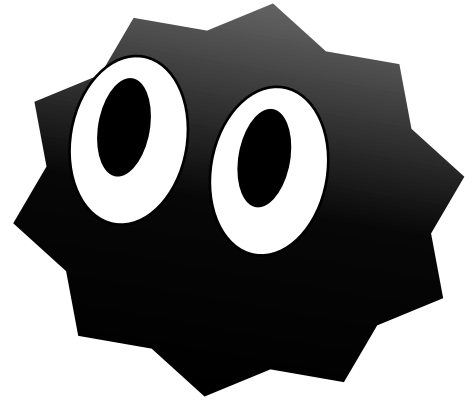
Power Automateの
フローの設計・実装における
私なりの
ロジックの組み立て方

気ままに勉強会 #94

気ままになんでもLT会～第5回～

2024/9/28



自己紹介

- 名前：

  わたるふ

- こんなひとです
  - SharePoint Server 2007 から
    お仕事でMicrosoft 製品に関わる
  - PowerShell が好き
  - Microsoft Teams が好き
  - 今はPower Automate に夢中
- 活動内容
  - 調査・検証したことなどをブログで解説しています
  - 開発したもの（Power Automate のフロー、PowerShellの
    ソースコード）をGitHubで公開しています

Microsoft MVP for
Business Applications

（2023/4～）

- 活動場所
  - ブログ （主な活動場所）
    ルドルフもわたるふもいろいろあってな
    https://wataruf.hatenablog.com
  - マシュマロ （フローについて質問を受け付けています）
    https://marshmallow-qa.com/wataruf01?utm_medium=url_text&utm_source=promotion
  - 過去の登壇資料
    https://www.docswell.com/user/wataruf01
  - X（旧 Twitter）
    https://twitter.com/wataruf01
  - GitHub
    https://github.com/wataruf01
  - Microsoft MVP プロフィールページ
    https://mvp.microsoft.com/ja-jp/PublicProfile/5005227?fullName=Wataru%20Fukai

注意事項

- 解説する内容は私の個人的な見解が含まれています
- 内容に誤りがある場合はご指摘をいただけると助かります

はじめに

今回お話する内容は個人ブログで今月投稿した内容です。

また、勉強会終了後に登壇資料を公開する予定です。

そのため、聞き逃したところや詳しく見直したい部分がでてきましたら

ブログもしく登壇資料でご覧ください。

今回のお話

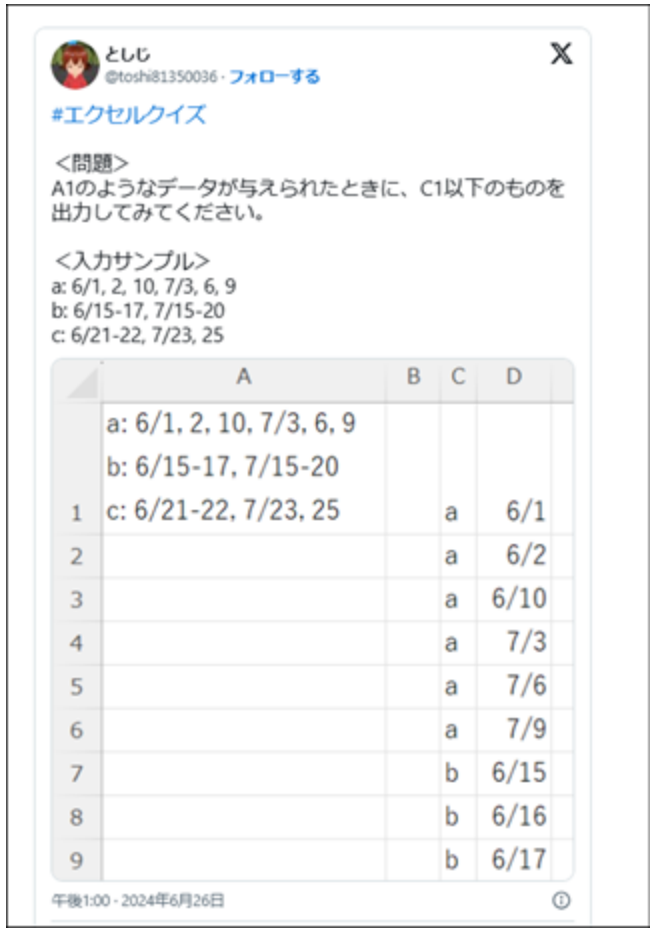
Power Automate における

フローのロジックの組み立てかたについて

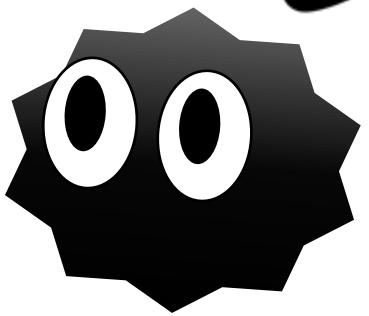
私の経験則をお話します。

背景

- X（旧：Twitter）でこんな投稿がありました。



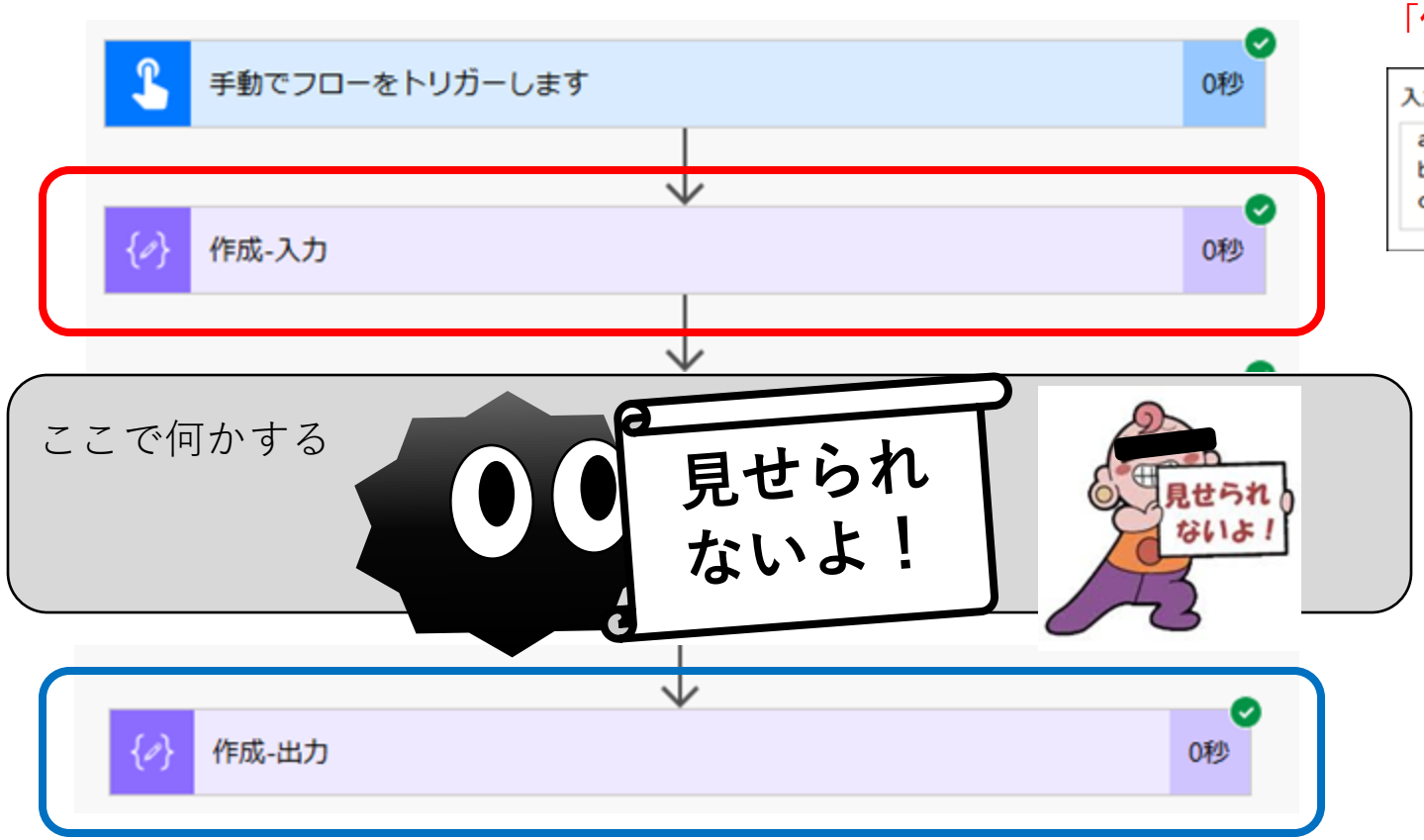
これは
「アルゴリズムクイズ」や
「コーディングチャレンジ」
と呼ばれる問題ですね。



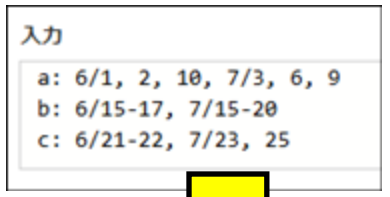
- **アルゴリズムクイズ:**
  与えられた入力に対して、効率的なアルゴリズムを用いて正しい出力を導く問題
- **コーディングチャレンジ**:
  プログラミングスキルを試すためのチャレンジ形式の問題

この問題をPower Automate用に読み変えるとこうなります

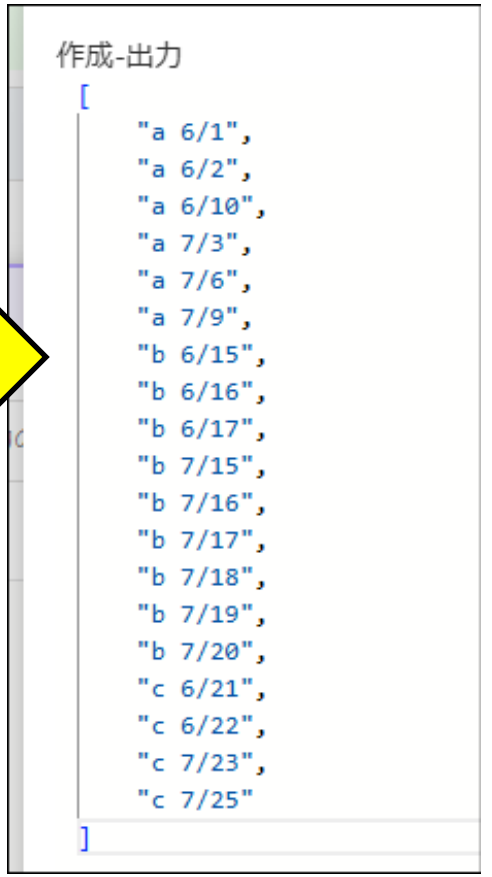
<問題>
下図の通り、「作成-入力」アクションの値を入力情報として「作成-出力」アクションの値を得ること



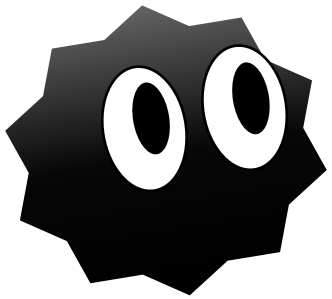
「作成-入力」アクション



「作成-出力」アクション

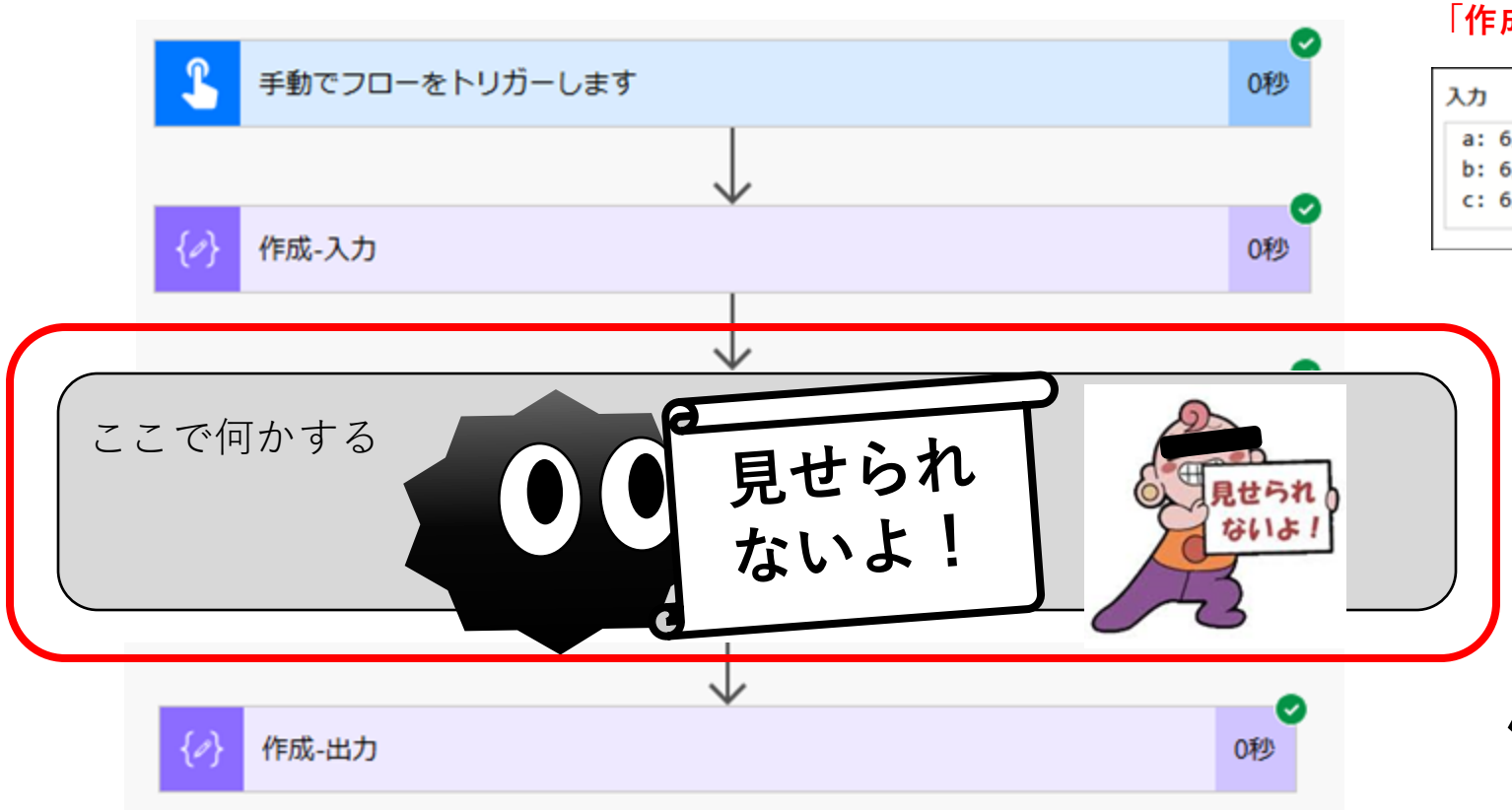


フローの
アクションを駆使します



赤枠部分で何をするか想像できますか？？

<問題>
下図の通り、「作成-入力」アクションの値を入力情報として「作成-出力」アクションの値を得ること



「作成-入力」アクション

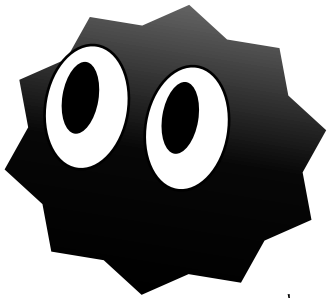
入力

```
a: 6/1, 2, 10, 7/3, 6, 9
b: 6/15-17, 7/15-20
c: 6/21-22, 7/23, 25
```

「作成-出力」アクション

作成-出力

```
[
    "a 6/1",
    "a 6/2",
    "a 6/10",
    "a 7/3",
    "a 7/6",
    "a 7/9",
    "b 6/15",
    "b 6/16",
    "b 6/17",
    "b 7/15",
    "b 7/16",
    "b 7/17",
    "b 7/18",
    "b 7/19",
    "b 7/20",
    "c 6/21",
    "c 6/22",
    "c 7/23",
    "c 7/25"
]
```



見せなさいよ

ロジックを組み立てるのは難しい

「a」「b」「c」
というアルファベットが
行ごとについている。
だから、行ごとに
区切って配列にする。

「作成-入力」アクション

入力

```
a: 6/1, 2, 10, 7/3, 6,
b: 6/15-17, 7/15-20
c: 6/21-22, 7/23, 25
```

日付にハイフンが
含まれる場合は
指定された範囲の日付を
展開して出力する。

個々の日付ごとの
値を出力するために
カンマで区切って
配列にする

日付にスラッシュが
含まれる場合は
月が変わる判定をいれる

「作成-出力」アクション

作成-出力


```
[
    "a 6/1",
    "a 6/2",
    "a 6/10",
    "a 7/3",
    "a 7/6",
    "a 7/9",
    "b 6/15",
    "b 6/16",
    "b 6/17",
    "b 7/15",
    "b 7/16",
    "b 7/17",
    "b 7/18",
    "b 7/19",
    "b 7/20",
    "c 6/21",
    "c 6/22",
    "c 7/23",
    "c 7/25"
]
```


必要な「繰り返し処理」や
「条件分岐」が
大まかには見えてくるけど

全体を通して
どんなフローになるのかは
イメージが難しい。。

避けたほうがよいこと

- はじめから完成形を目指して

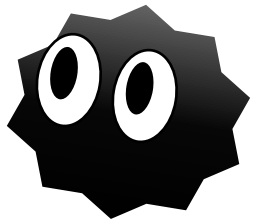
  上から順番にフローを作成すること

  - 入れ子になっている繰り返し処理
  - フローが使いそうな値をすべて変数アクションで扱おうとする

    （= 定数にするべきかを検討しない）

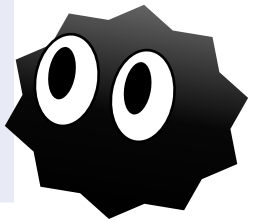
- フローがすべて組み終わってから

  はじめて動作確認をすること



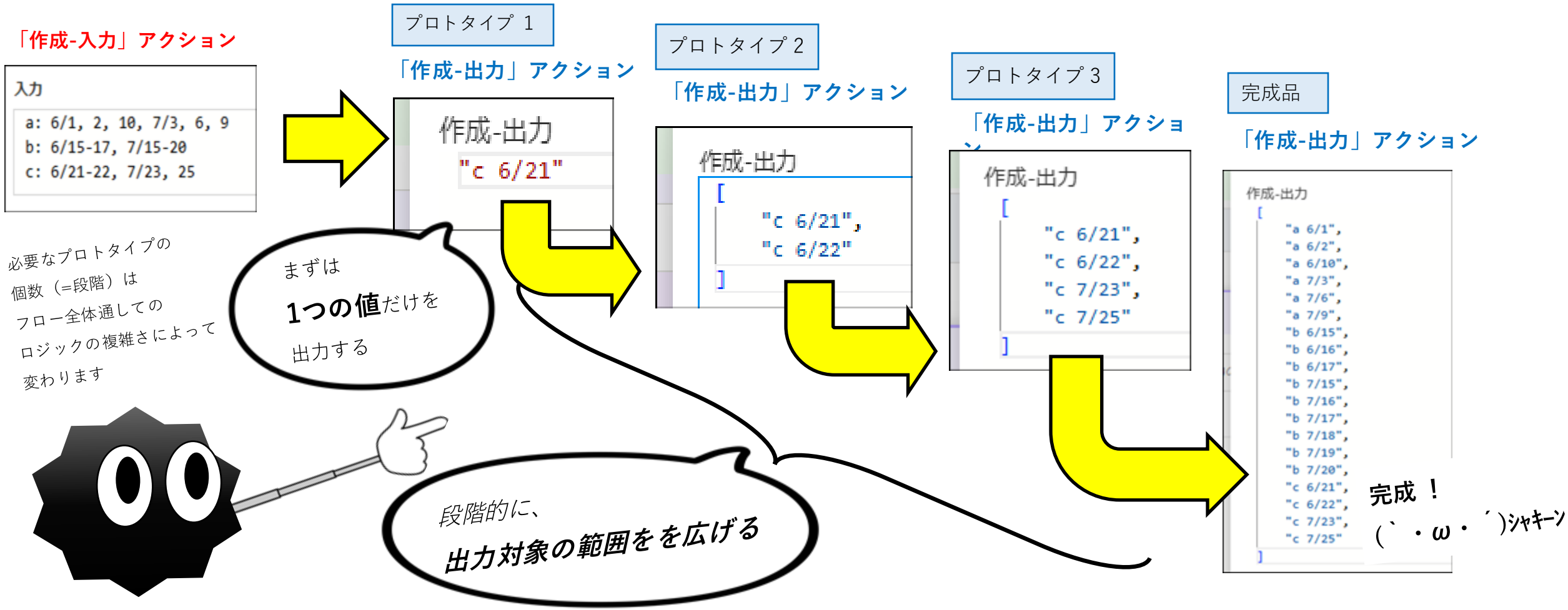
避けたほうがよい理由

避けたほうがよい理由	詳細
完成までの見通しがたてにくい	全体的な処理の流れや課題を洗い出さずに実装を進めるため、作業ボリュームの見通しがたてにくい。最悪の場合、作業時間をかけたにも関わらず『開発の進捗が進まない』という事態が起こり得る。
課題の発見が遅くなる可能性がある	開発の過程（特にフローの後半・終盤）でにおける課題の洗い出しが後回しにされることがあり、問題解決に時間を要することがある。
可読性やメンテナンス性が低下しやすい	フロー全体をイメージしない状態で開発を進めるため、一貫性が保たれず表記のブレや冗長な実装が発生する場合がある。
開発の引き継ぎが難しくなりやすい	途中で他のメンバーに開発を引き継ぐ際、最初に開発した人が後続処理をどのように考えていたかが伝わりづらい。

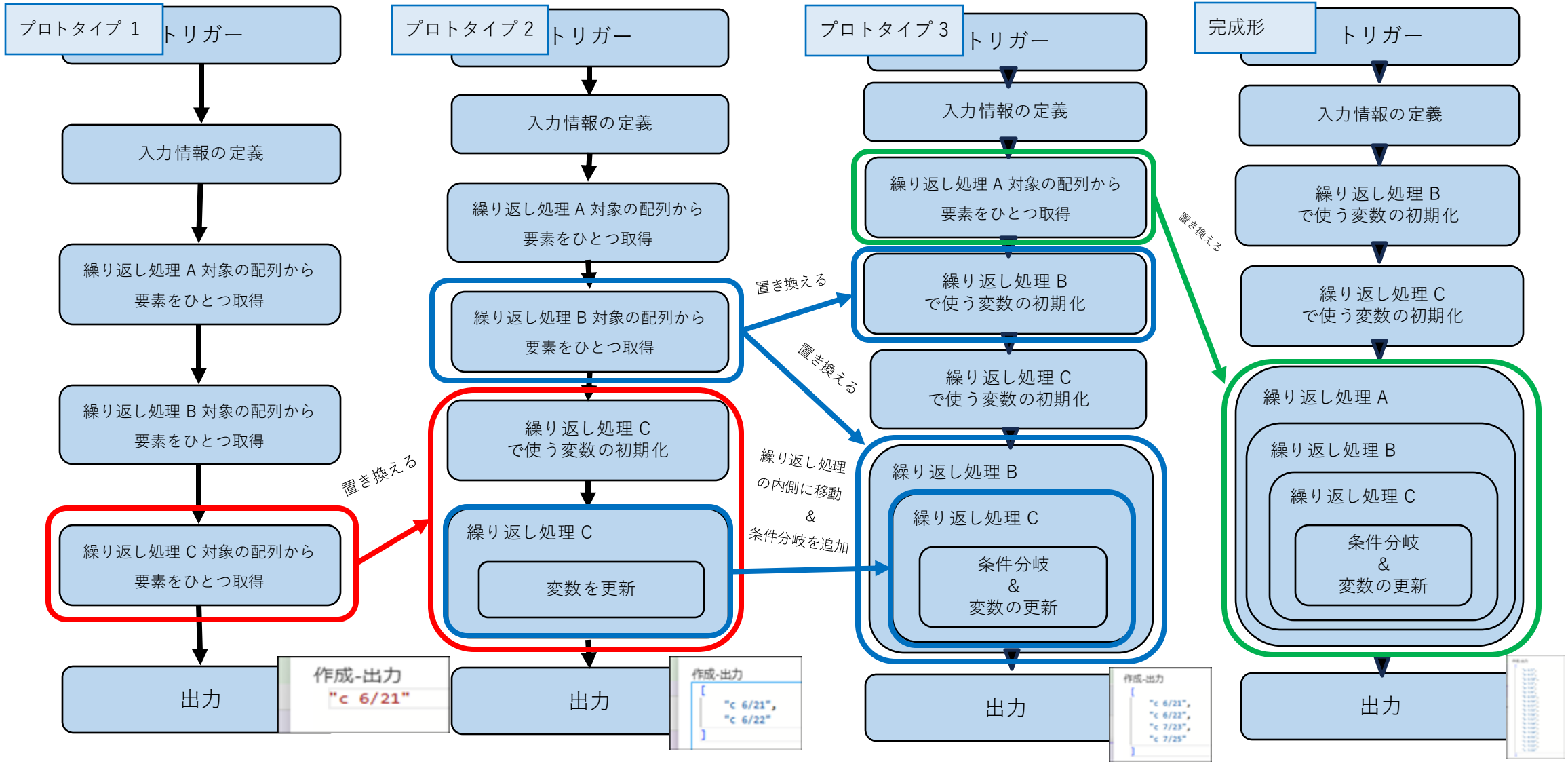


【結論・主旨】フローのロジックは段階的に組み立てよう

最初は出力の範囲を最小減に絞ってフローを作り、徐々に出力の対象範囲を広げていきます。



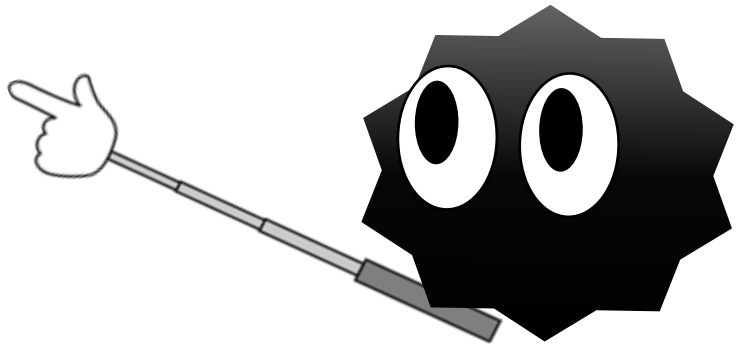
段階的なロジックの組み立てかたをフロー図で表すとこの通り



【段階的にフローを組み立てる5つのメリット】

メリットその1：機械的に設計・実装を進められる

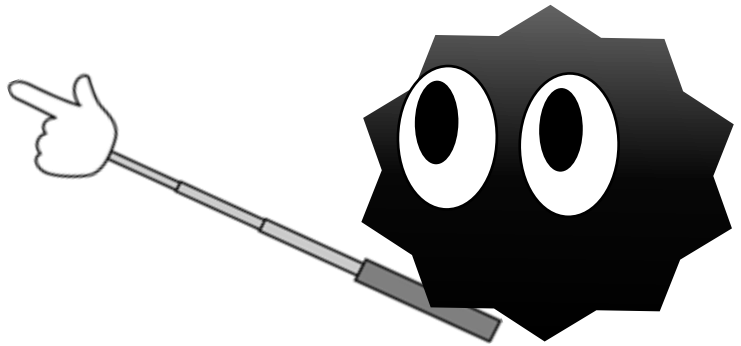
- この手法では、最初に小さな出力範囲で簡単なフローを作成し、実際に動作するか確認してから徐々に複雑な処理や出力対象を追加していきます。
- 開発者は、個々のステップごとに具体的なゴールを持って作業するため、迷うことなく効率的に作業が進みます。
- **段階的なアプローチにより、次に何をすべきかが明確で、フローの設計や実装が機械的に進められます。**



【段階的にフローを組み立てる5つのメリット】

メリットその2：かけた時間が無駄にならない

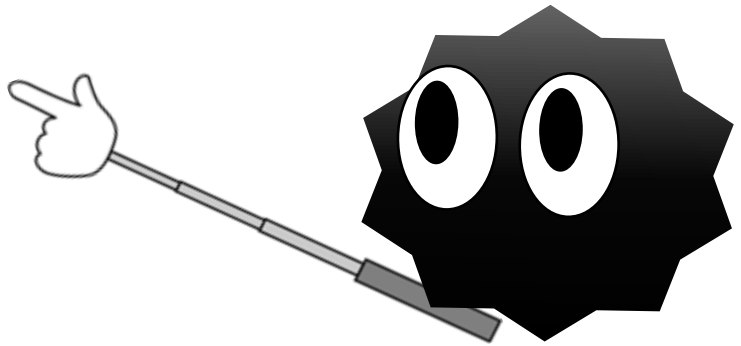
- 出力範囲を小さく設定し、動作確認を行ってから次のステップに進むため
  フローが予期せず動作しないケースを早期に発見できます。
- また、逐次確認することで、実装の過程で発生する問題を小さな単位で特定しやすく
  なり手戻りを最小限に抑えることができます。
- **この方法により、後になって大きなフロー全体を修正する必要がなく**
  初期段階で問題を解決できるため、かけた時間が無駄になりません。



【段階的にフローを組み立てる5つのメリット】

メリットその3：可読性・メンテナンス性を保ちやすい

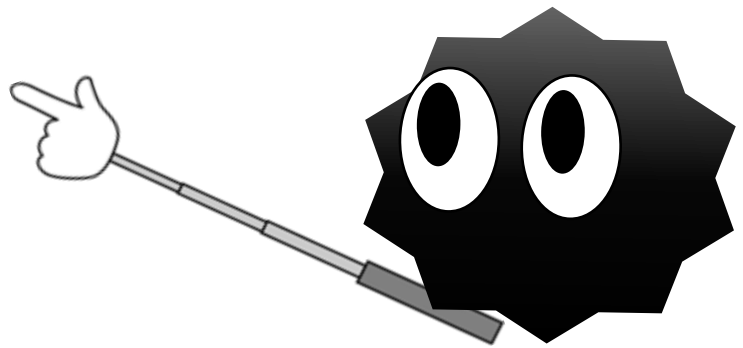
- **段階的な開発手法を採用することで、各段階のフローが整理された状態で作成されるため、可読性が向上します。**
- 各ステップで明確で簡潔なゴールをもとに開発が行われるため、
  フロー全体を通してロジックが煩雑になることが回避しやすいです。
- フローを段階的に拡張していくことを前提とした開発手法であるため
  要件の追加や変更があった場合にも、比較的柔軟に対応できます。



【段階的にフローを組み立てる5つのメリット】

メリットその4：ケース漏れが発生しにくい

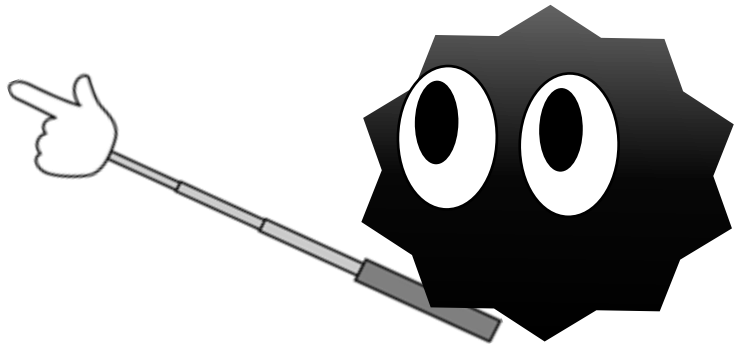
- 段階的にフローを拡張することで、各ステップごとに動作確認を行いながら開発を進めるため、ケース漏れが発生しにくくなります。
- 出力範囲を少しずつ広げていくことで、想定される処理や例外ケースも小さな単位で確認でき、結果としてすべてのケースが網羅されやすくなります。
- **最初から大規模なフローを構築する場合に比べて、細かく動作確認を行えるため、後からケース漏れが発覚するリスクを減らせます。**



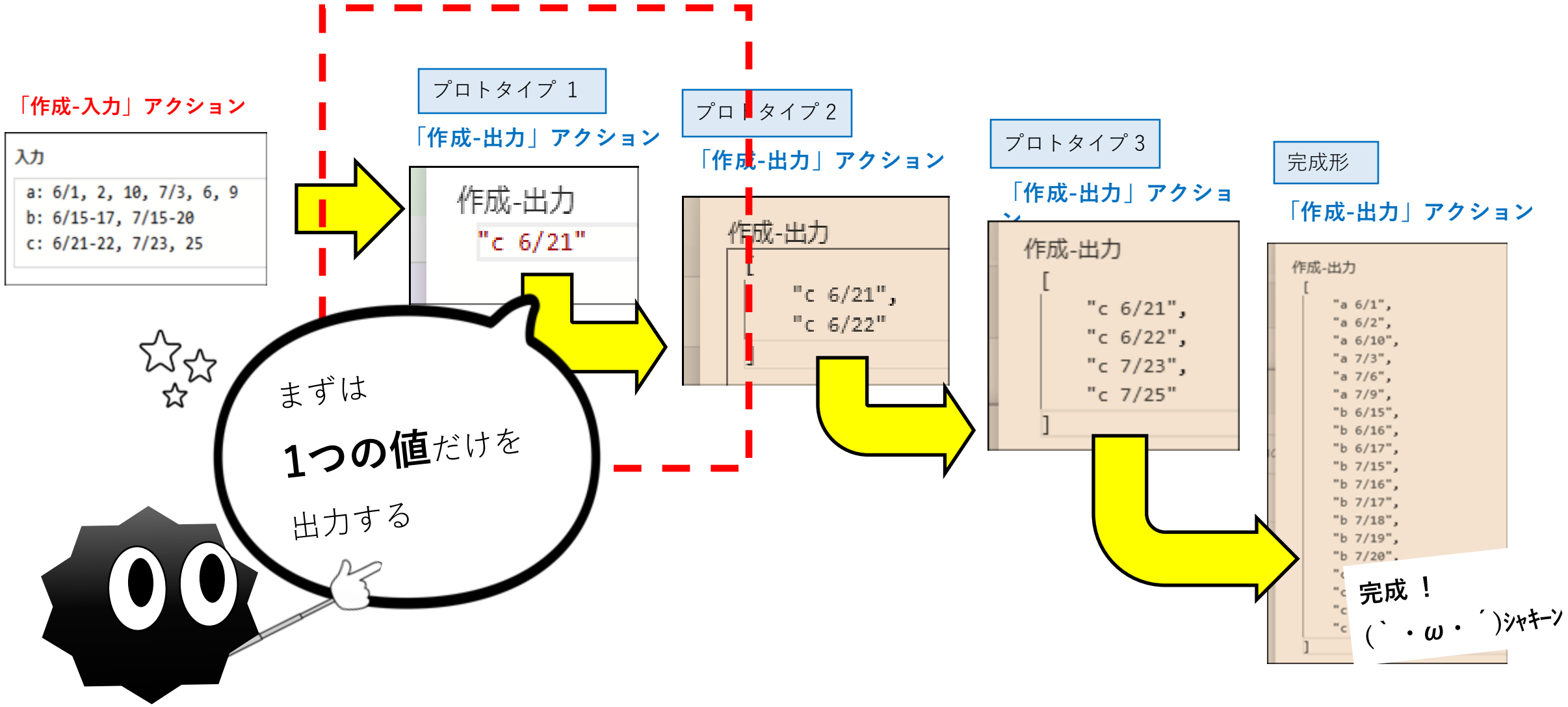
【段階的にフローを組み立てる5つのメリット】

メリットその5：開発途中での引継ぎが行いやすい

- この開発手法では、**小さな範囲ごとに確実に動作するフローが構築されているため途中で別の開発者に開発を引き継ぐ場合も引き継ぎが比較的スムーズに行えます。**
- 各ステップが明確に分かれており、それぞれの範囲で期待される出力が明示されているため、新しい開発者がフローの意図を理解しやすく、開発を継続しやすいです。



まずはプロトタイプ1を作ります



プロトタイプ1における入力と出力

「作成-入力」アクション

入力

```
a: 6/1, 2, 10, 7/3, 6, 9
b: 6/15-17, 7/15-20
c: 6/21-22, 7/23, 25
```

これがプロトタイプ1の
フローです。
完成形のフロー出力に
含まれる値のひとつを出力する
ことを目的としたフローです。

「作成-出力」アクション

作成-出力

```
"c 6/21"
```

どの値を対象とするかは
任意で大丈夫です

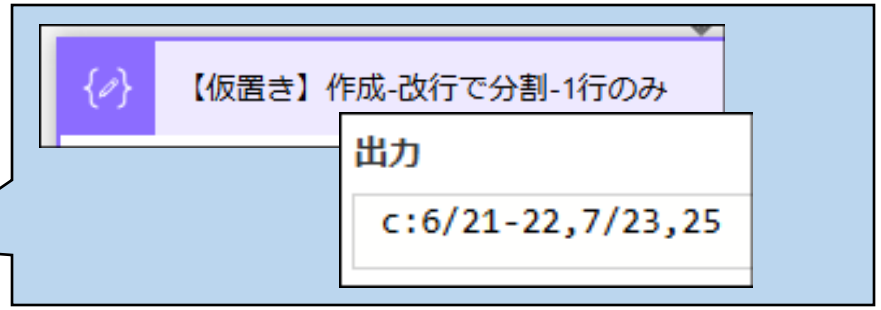
例として
この値を出力対象
にします。

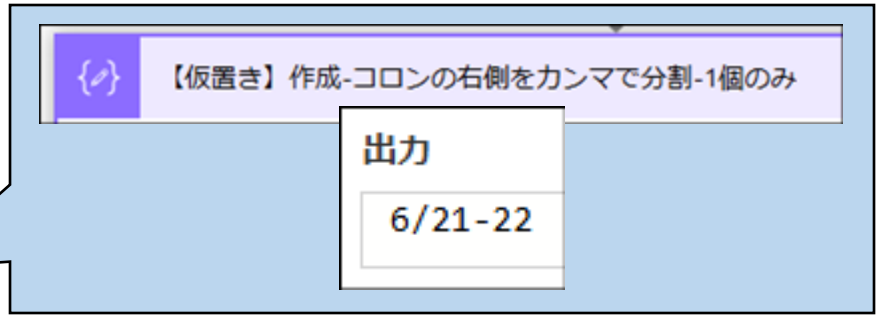
完成形

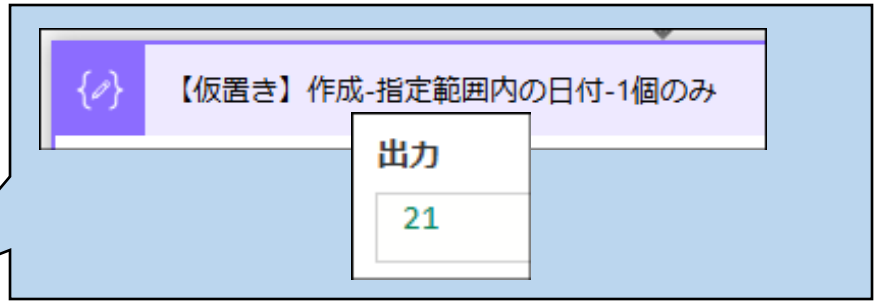
作成-出力

```
[
    "a 6/1",
    "a 6/2",
    "a 6/10",
    "a 7/3",
    "a 7/6",
    "a 7/9",
    "b 6/15",
    "b 6/16",
    "b 6/17",
    "b 7/15",
    "b 7/16",
    "b 7/17",
    "b 7/18",
    "b 7/19",
    "b 7/20",
    "c 6/21",
    "c 6/22",
    "c 7/23",
    "c 7/25"
]
```

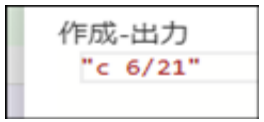
繰り返し処理を行うべき箇所を「ひとつの要素を対象にした処理」に仮置き







「作成-出力」アクション



このプロトタイプ1では

繰り返し処理を行うべき箇所を「ひとつの要素を対象にした処理」に置きかえてフローを作成しています。

“仮置き”目的の
アクションを 配置しています。

プロトタイプ１では変数・条件分岐を使用しない

フロー全体を通して

すべて固定値を使っている

(= 動的な値が無い)

プロトタイプ1では

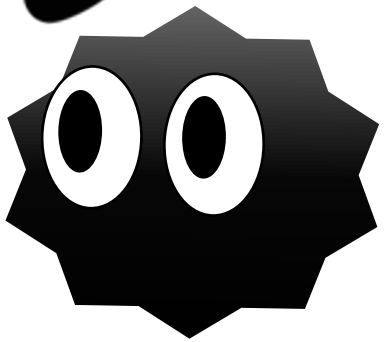
特定のひとつの値を

出力することを目的としているため

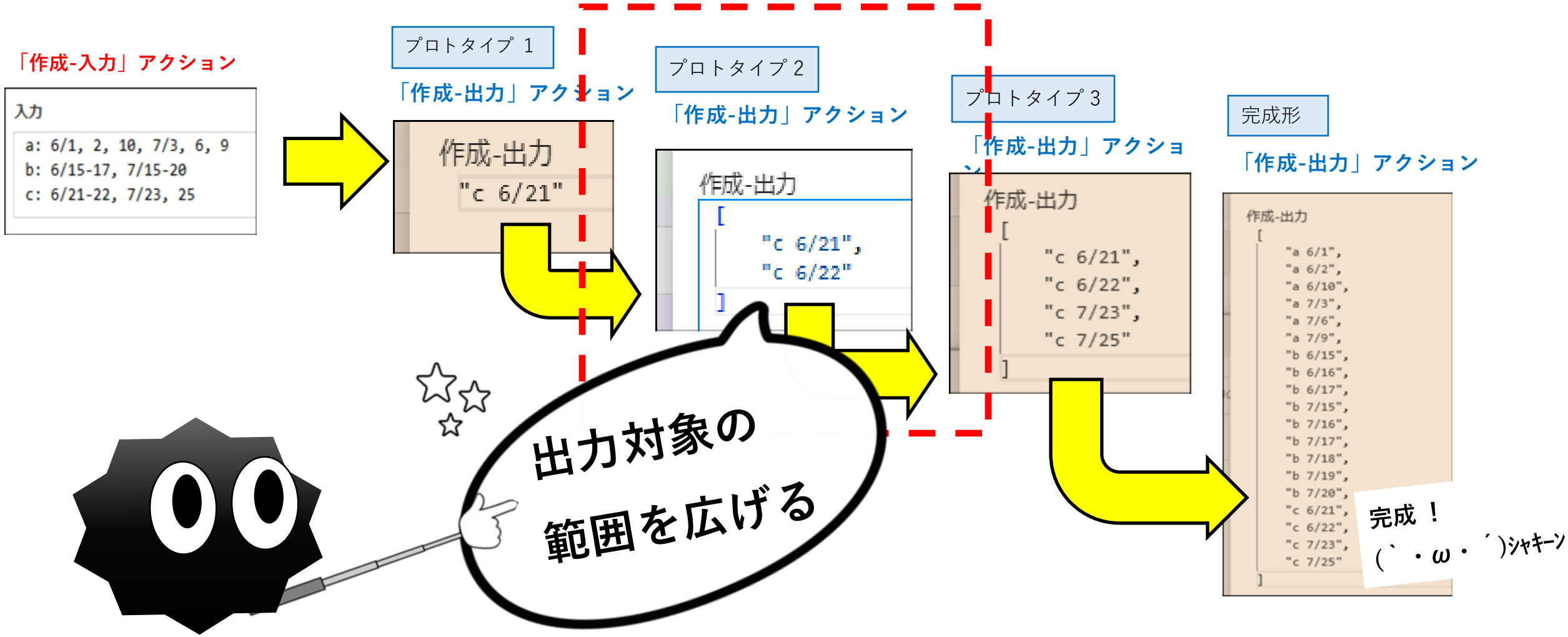
変数アクションと条件分岐を

使用していない

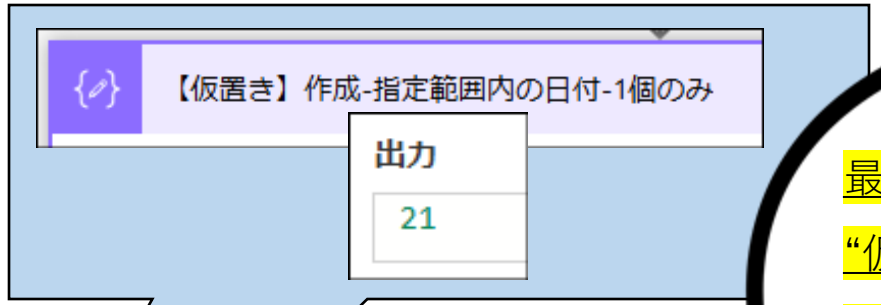
という特徴があります。



次にプロトタイプ2を作ります



プロトタイプ1 から プロトタイプ2への変化



最終的な出力に最も近い

“仮置き”のアクションを

本来実装したい

繰り返し処理に置き換えます。

ここに注目！

次のスライドで

注目箇所にズームします。

プロトタイプ1 から プロトタイプ2への変化

プロトタイプ 1

outputs('作成-指定範囲内の日付')[0]

@{outputs('作成-コロンの左側')} @{outputs('作成-月')}
/@{outputs('【仮置き】作成-指定範囲内の日付-1個のみ')}

このアクションで
配列から要素をひとつ
取得します。

outputs 関数の
末尾にある [0] は
配列にある最初の要素を
取得することを表しています

次スライド以降でこの
「**ひとつの要素を対象とした処理**」を
本来の目的である
「**配列を対象とした処理**」
に置き換えます。

プロトタイプ1の
最終的なアクション（出力）で
この要素を使用しています。

次のスライドへ

プロトタイプ1 から プロトタイプ2への変化

プロトタイプ 1

作成-指定範囲の終了日

作成-指定範囲内の日付

【仮置き】作成-指定範囲内の日付-1個のみ

入力 outputs(...)

作成-出力

入力 出力 出力 出力

@{outputs('作成-コロンの左側')} @{outputs('作成-月')}
/@{**outputs('【仮置き】作成-指定範囲内の日付-1個のみ')**}

outputs('作成-指定範囲内の日付')[0]

変更：

ひとつの要素 ⇒ 配列

「変数の初期化」アクション
に置き換え

「変数の設定」アクション
に置き換え

変数の値を出力する関数
に置き換え

プロトタイプ 2

作成-指定範囲内の日付

変数を初期化する-出力

名前 出力

種類 アレイ

値 初期値を入力

配列変数「出力」を初期化

Apply to each

出力

@outputs('作成-指定範囲内の日付')

配列変数に追加-出力

名前 出力

値 出力 出力 出力

@{outputs('作成-コロンの左側')} @{outputs('作成-月')}
/@{**item()**}

作成-出力

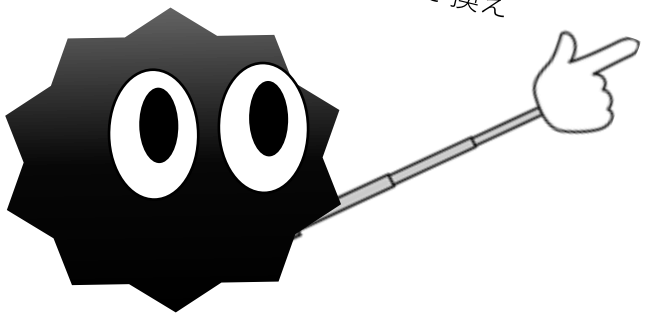
入力 出力

@variables('出力')

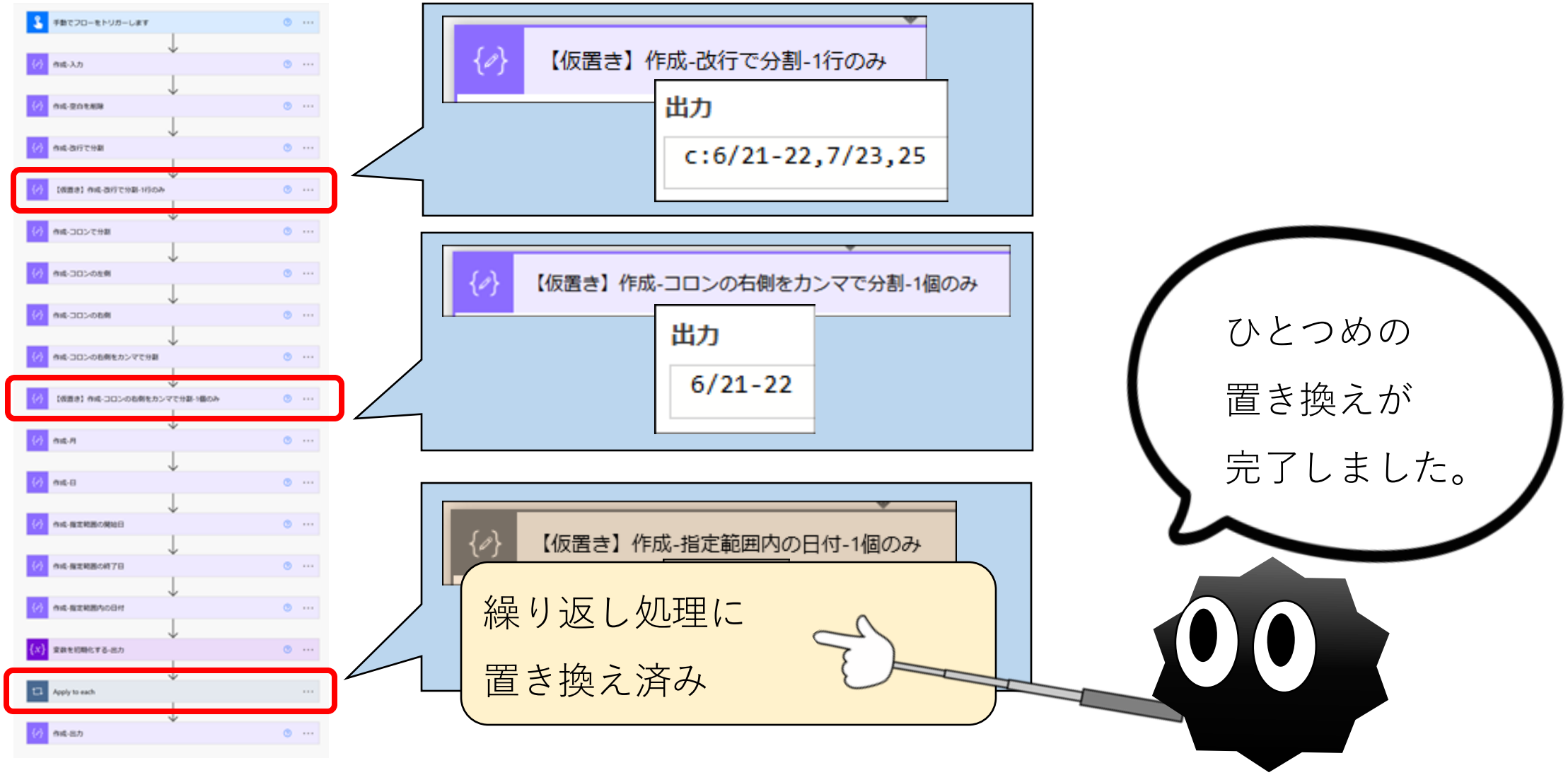
作成-出力

```
[
    "c 6/21",
    "c 6/22"
]
```

処理対象を配列に広げたことで
値が固定から動的に変化した箇所を
変数に置き換えます



仮置きのアクションを繰り返し処理に置き換え （ひとつめが完了）



次工程のプロトタイプ3でも置き換えを行います

【仮置き】作成-改行で分割-1行のみ

出力

c:6/21-22,7/23,25

【仮置き】作成-コロンの右側をカンマで分割-1個のみ

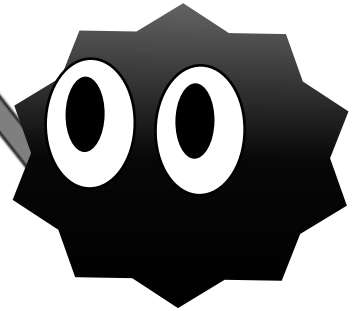
出力

6/21-22

【仮置き】作成-指定範囲内の日付-1個のみ

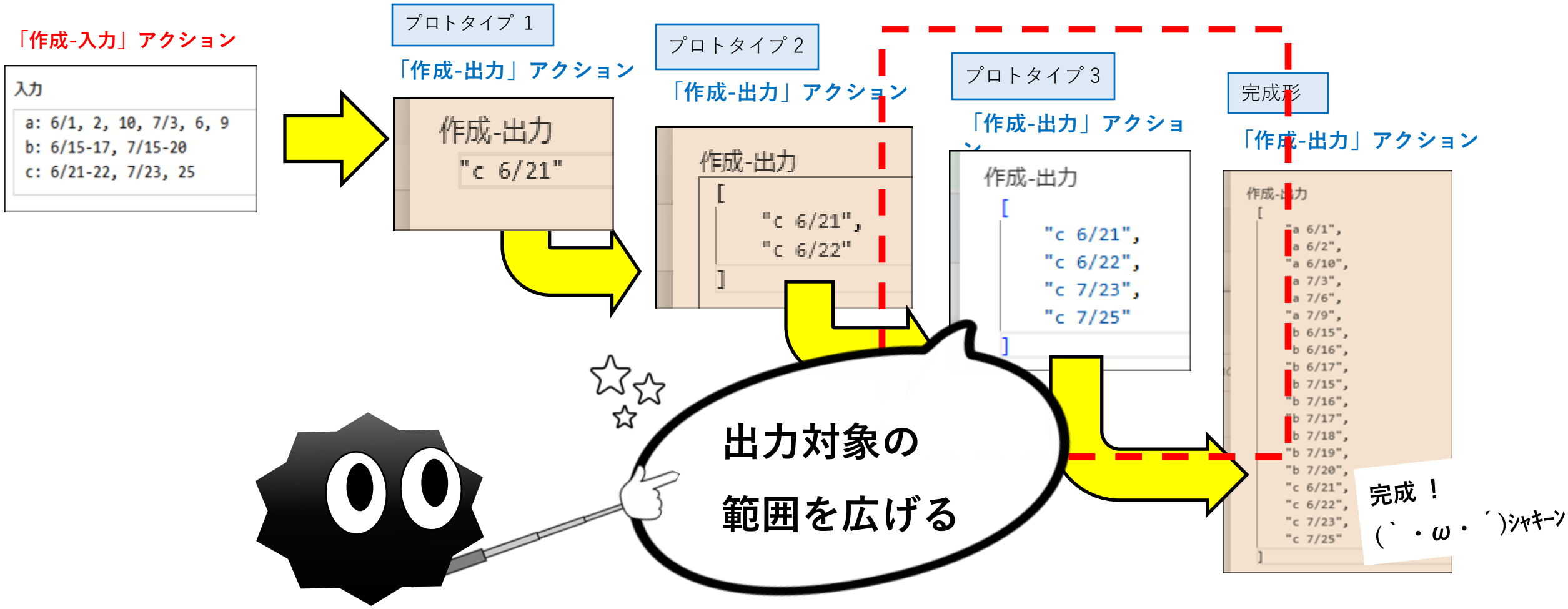
繰り返し処理に
置き換え済み

次に
この仮置きアクションを
繰り返し処理に
置き換えます。

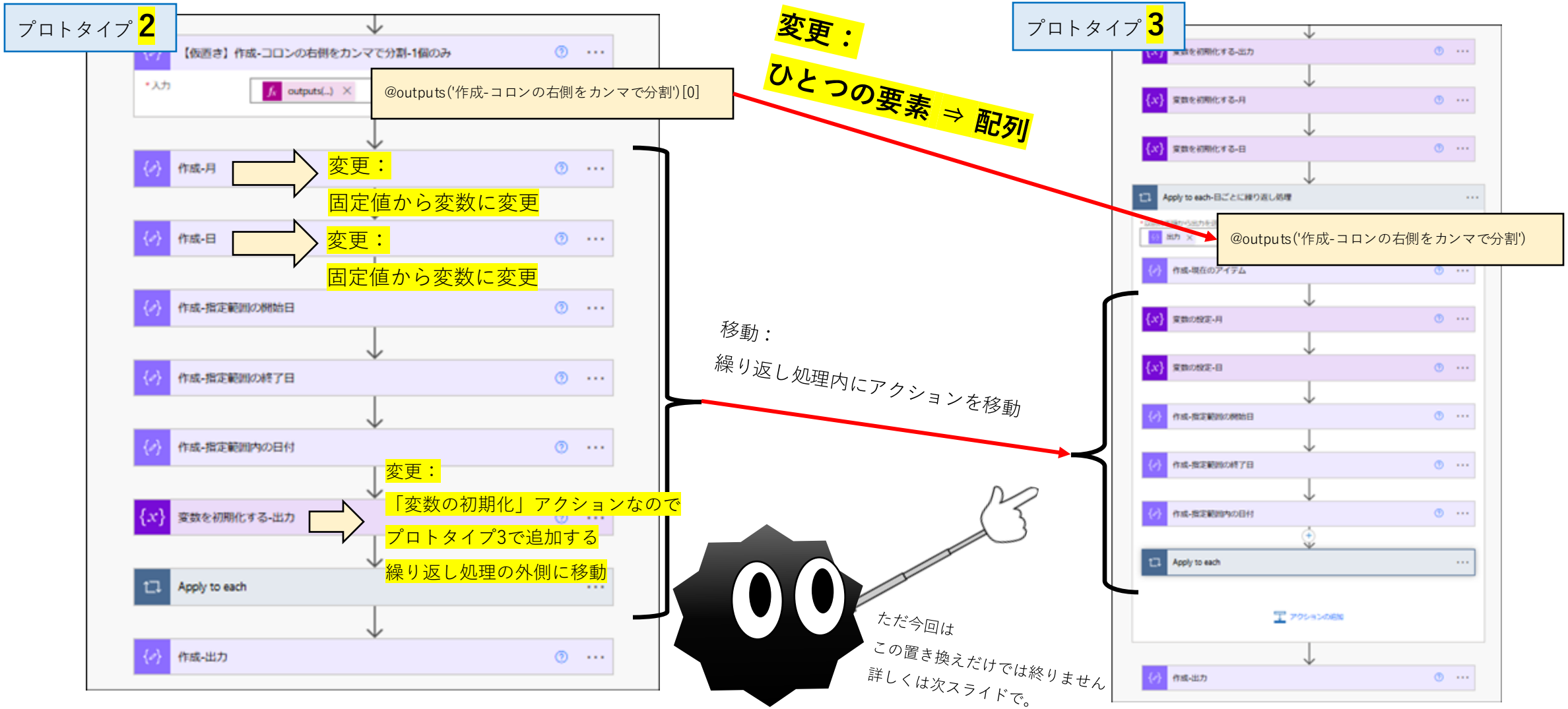


引き続き
フローの最終的な出力に
近い順に行います。

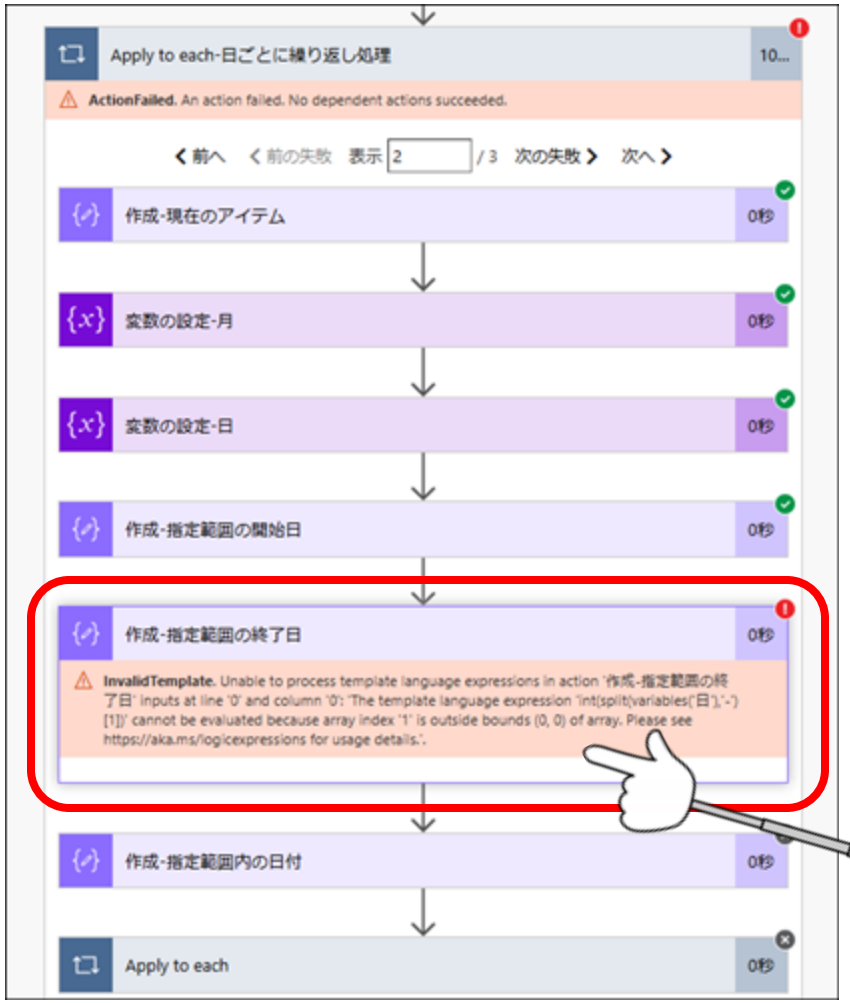
次にプロトタイプ3を作ります



プロトタイプ2 から プロトタイプ3への変化



追加実装の対応(= 入力値のパターン増加への対応)



処理対象の

入力値の範囲を広げることで

入力値のパターンが増え、

フローがエラーになる場合が

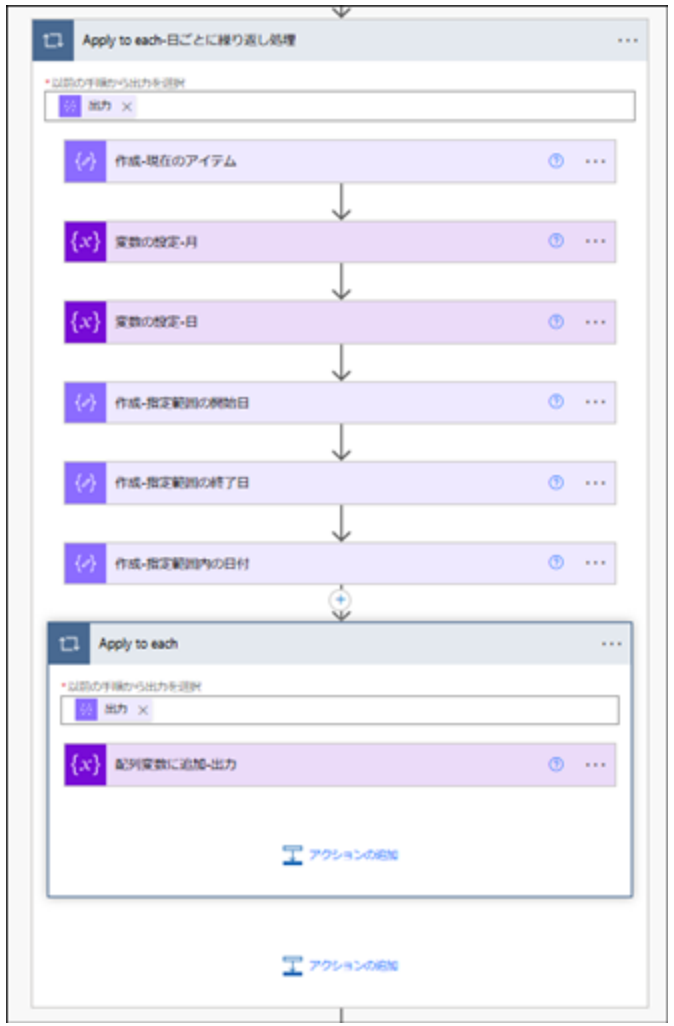
あります

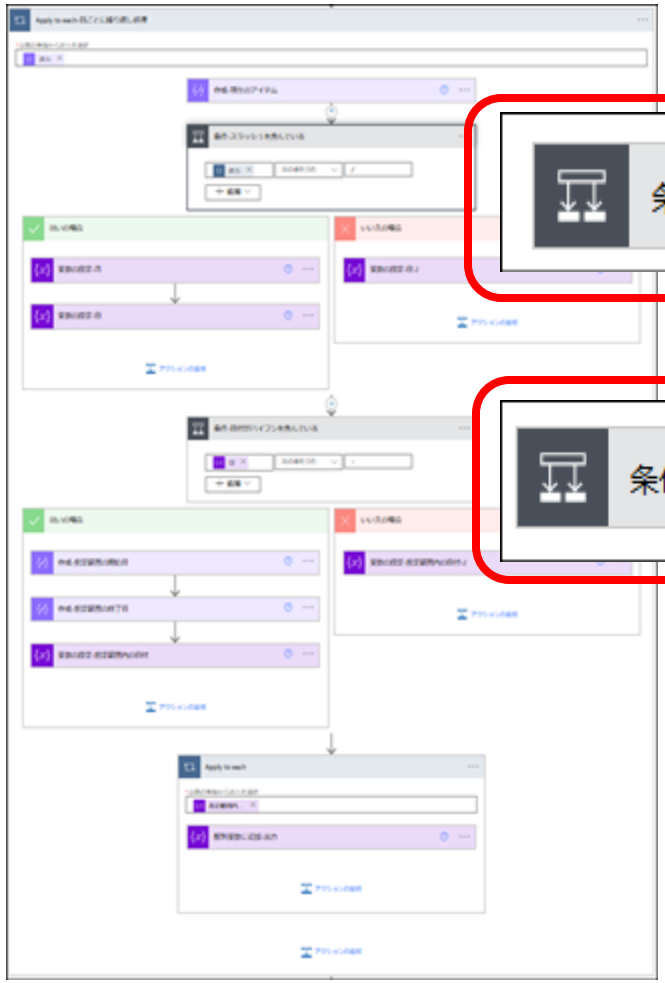
このエラーを

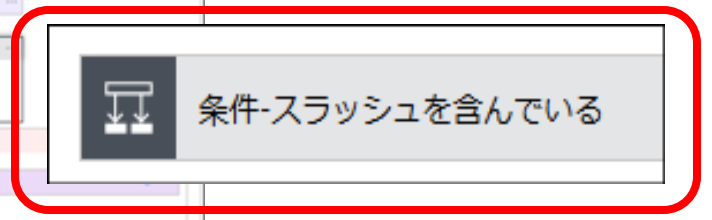
条件分岐の実装で対応します

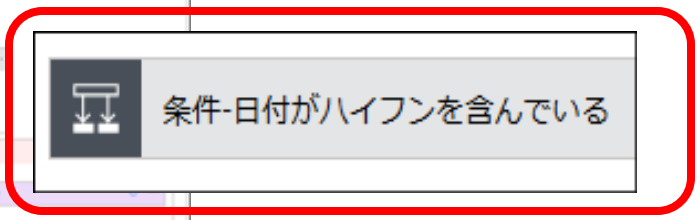
次のスライドで対応例を示します

追加実装の対応例





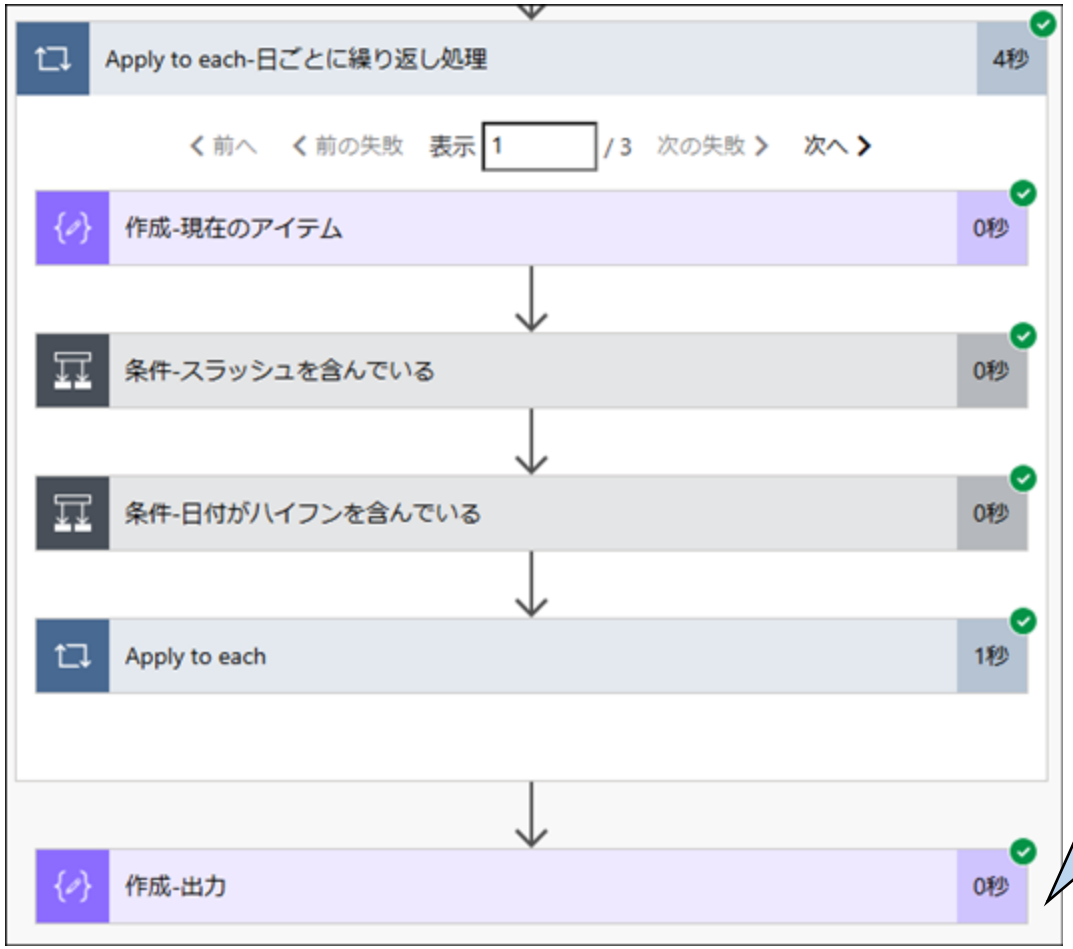


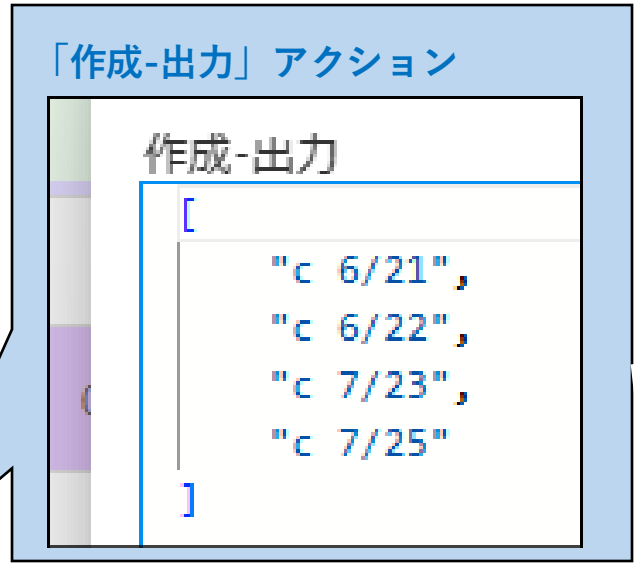


このフローでは
この2つの条件分岐で
対応します

追加実装の結果確認

エラーが回避され、期待通りの出力が得られたことを確認します

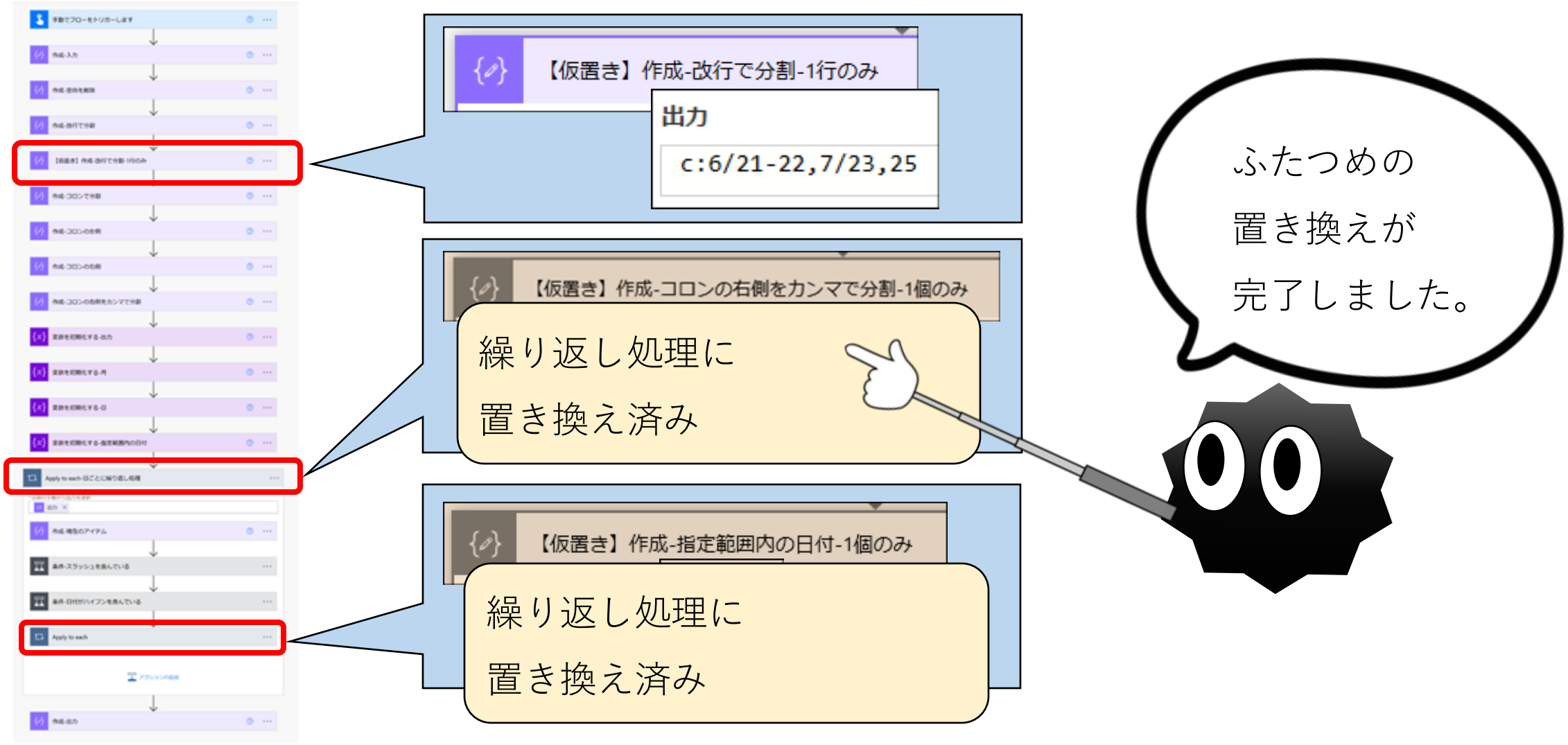




動作確認結果OK！

(｀・ω・´)ｼｬｷｰﾝ

仮置きのアクションを繰り返し処理に置き換え （ふたつめが完了）



次行程の完成形フローの作成でも置き換えを行います

【仮置き】作成-改行で分割-1行のみ

出力

c:6/21-22,7/23,25

【仮置き】作成-コロンの右側をカンマで分割-1個のみ

繰り返し処理に
置き換え済み

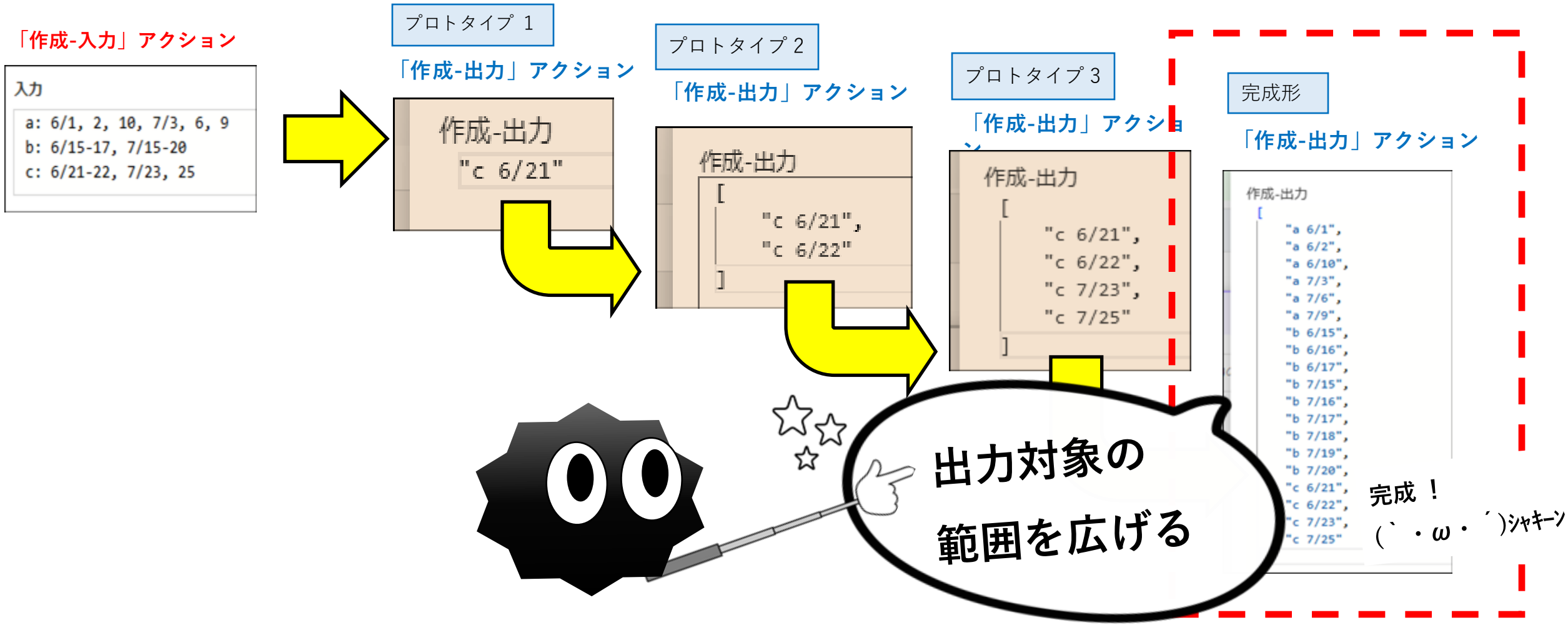
【仮置き】作成-指定範囲内の日付-1個のみ

繰り返し処理に
置き換え済み

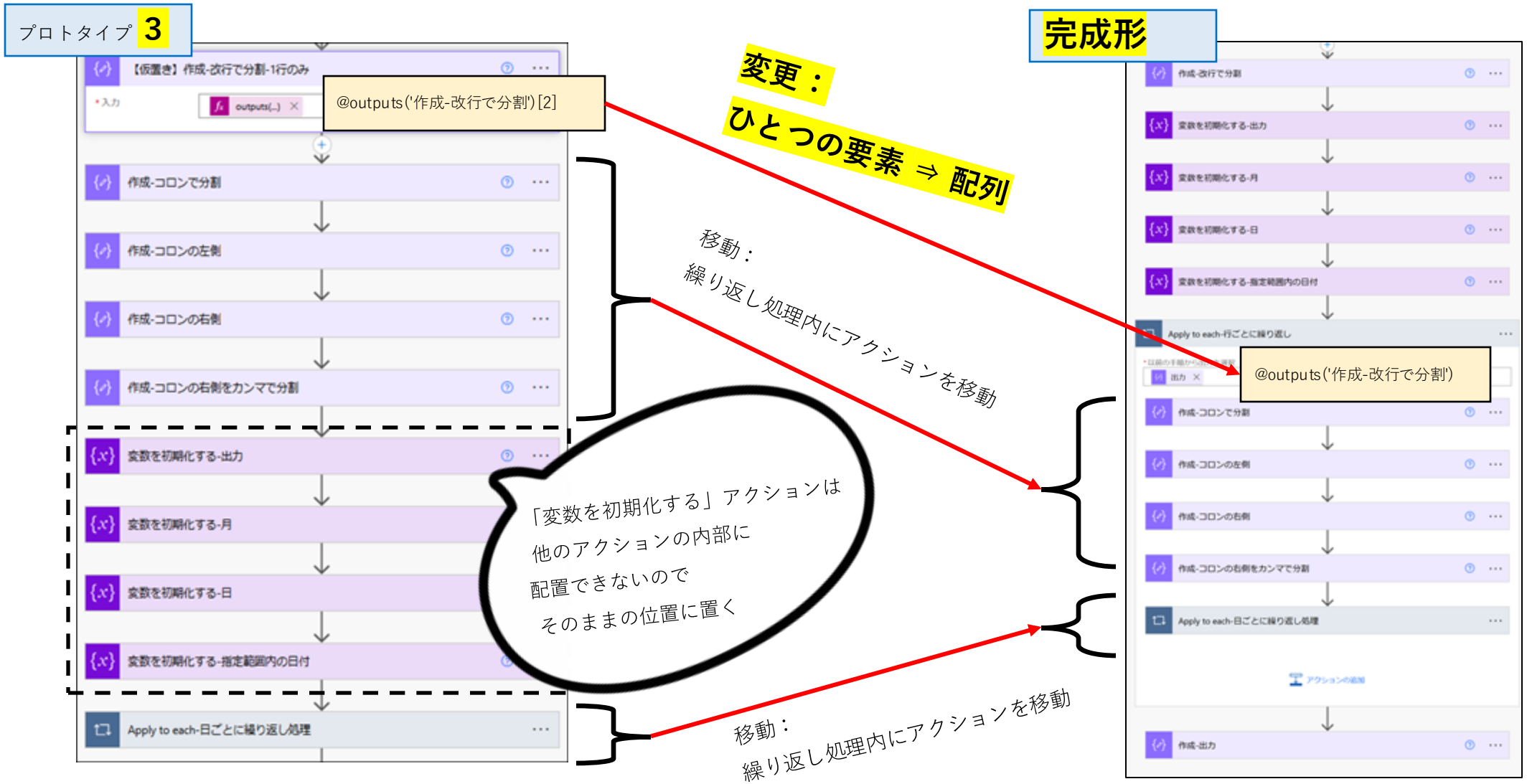
次に
この仮置きアクションを
繰り返し処理に
置き換えます。

この置き換えが完了したら
フローの完成です

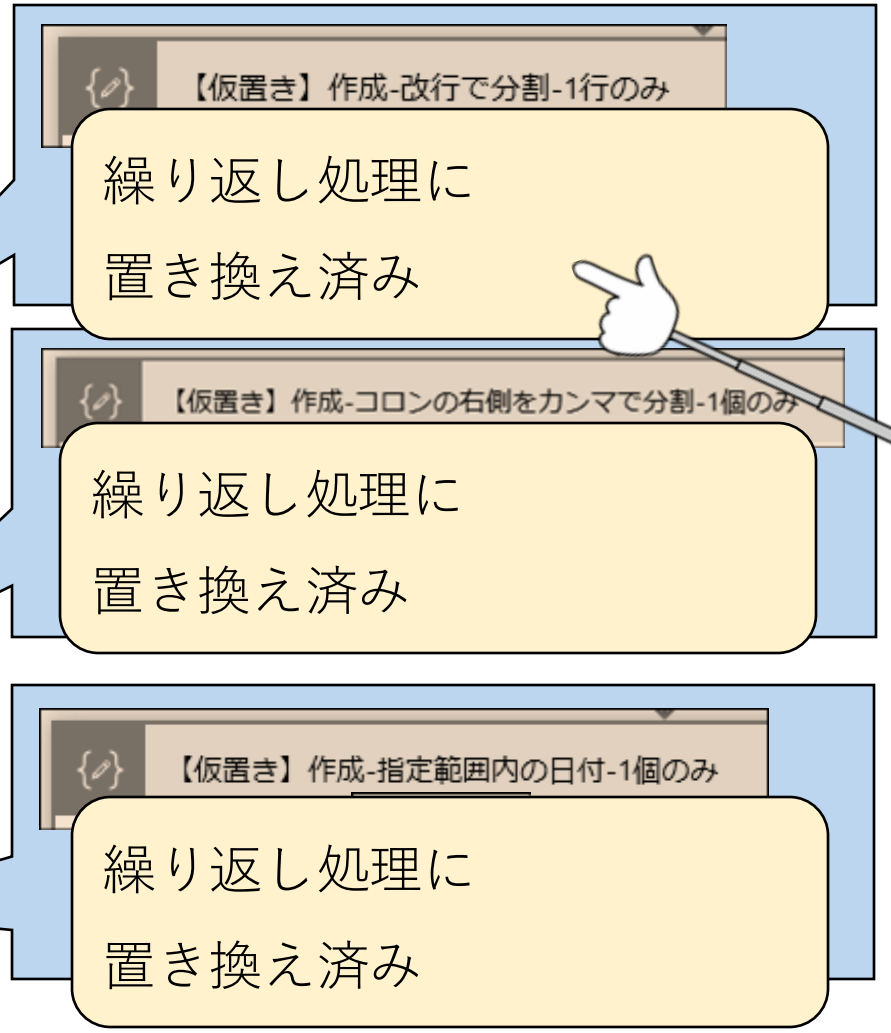
それでは完成形のフローを作ります



プロトタイプ3 から 完成形への変化



仮置きのアクションを繰り返し処理に置き換え （みっつめが完了）



完成

「作成-出力」アクション

作成-出力

```
[
    "a 6/1",
    "a 6/2",
    "a 6/10",
    "a 7/3",
    "a 7/6",
    "a 7/9",
    "b 6/15",
    "b 6/16",
    "b 6/17",
    "b 7/15",
    "b 7/16",
    "b 7/17",
    "b 7/18",
    "b 7/19",
    "b 7/20",
    "c 6/21",
    "c 6/22",
    "c 7/23",
    "c 7/25"
]
```

動作確認結果OK！

(｀・ω・´)ｼｬｷｰﾝ

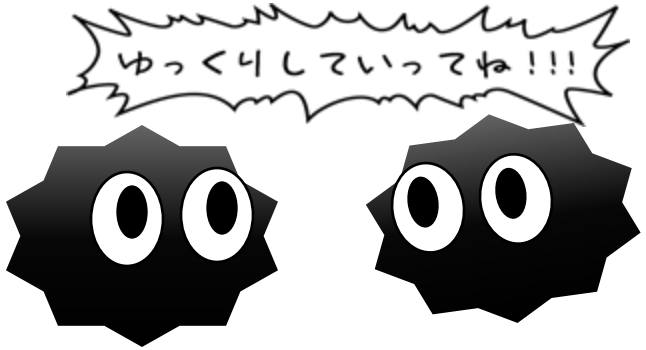
解説に使用したフローはGitHubで公開しています

解説内容の理解にご活用ください。このフローは下記ブログ記事で使用しています。

- 【Power Automate】【TIPS】【**連載その1**】フローの設計・実装における私なりのロジックの組み立て方
  https://wataruf.hatenablog.com/entry/2024/09/12/000000
- 【Power Automate】【TIPS】【**連載その2**】フローの設計・実装における私なりのロジックの組み立て方
  https://wataruf.hatenablog.com/entry/2024/09/14/000000
- 【Power Automate】【TIPS】【**連載その3**】フローの設計・実装における私なりのロジックの組み立て方
  https://wataruf.hatenablog.com/entry/2024/09/19/000000
- 【Power Automate】【TIPS】【**連載その4・最終回**】フローの設計・実装における私なりのロジックの組み立て方
  https://wataruf.hatenablog.com/entry/2024/09/25/000000

登壇で割愛した部分の
解説もあります。

GitHubのリンクも
ブログ記事内に
記載しています。



Power AutomateのフローやTIPSをブログで公開しています

- 興味をもっていただけたかたは下記のURLからいらしてください
  https://wataruf.hatenablog.com

フローに関する質問を受け付けています

- 匿名でのご質問も可です。
  - X（旧：Twitter）にマシュマロのリンクを公開しています。
  - 回答内容はブログで投稿します。

最後に

今回は私が普段は感覚的に行っていることを言語化してみました。

これがどなたかのお役に立てれば幸いです

「私はこの部分はこうやってます！」といった

フィードバック・ご意見をいただけると

とても嬉しいです。

終わりです。

(`・ω・´)ﾉｼ