

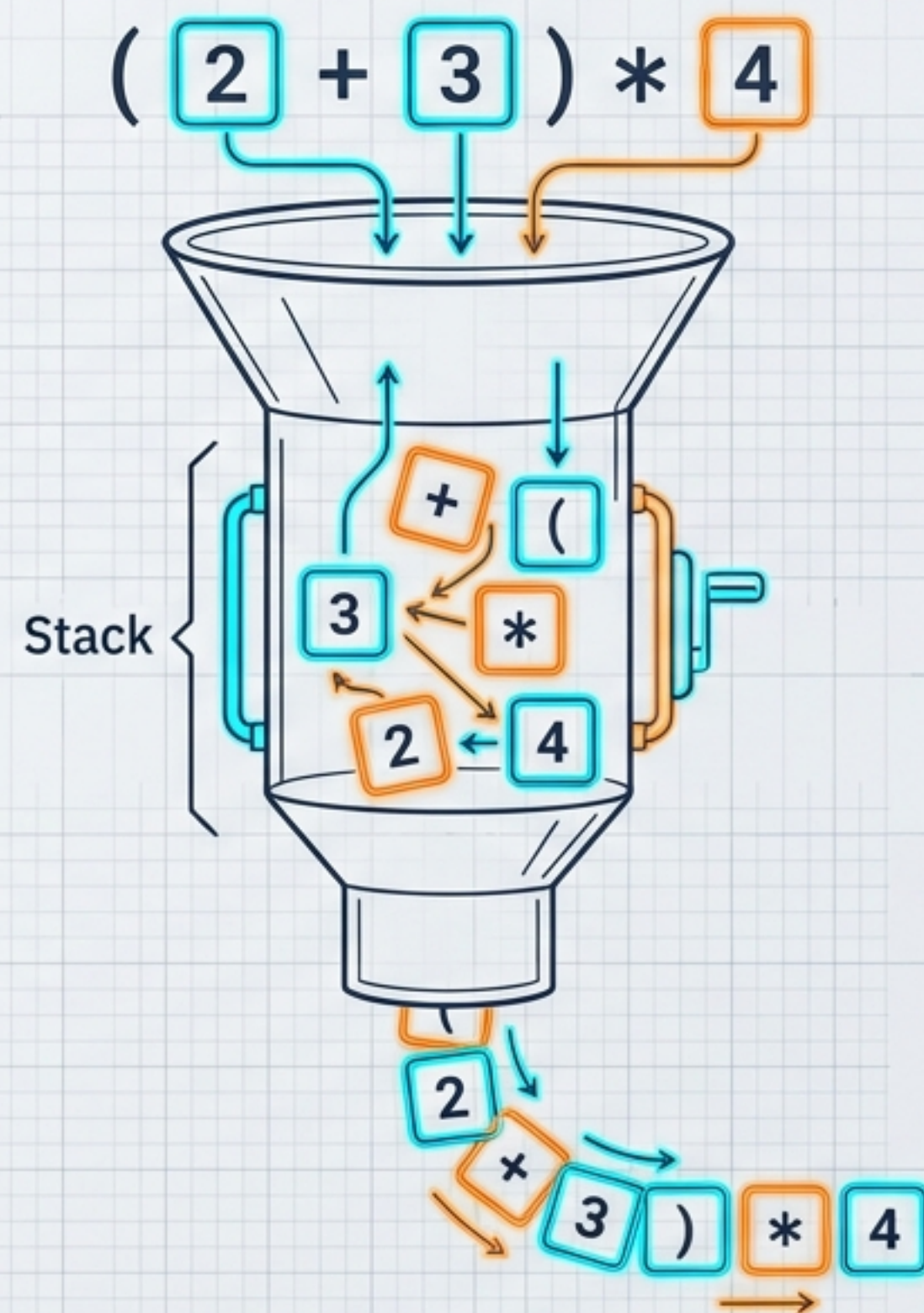
アルゴリズム構築の旅 と思考の解剖

逆ポーランド表記法への変換アルゴリズム

平成25年度 春期 応用情報技術者試験 午後 問2 解説

本資料のミッション：

- 解答そのものは一切提示しません。
- 問題に隠された「手がかり」と、自力で正解にたどり着くための「思考のプロセス」のみを解き明かします。



Reverse Polish Notation Conversion

逆ポーランド表記法のコアコンセプト

一般的な表記
(中置記法)

A + B

逆ポーランド表記
(後置記法)

A B +

1. 演算子の後置

演算子を対象となる要素の「後」に配置する。

2. 括弧からの解放

計算順序が記法自体に組み込まれるため、カッコが一切不要になる。

3. 機械との親和性

左から順に処理するだけで計算が完了するため、コンピュータ（スタック処理）にとって極めて都合が良い。

設問1(1) アプローチ：直感的な「塊」メソッド



Focus Point

優先度が最も高い「カッコの中」から順番に塊を作っていく。

Step 1: カッコ内の処理

$$(2 + 3) * 4$$



Step 2: 形成された塊を単一要素とみなす

$$[2 \ 3 \ +] * 4$$



Step 3: 残りの処理

$$[2 \ 3 \ +] 4 *$$

演算子 * が全体の最後尾へスライドしていく
(最終的な解答は自力で導き出そう)。

設問1(1) アプローチ：スタックの動きのトレース

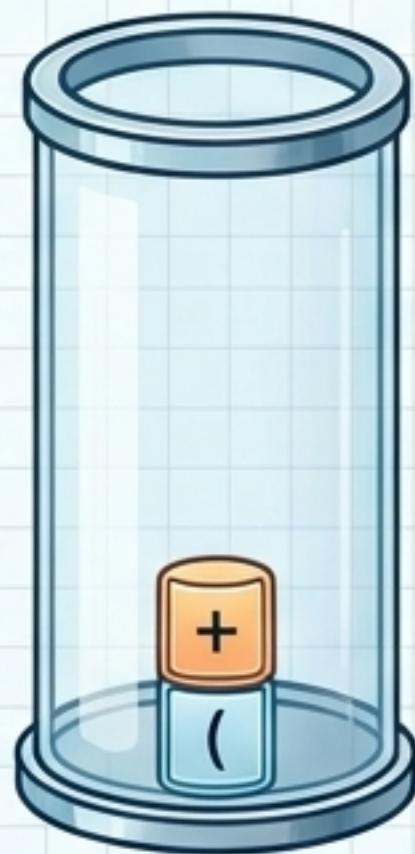
Rule Check

A. (数値):
そのまま横の変換後配列へ直行。

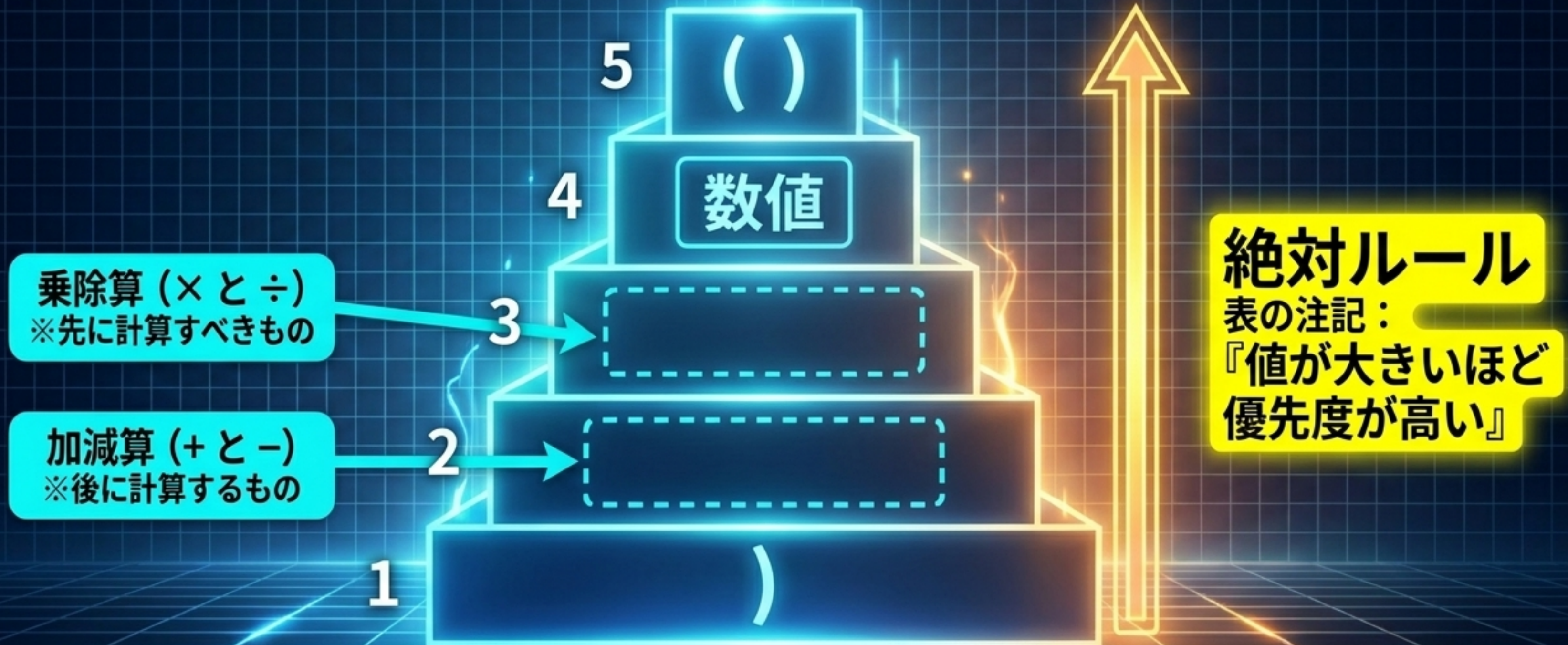
B. (演算子・カッコ):
一旦、縦のスタックへ Push。

 Focus Point: トリガー発動

閉じカッコ) が読み込まれた瞬間の挙動。
スタックの中にある (にぶつかるまで、スタック
の演算子が次々と Pop されて配列へ押し出される。



設問1(2) アプローチ：演算子の優先度階層



数学の常識（掛け算・割り算は、足し算・引き算より先に計算する）と、
表の絶対ルールを照らし合わせれば、[ア]～[エ]に何が入るかが自然と浮かび上がる。

設問2(1) アプローチ：番兵（Sentinel）を守る論理



表2 演算要素の優先度

演算要素	優先度
(	5
数値	4
ア, イ	3
ウ, エ	2
)	1

注記 値が大きいほど優先度が高い。

✓ Rule Check

スタックから要素がPopされる条件

スタック先頭の優先度 $\geq$ 参照した要素の優先度

🔍 The Mystery

この null（番兵）が誤って飛び出してしまうとはシステムが破綻する。絶対にPopさせないためには、どのような優先度を設定すべきか？

⚙️ Logical Process

登場するどんな演算要素と比較されても、上記の条件式が成立しないようにする。
つまり、表の中で最も低い優先度（レベル1）よりも、さらに『低い値』を null に設定すればよい。[オ]

設問2(2) アプローチ：番兵がもたらすコードの劇的な簡略化

⚠ 番兵なしの世界



要素を取り出す際、常に「底が抜けていないか（中身が空っぽではないか）」を毎回確認しなければ、システムがクラッシュする危険性がある。

✓ 番兵ありの世界

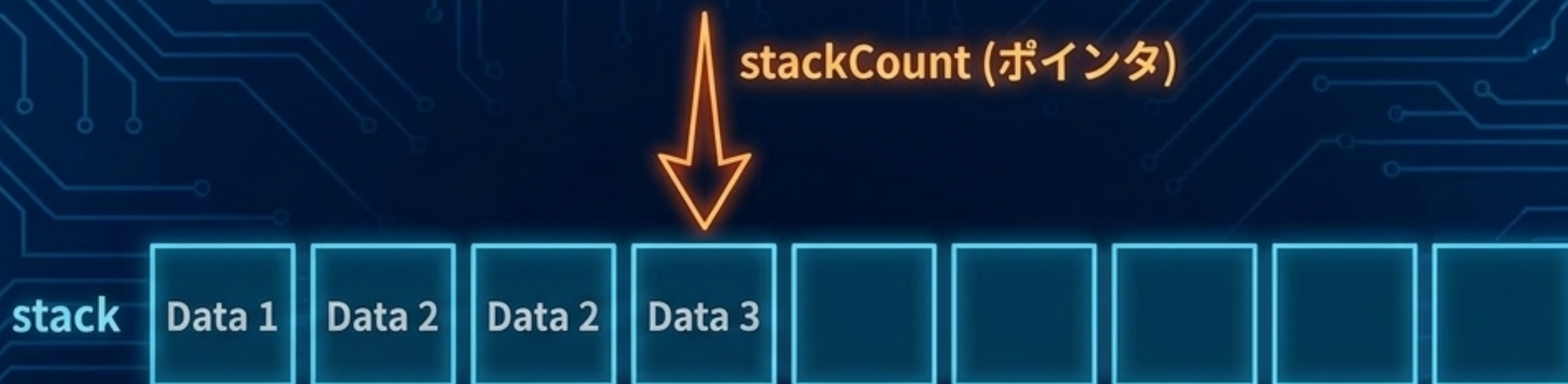


底に null が常駐しているため、スタックが「完全に空になる」という状態が物理的に発生しなくなる。

Logical Process: 記述の省略

この工夫により、本来の手順(2)の冒頭にあった「ある確認作業」をプログラムに記述する必要が消滅した。その省略された『○○であるか否か』という判定処理とは何か？ **[力]**

設問2(3) [キ] アプローチ：スタック先頭へのアクセス手法



stack	stackCount
データを順番に格納していくためのコンテナ（配列）。	現在データが何個入っているかを示し、同時に「今一番上にあるデータ」の位置を指し示すインジケータ（ポインタ）。

Logical Process

アルゴリズムを実行するために「スタックの先頭にある演算要素」を変数 `stackTop` に代入したい。
この配列名とポインタをどのように組み合わせれば、常に一番上の要素を的確に指定できるか？

設問2(3) [ク] アプローチ：ループ条件の翻訳

アルゴリズムの仕様

スタックの先頭の
演算要素の優先度が、
参照した演算要素の
優先度**以上**なら

プログラムの変数

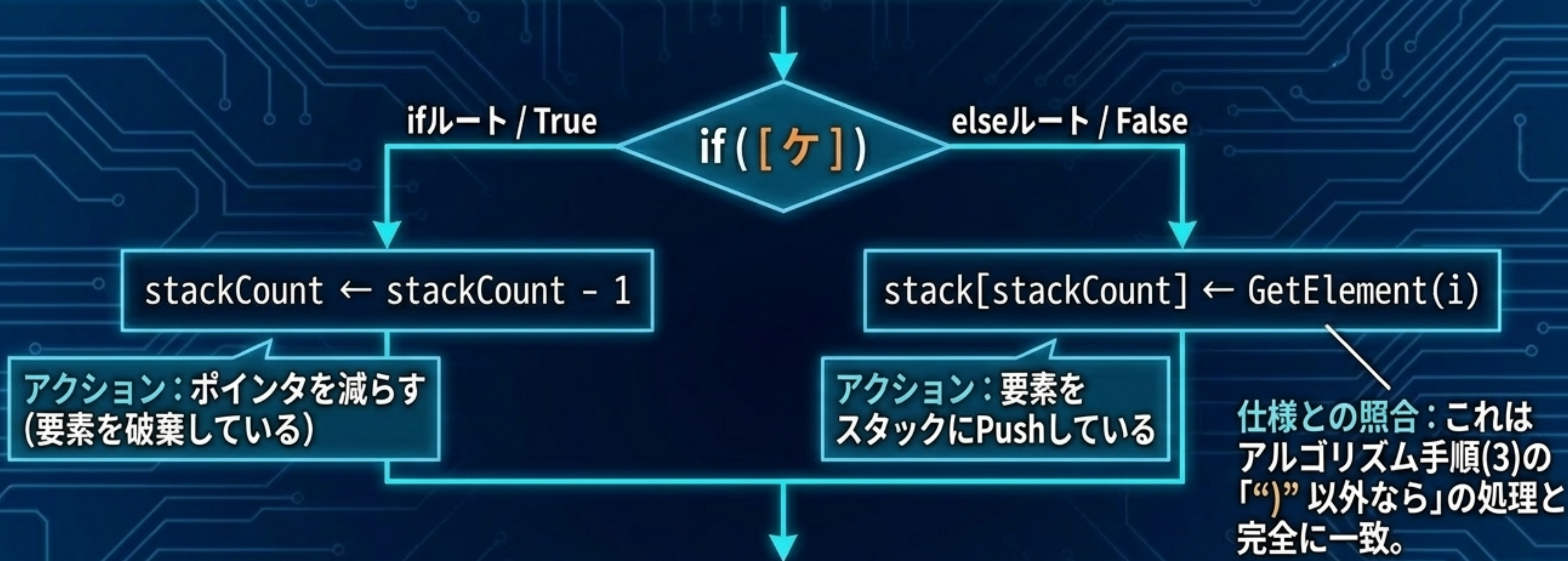
stackTop

elementPriority

Logical Process: 単纤の合纜

単純に変数を比較するのではなく、stackTop が持つ「優先度」を評価する関数 GetPriority() を適用した上で、これら2つを比較する関係式を構築する。

設問2(3) [ケ] アプローチ：条件分岐の逆算



Logical Process (Reverse Engineering)

else 側が「(」以外」を処理しているならば、if 側の条件は「まさに参照した要素が「)」である場合」となる。
これをプログラム内の変数 GetElement(i) や優先度 elementPriority を用いてどう表現するか？

設問3へのアプローチ：エラーチェックマトリクスの解剖

横軸：GetElement(i)
= 現在参照している要素

縦軸：GetElement(i-1)
= 1つ前の要素

	5 (“(”)	4 (数値)	3(ア, イ)	2(ウ, エ)	1 (“)”)
5 (“(”)					
4 (数値)					
3(ア, イ)					
2(ウ, エ)					
1 (“)”)					



The Diagnostic Goal

このマトリクスは、隣り合った2つの要素の並びが、四則演算の数式として「**文法的に正しいか (OK)**」それとも「**ありえない不正な並びか (Err)**」を判定する関所である。

Logical Process (Reverse Engineering)

このマトリクスは、隣り合った2つの要素の並びが、四則演算の数式として「**文法的に正しいか (OK)**」、それとも「**ありえない不正な並びか (Err)**」を判定する関所である。

設問3 [コ] アプローチ：隠された「大前提」のルール

Critical Condition (大前提ルール)

アルゴリズムの仕様：「数値は 0～9 の 1桁の数とする」



Logical Process

もし数式の中に **2** があり、その直後に **3** が来たらどうなるか？

- システムはこれを **23** という2桁の数字として結合・認識することができず、解釈が**破綻**する。
- 結論：ゆえに、「数値」の直後に「数値」が並ぶ交点の評価は、**不可避免的にエラー**として**処理**されなければならない。

設問3 [サ][シ][ス] アプローチ：数式文法のパターンテスト

Pattern 1 (数値 → 演算子) [サ]

2 *

この後に数値が続けば
数式として成立する。



判定：文法的に正しい

Pattern 2 (演算子 → 数値) [シ]

* 3

この前に数値があれば
数式として成立する。



判定：文法的に正しい

Pattern 3 (演算子 → 演算子) [ス]

+ *

二項演算子が連続して並ぶことは、
四則演算のルール上絶対にあり得ない。



判定：明らかな不正

これらのパターンの正誤を、表4の該当する交点に当てはめることで判定が完成する。

Synthesis: 堅牢なアルゴリズムを構成する3つの歯車

スタックというデータ構造を活用した、淀みのない**記法変換プロセス** (設問1)

Gear 1:
コアロジック
(The Engine)

Gear 2:
安全性
(The Shield)

null という番兵 (Sentinel) を配置することで、システムダウンの恐怖を排除し、コードを極限まで**シンプル**にする**ハック** (設問2)

不正な入力を水際で弾き返す、緻密な**文法エラーチェック機構** (設問3)

Gear 3:
堅牢性
(The Filter)

問題を解くための単なる空欄補充ではない。これら3つの要素が美しく統合されることで、初めて「**実用基準のThe Technical Blueprint (設計図)**」が完成する。