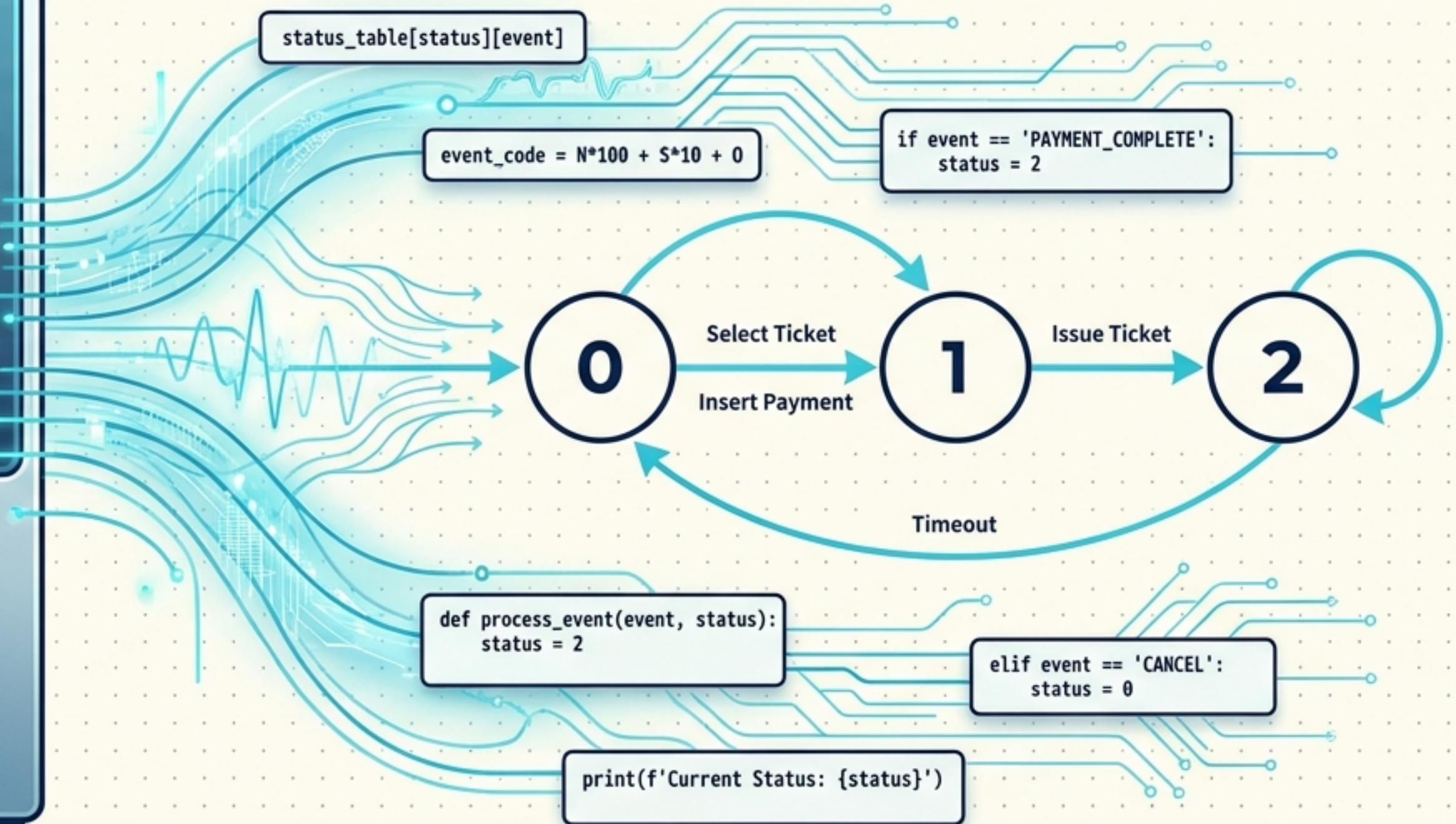


2019s 応用情報技術者試験 午後問3

券売機システムの状態遷移とPython実装を完全解剖



券売機システムの要件：商品は4つの属性を持ち、特定の組み合わせで値引きが発生する



メイン

+



サイド1

+



サイド2

=



1回の購入につき1回のみ適用

N: 分類番号

- 1: メイン
- 2: サイド1
- 3: サイド2
- 4: オプション

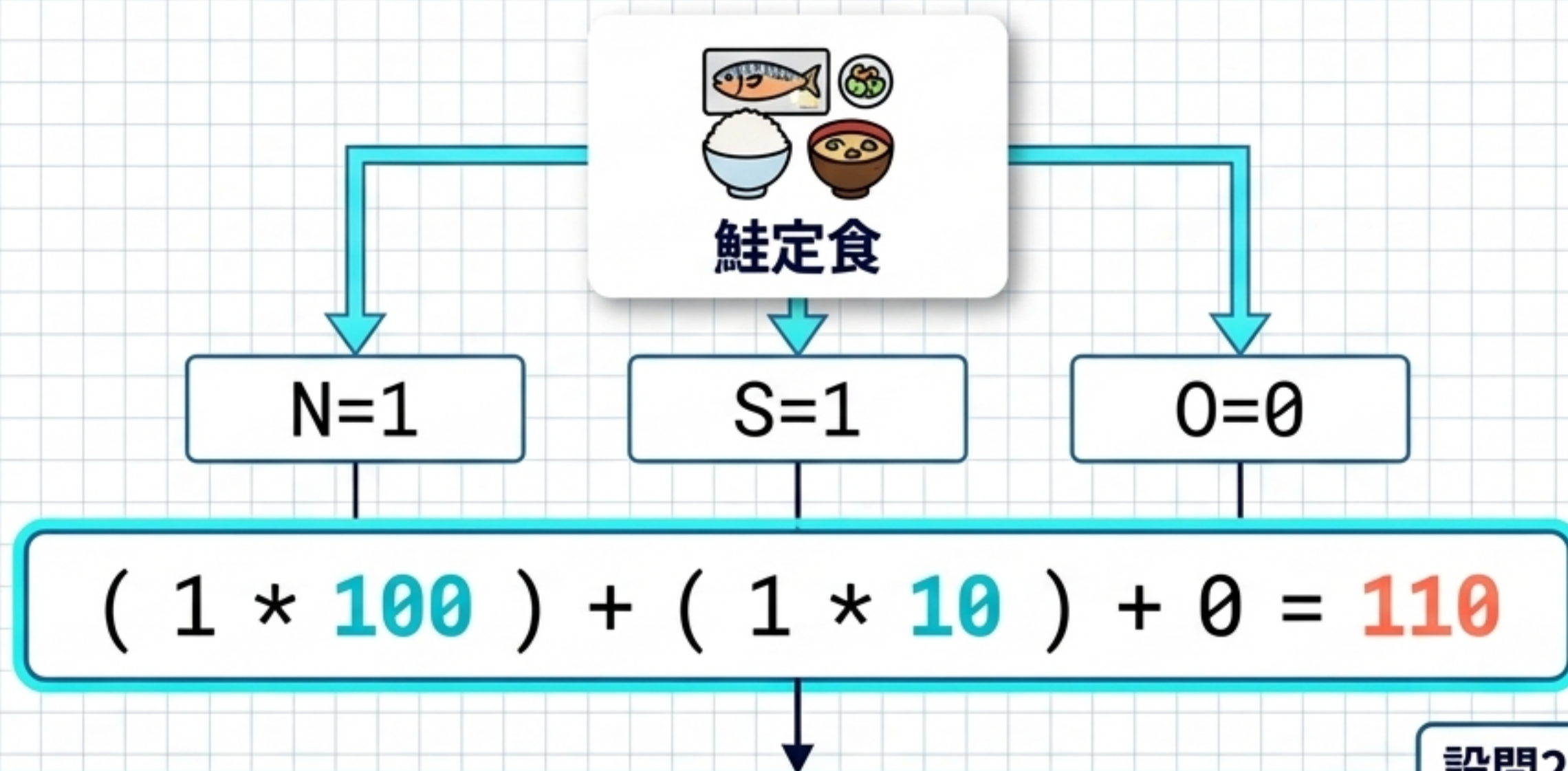
S: セット適用可否

- 1: 可
- 0: 不可

O: オプション情報

0以外は指定必須

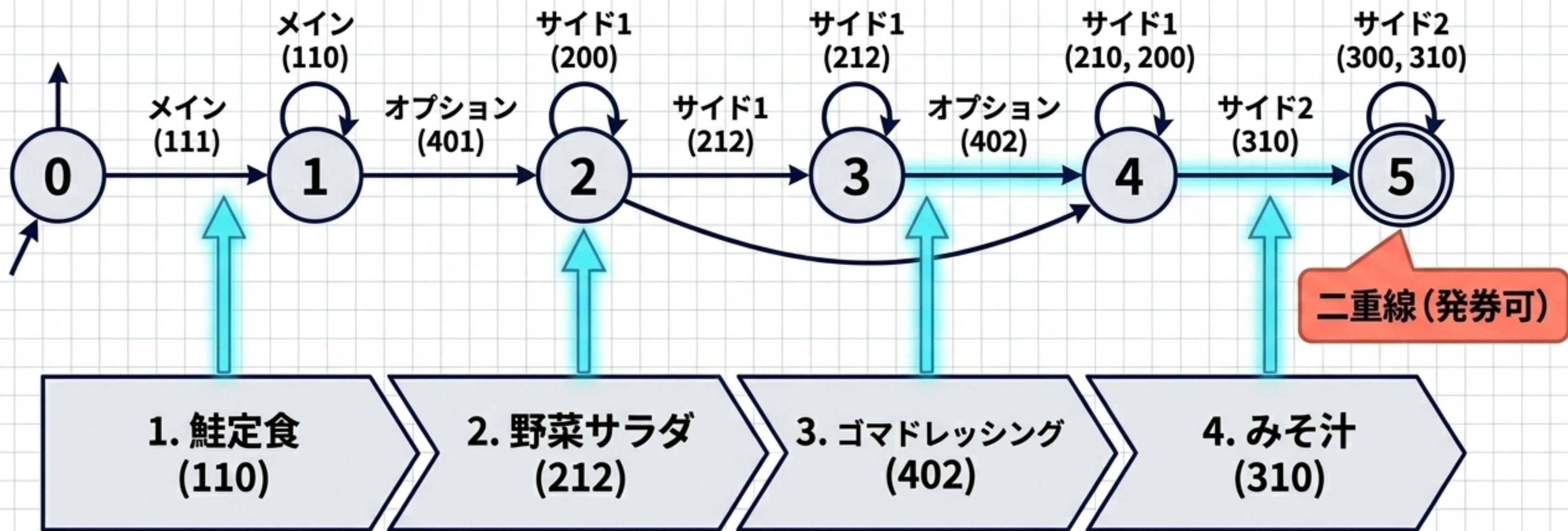
すべての注文は「3桁のイベントコード (N, S, O) 」に変換され、システムを駆動する



設問2 [エ] の解答構造

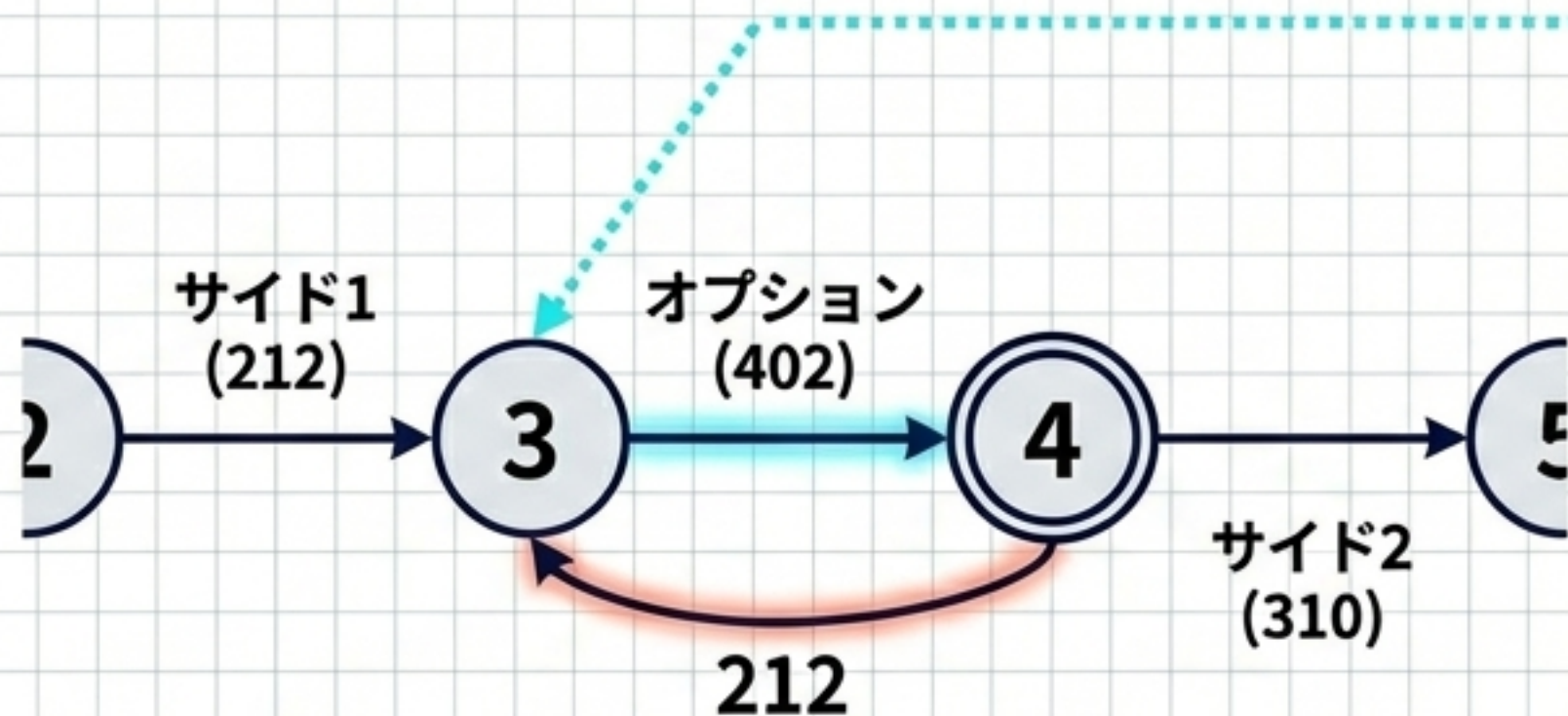
```
event_code ← order[i].N * 100 + order[i].S * 10 + order[i].O
```

注文順序に従って状態が遷移する。最終状態が「二重線」
なら発券可能（設問1-1）



解答: 0 → 2 → 3 → 4 → 5

図の論理をプログラムで処理可能な 「2次元配列（状態遷移表）」に落とし込む



状態遷移図の抜粋
(ノード3と4)

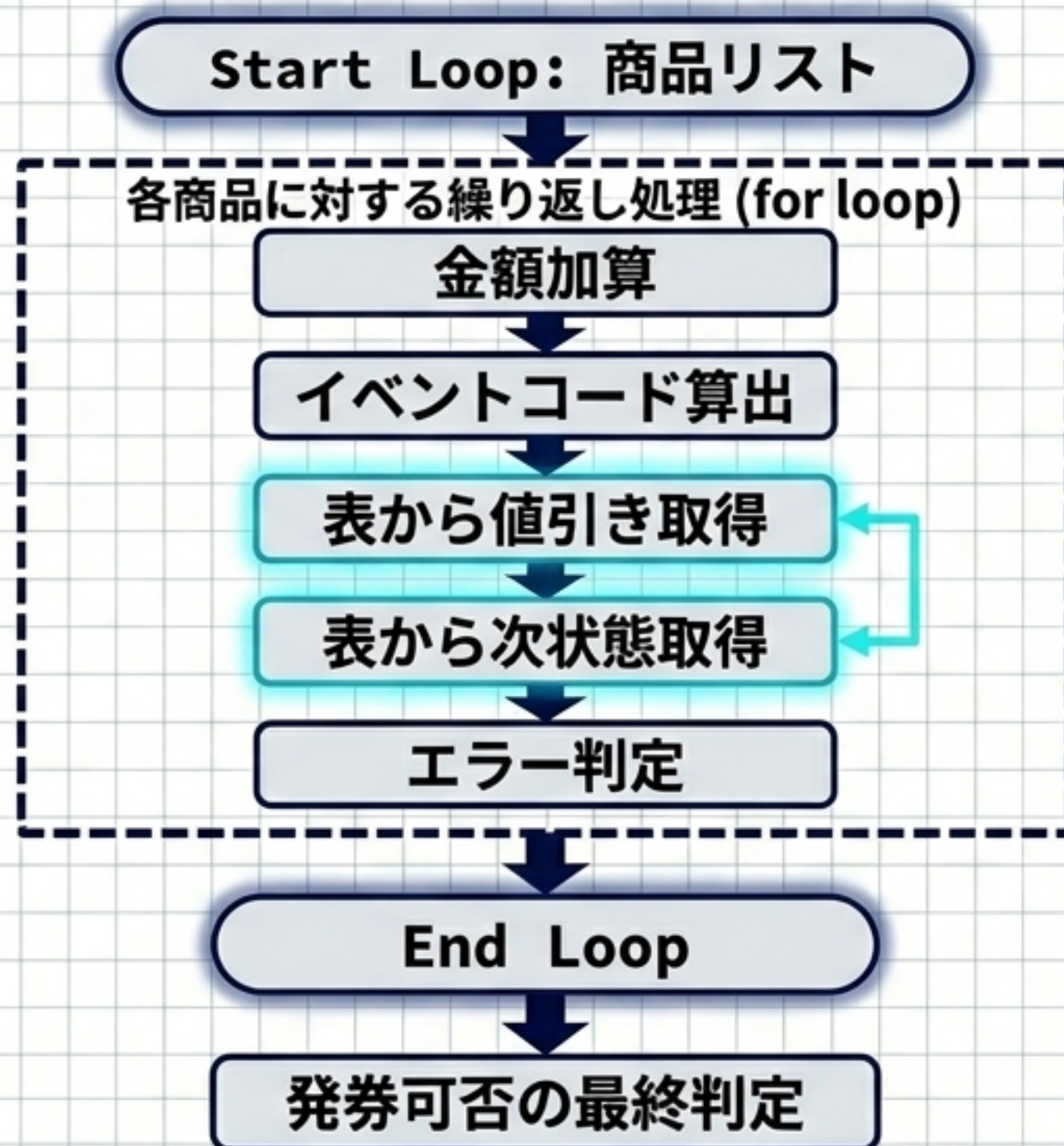
オプション未指定で注文未完了のため、発券不可。
[ア] = false

状態 番号	イベントコード									発券可
	110	111	200	210	212	300	310	401	402	
3	E:0	E:0	E:0	E:0	E:0	E:0	E:0	E:0	4:0	false
4	E:0	E:0	4:0	4:0	3:0	4:0	5:50	E:0	E:0	true

状態4から212が入力されると状態3へ戻る。
サイド2未選択のため値引きは発生しない。
[イ] = 3:0

状態遷移表の抜粋
(状態3と4の行)

判定プログラムの全体像：注文リストをループし、表を参照して状態を更新し続ける



Step 1 & 2 : 商品ごとに金額を加算し、固有のイベントコードを生成する

```
1 for( i を 1 から order_count まで繰り返す )
2
3     amount ← amount + order[i].A
4     event_code ← order[i].N * 100 + order[i].S * 10 + order[i].O
5 )
```

[ウ]の解答

商品の金額を加算。
解答は $\text{order}[i].A$

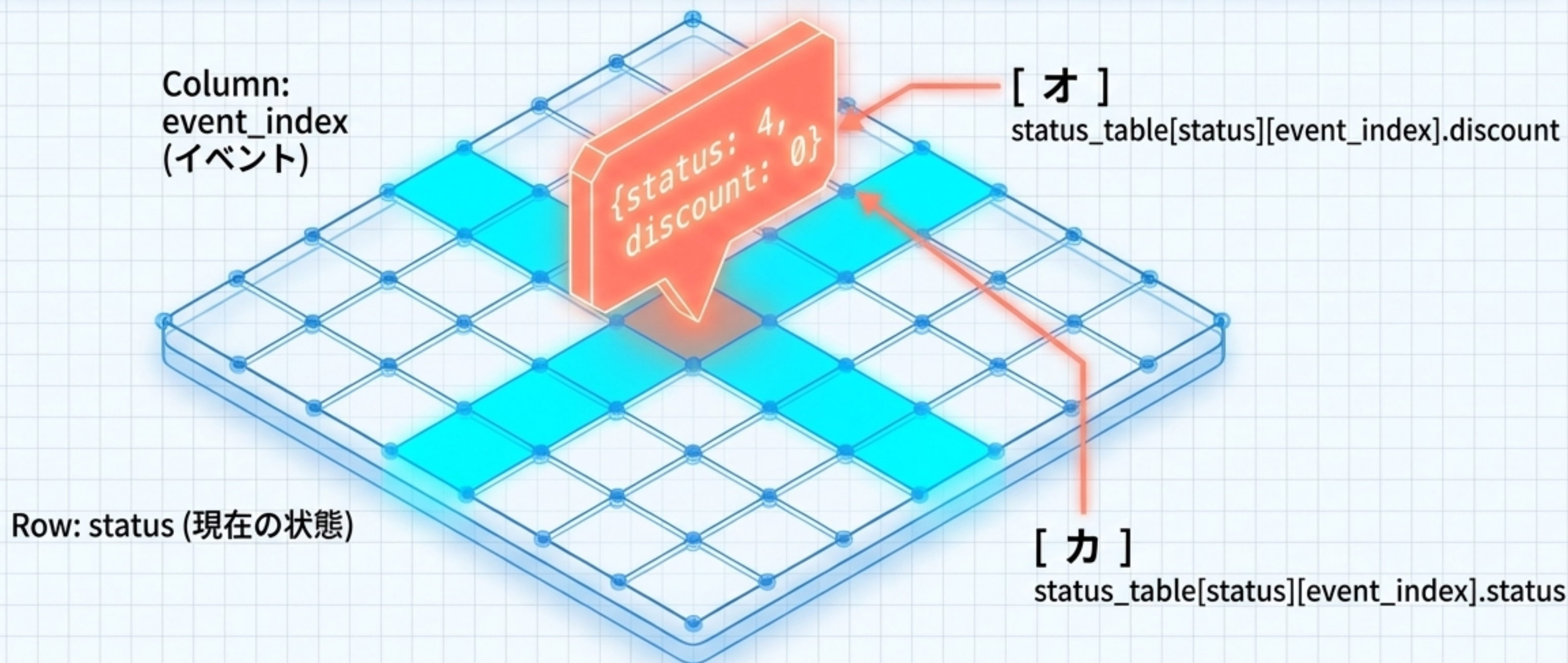
[エ]の解答

Slide 3で確認した3桁の生成式。
百の位(N)と十の位(S)を計算し、
一の位(O)を足す。

解答は
 $\text{order}[i].N * 100 + \text{order}[i].S * 10 + \text{order}[i].O$

Step 3 & 4 : 状態遷移表にアクセスし、「値引き」と「次の状態」を同時に取得する

```
1 discount ← discount + status_table[status][event_index].discount
2 status   ← status_table[status][event_index].status
```



最終評価：ループ終了後、現在の状態が「発券可」のフラグを持っているか判定する

Left Column (Code Viewer)

```
accepted ← status_table[status][ACCEPT_INDEX].accept  
if ( accepted == true )  
    発券ボタンを押せるようにする
```

Logic Visual

発券可
State 0: false
State 1: false
State 2: true
State 3: false
State 4: true
State 5: true

プログラムは『状態が2か4か5か』
をハードコードで判定しない。
単に表の論理値 (true/false) を
評価するだけ。

解答 [キ] = accepted == true

Python実装の裏側：状態遷移表は「辞書のネスト (Nested Dictionaries)」として表現される

```
1 status_table = {  
2     4: {  
4         '212': {'status': 3, 'discount': 0},  
5         '310': {'status': 5, 'discount': 50}, # 値引き発生  
6         ACCEPT_INDEX: {'accepted': True}  
7     }  
8 }
```

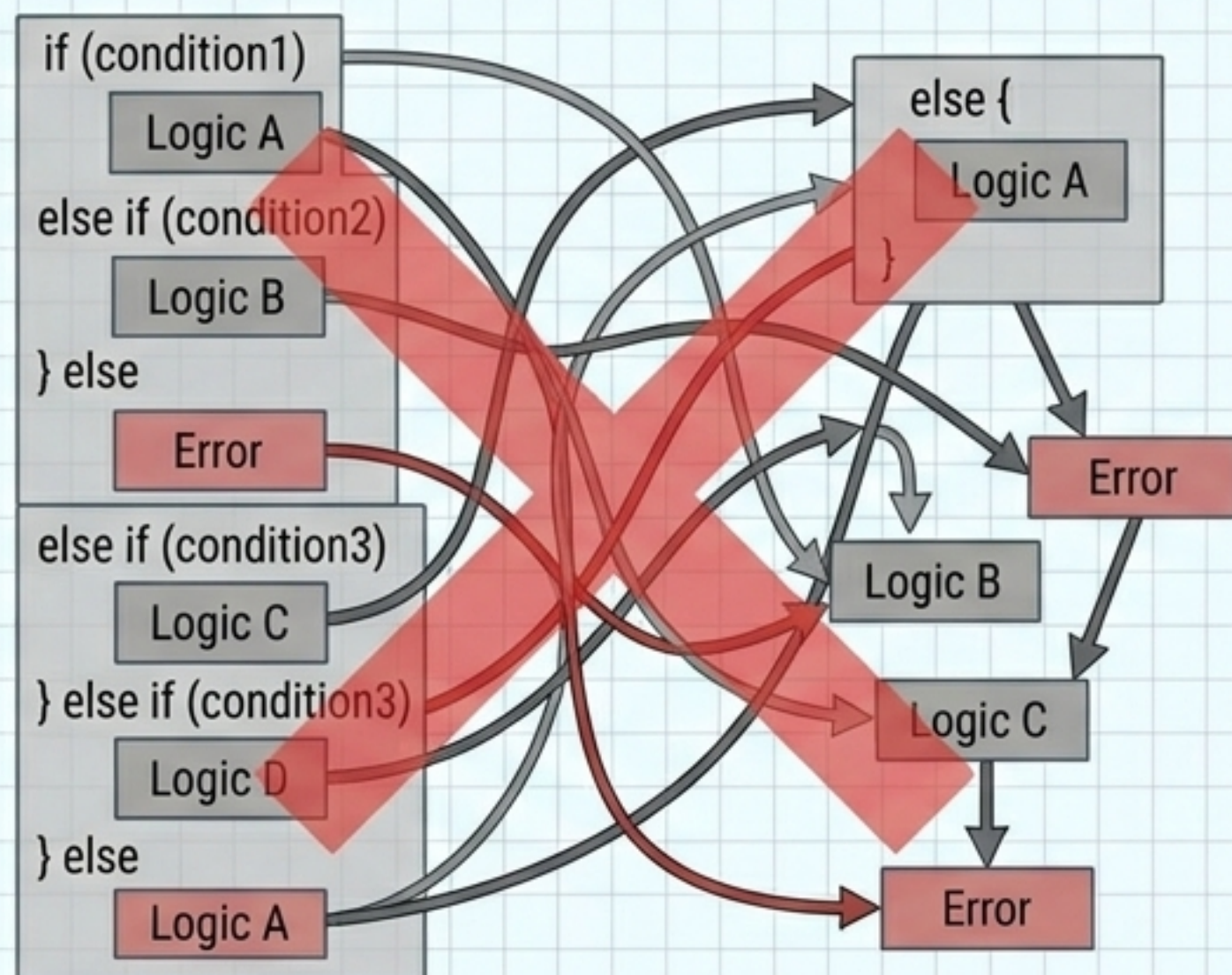
status_table[4]['310']['discount']

Pythonでは2次元配列の代わりに、
状態、番号を外側のキー、イベント
コードを内側のキーとするネストされ
た辞書で表現する。

なぜこのアーキテクチャなのか？

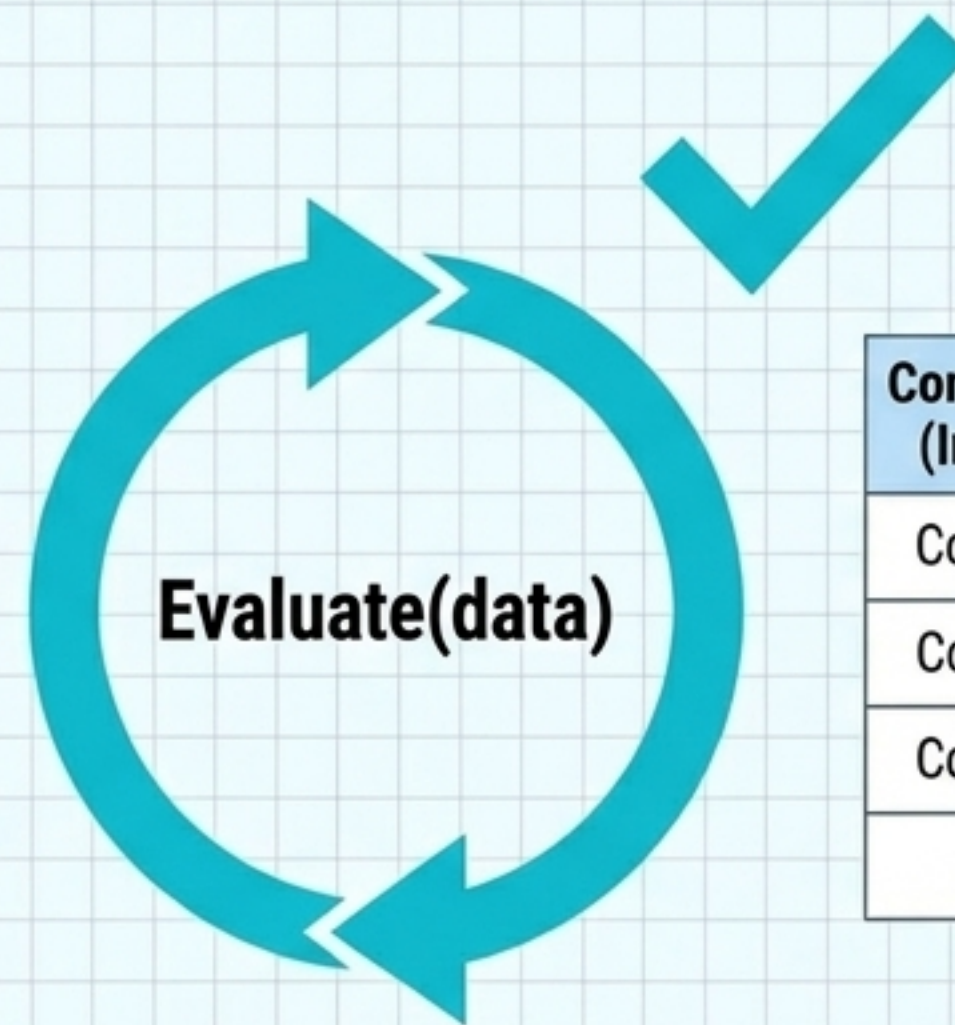
「ロジック（コード）を変えずに、データ（表）だけで拡張できる」から

従来の条件分岐（変更に弱弱）



仕様追加のたびにプログラム全体の修正と再テストが必要。

データ駆動設計（変更に強い）



ルールテーブル

Condition (Input)	State	Action (Output)
Cond A	State 1	Action X
Cond B	State 2	Action Y
Cond C	State 3	Action Z
...	...	...

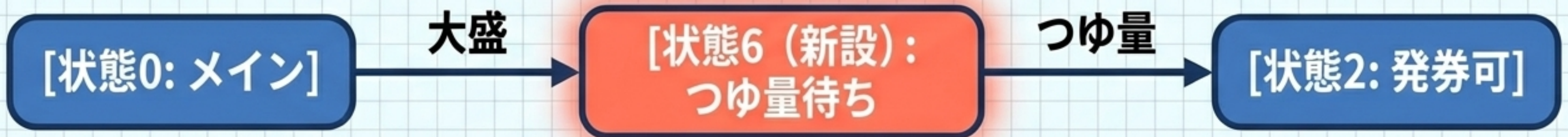
ビジネスルールが変わっても、更新するのは外部のデータ（表）のみ。コードは一切触らない。

拡張性の証明①：二段階のオプション指定を追加する（設問3イ）

現在の仕様：



つゆの量（普通・多め・少なめ）を追加した場合：



状態遷移図の経路（表のデータ）を増やすだけで対応可能。
図4のforループや判定プログラム自体は1行も書き換える必要がない。

拡張性の証明②：特定オプションを禁止した新商品を追加する（設問3 工）

		イベントコード: 111 (カレー)
大盛りが入力された場合	→	状態1へ遷移 (オプション待ち)
特盛りが入力された場合	→	E (エラー) へ遷移
...		...

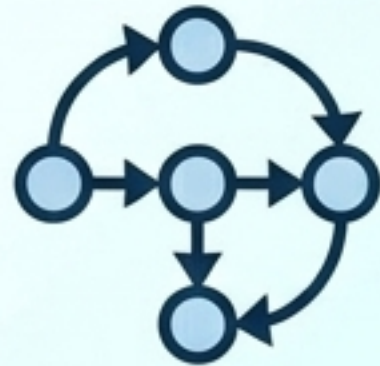
既存のシステム枠組みの中で「新しい商品データ(111)」と「エラー遷移(E)」を表に定義するだけで、特盛り拒否の仕様を実現できる。

複雑な条件分岐は、状態遷移モデルとデータ駆動設計によって美しく解決できる

01

視覚化 (Visualize)

複雑な仕様は「状態遷移図」を用いて、状態と遷移の経路としてモデリングする。



02

データ化 (Data-ify)

図の論理を「2次元配列（状態遷移表）」に変換し、状態と値引きのルールをデータとして分離する。



03

汎用化 (Generalize)

プログラムは「表を参照するだけ」のシンプルなループ構造にし、保守性と拡張性を最大化する。

