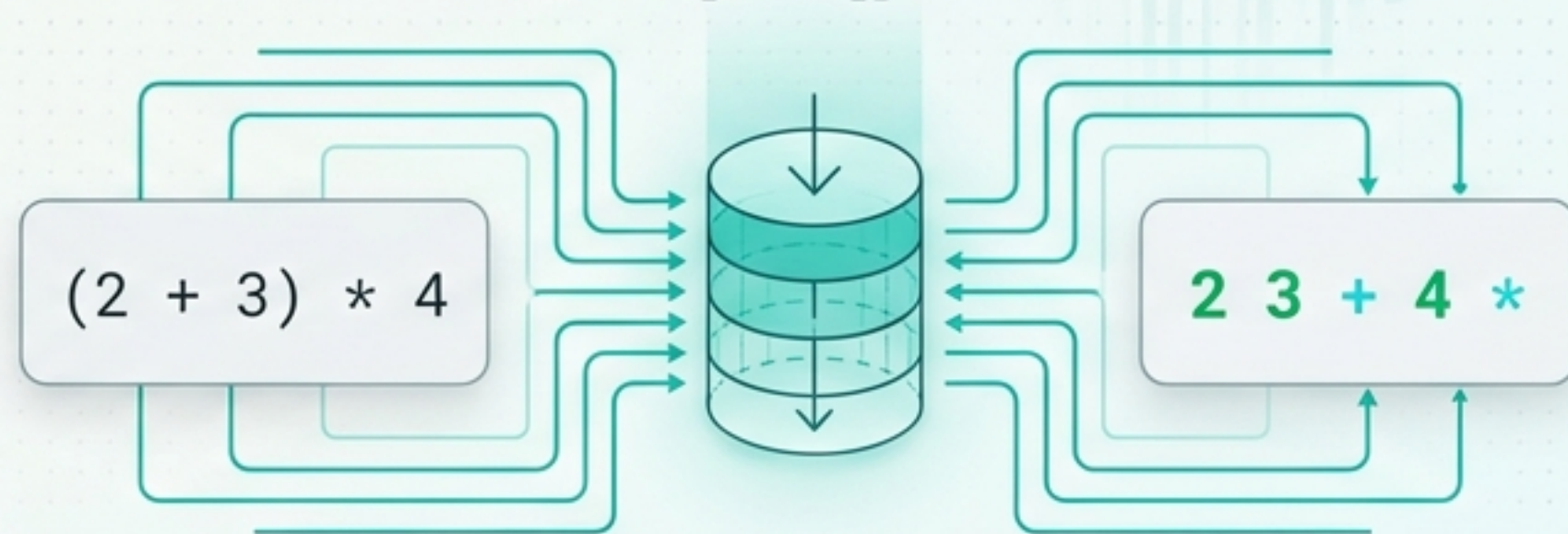


逆ポーランド表記法 (RPN) の解剖学

アルゴリズムの理論からPythonコード実装までを完全トレース



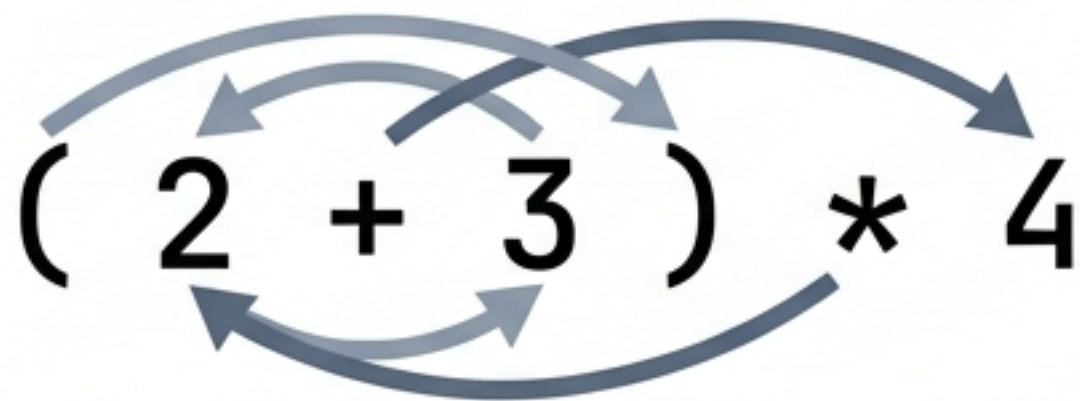
```
def convert_to_rpn(formula):  
    stack = [None] * 100  
    output_queue = []  
  
    if token.isdigit():  
        output_queue.append(token)  
    while stack:  
        token = stack.pop()
```

```
        if token.isdigit():  
            output_queue.append(token)  
        if stack[-1] == '(':  
            output(token)  
            stack.pop()  
        else:  
            while stack:  
                final_rpn_string = ' '.join(output_queue)
```

なぜコンピュータは「逆ポーランド表記法」を愛するのか？

ヒトの数式（中置記法）

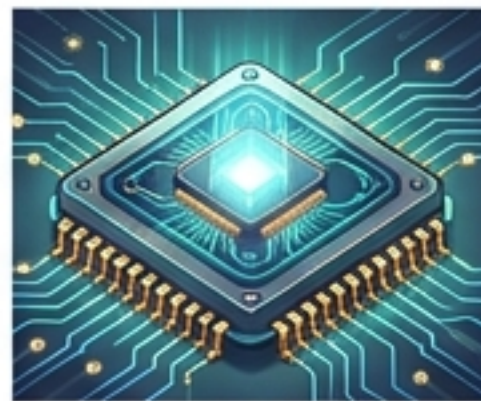
$$(2 + 3) * 4$$



「括弧」や「優先順位」を把握するため、式全体を行ったり来たりして見渡す必要がある。

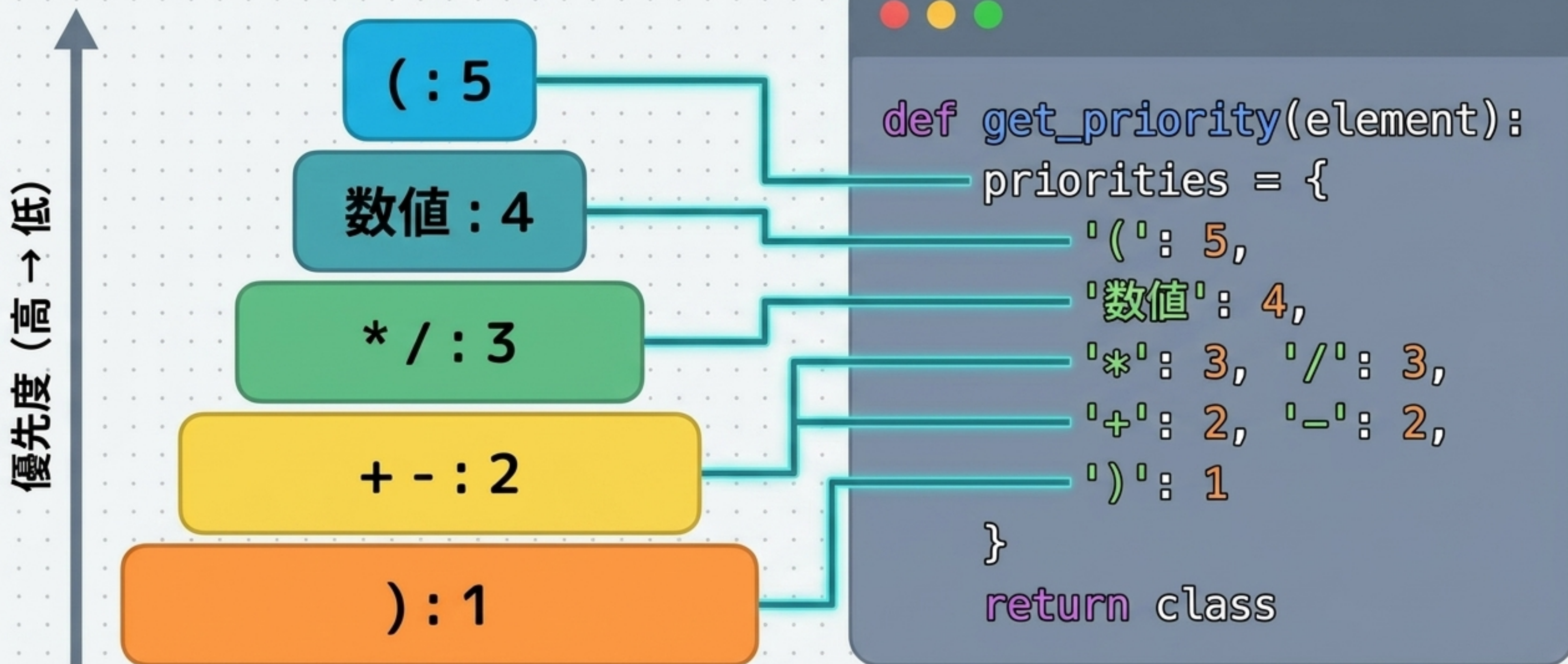
コンピュータの数式（RPN）

$$2 \ 3 \ + \ 4 \ *$$



左から右へ順番に読み込み、演算子が出たら直前の2つの数値を計算するだけ。括弧が一切不要。

変換の心臓部：演算子・要素の「優先度」階層



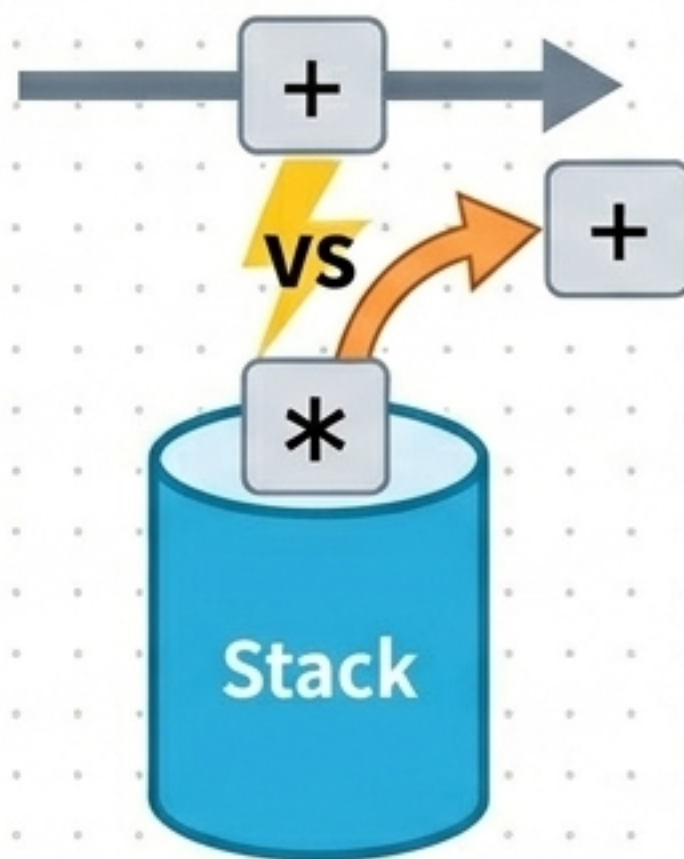
スタックを用いた変換アルゴリズムの全容

1. 参照 (Read)



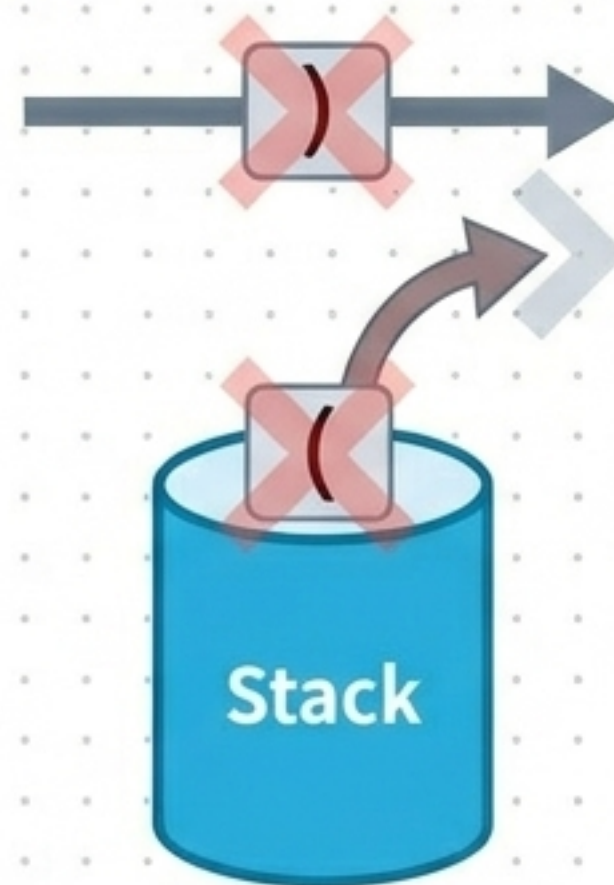
要素を左から1つ読む。
数値ならそのまま出力へ。

2. 判定 & Pop (Battle)



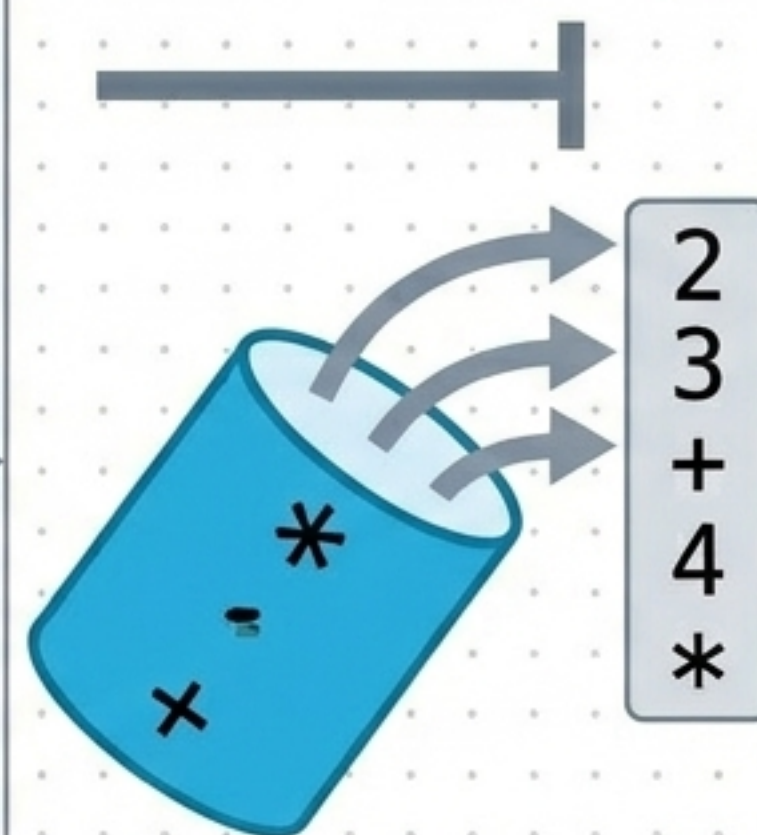
スタック先頭の優先度と
バトル。スタック側が強
ければ追い出す (Pop)。

3. 括弧の相殺 (Cancel)



') ' が来たら、 ' ('
が出るまでPopし、
両方とも破棄する。

4. 残骸の回収 (Flush)



全要素を読み終わったら、
スタックに残った演算子を
すべてPopして出力。

手動トレース： $(2 + 3) * 4$ の変換プロセス

人間の脳による塊（チャンク）での直感的な変換

$(2 + 3) * 4$

$[2 \ 3 \ +] * 4$

$2 \ 3 \ + \ 4 \ *$

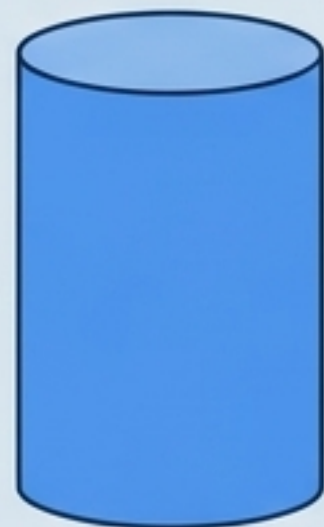
State A: '+' のPush



入力: +
出力: 2

State B: 括弧の相殺

~~(~~ ~~)~~



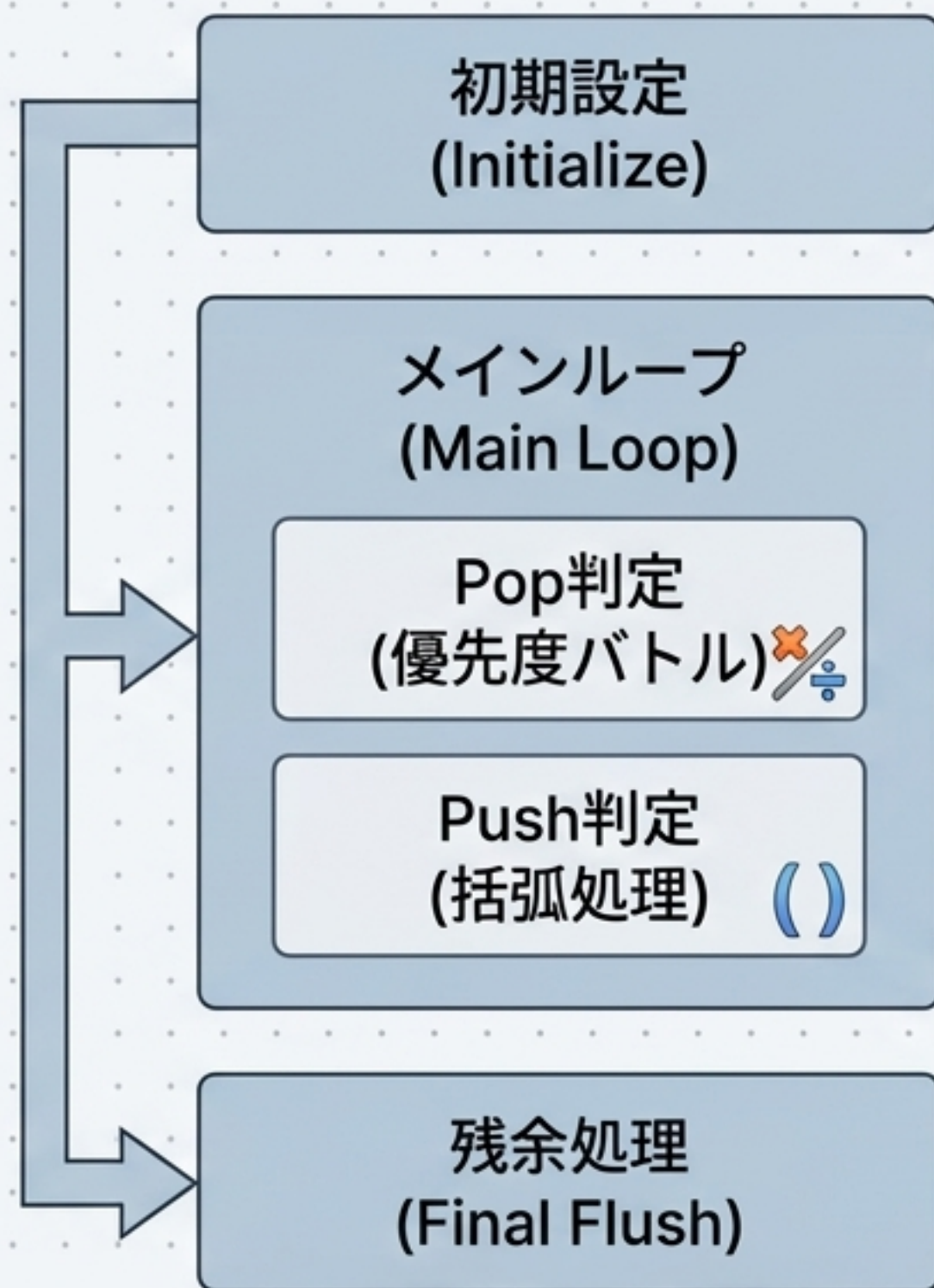
入力:)
出力: 2 3 +

State C: 最終Flush



入力: 終了
出力: 2 3 + 4 *

Algorithm to Code: プログラムラム構造の俯瞰



```
def convert_to_rpn(formula):  
    # 1. 初期設定  
    result = []  
    stack = [None] * 100  
  
    # 2. メインループ  
    for element in formula:  
        while ... (Pop処理) ... :  
            ...  
        if ... (Push/相殺処理) ... :  
            ...  
  
    # 3. 残余処理  
    while stack_top is not None:  
        ... (全回収) ...
```

巧妙な初期設定：エラーを防ぐ「番兵 (Null)」の導入

課題: 空のスタックへのアクセス



~~if stack_is_empty():~~

スタックからPopする際、毎回「空ではないか？」の判定（空欄[カ]）が必要となり、コードが煩雑になる。

解決: 番兵 (Sentinel) の配置



stack[0] = None # 空欄[オ]: 0

底に優先度0のダミーを配置。どんな演算子が来ても優先度バトルで勝てないため、絶対にPopされず、空判定のif文を排除できる。

優先度のバトル：スタックからの「追い出し (Pop)」

疑似コード: `while ( [ク] かつ stackTop が '(' 以外の間 )`

私の方が優先度が高い
(または同じ) だから先に出る!



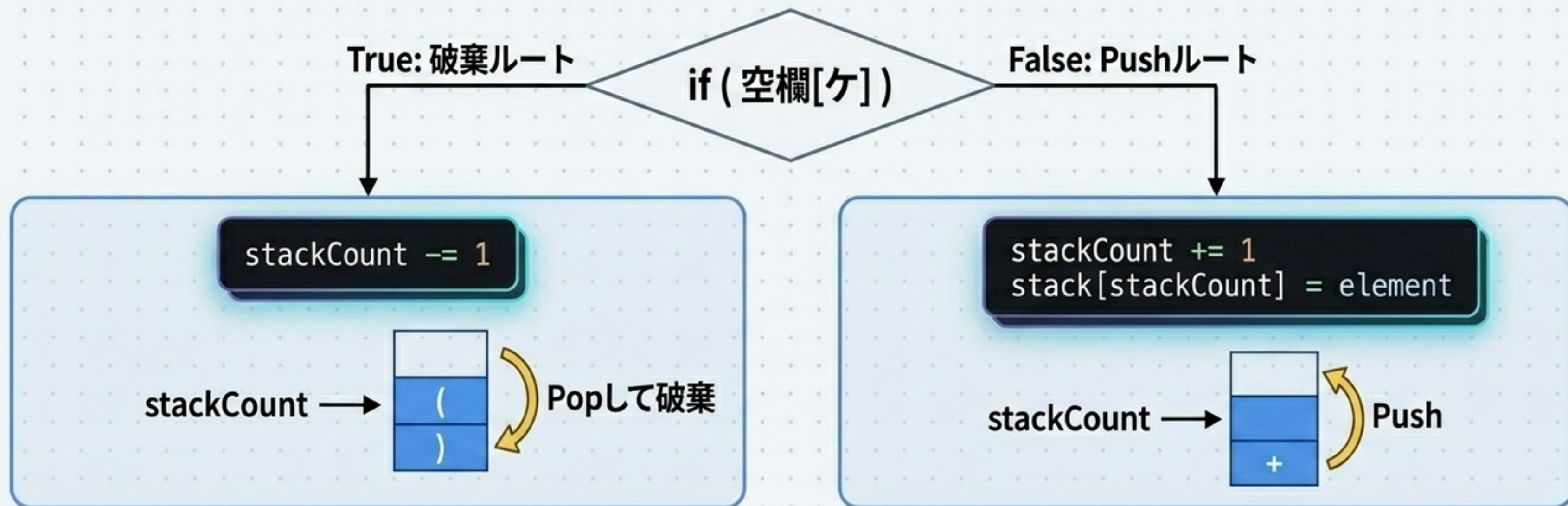
+

element
優先度: 2

```
def convert_to_rpn(formula):  
    # 1. 初期設定  
    result = []  
    stack = [None] * 100  
  
    while get_priority(stack_top) >= element_priority:  
        # 追い出し処理 (Pop)  
        result.append(stack_top)  
        stackCount -= 1
```

したがって条件[ク]は「GetPriority(stackTop) が elementPriority 以上」となる。

分岐処理：要素のPushと「括弧の相殺」



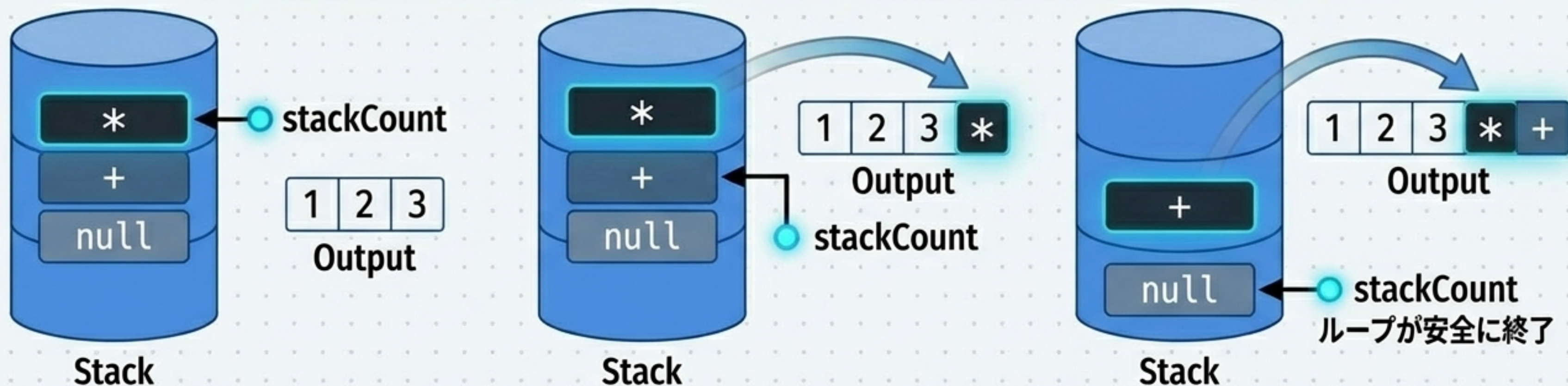
閉じ括弧 ')' が来た時、スタック内の '(' と相殺する。

それ以外の演算子はスタックへ積む。

結論: 相殺が発動する条件[ケ]は「GetElement(i) が ')' に等しい」

ループ終了後：スタックに残った要素の吐き出し

`while ( stack[stackCount] が null でない間繰り返す )`



全要素を読み終わった後、スタックに残存する演算子をすべて出力へ。ここでも底にある「番兵 (null)」のおかげで、配列の範囲外アクセスを防ぎつつ安全にループを終了できる。

実用性への昇華：簡易的な入力エラーチェック

		現在の要素 (GetElement(i))				
		5 (")	4 (数値)	3 (* , /)	2 (+ , -)	1 (")
直前の要素 (GetElement(i-1))	5 (")	OK	OK	OK	OK	OK
	4 (数値)	OK	Err	OK	OK	OK
	3 (* , /)	OK	OK	3 (* , /)	Err	OK
	2 (+ , -)	OK	OK	3 (* , /)	Err	OK
	1 (")	OK	OK	OK	OK	OK

直前の要素と現在の要素の組み合わせだけで、構文の正しさを網羅的に判定する状態遷移マトリクス。

エラー判定のロジック解剖

[コ] Err (数値 → 数値)

2~~×~~3

アルゴリズムの大前提として「数値は1桁限定」。演算子を挟まずに数字が連続するのは不正な式。

[サ] OK (数値 → 演算子)

2 * ✓

通常の数式の自然な流れ。数値の後に掛け算などの演算子が来るのは文法的に正しい。

[シ] OK (演算子 → 数値)

* 3 ✓

演算子の次に演算対象の数値が来る、数式として必要不可欠な正しい表現。

[ス] Err (演算子 → 演算子)

+~~×~~*

二項演算子が2つ以上連続して並ぶことは数式として絶対にあり得ないためブロック。

理論の証明：Pythonでのテスト実行結果



> 入力: ['1', '+', '2', '*', '3']

> 出力: ['1', '2', '3', '*', '+']

> 設問1(1)のテスト: ['(', '2', '+', '3', ')', '*', '4']

> 結果: ['2', '3', '+', '4', '*']



手動トレース（理論）
の結果と完全に一致

まとめ：アルゴリズム設計から学ぶプロの思考



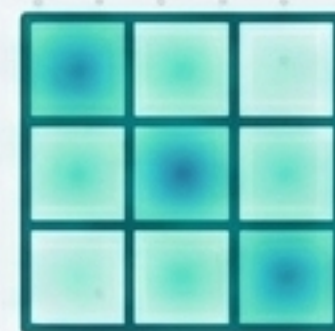
1. データ構造の力

スタックを利用することで、人間にとって複雑な括弧の構文解析が、機械にとって単純な「優先度バトル」へと劇的に簡略化される。



2. 番兵 (Sentinel) の美学

データ構造の底に「意図的なダミー」を仕込むことで、例外処理のif文を排除し、バグの入り込む隙のない堅牢で美しいループを実現する。



3. 状態遷移の網羅性

直前と現在の要素の組み合わせによるマトリクス評価は、すべてのエッジケースを漏れなく定義し、安全な処理を担保する強力な設計手法。