## O que é um analisador léxico?

Um analisador léxico (ou *lexer*) é como um **quebra-texto** inteligente: ele pega uma sequência de caracteres (por exemplo, `x = 3 + 42 * (s - 5)`) e separa em **tokens**.

**Token = pedacinho de texto com significado** → exemplo:

- `x` → identificador (`ID`)
- `=` → operador de atribuição (podemos chamar de `ASSIGN`)
- `3` → número (`NUMBER`)
- `+` → operador soma (`PLUS`)
- etc.

O lexer **não entende a lógica ainda** (isso é papel do *parser*), mas ele organiza o texto em unidades básicas.

---

### 1️⃣ Especificação dos tokens

No código:

```
TOKEN_SPEC = [
    ('NUMBER',    r'\d+(\.\d*)?'),    # Números inteiros ou decimais
    ('ID',        r'[A-Za-z_]\w*'),   # Identificadores (variáveis,
nomes)
    ('PLUS',      r'\+'),             # +
    ('MINUS',     r'-'),              # -
    ('TIMES',     r'\*'),             # *
    ('DIVIDE',    r'/'),              # /
    ('LPAREN',    r'\('),             # (
    ('RPAREN',    r'\)'),             # )
    ('WHITESPACE', r'\s+'),           # Espaços
    ('MISMATCH',  r'.'),              # Qualquer outro caractere
(erro)
]
```

Aqui estamos dizendo **quais padrões queremos reconhecer**.

Cada item tem:

- Um **nome do token** → ex: `NUMBER`, `PLUS`, `ID`
- Uma **expressão regular** → um padrão para encontrar isso no texto.

Exemplo:

- `\d+(\.\d*)?` → reconhece números como `42` ou `3.14`.
- `[A-Za-z_]\w*` → reconhece nomes como `x`, `var1`, `soma_total`.

---

### 2️⃣ Combinar todos os padrões em uma grande expressão

```
tok_regex = '|'.join(f'(?P<{name}>{pattern})' for name, pattern in
TOKEN_SPEC)
get_token = re.compile(tok_regex).match
```

Aqui usamos a biblioteca **`re` (regex)** para construir um único padrão que tenta todos os tokens ao mesmo tempo.
O `(?P<name>pattern)` nomeia cada grupo no regex, para sabermos **qual tipo de token foi encontrado**.

---

### 3️⃣ Ler o texto e identificar tokens

```
def lex(code):
    pos = 0
    tokens = []
    mo = get_token(code, pos)
    while mo:
        kind = mo.lastgroup
        value = mo.group()
        if kind == 'WHITESPACE':
            pass  # ignorar espaços
        elif kind == 'MISMATCH':
            raise RuntimeError(f'Caractere inesperado: {value!r} na
posição {pos}')
        else:
```

```
            tokens.append((kind, value))
        pos = mo.end()
        mo = get_token(code, pos)
    return tokens
```

👉 O que acontece aqui:

- Começamos no início (`pos = 0`).
- Usamos `get_token(code, pos)` → tenta achar um token a partir daquela posição.
- Se for espaço (`WHITESPACE`), ignoramos.
- Se for algo estranho (`MISMATCH`), lançamos erro.
- Se for um token válido, adicionamos na lista.

Esse processo continua até o final do texto.

---

### 4️⃣ Exemplo funcionando

Se rodarmos:

```
code = 'x = 3 + 42 * (s - 5)'
token_list = lex(code)
for token in token_list:
    print(token)
```

A saída seria algo como:

```
('ID', 'x')
('ID', '=')
('NUMBER', '3')
('PLUS', '+')
('NUMBER', '42')
('TIMES', '*')
('LPAREN', '(')
('ID', 's')
('MINUS', '-')
('NUMBER', '5')
('RPAREN', ')')
```