第10章 自顶向下分析

上一章中，介绍了自顶向下分析的基本方法和思路，自顶向下分析就是从起始符号开始，不断的挑选出合适的产生式，将中间句子中的非终结符的展开，最终展开到给定的句子。所以此方法的关键字为 **展开** ，难点在于如何挑选出合适的产生式，以展开到给定的句子。

LL(1) 分析法是一种自顶向下分析方法，它通过精心构造语法规则而使得每个推导步可以直接挑选出合适的产生式。本章将详细介绍此方法。

10.1 LL(1) 分析法基本流程

为了解释什么是 LL(1) 法，首先来看一个简单的例子，语法为：

```
S → aS | bS | c
```

需要解析的句子为 abac ，按上一章最后一节的方法，解析过程如下：

Working-string	**Production**
S	S -> aS
aS	S -> bS
abS	S -> aS
abaS	S -> c
abac	**ACCEPT**

下面，一步一步跟踪这个解析过程，查看每一步是如何选择出需要的产生式的。首先，我们的目标是将起始符号 S 展开到 最终句子 abac 。把它们写在同一行来进行比较，如下：

Working-string	**Final-string**	**Production**
S	**a** bac	

假设有一个 strcmp 函数来比较符号串 “S” 和 “abac” ，从左向右一个符号一个符号的比较，找到第一个不匹配的符号，也就是 “S” 和 “a” ，上面的表格中，将第一个不匹配的符号加粗表示了。

因此，此时必须将中间句子中的 “S” 展开，才能得到最终句子。那如何展开呢？将最终句子中不匹配的这个 “a” ，和 S 的三个产生式的右边 aS 、 bS 和 c 相比，可以看出，只能选择 S → aS 展开，才可以和 “a” 匹配上，展开后得到中间句子 aS ：

Working-string	**Final-string**	**Production**
S	**a** bac	S -> aS
a **S**	a **b** ac	

再次比较此时的中间句子 “aS” 和最终句子 “abac” ，找到的第一个不匹配的符号分别为 “S” 和 “b” ，将 “b” 和 S 的三个产生式比较，发现只能选择 S → bS ，展开后得到中间句子 abS ：

Working-string	**Final-string**	**Production**
S	**a** bac	S -> aS
a **S**	a **b** ac	S -> bS
ab **S**	ab **a** c	

按以上原则，每次都对中间句子和最终句子进行比较，找到第一个不匹配的符号，然后利用不匹配的符号挑选出需要的产生式，最终展开到最终句子：

Working-string	**Final-string**	**Production**
S	**a** bac	S -> aS
a **S**	a **b** ac	S -> bS
ab **S**	ab **a** c	S -> aS
aba **S**	aba **c**	S -> c
abac	abac	**ACCEPT**

因此 LL(1) 法的基本思路为：

> 每个推导步中，从左向右比较中间句子和最终句子，找到第一个不匹配的符号，如：中间句子为 u **X** v 、最终句子为 u **a** w 。显然，**a** 一定是终结符， **X** 则可能为非终结符，也可能为终结符，有以下 4 种情况：
>
> > 情况 A ： **X** 为终结符，这种情况表明无论怎么展开都不可能展开到最终句子，即最终句子不合语法，此时终止推导。

情况 B ： **X** 为非终结符，假设它的所有产生式为 X → u1 | u2 | ... | un ，将 **a** 和这些产生式的右边 u1, u2, ... , un 相比较，找出可以和 **a** 匹配的 ui ，将 ui 代替中间句子 u X v 中的 X ，得到下一个中间句子 u ui v，然后开始下一轮展开。

情况 C ： **X** 为非终结符，但它的所有产生式的右边 u1, u2, ... , un 中，没有一个可以和 “a” 匹配上，这种情况表明最终句子不合语法，此时终止推导。

情况 D ： **X** 为非终结符，但它的所有产生式的右边 u1, u2, ... , un 中，有两个或以上的 ui 可以和 “a” 匹配上，这种情况表明此语法不适于用 LL(1) 分析法，需要修改语法。

以上算法中，有一个重要的问题没有讲清楚，那就是怎么找出可以和 **a** 匹配的 ui 来，上面这个例子当然是简单的，直接比较产生式的第一个字符和 **a** 就可以找到，但遇到复杂的情况，比如产生式的最前面是一连串的非终结符怎么办？比如 **X** 可以展开成空串时怎么办？这个问题，我们稍后再讲，先来对这个算法稍微优化一下，先把基本的流程搞清楚。

上面的算法中可以优化的地方在于，其实没必要每次都从头开始比较中间句子和最终句子，上一轮推导步中已经比较过了的部分就没必要再比较了，比如说这一轮的中间句子为 u X v ，最终句子为 u a w ，可以应用的产生式是 X → ui ，那么下一轮，可以把中间句子改为 ui v ，把最终句子改为 a w ，也就是把已经匹配的符号都去掉。这样每次不匹配的第一个符号就是最左边的符号。

按此思路，在展开的过程中插入一个 **Match** 动作，将已经匹配的符号去掉。另外，在起始符号和最终句子的末尾添加一个结束符 EOF （用 **$** 表示），整个推导过程如下：

Stack	**Input**	**Action**
S $	a bac$	Predict S -> aS
a S$	a bac$	Match “a”
S $	b ac$	Predict S -> bS
b S$	b ac$	Match “b”
S $	a c$	Predict S -> aS

Stack	Input	Action
a S$	a c$	Match “a”
S $	c $	Predict S -> c
c $	c $	Match “c”
$	$	**ACCEPT**

上面的过程中，Match 动作是将中间句子和最终句子最左边已匹配的符号去掉，这样每次不匹配的第一个符号就是最左边的符号，因此只需要根据最左边的两个符号来选择需要的动作。

Predict 动作就是应用一个产生式，将中间句子中的最左边的非终结符替换成该产生式的右边。

上面的列表的表头中，原来的 Working-string 改成了 Stack ，原来的 Final-string 改成了 Input ，这是因为这两列的符号串的操作方式就像一个栈和一个输入流一样。

以上分析的具体步骤为：

(1) 将结束符（EOF） $ 和起始符号 S 压入栈中；

(2) 从输入流（token stream）中读入下一个终结符(token)，赋给 a ，也就是执行一次 a = yylex();

(3) 设栈顶符号为 X ，有以下三种情况：

情况 A ： X == a 且 a == $ ，解析成功，终止解析；

情况 B ： X == a 且 a != $ ，执行 Match 动作，将 X 出栈，转到 (2) ；

情况 C ： X != a 且 X 是非终结符，有三种情况：

情况 C1 ： 在 X 的所有产生式 X → u1 | u2 | ... | un 的右边中，只有一个 ui 可以与 a 匹配上。此时，执行动作 Predict X → ui ，将 X 出栈，将 ui 入栈，转到 (3) ；

情况 C2 ： 在 X 的所有产生式 X → u1 | u2 | ... | un 的右边中，没有一个 ui 可以与 a

匹配上。此情况表明最终句子不合语法，终止解析。

情况 C3 ： 在 X 的所有产生式 X → u1 | u2 | ... | un 的右边中，有两个或以上的 ui 可以与 a 匹配上。此情况表明此语法不适于用 LL(1) 分析法，需要修改语法。

情况 D ： X ≠ a 且 X 是终结符，输入不合语法，终止解析。

以上就是 LL(1) 分析法的基本流程，所谓的 LL(1) ，第一个 L 表示从左向右扫描输入流，第二个 L 表示每一步展开的时候取中间句子中左边第一个非终结符进行展开，括号里面的 1 表示每次只读入 1 个符号，每次只利用这 1 个符号的信息来挑选产生式。

事实上，还有 LL(2) 、LL(3) 、 LL(k) 等分析法，每次一次性读入多个符号，然后利用这些符号的信息来挑选产生式。

以上所说的 Predict 动作，之所以叫 Predict ，是因为这个动作是预测的，只看到了第一个符号 a ，就预测接下来的一串符号必须是 ui 。

10.2 首字符集合（first set）和后继字符集合（follow set）

上面的基本流程中，有一个重要的问题没有讲清楚，那就是怎么从 X 的所有产生式的右边 X → u1 | u2 | ... | un 中找出可以和 a 匹配的 ui 出来。为了解决这个问题，需要利用到首字符集合（first set）和后继字符集合（follow set）。首先介绍首字符集合的定义：

首字符集合（first set） ： 一个符号串 u 的首字符集合，用 **First(u)** 表示，是 u 可以推导出的所有句子的第一个终结符的集合，也就是说，若 u ⇒ v ，且 v 为一个句子，则 v 的第一个终结符属于 First(u) ，若 v 是一个空句子，则 ε 也在 First(u) 里面。

对于非终结符 A ，若其所有产生式为： A → u1 | u2 | ... | un ，则 First(A) 为 First(u1), First(u2), ... , First(un) 的并集。

在 LL(1) 解析过程中，假设栈顶为非终结符 X ，且此时读入了一个终结符 a 。如果 a 在某个 ui 的首字符集合 First(ui) 中，且 First(u1), First(u2), ... , First(un) 互不相交，那么此时只能挑选 X → ui 来进行展开了，否则将无法和 a 匹配上。

例如，对上一节中的例子： S → aS | bS | c ， aS, bS, c 的首字符集合分别为 {a}, {b}, {c} ，当栈顶为 S 时，若读入的是 a ，则应选择产生式 S → aS ，将栈顶的 S 替换成 aS ，才可以和 a 匹配上，若读入的是 b 或 c ，则应选择 S → bS 或 S → c 。

当 First(u1), First(u2), ... , First(un) 有相交的情况时怎么办？如 S → aS | a | c 。此时就不能使用 LL(1) 法了，因为当栈顶的 S 遇到 a 时，无法判断出应按 S → aS 展开，还是按 S → a 展开。因此，含左递归的语法是不能使用 LL(1) 法来解析的，因为一个含左递归的语法（如：A → Aa | c）中，必然存在相交的现象。

如果一种语法可以用 LL(1) 法来解析，则称此语法为一种 **LL(1) 语法（LL(1) grammar）** ，LL(1) 语法需要的特性将在本章第 4 节的最后讲。

如果 a 不在任何 ui 的首字符集合中呢？此时要分两种情况考虑：

> 情况 A ： 没有任何 First(ui) 含有 ε ，也就是 X 不能用空串代替，此情况表明最终句子不符合语法，因为用任何一个产生式 X → ui 展开都不可能和 a 匹配上。
>
> 情况 B ： 有一个 First(ui) 中含有 ε ，也就是 X 可以用空串代替，此时如果 a 在 X 的后继字符集合 Follow(X) 中，则可以应用 X → ui 展开，进一步解析接下的符号，如果 a 不在 Follow(X) 中，则最终句子不合语法。

那么什么是后继字符集合？

> **后继字符集合（follow set）** ： 一个非终结符 A 的后继字符集合，用 **Follow(A)** 表示，是一个语法中可能推导出来的所有中间句子中，位于 A 后面的终结符（包括结束符 $ 、但不包括 ε ）的集合，或者说，对于所有从起始符号推导出来的中间句子，若其形式为 u A a w ，即若 S ⇒ u A a w ，则 a 属于 Follow(A)。

后继字符集合可以看成所有可以合法的站在非终结符 A 的后面的终结符（可能包括结束符 $ 、但不包括 ε ）的集合。

因此，当栈顶为 X ，读入的符号为 a ， a 不在任何 First(ui) 中，且 X ⇒ ε 的时候，那么 a 必须是 X 的后继字符才能保证最终句子是一个符合语法的句子。

若一个符号串 u = X1 X2 ... Xn ，则 First(u) 的计算步骤如下：

(1) 置 i = 1 ；

(2) 若 i == n + 1，则将 ε 加入 First(u) ，终止计算；

(3) 若 Xi 是终结符，则将 Xi 加入 First(u) ，终止计算；

(4) 若 Xi 是非终结符，则将 First(Xi) - ε 加入 First(u)，

4.1 若 First(Xi) 不含 ε ，则终止计算；

4.2 若 First(Xi) 中含有 ε ，则置 i = i + 1 ，转到 (2) 。

一个语法中所有非终结符的 follow set 的计算步骤如下：

(1) 将 $ 加入到 Follow(S) 中， S 为起始符号， $ 为结束符 EOF ；

(2) 对每条形如 A → u B v 的产生式，将 First(v) - ε 加入到 Follow(B) ；

(3) 对每条形如 A → u B 的产生式，或 A → u B v 的产生式（其中 First(v) 含 ε ），将 Follow(A) 加入到 Follow(B) 。

以下为一个计算 first set 和 follow set 的实例，语法为：

```
S  ->  AB
A  ->  Ca | ε
B  ->  cB'
```

```
B' ->  aACB' | ε
C  ->  b | ε
```

计算结果如下：

```
First(C)   =  {b, ε}
First(B')  =  {a, ε}
First(B)   =  {c}
First(A)   =  {b, a, ε}
First(S)   =  {b, a, c}

Follow(S)  = {$}
Follow(B)  = {$}
Follow(B') = {$}
Follow(C)  = {a, $}
Follow(A)  = {c, b, a, $}
```

10.3 LL(1) 动作表及 LL(1) 解析

LL(1) 动作表用 M 表示，可以看成一个二维数组，或一个字典，动作表中的 M[A, a] 保存了在解析过程中当栈顶为非终结符 A 、读入的符号为 a 时所应采取的动作。动作表的构造过程为：

对语法中的每条产生式： A → u ：

(1) 对 First(u) 中的所有终结符 a （不含 ε ），置 M[A, a] = “A → u” ；

(2) 若 First(u) 含 ε ，则对 Follow(A) 中的所有符号 a （可含 $ ），置 M[A, a] = “A → u” 。

构造出动作表后，解析步骤为：

(1) 将结束符 $ 和起始符号 S 压入栈中；

(2) 从输入流中读入下一个终结符，赋给 a ，也就是执行一次 a = yylex() ；

(3) 设栈顶符号为 X ，有以下三种情况：

情况 A ： X == a 且 a == $ ，解析成功，终止解析；

情况 B ： X == a 且 a ≠ $ ，执行 Match 动作，将 X 出栈，转到 (2) ；

情况 C ： X ≠ a 且 X 是非终结符，有两种情况：

情况 C1 ： M[X, a] = “X → u”，执行 Predict 动作，将 X 出栈，压入 u ，转到 (3) ；

情况 C2 ： M[X, a] 未定义，输入不合语法，终止解析；

情况 D ： X ≠ a 且 X 是终结符，输入不合语法，终止解析。

下面来练习一下 LL(1) 分析法，语法为：

```
E   ->  T E'
E'  ->  + T E' | ε
T   ->  F T'
T'  ->  * F T' | ε
F   -> ( E ) | int
```

要解析的句子为： int + int * int 。

首先计算出所有非终结符的 first set 和 follow set ：

```
First(E)  = First(T) = First(F) = { ( int }
First(T') = { * ε}
First(E') = { + ε}
Follow(E) = Follow(E') { $ ) }
Follow(T) = Follow(T') = { + $ ) }
Follow(F) = { * + $ ) }
```

下面开始构造动作表 M 。

首先看第一个产生式： E → T E’ 。 First(T E’) = { (int } 。因此 M[E, (] = “E → T E’” ， M[E, int] = “E → T E’” 。写成表格的形式：

	int	**+**	*****	**(**	**)**	**$**

E	E -> T E′			E -> T E′		
E′						
T						
T′						
F						

接下来看第二个产生式： E′ -> + T E′ 。 First(+ T E′) = { + } 。因此 M[E′, +] = “E′ → + T E′”。

再看第三个产生式： E′ → ε 。 First(ε) = { ε } ，且 Follow(E′) = { $) } ，因此 M[E′, $] = M[E′,)] = “E′ → ε” 。都写到表格里面：

	int	**+**	*****	**(**	**)**	**$**
E	E -> T E′			E -> T E′		
E′		E′ -> + T E′			E′ -> ε	E′ -> ε
T						
T′						
F						

重复以上方法，最终的分析表为：

	int	**+**	*****	**(**	**)**	**$**
E	E -> T E′			E -> T E′		
E′		E′ -> + T E′			E′ -> ε	E′ -> ε
T	T -> F T′			T -> F T′		
T′		T′ -> ε	T′ -> * F T′		T′ -> ε	T′ -> ε
F						

F	F -> int			F -> (E)		

下面来解析句子 int + int * int ：

Parse Stack	Remaining Input	Parse Action
E $	int + int * int $	Predict E -> T E′
T E′ $	int + int * int $	Predict T -> F T′
F T′ E′ $	int + int * int $	Predict F -> int
int T′ E′ $	int + int * int $	Match int
T′ E′ $	+ int * int $	Match int
T′ E′ $	+ int * int $	Predict T′ -> ε
E′ $	+ int * int $	Predict E′ -> + T E′
+ T E′ $	+ int * int $	Match +
T E′ $	int * int $	Predict T -> F T′
F T′ E′ $	int * int $	Predict F -> int
int T′ E′ $	int * int $	Match int
T′ E′ $	* int $	Predict T′ -> * F T′
* F T′ E′ $	* int $	Match *
F T′ E′ $	int $	Predict F -> int
int T′ E′ $	int $	Match int
T′ E′ $	$	Predict T′ -> ε
E′ $	$	Predict E′ -> ε
$	$	Match $, **ACCEPT**

10.4 LL(1) 语法的特性

可以用 LL(1) 法来解析的语法被称为 LL(1) 语法（ LL(1) grammar ），当且仅当一种语法具有以下特性时，此语法才是 LL(1) 语法:

对该语法中的任何非终结符 A ，若其所有产生式为： A → u1 | u2 | ... | un ，则：

（1） First(u1), First(u2), ... First(un) 互不相交；

（2） 若有一个 First(ui) 中含 ε ，则 Follow(A), First(u1), First(u2), ... First(un) 互不相交。

判别一种语法是否是 LL(1) 语法的方法就是构造其分析表，若构造过程中没有发现冲突，也就是表中的任何元素 M[X, a] 最多只有一个动作，那么此语法就是 LL(1) 语法。

10.5 LL(1) 分析法的优缺点

以上就是 LL(1) 分析法的具体步骤，用代码来实现以上步骤的难度应该不大。

LL(1) 分析法的优点是不需要回溯，构造方法较简单，且分析速度非常快，每读到第一个符号就可以预测出整个产生式来。缺点是对语法的限制太强，它要求同一个非终结符的不同产生式的首字符集合互不相交，能满足此要求的语法相当少，而将一个不满足此要求的语法改写到满足要求也相当不容易。因此，LL(1) 分析法目前已经应用的比较少了，下一章将开始介绍目前广泛使用的自底向上的 LR 分析法。

第 10 章完