LL剖析器

维基百科，自由的百科全书

LL分析器是一种处理某些上下文无关文法的自顶向下分析器。因为它从**左**（**Left**）到右处理输入，再对句型执行**最左推导**出语法树（**Left** derivation，相对于LR分析器）。能以此方法分析的文法称为**LL 文法**。

本文中将讨论表格驱动的分析器，而非通常由手工打造（非绝对，参看如ANTLR等的 LL(*) 递归下降分析器生成器）的递归下降分析器。

一个 LL 分析器若被称为 LL(k) 分析器，表示它使用 k 个词法单元作向前探查。对于某个文法，若存在一个分析器可以在不用回溯法进行回溯的情况下处理该文法，则称该文法为 **LL(k) 文法**。这些文法中，较严格的 **LL(1) 文法**相当受欢迎，因为它的分析器只需多看一个词法单元就可以产生分析结果。那些需要很大的 k 才能产生分析结果的编程语言，在分析时的要求也比较高。

上下文无关文法语法分析器

- LL剖析器
- 算符优先分析器
- LR剖析器
- SLR剖析器
- LALR剖析器

目录

概览

对于给定的上下文无关文法，分析器尝试寻找该文法的最左推导。例如，给定一个文法G：

1. $S \to E$
2. $E \to (E + E)$
3. $E \to i$

对$w = ((i + i) + i)$的最左推导如下：

$$S \overset{(1)}{\Rightarrow} E \overset{(2)}{\Rightarrow} (E + E) \overset{(2)}{\Rightarrow} ((E + E) + E) \overset{(3)}{\Rightarrow} ((i + E) + E) \overset{(3)}{\Rightarrow} ((i + i) + E) \overset{(3)}{\Rightarrow} ((i + i) + i)$$

通常, 选择一条规则来展开给定的（最左的）非终结符时，有多个选择的可能。前一个关于最左推导的例子中， 在第2步：

$$S \overset{(1)}{\Rightarrow} E \overset{(?)}{\Rightarrow} ?$$

我们有两条规则可以选择：

2. $E \to (E + E)$
3. $E \to i$

为了提高分析的效率，分析器必须能够尽可能确切地、无回溯地进行规则的选择。对于一些文法，它可以透過偷看不回推（即读取之后不将它退回输入流）的输入符号来做到这点。在我们的例子中，如果分析器知道下一个无回推符号是 (，那么唯一正确可用的就是规则 2。

通常，$LL(k)$ 分析器可以向前探查 k 个符号。然而，给定一个文法，若存在一个能识别该文法 $LL(k)$ 分析器，则其 k 值的确定问题是不可判定的。也就是说，无法判定需要向前探查多少个符号才能识别它。对于每一个 k 的取值，总存在无法被 $LL(k)$ 分析器识别的语言，而 $LL(k+1)$ 分析器却可以识别它。

通过上述梗概，下面我们给出 $LL(k+1)$ 的形式化定义：

设 G 是一个上下文无关文法，且 $k \geq 1$。对于任意两个最左推导，当且仅当满足下述条件时，我们称 G 是 $LL(k)$ 文法：

1. $S \Rightarrow \ \ldots \ \Rightarrow wA\alpha \Rightarrow \ \ldots \ \Rightarrow w\beta\alpha \Rightarrow \ \ldots \ \Rightarrow wx$
2. $S \Rightarrow \ \ldots \ \Rightarrow wA\alpha \Rightarrow \ \ldots \ \Rightarrow w\gamma\alpha \Rightarrow \ \ldots \ \Rightarrow wy$

以下条件成立：串 x 中长度为 k 的前缀等价于串 y 中长度为 k 的前缀，表明 $\beta = \gamma$.

在该定义中， S 文法的开始符号， A 是任意非终结符。之前取得的输入 w，以及还没回推的 x 和 y 均为终结符串。希腊字母 α, β 和 γ 代表任意终结符和非终结符组成的串（也可能是空串）。前缀长度与用于保存向前探查结果的缓冲区尺寸一致，并且该定义表明了，缓冲区足以区分任意两个不同单词的推导。

本分析器可以处理特定形式文法的符号串。

本分析器由以下部件組成：

- 一个*输入缓冲区*，存放输入符号串（由语法建立的）。
- 一个*分析栈*，用于储存等待处理的终结符与非终结符的。
- 一张*分析表*，标记了是否存在可用于目前分析栈与下一个输入符号的语法规则。

分析器根据分析栈的栈顶符号（行）以及当前输入流中的符号（列）来决定使用哪一条规则。

当分析器一开始执行时，分析栈中已经有两个符号：

```
[ S, $ ]
```

'$'时一个特殊的终结符，用于表示分析栈的栈底或者输入的结束；而'S'则时文法的开始符号。分析器会尝试根据它在输入流中看到的符号来改写分析栈中的数据，但只会将仍需修改的数据存回分析栈中。

實際的例子

設定

為解釋LL剖析器的工作方式，我們創造了以下這個小語法：

1. S → F
2. S → (S + F)
3. F → 1

並處理以下輸入：

(1 + 1)

這個語法的剖析表如下：

	(	)	1	+	$
S	**2**	**-**	**1**	**-**	-
F	**-**	**-**	**3**	**-**	-

（注意到有一列特殊終端符號，在這裡表示為$，是用來標示輸入結束的。）

剖析流程

剖析器先從輸入資料流中讀到第一個 '('，以及堆疊中的'S'。從表格中他發現必須套用規則 (2)；它必須將堆疊中的'S'重寫為 '(S + F)'，並將規則的號碼輸出。最後堆疊變成：

```
[ (, S, +, F, ), $ ]
```

再來它移除輸入及堆疊中的 '(' ：

```
[ S, +, F, ), $ ]
```

現在剖析器從輸入資料流中抓到一個'1'，所以他知道必須套用規則 (1)與規則 (3)，並將結果輸出。則堆疊變成：

```
[ F, +, F, ), $ ]
[ 1, +, F, ), $ ]
```

接下來的兩個步驟中，剖析器讀到'1'及 '+'，因為他們跟堆疊中的資料一樣，所以從堆疊中移除。最後堆疊剩下：

```
[ F, ), $ ]
```

再接著的三個步驟中，堆疊中的'F'會'1'被取代，而規則 (3)會被輸出。再來堆疊與輸入資料流中的'1'與')'都會被移除。而剖析器看到堆疊與輸入資料流都只剩下'$'的時候，就知道自己的事情做完了。

在這個例子中，剖析器接受了輸入資料，並產生以下輸出（規則的代號）：

[2, 1, 3, 3]

這的確是從輸入的左邊優先推導。我們可以看出由左至右的輸入順序為：

S → (S + F) → (F + F) → (1 + F) → (1 + 1)

备注

由以上範例可以看出剖析器根據堆疊最上層為非終端符號、終端符號、還是特殊符號$來決定採取三種不同的步驟：

- 若堆疊最上層為非終端符號，則根據輸入資料流中的符號對照剖析表，決定要用語法中的哪條規則來取代堆疊中的資料，順帶輸出規則的號碼。若表格中並沒有這麼個規則，則回報錯誤並終止執行。
- 若堆疊最上層為終端符號，則與輸入資料流中的符號比較。若相同則移除，若不同則回報錯誤並終止執行。
- 若堆疊最上層為'$'，並且輸入資料流中也是'$'，則表示剖析器成功的處理了輸入，否則將回報錯誤。不管怎樣，最後剖析器都將終止執行。

這些步驟會持續到輸入結束，然後剖析器成功處理了一則左邊優先推導，或者會回報錯誤。

建構LL(1)剖析表格

為了要填滿剖析表格，我們必須決定剖析器在堆疊看到非終端(nonterminal)符號A又在輸入資料流看到a的時候應該選用哪一條文法規則。我們可以輕鬆的發現到這種規則應該有$A \rightarrow w$一類的格式，並且語言中的w應至少有一個字串由a開頭。為了這個目的，我們設定 *第一個集合*(first set)的w，記作 **Fi**（w），表示可以在w中找到的所有字串的集合，如果空字串也屬於w的話還要再加上ε。而透過文法規則$A_1 \rightarrow w_1, ..., A_n \rightarrow w_n$，就可以使用以下方法演算每條規則的**Fi**($w_i$)及**Fi**($A_i$)了：

1. 將每個**Fi**(w_i)及**Fi**(A_i)初始成空集合
2. 將*Fi*(w_i)加入每條$A_i \rightarrow w_i$規則中的**Fi**(A_i)，*Fi*定義如下：
   - 所有的a皆為終端符號時，*Fi*（$a\ w'$）= { a }
   - **Fi**（A）不包含ε時，相對於每個非終端符號A，*Fi*（$A\ w'$）= **Fi**（A）
   - **Fi**（A）包含ε時，相對於每個非終端符號A，*Fi*（$A\ w'$）= **Fi**（A）\ { ε } ∪ *Fi*（w'）
   - *Fi*(ε) = { ε }
3. 針對每條$A_i \rightarrow w_i$規則，將**Fi**(w_i)加入**Fi**(Ai)
4. 重複步驟2與步驟3，直到所有**Fi**集合固定下來。

不幸的是，第一集合還不夠用來產生出剖析表。由於規則中右手邊的w可能無限制的被覆寫成空字串，所以剖析器也在ε位於**Fi**（w）並且輸入資料流中的符號可以符合A的時候套用$A \rightarrow w$。所以還需要一個記作**Fo**（A）的A的*跟隨集合*(follow set)，表示可以由開始的符號衍生出$\alpha A a \beta$字串的終端符號a的集合。非終端符號的跟隨集合可以用以下方法得出：

1. 將每個**Fo**(A_i)初始成空集合
2. 若存在$A_j \rightarrow wA_iw'$ 格式的規則，則
   - 若終端符號a存在Fi（w'）中，則將a加入**Fo**(A_i)
   - 若ε存在Fi（w'）中，則將**Fo**(A_j)加入**Fo**(A_i)
3. 重複步驟2直到所有**Fo**集合固定下來

現在我們可以清楚定義每條規則要放在剖析表的哪裡了。若$T[A,a]$用以表示表格中代表非終端符號A及終端符號a的規則，則

> $T[A,a]$包含$A \rightarrow w$規則，若且唯若
>
> > a在**Fi**（w）之中，或
> > ε在**Fi**（w）之中，且a在**Fo**（A）之中。

若表格的每格中都僅包含一個規則，則剖析器總是知道該套用什麼規則，所以可在不用回溯的前提下剖析字串。在此情形下，這個語法可以稱為*LL(1)語法*。

建構LL(*k*)剖析表格

剖析表格可能（一般來說，在最差狀況下）必須有k次的指數複雜度的觀念在1992年左右PCCTS發表後改觀，它示範了許多程式語言可以用LL(k)來有效率的處理，而不會觸發剖析器的最差狀況。再者，在某些必須無限前瞻的狀況下，LL剖析也是合理的。相反的，傳統剖析器產生器，如yacc使用LALR(1)剖析表格建立被限制的LR剖析器，這種剖析器只能向後看固定的一個語彙符號。

参见

- 編譯器剖析器比較表
- 抽象語法樹
- 由上而下剖析
- 由下而上剖析

外部链接

- An easy explanation of First and Follow Sets (http://www.jambe.co.nz/UNI/FirstAndFollowSets.html)（使用一種比c較直觀的方法解釋產生First與Follow集合的過程）
- A tutorial on implementing LL(1) parsers in C# (https://web.archive.org/web/20080916052313/http://www.itu.dk/people/kfl/parsernotes.pdf)
