

这看起来像注释



这些抽象的结构在我大脑中理解的很费力，我理解的是:我们有一个输入的token stream，这个流是由next指针指向的，next指针指向的是待处理的token，而我们的match () 中expected的字符是根据产生式的右部规则所产生的，一旦两者相符合，那就match()解析成功

已推理，持续 10 秒

你的理解大方向是对的！你可以把它想象成：

1. 有一串按顺序排列好的 token（就像排队等候的客人），这就是 **token stream**。
2. 一个指针（`next`）指着队列里“最前面”的那个人，表示“下一个要处理的 token”。
3. 当解析器根据产生式规则需要某个特定 token（例如 `INT`），就调用 `match(INT)`。
4. `match(INT)` 会检查 `next` 指向的那个 token 是不是 `INT`：
 - 如果是，就表示**匹配成功**，并让 `next` 往后移一步（就好像让那位客人离开队伍，下一个人上来）。
 - 如果不是，就表示**匹配失败**，抛出语法错误（或者在回溯解析里，会回溯到上一步继续尝试别的规则）。

下面给你一个简单的**具体例子**，帮助你把这些概念放到实际场景里。

示例：解析简单表达式

1. 语法 (Grammar)

假设我们有一个极简的文法 (LL(1))：

```
r
E → T + E | T
T → INT
```

意思是：

- 一个表达式 `E` 可以是 `T + E` 或者 `T`。
- `T` 就是一个 `INT`。

2. Token Stream (输入标记流)

假设我们输入的代码片段是：

```
cpp
int + int
```

词法分析后，得到的 token 流是：

```
sql

[INT, PLUS, INT, EOF]
```

其中：

- `INT` 表示数字或整数标记
- `PLUS` 表示 `+`
- `EOF` 表示输入结束

3. `next` 指针

一开始：

```
lua

next -> 第一个 token (INT)
```

也就是指向 `[INT, PLUS, INT, EOF]` 里的第一个 `INT`。

4. 解析过程

我们要解析一个表达式 `E`，因此从 `E` 开始。

步骤 A：解析 `E`

根据文法， $E \rightarrow T + E \mid T$ 。

- 在没有回溯的 LL(1) 解析器中，我们会先看当前 token (`INT`)，然后决定用哪个产生式。
- 因为看到的是 `INT`，它匹配 `T`（不是 `+`），所以我们先去解析 `T`。

步骤 B：解析 `T`

文法： $T \rightarrow INT$ 。

- 调用 `match(INT)` 来匹配当前的 token。
- `match(INT)` 检查 `next` 是否是 `INT`。
 - 如果是，匹配成功，`next` 往后移，指向 `PLUS`。
 - 解析器返回“匹配成功”给 `T`。

现在 token 流状态：

```
sql

[~~INT~~, PLUS, INT, EOF]
      ^
      next
```

（我用 `~~INT~~` 表示已经被匹配并消费掉的 token。）

步骤 C：回到 `E` 的规则

我们刚刚匹配了 `T`，但 `E` 的规则是 $T + E \mid T$ 。

- 此时，我们看下一个 token（`PLUS`）。
- 发现是 `PLUS`，说明应该走 $T + E$ 这个分支，而不是只匹配 `T` 就结束。
- 所以接下来要匹配 `+`，再解析一个新的 `E`。

步骤 D：匹配 `+`

- 调用 `match(PLUS)`。
- `next` 是 `PLUS`，符合预期，匹配成功，`next` 往后移，指向第二个 `INT`。

现在 token 流状态：

```
sql
[~~INT~~, ~~PLUS~~, INT, EOF]
                ^
                next
```

步骤 E：解析新的 `E`

在 $T + E$ 里，我们还要解析后面的那个 `E`。

- 再次进入 `E` 的规则： $E \rightarrow T + E \mid T$ 。
- 看当前 token 是 `INT`，匹配 `T`。
- 于是去解析 `T`。

步骤 F：解析 `T`

- $T \rightarrow INT$ 。
- 调用 `match(INT)`，`next` 是 `INT`，匹配成功，指针移动到 `EOF`。

现在 token 流状态：

```
sql
[~~INT~~, ~~PLUS~~, ~~INT~~, EOF]
                ^
                next
```

步骤 G：看是否还有 `+ E`

- 现在 `next` 是 `EOF`，不是 `+`，所以 $T + E$ 这个分支走不下去了，说明表达式结束了。
- 解析成功，整条表达式 `int + int` 被正确识别。

为什么说“一旦两者相符合，那就 `match()` 解析成功”？

正如你理解的，`match(expected)` 做的事情就是：

1. 看看 `next` 指向的 token 是否是 `expected`。
2. 如果匹配，就表示当前产生式需要的 token 确实在输入里找到了，解析器可以前进到下一个 token。

3. 如果**不匹配**，就说明输入跟语法要求的 token 不一致，发生了**语法错误**。

在一些更复杂的（回溯）解析器里，若不匹配，会退回到之前保存的 ``next`` 位置，去尝试别的产生式分支；在纯粹的 LL(1) 解析器里，通常会直接报错。

总结

- 你可以把 ``next`` 想成**“指向队列前排”**的指针，每次匹配成功，就让这个指针往后挪一格。
- ``match(expected)`` 就是**检查当前排在前面那个人是不是你要找的那个**。
- 如果是，就**让他出列**（``next`` 往后移），如果不是，就报错或者回溯。

这样把“token 队列”+“指针”+“期望的 token”结合起来，你就能更好地在脑海中想象出这个流程了。希望这样解释能帮助你把抽象的结构在脑海里具象化。