z
```



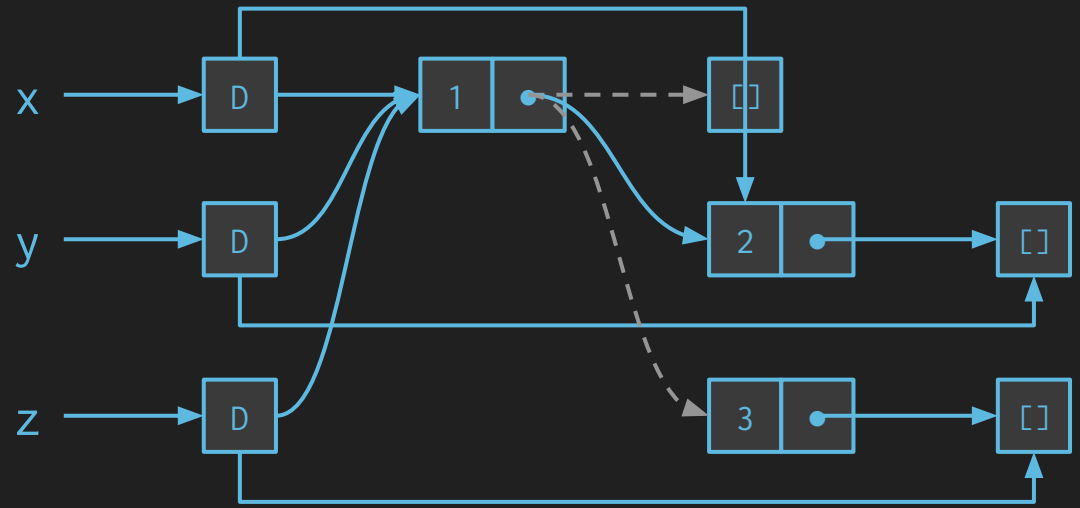
ここまではさっきまでと同じ

```
x = 1::[]
y = x ++ 2::[]
z = x ++ 3::[]
print y
print z
```



x に 3::[] を連結した z を新たに生成する
このとき，y を先頭から辿っても y の末尾には辿り着かない
- y が欲しくなったら前の状態へ戻す必要がある

```
x = 1::[]
y = x ++ 2::[]
z = x ++ 3::[]
print y
print z
```



y が欲しくなったので前の状態へ戻した

例題 2 | 結論

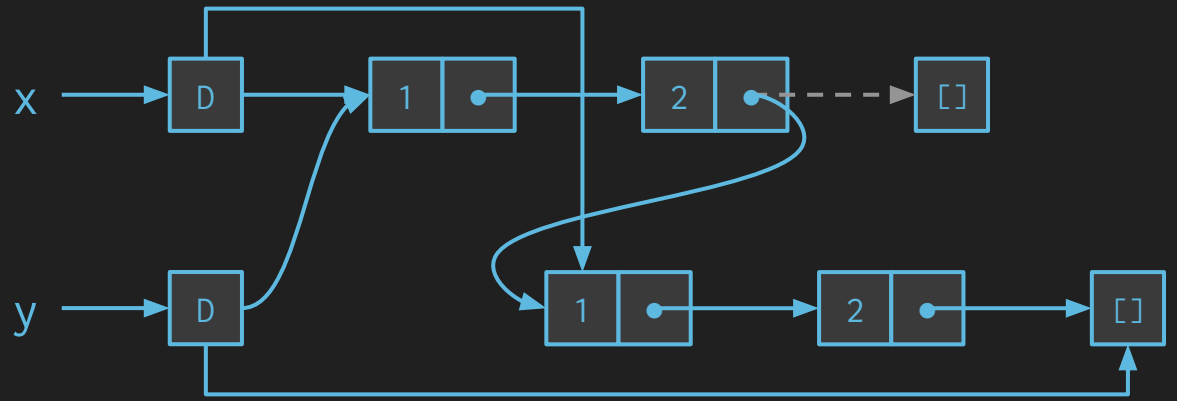
2 回以上同じ差分リストに対して連結した場合は
戻せなくてはいけない

1. 例題 1
2. 例題 2
3. 例題 3
4. 提案手法〉履歴管理木を用いたアルゴリズム
5. 提案手法の改良すべき点と懸念
6. まとめ

```
x = 1::2::[]
y = x ++ x
print y
```

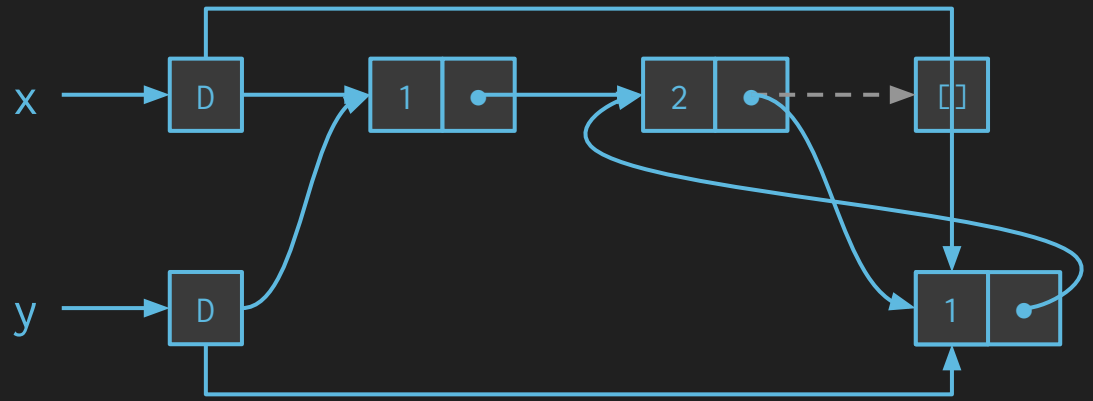
差分リストを生成し，x に束縛する

```
x = 1::2::[]
y = x ++ x
print y
```



x と x を連結して，1::2::1::2::[] という差分リストを生成したい
理想的には右上の図↗のようになって欲しい

```
x = 1::2::[]
y = x ++ x
print y
```



しかし，提案手法では右上の図↗のような循環グラフになってしまう
こうなってしまったデータをうまく扱う手法は，
少なくとも現時点では思いつかない（こうなってはマズい）

Case 3 | 結論

自分自身を含むリストの連結は
（恐らく）破壊的にはできない

- 普通の append を行う必要がある

動的に検出するには

- 全てのリストが固有な id を持ち（メモリアドレスを活用）
- Union-Find を用いて連結するリストに「重なり」がないか
  チェックすれば良さそう

例題から得られる結論のまとめ

どんな動的手法を用いるにせよ，
静的なフロー解析は重要

- そもそも move できるなら，戻す心配はいらない
- 「重なり」がないことのチェックもほぼ静的にできるはず

ただし，それだけでは解決できない場合も
ある程度以上の効率で安全にできるようにしたい

1. 例題 1
2. 例題 2
3. 例題 3
4. 提案手法〉履歴管理木を用いたアルゴリズム
5. 提案手法の改良すべき点と懸念
6. まとめ

やりたいこと

現実的には
「過去のものではなく，新しく作った物を使う」
可能性が高いと思われるので

1. 破壊的更新を行い評価しても，使った人は戻さない
2. 戻す必要があるなら，次に使う人が戻して評価

したい

→ 操作の履歴を残しておき，戻せるようにする

履歴管理木（提案手法）の概要

差分リストに対して行った操作の履歴を
木構造で管理するメタオブジェクト

現時点でのデータに対応する葉と
欲しいデータに対応する葉の間のパスの
ノードにある操作をそれぞれ逆・順実行する

素朴な手法なので要改良

履歴管理木のノードの概要

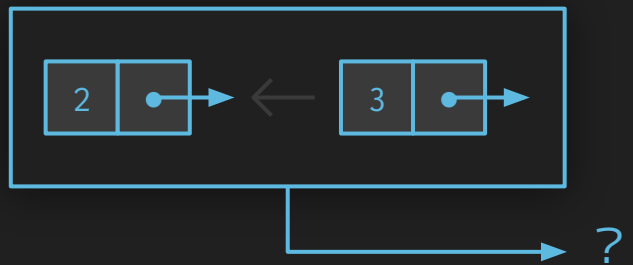
ノードが持つデータ

差分リスト末尾の破壊的更新の履歴

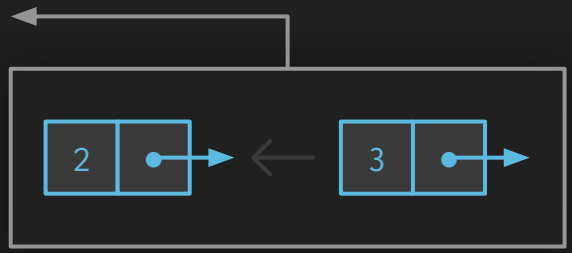
この例では [2|·] に [3|·] を代入する操作を記録している

ただし，[] の更新操作は記録しなくて良い

ノードの種類（2種類）

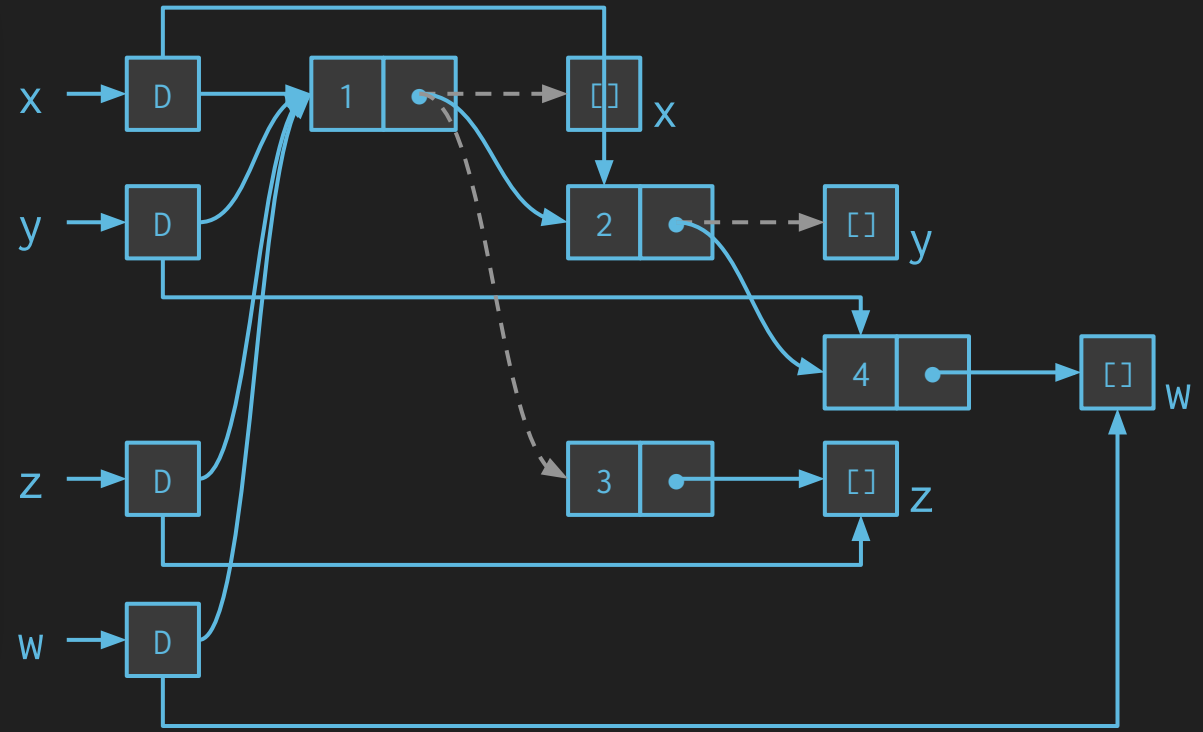


本流（最後に更新したブランチ）
次の操作のノードへのポインタ
（NULL を含む）を持つ



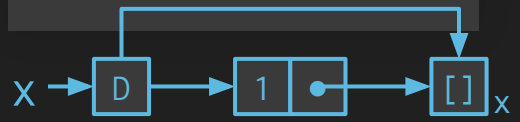
支流（master に取り残されたブランチ）
過去の操作のノードへのポインタを持つ

```
x = 1::[]x
y = x ++ 2::[]y
z = x ++ 3::[]z
w = y ++ 4::[]w
```



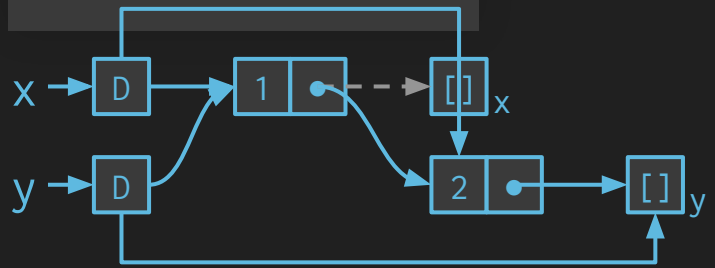
先に紹介した例題にもう1操作追加しただけ
この例題で履歴管理木のアルゴリズムを解説する

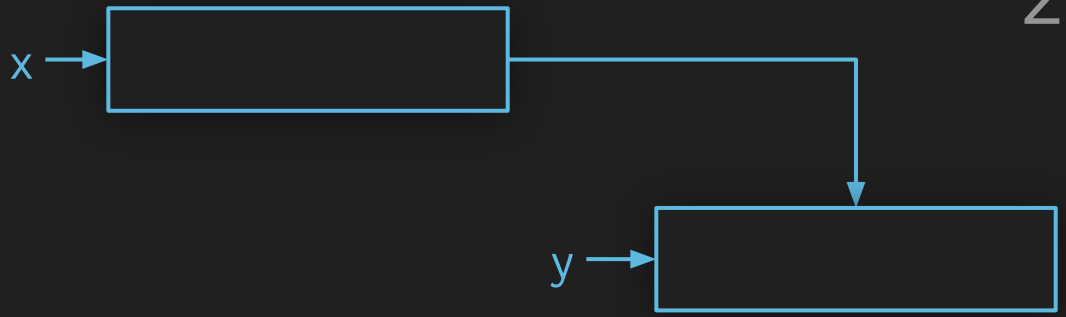
```
x = 1::[]x
y = x ++ 2::[]y
z = x ++ 3::[]z
w = y ++ 4::[]w
```



初期コミット（master ブランチ）を生成
ノードが持つポインタは NULL

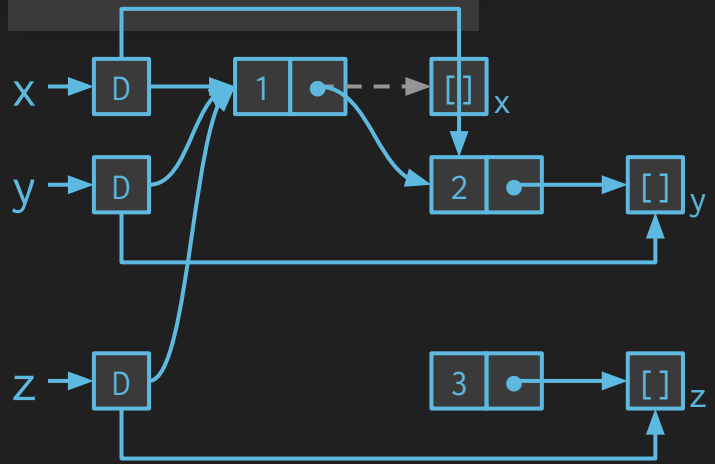
```
x = 1::[]x
y = x ++ 2::[]y
z = x ++ 3::[]z
w = y ++ 4::[]w
```

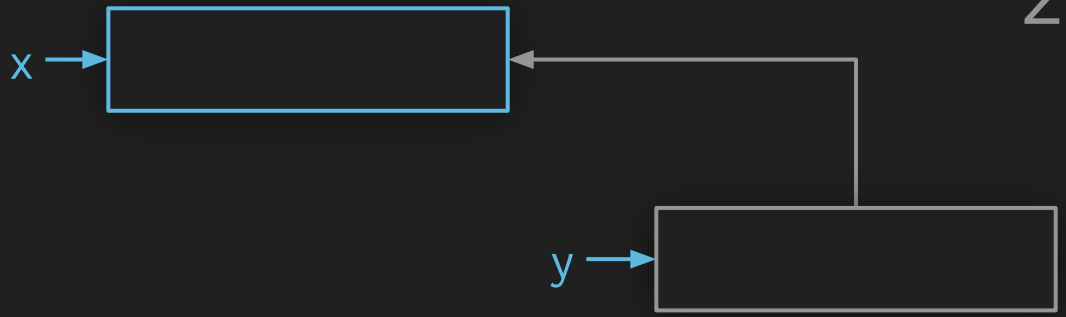




master ブランチに y から参照されるノードを追加

```
x = 1::[]x
y = x ++ 2::[]y
z = x ++ 3::[]z
w = y ++ 4::[]w
```

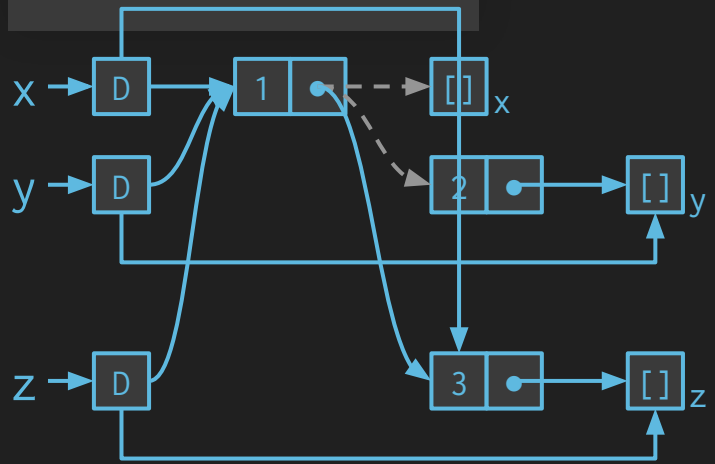


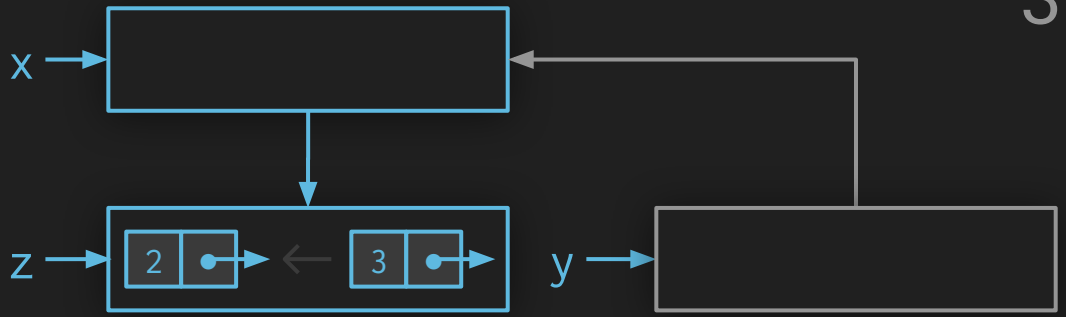


x の末尾を破壊的に更新する
元々のデータが [] でないので操作を履歴に残す必要がある

1. x が参照するノードから本流を下へ，y が参照するノードまで辿り，逆順につなぎ直す
2. x の末尾の更新操作をコミットして z からのポインタをはる

```
x = 1::[]x
y = x ++ 2::[]y
z = x ++ 3::[]z
w = y ++ 4::[]w
```

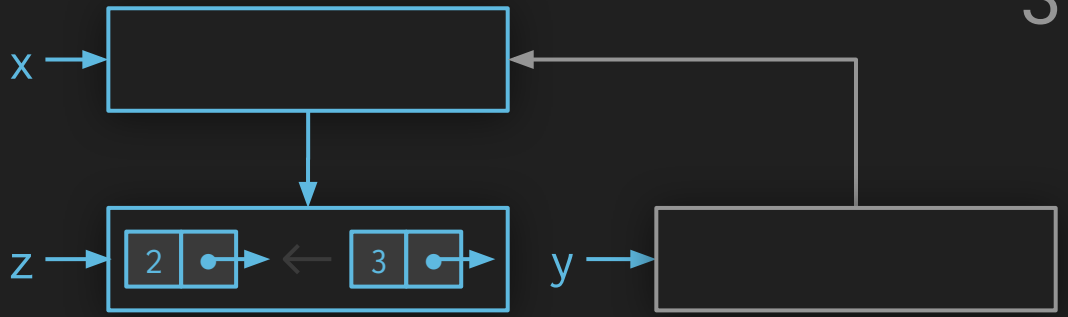


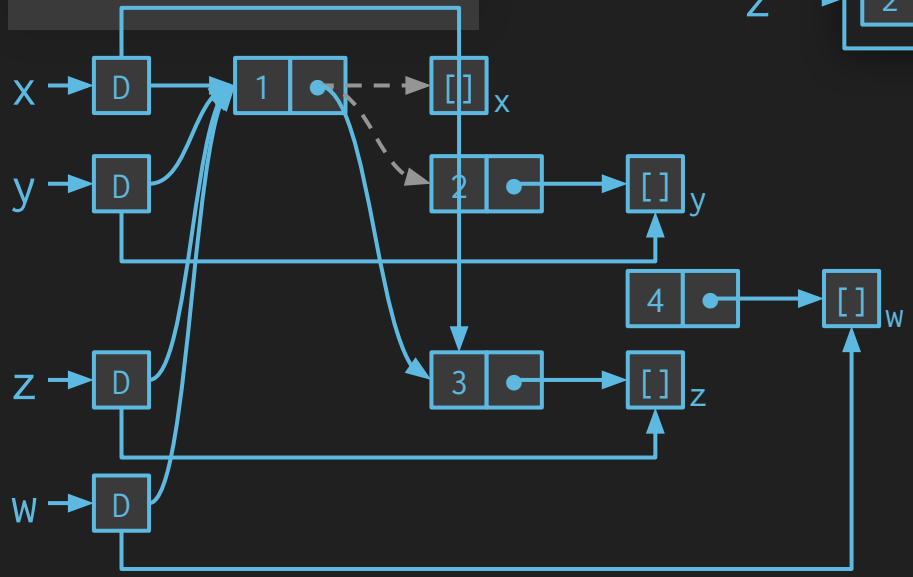


x の末尾を破壊的に更新する
元々のデータが [] でないので操作を履歴に残す必要がある

1. x が参照するノードから本流を下へ，y が参照するノードまで辿り，逆順につなぎ直す
2. x の末尾の更新操作をコミットして z からのポインタをはる

```
x = 1::[]x
y = x ++ 2::[]y
z = x ++ 3::[]z
w = y ++ 4::[]w
```

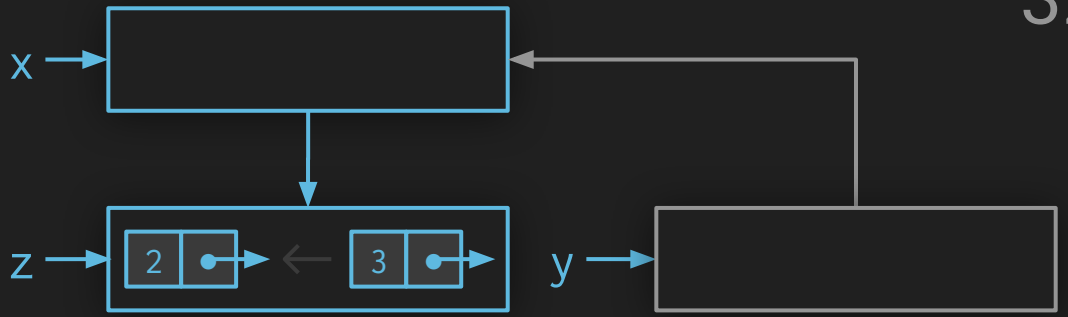


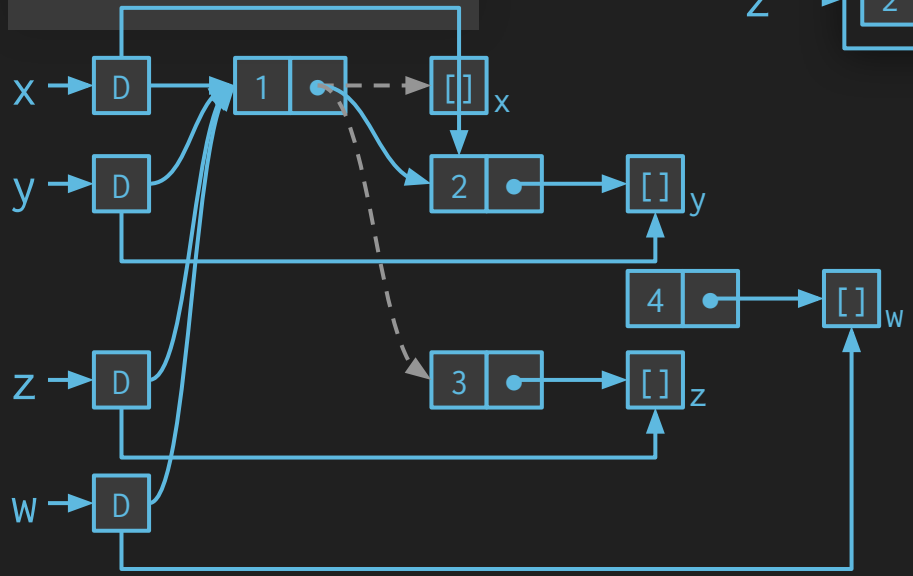


y を戻してから，y の末尾を更新し，w を生成する．y を戻すには

1. y が参照するノードを上へ本流に辿り着くまで 1 回たどり
2. 本流を下って，直前の操作を記録する葉（z が参照するノード）から逆順に逆操作を行い
3. 辿ったノード逆に繋ぎ直して，w のためのノードを追加する

```
x = 1::[]x
y = x ++ 2::[]y
z = x ++ 3::[]z
w = y ++ 4::[]w
```

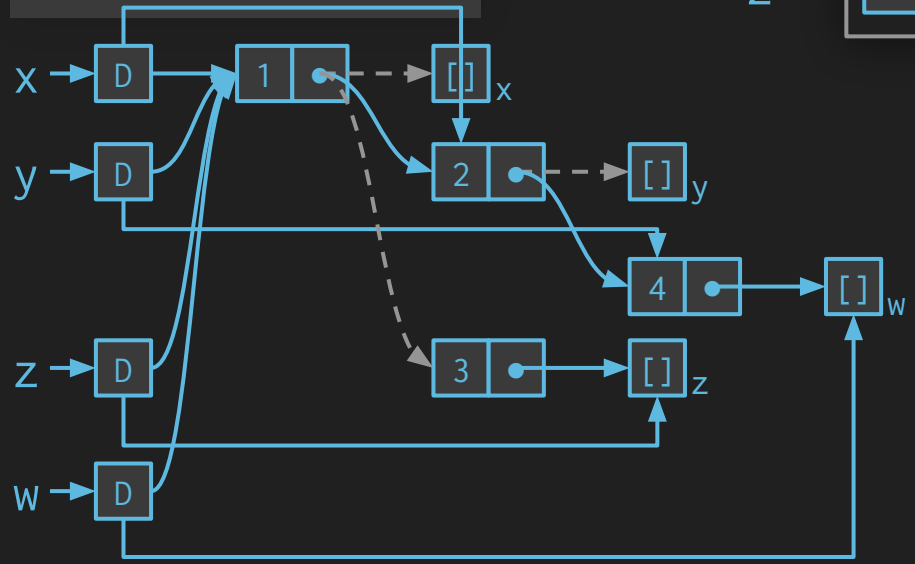


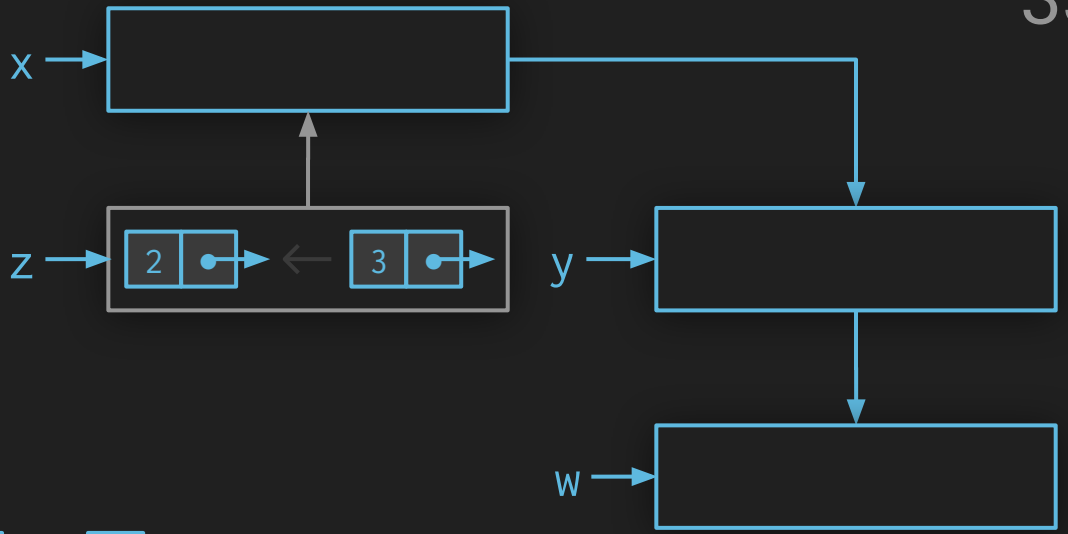


y を戻してから，y の末尾を更新し，w を生成する．y を戻すには

1. y が参照するノードを上へ本流に辿り着くまで 1 回たどり
2. 本流を下って，直前の操作を記録する葉（z が参照するノード）から逆順に逆操作を行い
3. 辿ったノード逆に繋ぎ直して，w のためのノードを追加する

```
x = 1::[]x
y = x ++ 2::[]y
z = x ++ 3::[]z
w = y ++ 4::[]w
```





y を戻してから，y の末尾を更新し，w を生成する．y を戻すには

1. y が参照するノードを上へ本流に辿り着くまで 1 回たどり
2. 本流を下って，直前の操作を記録する葉（z が参照するノード）から逆順に逆操作を行い
3. 辿ったノード逆に繋ぎ直して，w のためのノードを追加する

1. 例題 1
2. 例題 2
3. 例題 3
4. 提案手法〉履歴管理木を用いたアルゴリズム
5. 提案手法の改良すべき点と懸念
6. まとめ

提案手法の改良すべき点

操作（[] への代入）を記録していないノードがある

- 一般のグラフ（末尾が必ず [] である保証がない）なら必要
- 差分リストの場合は，ムダ
- GC を応用（介入）することで動的に削減可能だと思われるが，そもそもこれらを生成しないアルゴリズムが欲しい
- （静的解析により move できるなら不要）

提案手法の懸念

更新前と更新後のデータを両方同時に扱う場合

- map2, zip など

入れ子になったリストの走査の安全性

1. 例題 1
2. 例題 2
3. 例題 3
4. 提案手法〉履歴管理木を用いたアルゴリズム
5. 提案手法の改良すべき点と懸念
6. まとめ

応用例 | リスト系の関数の末尾再帰化

提案手法（動的な履歴管理）と所有権解析により
リストの操作を行う関数はほぼ全て末尾再帰化可能と期待する

```
let rec map f = function
  | [] -> []
  | h::t -> f h :: map f t
```

```
let map f =
  let rec helper acc = function
    | [] -> acc
    | h::t -> helper @@ acc ++ [f h]
  in helper []
```

まとめ

差分リストを純粋風に扱う手法に関して考えた

- 履歴を管理することで，元のデータに復元する
- 復元に多少のコストはかかるが，
  大抵の部分は move できるはずなので
  全体として理想的な効率で動くことを期待する

差分リストだけでもまだまだ考えるべきことは多いが
履歴管理による復元は一般のグラフに応用できる可能性も高いと期待する

参考文献

Purely functional data structure

Reflection without Remorse

など（あまりない）

予備スライド：

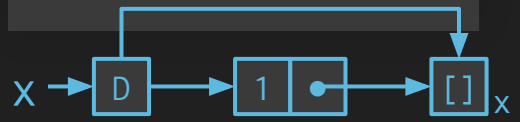
アルゴリズム改良の試み

アイデア

ノードを参照するポインタが二つあるときに
ノードにどちらが本流かの情報を持たせる

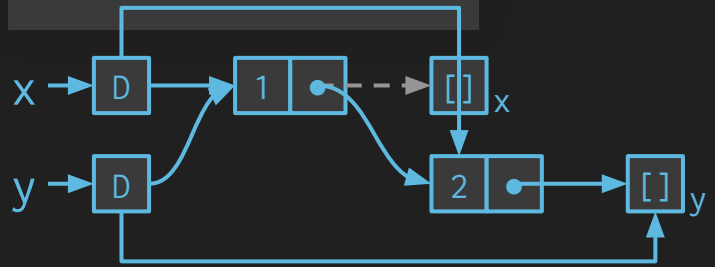
- 二つ以上のノードを用意しなくとも
  自分が本流を参照しているか決定可能
- 図では青いポインタでノードを指しているときに
  そっちが本流であるものとする

```
x = 1::[]x
y = x ++ 2::[]y
z = x ++ 3::[]z
w = y ++ 4::[]w
```



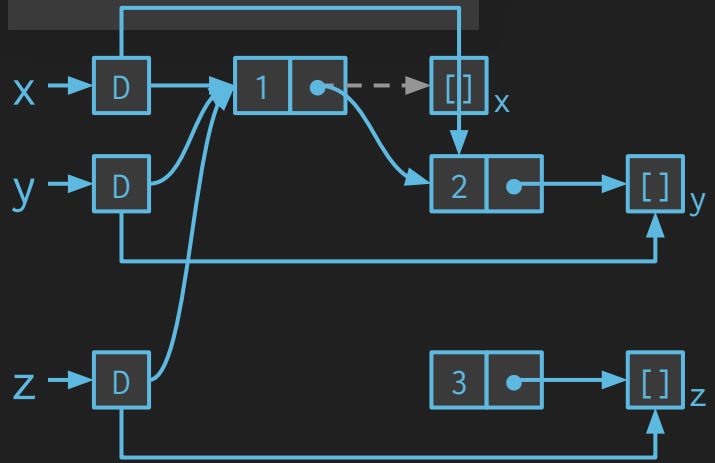
初期コミット（master ブランチ）を生成
ノードが持つポインタは NULL

```
x = 1::[]x
y = x ++ 2::[]y
z = x ++ 3::[]z
w = y ++ 4::[]w
```



master ブランチに y から参照されるノードを追加

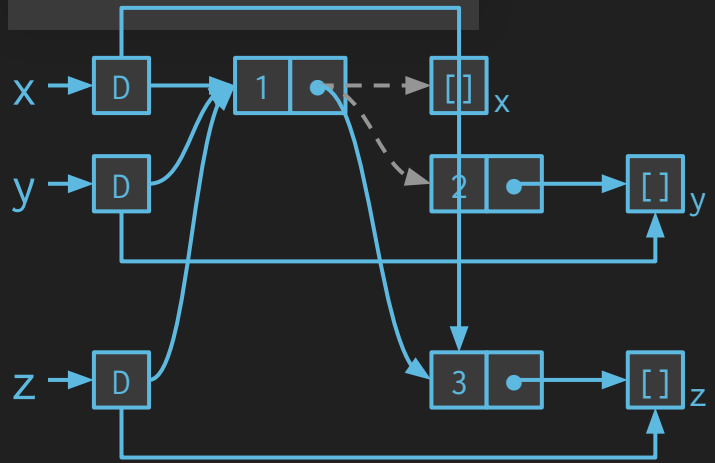
```
x = 1::[]x
y = x ++ 2::[]y
z = x ++ 3::[]z
w = y ++ 4::[]w
```

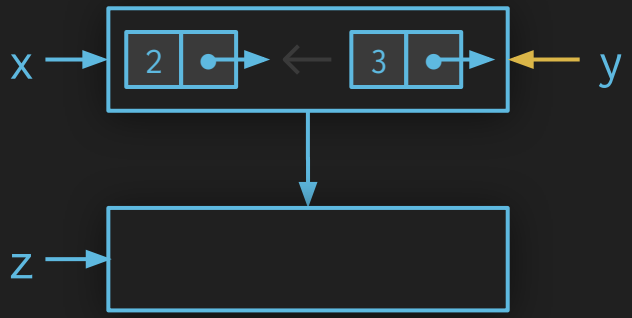


x の末尾を破壊的に更新する
元々のデータが [] でないので操作を履歴に残す必要がある

1. x が参照するノードから本流を下へゼロ回辿り
2. x の末尾の更新操作をコミットして z からのポインタをはる

```
x = 1::[]x
y = x ++ 2::[]y
z = x ++ 3::[]z
w = y ++ 4::[]w
```

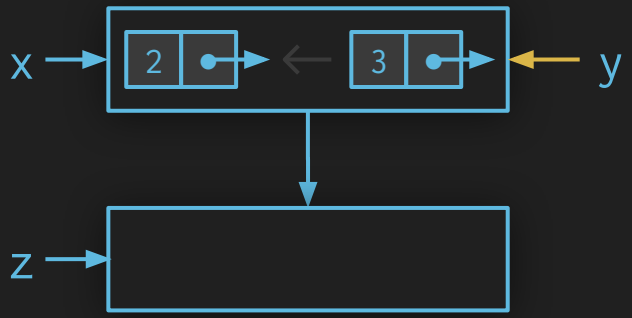


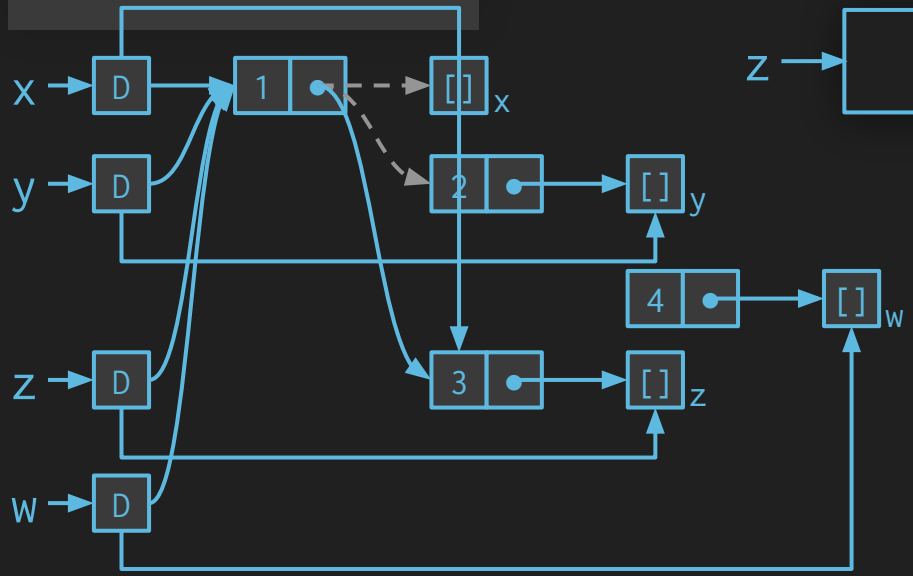


x の末尾を破壊的に更新する
元々のデータが [] でないので操作を履歴に残す必要がある

1. x が参照するノードから本流を下へゼロ回辿り
2. x の末尾の更新操作をコミットして z からのポインタをはる

```
x = 1::[]x
y = x ++ 2::[]y
z = x ++ 3::[]z
w = y ++ 4::[]w
```

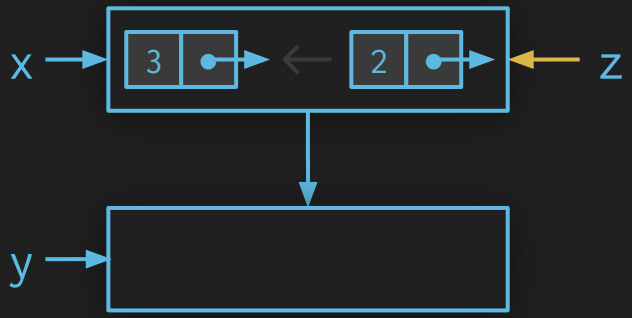


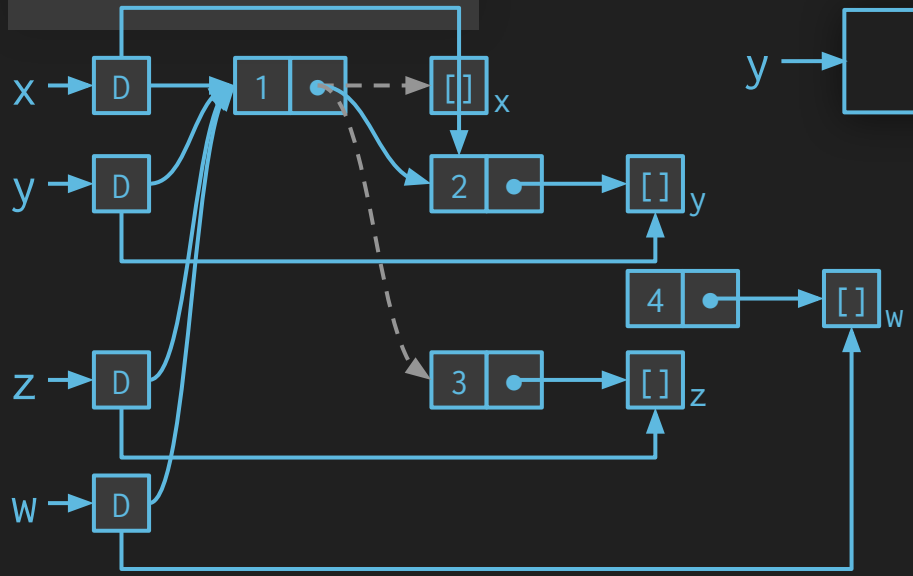


y を戻してから，y の末尾を更新し，w を生成する．y を戻すには

1. y が参照するノードを上へ本流に辿り着くまでゼロ回たどり
2. 本流を下って，直前の操作を記録する葉（z が参照するノード）から逆順に逆操作を行い
3. w からの参照を追加する

```
x = 1::[]x
y = x ++ 2::[]y
z = x ++ 3::[]z
w = y ++ 4::[]w
```

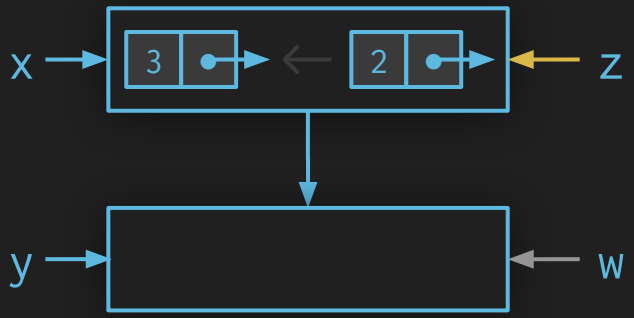


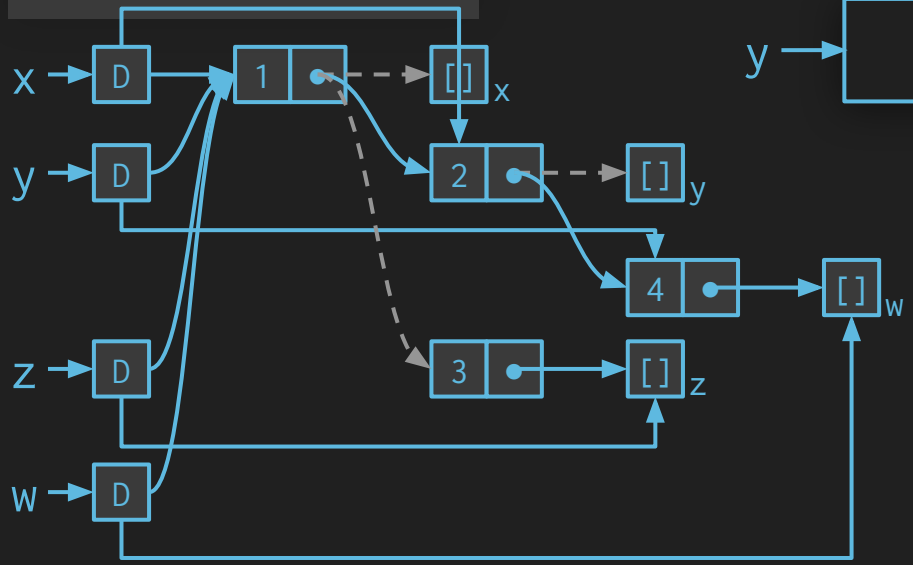


y を戻してから，y の末尾を更新し，w を生成する．y を戻すには

1. y が参照するノードを上へ本流に辿り着くまでゼロ回たどり
2. 本流を下って，直前の操作を記録する葉（z が参照するノード）から逆順に逆操作を行い
3. w からの参照を追加する

```
x = 1::[]x
y = x ++ 2::[]y
z = x ++ 3::[]z
w = y ++ 4::[]w
```





y を戻してから，y の末尾を更新し，w を生成する．y を戻すには

1. y が参照するノードを上へ本流に辿り着くまでゼロ回たどり
2. 本流を下って，直前の操作を記録する葉（z が参照するノード）から逆順に逆操作を行い
3. w からの参照を追加する

結論

三つ以上に分岐した場合にうまくいくか不明

多少メモリとトラバース回数を削減することができているが
本質的な部分は改善できていない（もしかしたらこれを応用すればできるかも知れないと思ったが）

- 依然，n 回の push_back で n に比例した数のノードが生成されてしまう

特に静的フロー解析が効かない・動的言語である場合は
GC に介入してスコープを抜けた差分リストの中間状態のための無駄な履歴ノードを除去して経路を圧縮する必要がありそう