

```
def parse_syntax(tokens):
```

```
    tokens = 0
```

```
    tokens = 'token':
```

```
        token = tokens_name()
```

```
        .keyNode=()
```

```
        return [' ']
```

Pythonで読み解く 構文解析の実装

2010年代のコードブロックごとの役割と「文法理解」のメカニズム

Syntax Analysis Explainer Deck

```
class ASTNode:
```

```
    if token.type == 'KEYWORD':
```

```
        return SyntaxError('Unexpected token')
```

```
class ASTNode:
```

```
    if token.type == 'KEYWORD':
```

```
        token
```

```
    return SyntaxError('Unexpected token')
```

はじめに：本コードの位置づけ

情報処理試験の理論をコード化

本スライドで解説するPythonコードは、平成22年秋期応用情報技術者試験 午後問2に登場する「構文解析プログラム」の処理ロジックを、現代的なPythonのクラスベース実装で再現したものです。

```
<プログラム> ::= <宣言部> <実行部>
<宣言部> ::= <宣言部記述> {< ア >}
<実行部> ::= <文> {<文>}
<宣言部記述> ::= <宣言記述子> < イ > ';'
<宣言記述子> ::= 'short' | 'long'
<文> ::= <識別子> '=' <式> ';'
<式> ::= <項> {'+' <項> | '-' <項>}
<項> ::= < ウ > {'*' <因子> | '/' <因子>}
<因子> ::= <数> | <識別子>
```

図2 構文規則

理論から実装へ

解析対象の文字列

今回は以下のコードが、文法規則（BNF）に従って正しく記述されているかをプログラムがどう判定するか、その裏側をブロックごとに解剖します。

```
1
2
3
4 code = "aa = aa + 2 * b1 ;"
5
```

全体構造：2つの大きなブロック

① パーサ定義部

解析器 (Parser) の設計図。
初期化処理と、構文を解析するためのメソッド (ルール) が定義されています。

② 実行・判定部

実際に文字列を流し込み、
解析を実行し、その結果を評価して出力するブロックです。

```
class Parser:
    def _init_(self, self):
        self.paver(cot) = nolf

    def test-out(self, arng):
        self.parser = self.set<x.t, =1F "[ルール]"

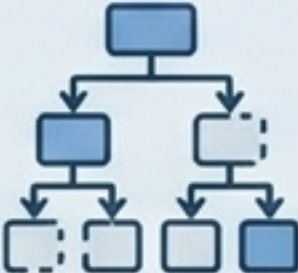
    def rutn-on(self):
        return paserout(self, forever(["/ルール"]))

code = "aa = aa + 2 * b1 ;"
parser = Parser(code)

def mer_inder(senc):
    code = Parser(code)
    parser.simulation("! " + 0marsen(auu, * b1)")
    return code
```

事前知識：コードが担う2つの解析フェーズ

コードの役割を理解するため、コンパイラ理論における「字句解析」と「構文解析」の違いを整理します。

役割の比較	字句解析 (Lexical Analysis) 	構文解析 (Syntax Analysis) 
担当メソッド	<code>_lexer()</code> / <code>gettoken()</code>	<code>bun()</code> / <code>shiki()</code> など
入力	単なる文字列 (Raw String)	トークン (字句) の配列
目的	文字を意味のある最小単位 (トークン) に分割する	トークンの並び順が文法規則 (BNF) 通りかチェックする
例え	文章を「単語」ごとに切り分ける	単語の並びが「主語・述語」等のルールに沿っているか確認する

ブロック詳解①：パーサの初期化と字句解析

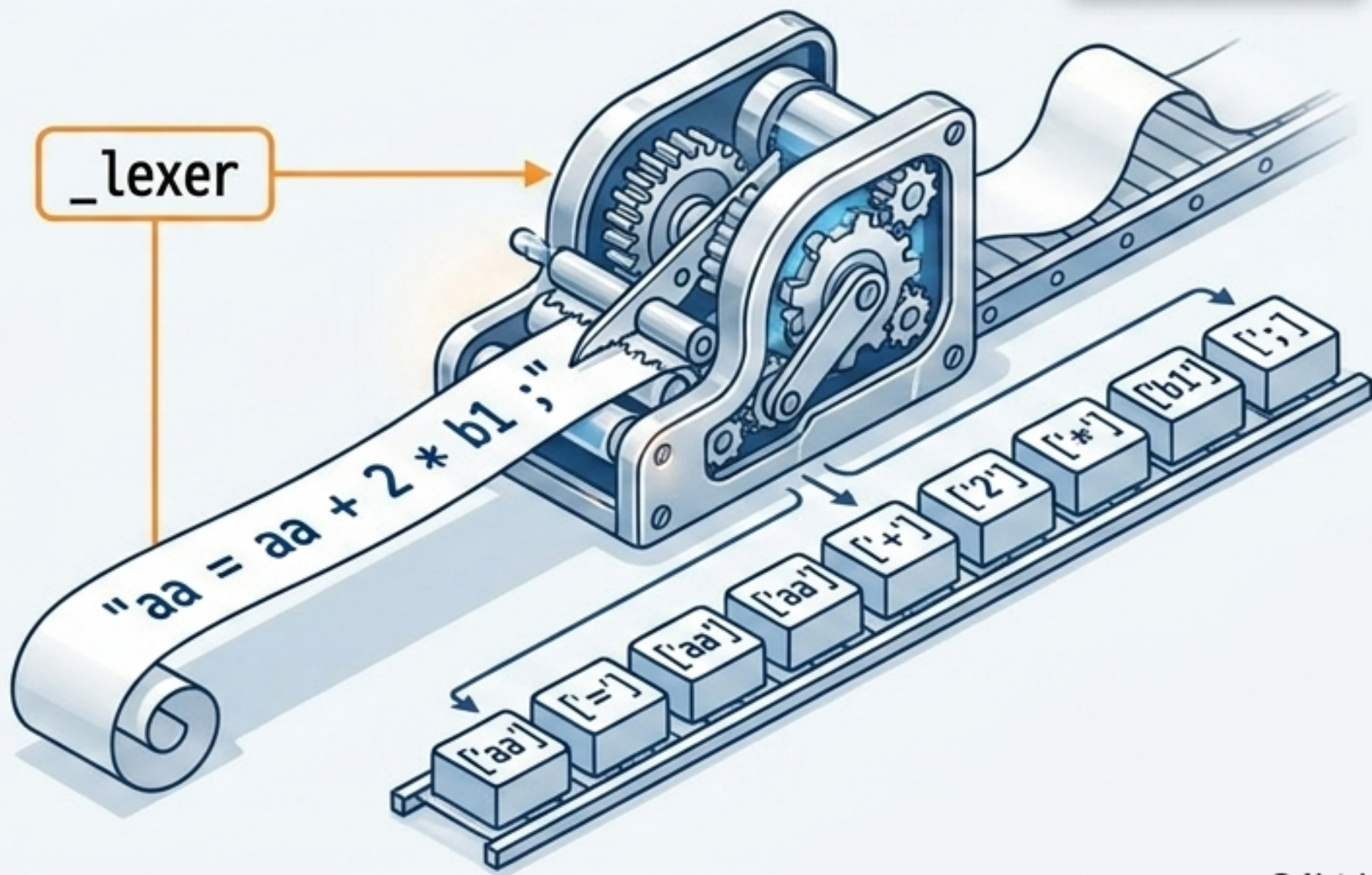
コンストラクタ `__init__`

```
class Parser:  
    def __init__(self, text):  
        self.tokens = self._lexer(text)
```

`text`（生の文字列）を受け取り、クラスの初期状態を構築します。

内部の動き：文字列から配列へ

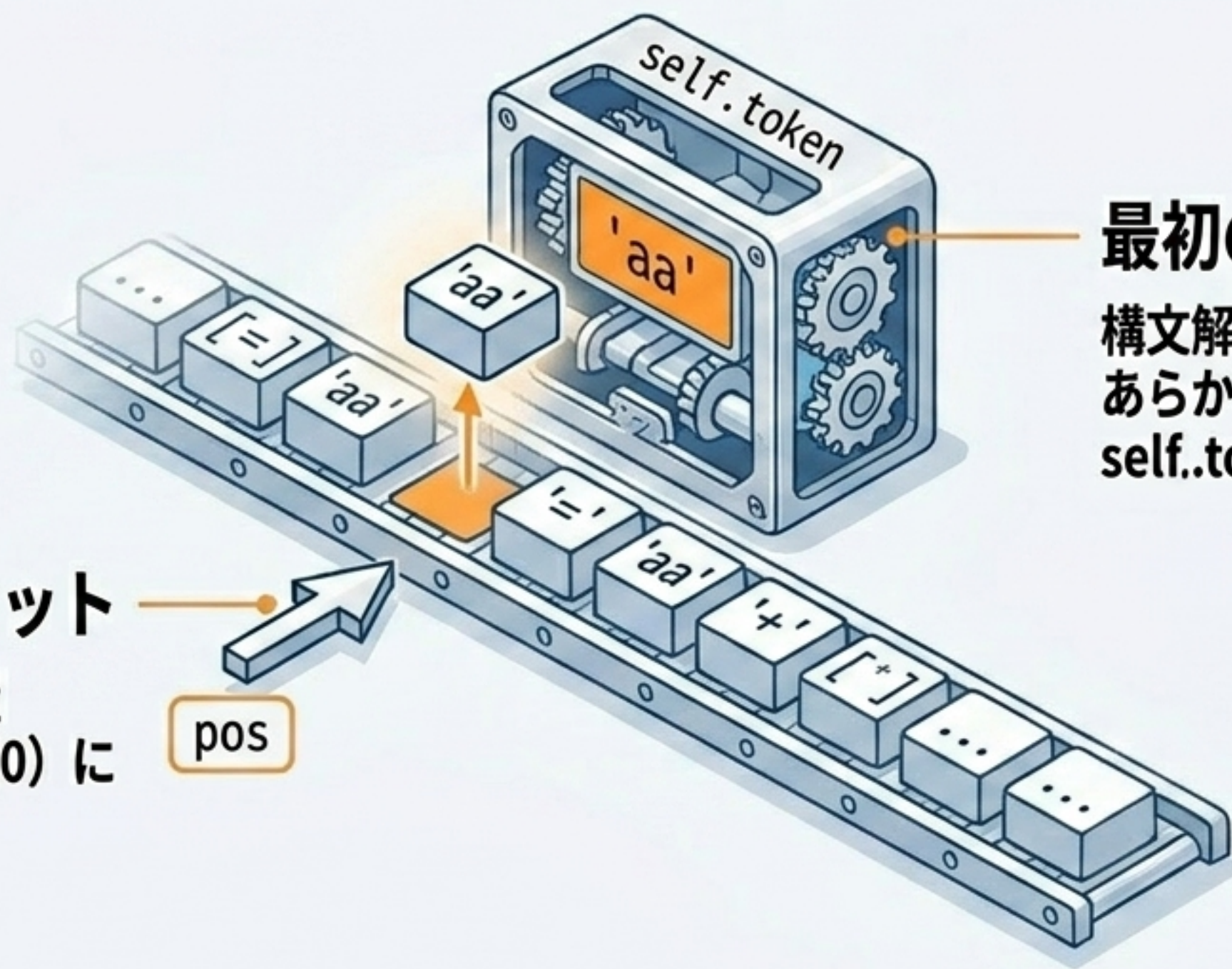
この段階で、対象文字列は `_lexer(text)` によって空白で分割され、扱いやすい「トークンのリスト」に変換されます。



ブロック詳解②：解析の「発射準備」

```
self.pos = 0  
self.token = self.gettoken()
```

Minimap



最初のトークンの装填

構文解析をスタートさせるために、あらかじめ最初のトークンを `self.token` に読み込んでおきます。

読み込み位置のリセット

トークンリストの現在位置（インデックス）を先頭（0）にセットします。

pos

ブロック詳解③：解析対象のセットアップ

実行部の開始

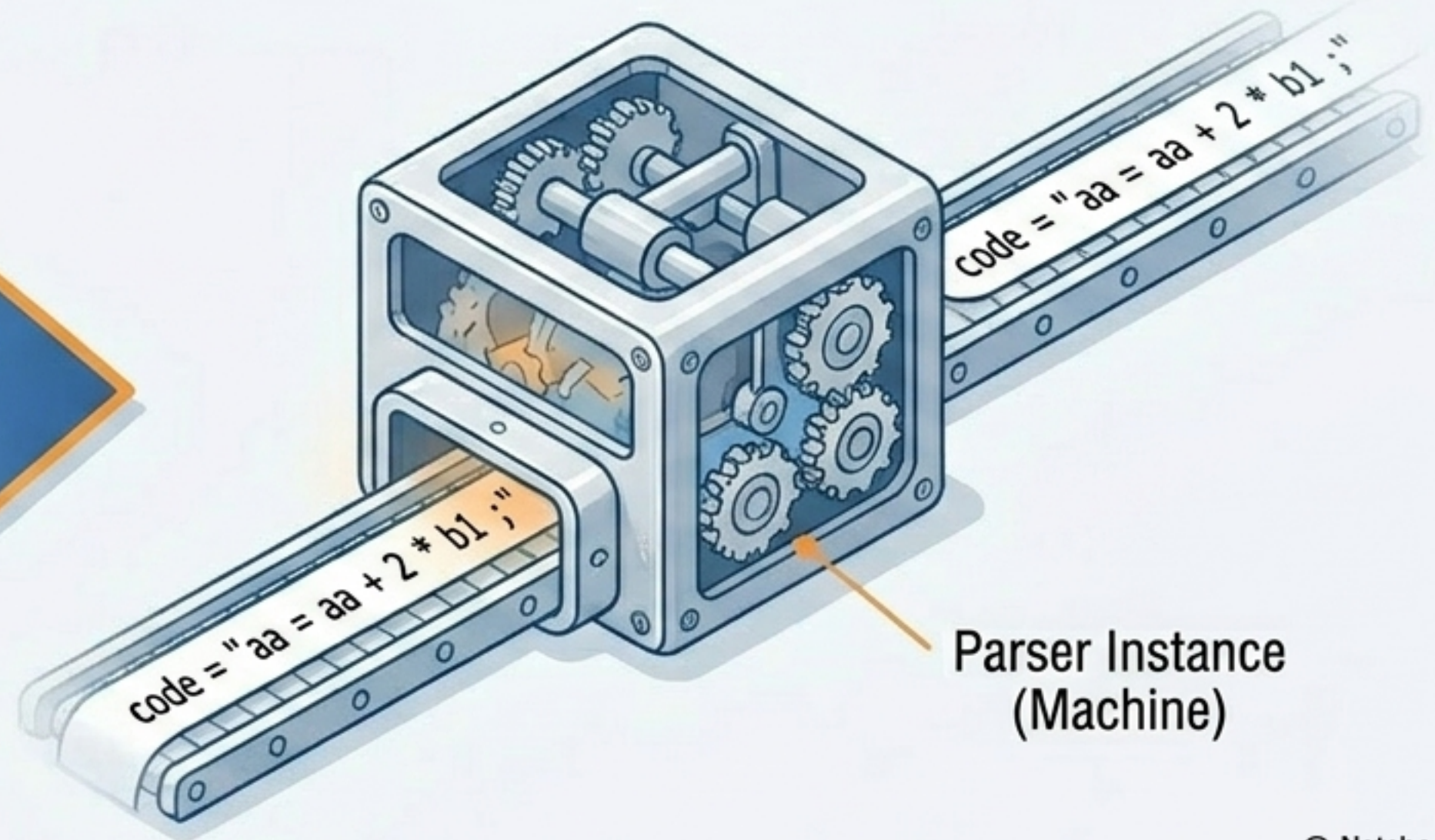
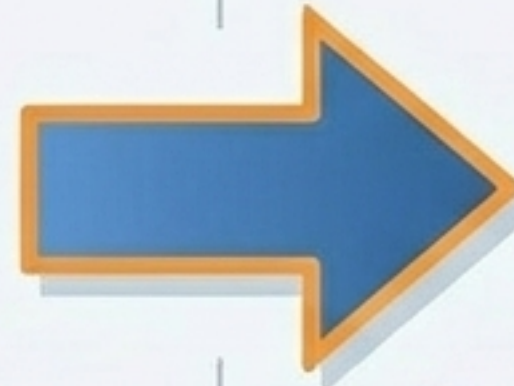
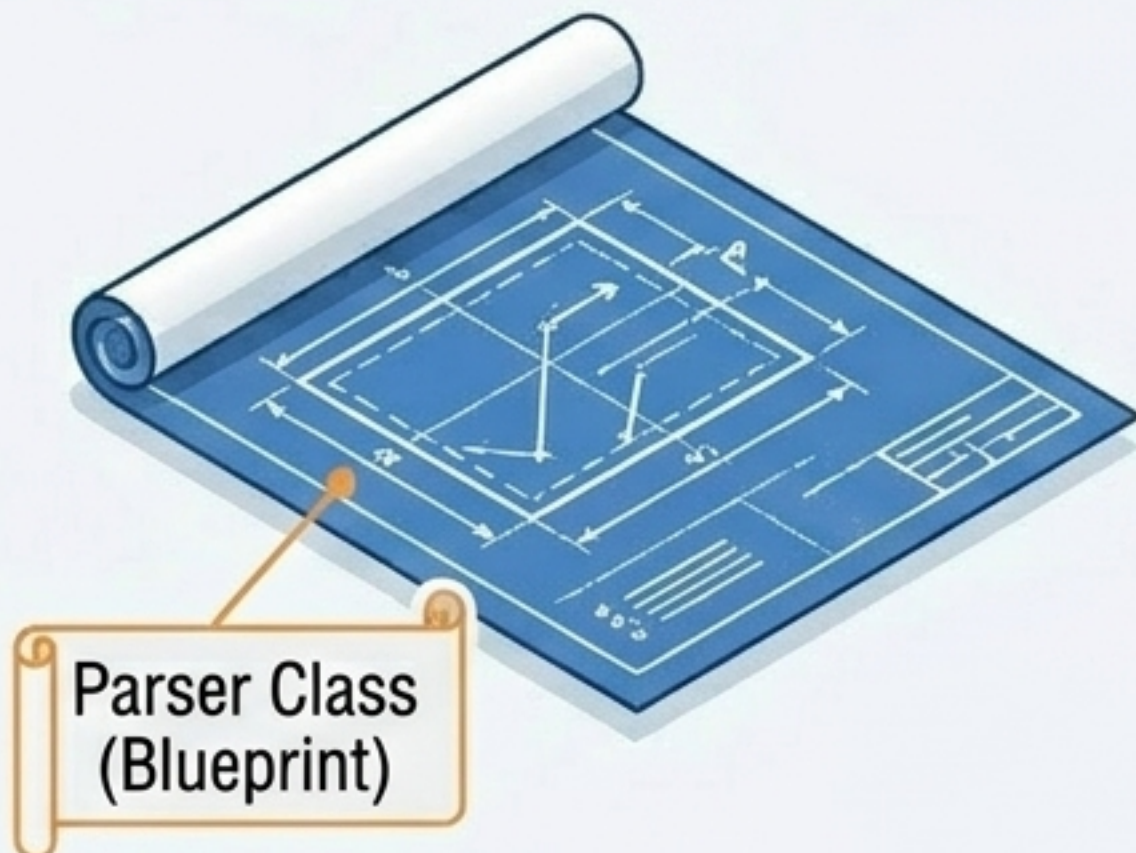
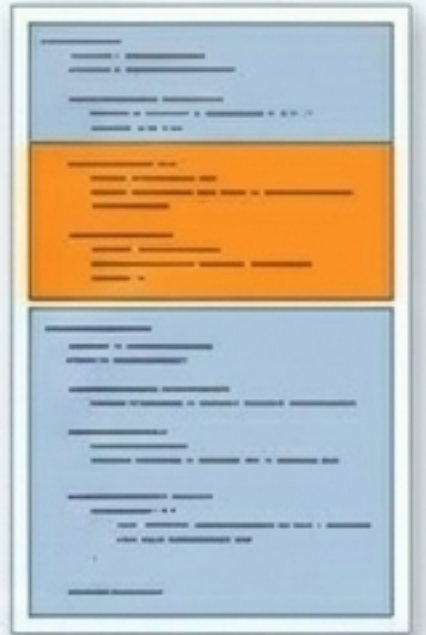
```
code = "aa = aa + 2 * b1 ;"  
parser = Parser(code)
```

パーサのクラス定義が終わり、実際にプログラムを動かすブロックに入ります。

インスタンスの生成

変数codeに検証したいプログラム文を格納し、Parserクラスに渡してインスタンス化します。この1行が実行される裏で、字句解析と初期トークンの装填が完了しています。

Minimap



ブロック詳解④：構文解析のキックオフ

Minimap

トップダウン解析の開始

```
result = parser.bun()
```

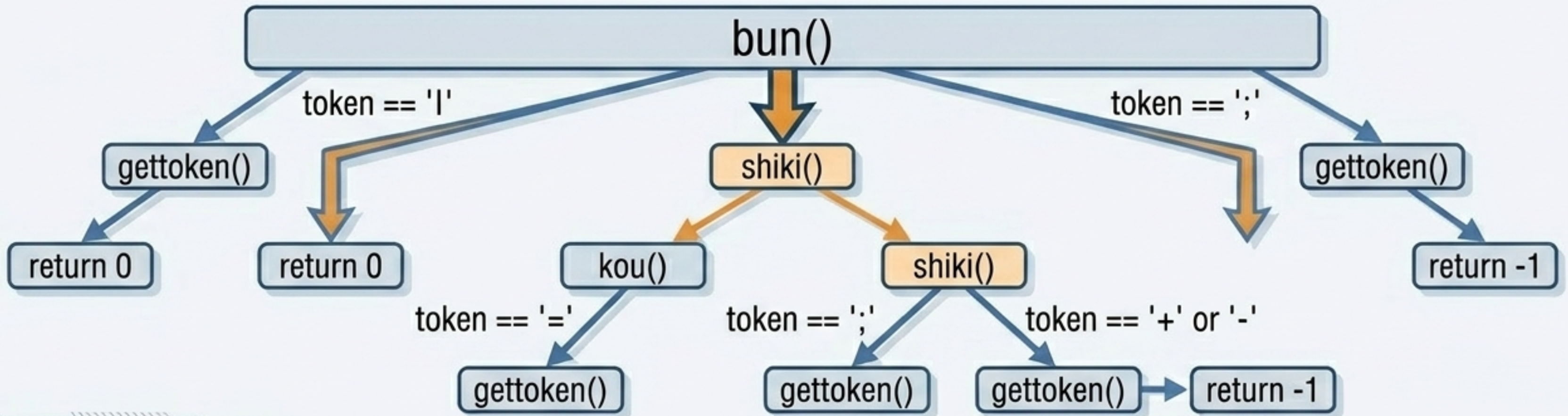
この1行が、構文解析の心臓部です。
bun() (文) メソッドを呼び出すことで、
ドミノ倒しのように解析が始まります。

再帰的下降構文解析の連鎖

試験理論に基づくBNF規則：

<文> ::= <識別子> '=' <式> ';' ;

bun() メソッドは内部で、自分が処理すべき
トークンを確認しながら、さらに深いルールである
shiki() (式) などを連鎖的に呼び出していきます。



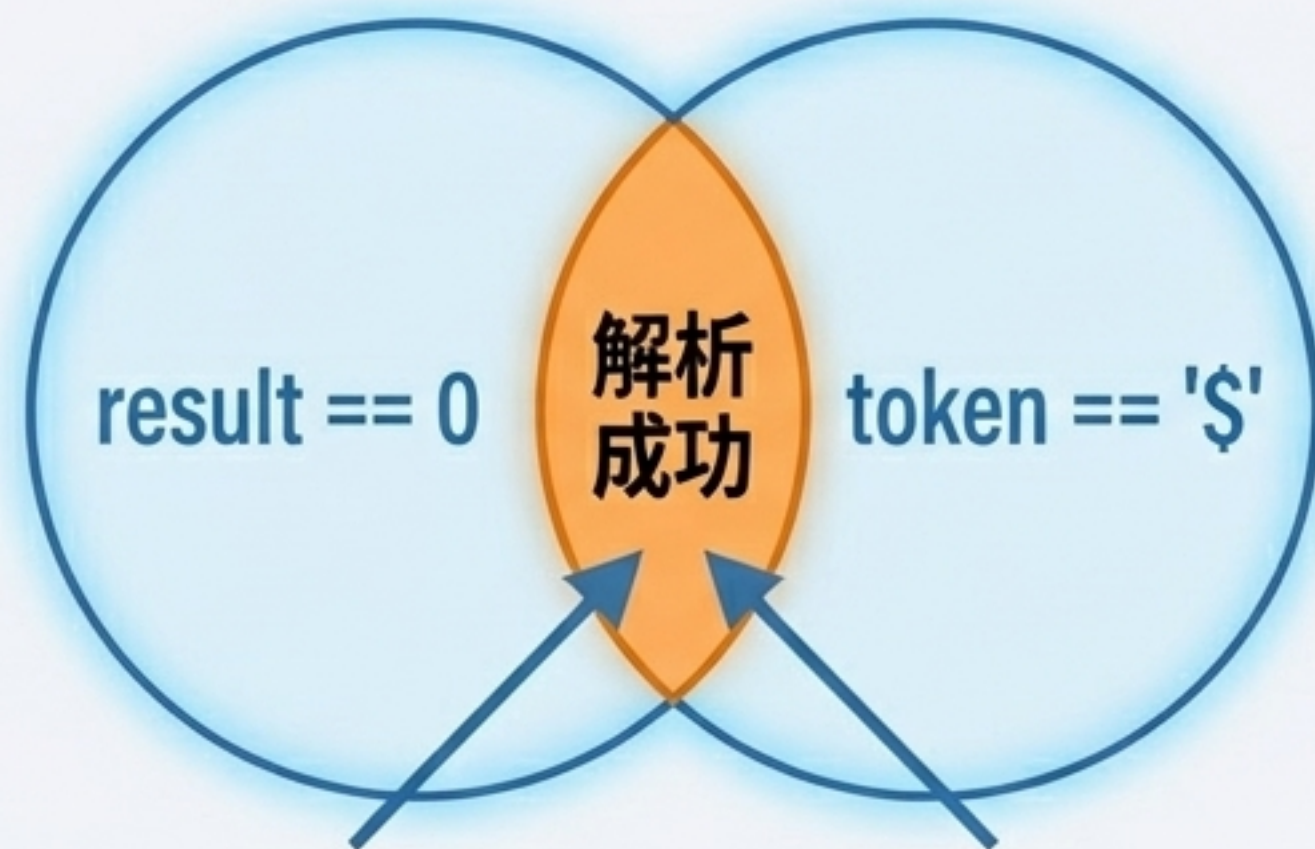
ブロック詳解⑤：解析結果の判定条件

Minimap

厳密な成功条件

```
if result == 0 and parser.token == '$':  
    print("解析成功...")
```

解析が正常終了したかを確認するブロックです。ここでは2つの条件がANDで結ばれていることが極めて重要です。



2つの条件の役割



result == 0

構文解析メソッド（bunなど）が文法エラーを検知せず、正常に処理を終えたことを示します。



parser.token == '\$'

\$は「プログラム記述の終わり（EOF）」を示す終端記号です。未処理の文字列が残っていないことを保証します。

診断：なぜ終了記号 ('\$') の確認が必要なのか？

Minimap

もし `result == 0` だけで判定してしまうと、どうなるでしょうか？
以下のテストケースで検証します。

入力コード	構文解析 (result)	最終トークン	判定結果
<code>aa = 3 ; \$</code>	0 (文法通りで成功)	'\$' (最後まで到達)	解析成功 
<code>3 = aa ; \$</code>	-1 (代入先が識別子ではない)	'=' (途中でエラー終了)	解析失敗 (文法エラー) 
<code>aa = 3 ; extra \$</code>	0 (前半部分は文法通りで成功!)	'extra' (ゴミが残っている)	解析失敗 (未到達エラー) 

ブロック詳解⑥：結果の出力

```
else:
    print(f"解析失敗...")
```

エラーハンドリング

条件のどちらか一方でも満たさなかった場合、構文規則に反している、あるいは余計な文字列が混入しているとみなし、エラーを出力してプログラムを終了させます。

Minimap



```
> python parser.py
```

解析成功: 'aa = aa + 2 * b1 ;' は文法的に正しいです。

まとめ：構文解析のパイプライン

今回読み解いたPythonコードの各ブロックが連動し、文字列が「意味」として評価されるまでの全工程です。

