Syntax Analysis

引论

文法：一种用于描述程序设计语言语法的表示方法，能够自然地描述程序设计语言构造的层次化语法结构

- 文法给出了程序设计语言的**语法规约**
- 可以基于文法构造语法分析器
- 语法分析器有助于源程序翻译代码
- 文法的扩展性有助于语言的演化

语法分析器接受词法单元序列，输出其语法树表示。一般可分为

- 通用型：没有实践性
- 自顶向下：处理 LL 文法
- 自底向上：处理 LR 文法

上下文无关文法（Context Free Grammar, CFG）

Definition

> A context-free grammar G is defined by the 4-tuple
>
> $$G = (V, \Sigma, R, S)$$
>
> where
>
> 1. V is a finite set; each element $v \in V$ is called *a nonterminal character* or a *variable*. Each variable defines a sub-language of the language defined by G
> 2. Σ is a finite set of *terminals*, disjoint from V, which make up the actual content of the sentence. The set of terminals is the alphabet of the language defined by the grammar G
> 3. R is the finite relation from V to $(V \cup \Sigma)^*$, where the asterisk represents the Kleene star operation. The members of R are called the *(rewrite) rule*s or *production*s of the grammar. (also commonly symbolized by a P)

> 4. S is the start variable (or start symbol), used to represent the whole sentence (or program). It must be an element of V

Derivations

从开始符号出发，每次将待处理串中的某个非终结符号替换为其某个产生式的体，这样的推导思路对应于**自顶向下**构造语法分析树的过程

推导：考虑一个文法符号序列 $\alpha A\beta$，其中 A 为非终结符，α, β 为任意文法符号串。设 $A \to \gamma$ 是一个产生时，则推导可写作

$$\alpha A\beta \Rightarrow \alpha\gamma\beta$$

其中 $\Rightarrow$ 表示经一步推导出

> 经零步或多步推导出 (reflexive transitive closure)：$\overset{*}{\Rightarrow}$
>
> - 对任何串 α，有 $\alpha \overset{*}{\Rightarrow} \alpha$
> - 如果 $\alpha \overset{*}{\Rightarrow} \beta, \beta \Rightarrow \gamma$，则 $\alpha \overset{*}{\Rightarrow} \gamma$
>
> 经一步或多步推导出 (transitive closure)：$\overset{+}{\Rightarrow}$
>
> 最左推导 (leftmost derivation)：$\underset{\text{lm}}{\Rightarrow}$：推导时选择最左的非终结符
>
> 最右推导 (rightmost derivation)：$\underset{\text{rm}}{\Rightarrow}$：推导时选择最右的非终结符，也被称为规范推导 (canonical derivation)
>
> 每一步最左推导可以写作
>
> $$w A\gamma \underset{\text{lm}}{\Rightarrow} w\delta\gamma$$
>
> 其中 w 只包含终结符

句型（sentential form）：如果 $S \overset{*}{\Rightarrow} \alpha$，则 α 是文法的一个句型。句型可以包含终结符和非终结符，也可以是空串

句子（sentence）：不包含非终结符的句型

一个文法 G 生成的语言是其所有句子的集合，记作 $L(G)$

$$\omega \in L(G) \iff S \overset{*}{\Rightarrow} \omega$$

从推导的角度看，语法分析的任务是接受一个终结符串作为输入，找出从文法的开始符号推导出这个串的方法。

Parse tree

语法分析树（parse tree）是推导的图形表示形式，过滤了推导过程中对非终结符应用产生式的顺序。

- 根节点是文法的开始符号
- 叶子节点是非终结符，终结符或空串 ϵ
- 内部节点是非终结符
- 每个内部节点表示一次产生式的应用，其标号为产生式头，子节点从左到右是产生式体

从左到右排列叶节点的符号，得到根的一个句型，称为树的结果（yield）或边缘（frontier）

parse tree 可以反应串的语法层次。

> Derivation 与 Parse tree 的关系
>
> 考虑推导过程 $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \Rightarrow \alpha_n$ ，对于其中每个句型 α_i ，均可构造出一个结果为 α_i 的语法树，其过程是对 i 的归纳
>
> Basis. $\alpha_1 = A$ 的语法分析树是标号为 A 的单个节点
>
> Induction. 假设已经构造了结果为 $\alpha_{i-1} = X_1X_2X_3 \dots X_k$ 的语法分析树，设 α_i 是将 α_{i-1} 中的某个符号 X_j 替换为 $\beta = Y_1Y_2 \dots Y_m$ 得到的句型，即对 α_{i-1} 应用 $X_j \to \beta$ ，得到 $X_1X_2 \dots X_{j-1}\beta X_{j+1} \dots X_k$ ，则在语法分析树中找到左起第 j 个非 ϵ 叶节点，向其添加 m 个子节点 $Y_1Y_2 \dots Y_m$ 。若 $\beta = \epsilon$ ，则添加一个为 ϵ 的子节点

Ambiguity

若一个文法可以为一个句子生成多个语法分析树，则这个文法是二义性的（ambiguous）。

对于编程语言的文法，需要其无二义性。或者使用消二义性规则（disambiguating rules）来消除二义性。 e. g.

- 运算时先乘除后加减
- 没有大括号时嵌套的 if-else 结构的匹配问题

Language Generated by a Grammar

一个文法的语言是从其开始符号出发，能推导得到的所有句子的集合。

验证文法生成的语言一般分为两部分

- 证明 G 生成的每个串都在 L 中
- 证明 L 的每个串都可由 G 生成

证明过程通常使用数学归纳法，对推导序列长度/串的长度归纳

CFG and RE

CFG 的表达能力强于 RE，每个 RE 都可以用一个 CFG 描述，但反之不成立。

> E. g.
>
> $$L = \{a^n b^n : n \geqslant 1\}$$
>
> 上述语言可以用 CFG 表示：
>
> $$S \to aSb \mid ab$$
>
> 但是不能用 RE 表示

RE 适合描述词法结构，CFG 适合描述嵌套结构

从一个 RE 的 NFA 可以构造一个 CFG

- 对 NFA 每个状态 i，创建非终结符 A_i
- 如果有 i 在输入 a 上到达 j 的 transition，增加产生式 $A_i \to aA_j$
- 如果 i 在输入 ϵ 上到达 j，增加产生式 $A_i \to A_j$
- 如果 i 是接受状态，增加产生式 $A_i \to \epsilon$
- 如果 i 是开始状态，令 A_i 为开始符号

文法的设计

为了进行高效的语法分析，需要对文法做一定的处理，如消除二义性，消除左递归，提取左公因子等

消除二义性

文法的二义性是指文法可以为一个句子生成多颗不同的语法分析树，如下图的文法

$$\begin{array}{lcl} stmt & \to & \textbf{if } expr \textbf{ then } stmt \\ & | & \textbf{if } expr \textbf{ then } stmt \textbf{ else } stmt \\ & | & \textbf{other} \end{array}$$

在分析嵌套的 if-then-else 结构时会产生 else 与哪个 then 匹配的问题。类 C 语言的规则是每个 else 与最近尚未匹配的 then 相匹配。

可以通过改写文法消除二义性

$$
\begin{array}{rcl}
stmt & \rightarrow & matched\_stmt \\
 & | & open\_stmt \\
matched\_stmt & \rightarrow & \textbf{if } expr \textbf{ then } matched\_stmt \textbf{ else } matched\_stmt \\
 & | & \textbf{other} \\
open\_stmt & \rightarrow & \textbf{if } expr \textbf{ then } stmt \\
 & | & \textbf{if } expr \textbf{ then } matched\_stmt \textbf{ else } open\_stmt
\end{array}
$$

但是二义性的消除方法没有统一的规律可循，故一般不通过改变文法来消除二义性

消除左递归

如果一个文法中存在一个非终结符号 A 使得对某个串 α 存在推导 $A \overset{+}{\Rightarrow} A\alpha$，则称这个文法是左递归（left recursive）的

立即左递归：形如 $A \Rightarrow A\alpha$

立即左递归可通过如下的方法消除，首先将 A 的产生式分组

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

其中 β_i 都不以 A 开头。则可以将产生式替换为

$$
\begin{aligned}
A &\rightarrow \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A' \\
A' &\rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \epsilon
\end{aligned}
$$

这样做的前提是 $\alpha_i \neq \epsilon$

这样可以消除立即左递归，对于多步推导产生的左递归，可以通过以下的算法消除，算法要求文法中不出现**环**（形如 $A \overset{+}{\Rightarrow} A$ 的推导）或 ϵ **产生式**（形如 $A \rightarrow \epsilon$）

> 按照某个顺序将非终结符排序为 $A_1, A_2, \ldots, A_n$
>
> i 从 1 到 n，若存在 $A_i \rightarrow A_j\gamma, (i > j)$，则将其替换为
>
> $$A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \cdots \mid \delta_k\gamma$$
>
> 其中 $A_j \rightarrow \delta_1 \mid \delta_2 \mid \cdots \mid \delta_k$ 是所有 A_j 的产生式
>
> 经过这样的过程，所有形如 $A_i \rightarrow A_k\gamma$ 的产生式都满足 $k \geqslant i$，其中若存在 A_i 的立即左递归则将其消除。
>
> 消除立即左递归后所有形如 $A_i \rightarrow A_k\gamma$ 的产生式满足 $k > i$，循环结束后不会有左递归

提取左公因子

提取左公因子是一种文法转换方法，可以产生适用于自顶向下语法分析技术的文法。主要用于在不清楚两个产生式中如何选择时，可以通过改写产生式来推后决定

对于一个非终结符 A，提取左公因子的过程如下。

找出它的多个可选项之间的最长公共前缀 α，并且满足 $\alpha \neq \epsilon$，则将其所有的产生式

$$A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_n \mid \gamma$$

替换为

$$A \to \alpha A' \mid \gamma$$
$$A' \to \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

自顶向下分析技术

自顶向下语法分析可以看作是为输入串构造一个语法分析树的问题，从语法分析树的根节点开始构造，也可以看作寻找输入串的最左推导的过程。

递归下降分析框架

自顶向下的语法分析技术可以通过一个递归下降的分析框架实现。

- 框架由一组过程组成，每个非终结符对应一个过程
- 程序从开始符号的过程开始
- 每个过程选择一个产生式体，扫描相应的句子，遇到非终结符则调用符号对应的过程

非终结符 A 的过程可描述为

```
void A(){
    选择一个产生式 A -> X_1 X_2 ... X_k
    for(i = 1 to k){
        if(X_i 是非终结符)
            调用 X_i
        else if(X_i 等于当前输入 a)
            读入下一个输入
        else
            错误
    }
}
```

但是在 A 有不止一个产生式时，就可能需要回溯，因为当前产生式产生错误不代表其他产生式不能匹配。然而在进行语法分析时一般不回溯。通过预测分析技术配合对文法的修改可以实现不回溯而生成语法分析树

递归下降的分析过程遇见左递归时会陷入死循环，故需要**消除左递归**

> 需要一个好的方式来选择 A 的产生式

预测分析法

预测分析技术通过在输入中向前看固定多个符号来选择正确的产生式。通常情况下只用看一个。

预测分析技术通过

- 消除二义性：避免歧义
- 消除左递归：防止递归无法退出
- 提取左公因子：查看下一个输入符号时保证满足的产生式唯一

来实现**确定性**的，**无回溯**的分析技术

> 递归下降分析一般只适合于每个子表达式的第一个终结符能够为产生式选择提供足够信息的那些文法

在实现预测分析技术时，借助两个函数 $FIRST(), FOLLOW()$ 来根据下一个输入符号确定应用哪个产生式。

当前句型为 $xA\beta$，而输入是 xa，则选择产生式 $A \to \alpha$ 的必要条件是下列之一

- $A \overset{*}{\Rightarrow} \epsilon$ 且 β 以 a 开头，即某个句型中 a 在 A 之后
- $A \overset{*}{\Rightarrow} a\ldots$

当按以上两个条件选择时能保证唯一性，就可以避免回溯

First

$First(\alpha)$ 是可从 α 推导得到的串的首符号的集合，如果 $\alpha \overset{*}{\Rightarrow} \epsilon$，则 ϵ 也在 $FIRST(\alpha)$ 中

在预测分析时，考虑两个产生式 $A \to \alpha \mid \beta$，其中 $FIRST(\alpha), FIRST(\beta)$ 是不相交的集合，则只用查看下一个输入符号就可以在两个产生式之间选择

对于文法符号 X 的 $FIRST(X)$，不断应用下列规则直至没有终结符或 ϵ 可加入为止

- 如果 X 是终结符，$FIRST(X) = \{X\}$
- 如果 X 是非终结符，且有规则 $X \to a\ldots$，则将 a 添加到 $FIRST(X)$
- 如果 $X \to \epsilon$，将 ϵ 加入 $FIRST(X)$
- 如果 X 是非终结符且有产生式 $X \to Y_1Y_2\ldots Y_k$
    - 如果对某个 i，a 在 $FIRST(Y_i)$ 且 ϵ 在所有的 $FIRST(Y_1), FIRST(Y_2), \ldots, FIRST(Y_{i-1})$ 中，则 a 在 $FIRST(X)$ 。

- 即 $FIRST(Y_1)$ 的符号都在 $FIRST(X)$，而若 $Y_1 \overset{*}{\Rightarrow} \epsilon$，则将 $FIRST(Y_2)$ 也加入 $FIRST(X)$，以此类推。
- 只有在对所有 $j = 1, 2, \ldots, k$ 均有 $\epsilon \in FIRST(Y_j)$ 时，才将 ϵ 加入 $FIRST(X)$

对于一个文法符号串 $X_1 X_2 \ldots X_n$，计算 $FIRST(X_1 X_2 \ldots X_n)$ 的过程同理，首先将 $F(X_1)$ 加入，若 ϵ 在 $FIRST(X_1)$，则再加入 $FIRST(X_2)$，以此类推。如果对所有 $i = 1, 2, \ldots, n$ 有 $\epsilon \in FIRST(X_i)$，则将 ϵ 加入

Follow

对于非终结符 A，$FOLLOW(A)$ 是在某些句型后紧跟在 A 之后的终结符的集合（没有 ϵ）

在预测分析时，若 $A \to \alpha$，当 $\alpha \overset{*}{\Rightarrow} \epsilon$ 时，$FOLLOW(A)$ 可以用于选择恰当的产生式

对于非终结符 A，$FOLLOW(A)$ 的计算为不断应用如下规则直至没有终结符可以加入

- 将 $ 加入 $FOLLOW(S)$，其中 $ 是标志输入的结束标记
- 如果存在产生式 $A \to \alpha B \beta$，则 $FIRST(\beta)$ 中除 ϵ 以外的符号都在 $FOLLOW(B)$
- 如果存在产生式 $A \to \alpha B$ 或存在产生式 $A \to \alpha B \beta$ 且 $\epsilon \in FIRST(\beta)$，则 $FOLLOW(A)$ 所有符号都在 $FOLLOW(B)$ 中

LL(1) 文法

LL(1) 意为从左向右扫描输入，产生最左推导，且每一步中只用向前看 1 个输入符号

LL(1) 文法利用不回溯的确定性的预测分析技术

一个文法是 LL(1) 的 $\iff$ 其任意两个不同的产生式 $A \to \alpha \mid \beta$ 满足

1. 不存在终结符 a 使得 α 和 β 都能推出以 a 开头的串
2. α, β 中最多只有一个可以推导出 ϵ
3. 如果 β 可以推出 ϵ，则 α 不能推出任何以 $FOLLOW(A)$ 中的终结符开头的串，反之亦然

上述三个条件中，前两个条件等价于 $FIRST(\alpha) \cap FIRST(\beta) = \varnothing$，第三个条件等价于如果 $\epsilon \in FIRST(\beta), FIRST(\alpha) \cap FOLLOW(A) = \varnothing$，反之亦然

对于一个 LL(1) 文法，可以通过当前的输入为非终结符选择唯一的产生式，具体而言就是构造一个**预测分析表**，每个表项 $[A, a]$ 表示当最左推导的非终结符为 A，当前输入为 a 时应该为 A 选择哪个产生式。其构造方法为：对于文法的每个产生式 $A \to \alpha$

- 对于 $FIRST(\alpha)$ 中的每个终结符 a，将 $A \to \alpha$ 加入 $[A, a]$
- 若 $\epsilon \in FIRST(\alpha)$，对于 $FOLLOW(A)$ 中的每个终结符 b（包括结束符 $）$，将 $A \to \alpha$ 加入 $[A, b]$

构造完成后，表中的空条目即为发生语法错误的情况

任何文法都可以通过上述算法生成预测分析表，但 LL(1) 的分析表中每个条目都**唯一地**指定了一个产生式，或是标明一个语法错误

非递归的预测分析

自顶向下的分析过程需要用到递归下降分析框架。通过消除左递归，消除二义性，提取左公因子可使其过程确定无回溯。

在自顶向下分析过程中，总是匹配掉句型中左边的所有终结符，对于最左的非终结符，使用恰当的产生式将其展开，而匹配后的终结符不再考虑，只需记住**句型余下的部分以及尚未匹配的输入串**，由于展开总是在最左端的非终结符，故可以使用栈来存放余下的句型，实现非递归的预测分析。

考虑已经匹配的终结符部分 w 和栈中的文法符号串 α，满足 $S \overset{*}{\Rightarrow} w\alpha$

具体实现时：

- 栈中初始为开始符号 S
- 当栈非空时，设栈顶的符号为 X
  - 若 X 为终结符，且 X 等于当前输入符号 a，则将其弹出，读入下一个符号
  - 若 X 为终结符，且 X 不等于当前输入符号 a，则发生错误
  - 若 X 为非终结符，且 $[X, a]$ 为产生式 $X \to Y_1 Y_2 \dots Y_k$，则按照 $Y_k, Y_{k-1}, \dots Y_1$ 的顺序将其入栈，即栈顶为 Y_1
  - 若 X 为非终结符，且 $[X, a]$ 为空，则发生错误

最终栈为空时将输出一个输入字符的最左推导，或是发生错误

当发生错误时，从错误中恢复的基本思想为：语法分析器忽略输入中的一些符号，直到输入中出现由设计者选定的同步词法单元集合中的某个词法单元。从错误中快速恢复的关键是**同步词法单元**的选取：

- 可以将 $FOLLOW(A)$ 中的终结符都放到 A 的同步集合中，即略过一些输入直到出现 $FOLLOW(A)$ 中的符号，然后将 A 弹出，很可能分析过程就能继续进行
- 根据语言中的层次结构，可将较高层次构造的开始符号加入较低层次构造的同步集合
- 可将 $FIRST(A)$ 中的终结符加入 A 的同步集合，这样出现其中符号时可根据其继续分析 A
- 对于可推出 ϵ 的非终结符，可以将 ϵ 当作默认值

自底向上分析技术

自底向上分析技术从叶节点开始为输入串构建语法分析树

归约与句柄剪枝

自底向上的分析过程可以看作将一个串**归约**为开始符号的过程

归约（reduction）：将与某个特定产生式体匹配的子串替换为对应产生式的头

归约是推导的反向操作，将串归约为开始符号等价于找一个从开始符号到串的推导序列（最右推导）

对输入从左至右扫描，并且在扫描过程中使用自底向上的分析技术，可以反向构造一个最右推导

句柄（handle）：如果有

$$S \underset{\text{rm}}{\overset{*}{\Rightarrow}} \alpha A w \underset{\text{rm}}{\Rightarrow} \alpha \beta w$$

则产生式 $A \to \beta$ 是 $\alpha\beta w$ 的一个句柄。注意串 w 中仅有终结符。对句柄的归约代表了**最右推导**的一个反向步骤

逐渐将输入串中的句柄归约，最终得到一个从开始符号最右推导至串的反向序列。实现该过程可以使用**移入-归约框架**

需要解决的问题是何时归约以及使用哪个产生式归约

移入-归约框架

移入-归约语法分析（shift-reduce）是自底向上语法分析的具体形式之一。其使用一个栈存储文法符号，**句柄在被识别前，总是出现在栈顶**

在从左向右扫描输入串时，将符号放入串直至可以将栈顶的一个文法符号串 β 归约为某个产生式的头。重复该过程直至产生语法错误（没有可供归约的产生式），或是栈中有开始符号且输入为空（识别成功）

由于每次归约时归约的串右部都是终结符号，故最终得到的是一个反向的最右推导序列。移入-归约语法分析器有四种主要动作

- 移入（shift）：将下一个输入符号放入栈顶
- 归约（reduce）：被归约的串一定有一端在栈顶，找到其左端并决定用哪个产生式归约
- 接受（accept）：语法分析成功
- 报错（error）：发生语法错误

> 可以证明被识别的句柄都在栈顶。考虑连续的最右推导中的两种情况

> $$S \underset{\text{rm}}{\overset{*}{\Rightarrow}} \alpha Az \underset{\text{rm}}{\Rightarrow} \alpha\beta Byz \underset{\text{rm}}{\Rightarrow} \alpha\beta\gamma yz$$
>
> $$S \underset{\text{rm}}{\overset{*}{\Rightarrow}} \alpha BxAz \underset{\text{rm}}{\Rightarrow} \alpha Bxyz \underset{\text{rm}}{\Rightarrow} \alpha\gamma xyz$$
>
> 第一种情况，当栈中为 $\$\alpha\beta\gamma$ 时，将其归约为 $\$\alpha\beta B$ ，此时需要通过零次或多次移入将 y 移入栈，然后将栈顶的句柄 βBy 归约为 A
>
> 第二种情况类似。在完成一次归约后必须经过零次或多次归约才能在栈顶找到下一个句柄。故句柄被识别时一定在栈顶

移入-归约分析中的冲突

在移入-归约分析中会有两种冲突

- 移入/归约（shift/reduce）：不能判断应该继续移入还是进行归约
- 规约/规约（reduce/reduce）：不能在多个可能的归约中做出正确选择

在已知栈中情况和未来 k 个输入的情况下，仍会出现冲突的 CFG 不是 LR(k) 文法

简单 LR 技术

目前自底向上的语法分析都基于 LR(k) 语法分析的概念，其中 L 代表从左到右扫描输入，R 代表反向构造最右推导序列，k 表示向前看 k 个符号，一般只有 $k = 0, k = 1$ 具有实践意义

一个文法若能为其构造出 LR 分析器，即是 LR 文法

LR 语法分析有优点

- 表驱动
- 对几乎所有编程语言，写出 CFG 即可构造出对应 LR 语法分析器
- LR 语法分析是最通用的无回溯的移入-归约技术，且高效
- 能尽早检测到错误
- LR 语法分析能分析的集合是 LL(k) 的真超集

项与 LR(0) 自动机

一个 LR 语法分析器维护一些状态， 表明当前语法分析过程，以及做出移入-归约决定

分析器维护的状态是**项（Item）**的集合。文法 G 的一个项是其某个产生式加上一个位于其产生式体中某处的点，如产生式 $A \to XY$ 的项为

$$A \to \cdot XY$$
$$A \to X \cdot Y$$
$$A \to XY \cdot$$

item 指明了在语法分析的过程中，已经看到了一个产生式的哪一部分

规范 LR(0) 项集族（canonical LR(0) collection）的一组项集提供了构建自动机的基础，这样的自动机叫做 LR(0) 自动机。

为了构建一个文法的规范 LR(0) 项集族，需要

- 增广文法（augmented grammar）
- CLOSURE 函数
- GOTO 函数

增广文法

G 的增广文法 G' 是在 G 中增加新符号 S' 和产生式 $S' \to S$

加入新产生式的意义在于告诉语法分析器何时停止语法分析并接受串，即分析器使用 $S' \to S$ 归约时 $\iff$ 分析器接受输入串

CLOSURE

如果 I 是一个项集，则 $CLOSURE(I)$ 根据以下两个规则构造

- I 中的项都在 $CLOSURE(I)$
- 若 $[A \to \alpha \cdot B\beta]$ 在 $CLOSURE(I)$，$B \to \gamma$ 是一个产生式且 $[B \to \cdot\gamma]$ 不在 $CLOSURE(I)$，则将 $[B \to \cdot\gamma]$ 加入 $CLOSURE(I)$

重复应用这些规则直至没有项可以加入

直观来说，$CLOSURE(I)$ 可以表示当前节点期待的输入有哪些（e. g. $[A \to \alpha \cdot B\beta]$ 期待输入是一个从 $B\beta$ 推导的子串，故 $[B \to \cdot\gamma]$ 也在 closure 中）

如果包含 B 且点在 B 最左端的项（e. g. $[A \to \alpha \cdot B\beta]$）被加入 closure，则对于所有 B 的产生式 $B \to \gamma$，$[B \to \cdot\gamma]$ 都会被加入 closure

因此可以将项分为

- kernel item：$[S' \to \cdot S]$ 以及所有点不在产生式最左端的项（e. g. $[A \to \alpha \cdot B\beta]$）
- non-kernel item：除 $[S' \to \cdot S]$ 以外所有点在最左端的项（e. g. $[B \to \cdot\gamma]$）

每个项集都是某个 kernel项集合的 closure

GOTO

对于函数 $GOTO(I, X)$，其中 I 是某个项集，X 是文法符号，其结果为 I 中所有的形如 $[A \to \alpha \cdot X\beta]$ 的项对应的项$[A \to \alpha X \cdot \beta]$ 的 closure

GOTO 定义了状态机中的状态随着输入符号的转换

LR(0) 自动机

对于为一个文法 G 构造规范 LR(0) 项集族 C ，可以按照以下步骤

1. 构造增广文法 G'
2. 构造 $CLOSURE([S' \to \cdot S])$ ，其为 C 的初始项集
3. 对 C 中的每个项集 I_i 和每个文法符号 X_j ，若 $GOTO(I_i, X_j)$ 非空且不在 C 中，将其加入 C
4. 重复 3 直至没有项集被加入 C

由此可构造文法 G 的 LR(0) 自动机，令规范 LR(0) 项集族的项集为状态机的状态（I_i 对应状态 i） ，状态的转换根据 GOTO 函数给出（$GOTO(I_i, X) = I_j$ 意为状态 i 在输入为 X 时转换到 j）。初始状态是 $CLOSURE([S' \to \cdot S])$ 项集，而接收状态为包含 $S' \to S\cdot$ 的项集

对于 LR(0) 自动机，假设当前处于状态 j 且下一个输入为 a ，假设 j 有一个在 a 上的转换，则移入 a， 否则归约

LR(0) 自动机的栈中保存的不是文法符号而是状态，栈中的状态记录的是自动机运行时经过的路径，且每个状态对应唯一确定的文法符号，故不需要在栈中存储文法符号

LR 语法分析算法

LR 语法分析器是表驱动的。分析器由一个栈（存储状态）和两个语法分析表（ACTION, GOTO） ，以及语法分析驱动组成的。对于任何 LR 分析器来说，驱动都相同，但是不同的 LR 分析器表不同

分析器的栈中存储的是状态序列 $s_0 s_1 \dots s_m$ ，栈顶为 s_m 。根据构造状态的过程，每个状态都有一个对应的文法符号，有一个从 i 到 j 在 X 上的转换当且仅当 $GOTO(I_i, X) = I_j$ ，则所有到达 j 的转换都对应 X，即 j 对应的文法符号为 X 。故除开始状态外每个状态都和唯一的文法符号对应。

语法分析表的结构

语法分析借助两个表 ACTION 和 GOTO

ACTION 有两个参数：状态 i 和输入符号 a ，对于一组输入，有四种可能的动作

- 移入 j ： j 是一个状态，代表把 a 移入栈，但是用 j 代表 a
- 归约 $A \to \beta$ ： 将栈顶的 β 归约为 A
- 接受：接受输入并完成语法分析
- 报错：分析过程中发现错误

GOTO 由项集上的 GOTO 扩展，如果 $GOTO(I_i, X) = I_j$ ，则在表中 $GOTO[i, X] = j$

语法分析器的状态

语法分析器的完整状态包括栈和余下的输入，形如

$$s_0 s_1 \dots s_m, a_i a_{i+1} \dots a_n \$$$

对应一个最右句型

$$X_1 X_2 \dots X_m a_i a_{i+1} \dots a_n$$

每个状态 s_i 都代表一个文法符号 X_i ，开始状态 s_0 不代表任何文法符号，仅是作为栈底标记

语法分析器的行为

根据上述状态， 语法分析器读入下一个输入 a_i 和栈顶符号 s_m ，然后根据 $ACTION[s_m, a_i]$ 的结果做出行为

如果是移入 s ，则将状态 s 移入栈，当前状态为

$$s_0 s_1 \dots s_m s, a_{i+1} \dots a_n \$$$

下一个输入为 a_{i+1} 。a_i 不需要存放在栈中，如果需要（实践中从不需要 a_i）可以从 s 中恢复

如果是归约 $A \to \beta$ ，则执行一次归约动作，当前状态为

$$s_0 s_1 \dots s_{m-r} s, a_i a_{i+1} \dots a_n \$$$

语法分析器将 r 个符号弹出栈，其中 r 为 $|\beta|$ ，然后将状态 s 压入栈，其中 $s = GOTO[s_{m-r}, A]$ ，在归约是当前输入不变化，对于弹出的状态序列 $s_{m-r+1} \dots s_m$ 对应的文法符号序列 $X_{m-r+1} \dots X_m$ 总是等于归约时的产生式体 β

如果是接受，则语法分析过程完成

如果是错误，则说明语法分析器发现一个语法错误

所有 LR 语法分析器都按照如上的方式运行，区别只在于语法分析表的内容不同

SLR 构造方法

SLR 构造方法是一种构造语法分析表的算法，通过这种方式构造的语法分析表称为 SLR 语法分析表，且将使用 SLR 语法分析表的分析器称为 SLR 语法分析器。

SLR 方法以 LR(0) 项和 LR(0) 自动机为基础，且使用了额外的信息：每个文法符号的 FOLLOW 集合。构造过程如下

1. 构造增广文法 G' 的规范 LR(0) 项集族 $C = \{I_0, I_1, \dots I_n\}$
2. 根据项集 I_i 得到状态 i ，ACTION 根据以下原则确定

1. 如果 $[A \to \alpha \cdot a\beta]$ 在 I_i 并且有 $GOTO(I_i, a) = I_j$，则令 $ACTION[i, a]$ 为移入 j，这里的 a 必须是终结符
2. 如果 $[A \to \alpha\cdot]$ 在 I_i，对于 $FOLLOW(A)$ 中所有 a，将 $ACTION[i, a]$ 设为归约 $A \to \alpha$，这里的 $A \neq S'$
3. 如果 $[S' \to S\cdot]$ 在 I_i，则将 $ACTION[i, \$]$ 设为接受

   如果在上述过程中出现冲突，说明文法不是 SLR(1) 的，不能生成分析器

3. GOTO 的确定为如果 $GOTO(I_i, A) = I_j$，则 $GOTO[i, A] = j$
4. 上述过程后未定义的条目都为报错
5. 初始状态即 $[S' \to \cdot S]$ 所在状态

算法得到的分析表称为文法 G 的 SLR(1) 分析表，常常省略 (1) 因为不处理向前看更多符号的状况

可行前缀

LR(0) 自动机刻画了可能出现在语法分析栈中的文法符号串。栈中的内容一定是某个最右句型的前缀，即栈中内容为 α 而余下输入为 x，则存在一个将 αx 归约到开始符号的序列，即 $S \underset{\text{rm}}{\overset{*}{\Rightarrow}} \alpha x$

但是不是所有的最右句型前缀都能出现在栈中，因为栈中的前缀**不能越过句柄**，语法分析器必须在移入下一个符号前将句柄归约

可行前缀：一个最右句型的前缀，且没有越过该最右句型的最右句柄的右端

SLR 分析技术基于 LR(0) 自动机能够识别可行前缀。如果存在推导过程 $S \underset{\text{rm}}{\overset{*}{\Rightarrow}} \alpha A w \underset{\text{rm}}{\Rightarrow} \alpha\beta_1\beta_2 w$，则称项 $A \to \beta_1 \cdot \beta_2$ 对可行前缀 $\alpha\beta_1$ 有效

可行前缀的有效项可以决定当前栈中为可行前缀时应该采取的行动（归约/移入），如果 $\beta_2 \neq \epsilon$，说明此时句柄还未完全出现，应该移入，否则应该归约。事实上，如果在某个文法的 LR(0) 自动机中从初始状态沿标号为 γ 的某个可行前缀的路径到达一个状态，该状态对应的项集就是 γ 的**有效项集**

有时在某个时刻会有两个有效项要求执行不同的动作，此时即产生了冲突。冲突意味着可行前缀可能是两个最右句型的前缀，其中一个包含了句柄，而另一个没有。对于 SLR 而言，如果要按照 $A \to \beta$ 归约，则新的句型 A 之后跟随着下一个输入符号，故仅当下一个输入符号在 $FOLLOW(A)$ 时才归约

SLR 分析器的弱点

SLR 分析器按照 $A \to \alpha$ 归约的条件是下一个输入 a 可以在某个句型中出现在 A 之后。考虑此时栈中符号串为 $\beta\alpha$，如果 $\beta A a$ 不可能是某个最右句型的前缀，即使 a 在 $FOLLOW(A)$ 中也不应当按照 $A \to \alpha$ 归约

需要**归约的条件更为严格**

更强大的 LR 语法分析器

更强大的 LR 分析器不光利用当前的信息，还在输入中向前看一个符号，有两种方法

- 规范 LR 方法，或直接称为 LR 方法，利用了一个被称为 LR(1) 的项集
- 向前看 LR 方法，或 LALR，基于 LR(0) 项集族，状态数少于基于 LR(1) 的方法，且能处理的文法多于 SLR

规范 LR(1) 项

SLR 中给出的根据 FOLLOW 集的归约条件过松，希望每个状态能明确指出哪些输入符号可以跟在句柄 α 后，使 α 可以归约为 A

LR(1) 的思想是对项精化，使其包含第二个分量，这个分量是一个终结符，项的形式变为了

$$[A \to \alpha \cdot \beta, a]$$

其中 $A \to \alpha\beta$ 是一个产生式，而 a 是一个终结符或结束标记 $\$$ 。a 又被称为向前看符号，在形如 $[A \to \alpha \cdot \beta, a]$ 且 $\beta \neq \epsilon$ 的项中向前看符号没有作用，但是一个形如 $[A \to \alpha \cdot, a]$ 的项只有在下一个输入为 a 时才要求按照 $A \to \alpha$ 归约。这样的 a 的集合是 FOLLOW 的一个子集

一个 LR(1) 项 $[A \to \alpha \cdot \beta, a]$ 对一个可行前缀 γ 有效的条件是存在

$$S \underset{\text{rm}}{\overset{*}{\Rightarrow}} \delta A w \underset{\text{rm}}{\Rightarrow} \delta\alpha\beta w$$

其中

1. $\gamma = \delta\alpha$
2. 或者 a 是 w 第一个符号，或者 $w = \epsilon, a = \$$

LR(1) 项集的构造

LR(1) 项集构造过程和 LR(0) 类似，只是修改了 CLOSURE 和 GOTO，加入了对于向前看符号的计算

$CLOSURE(I)$:

- I 中每个项都在 closure
- 对于 I 中的 $[A \to \alpha \cdot B\beta, a]$ ，若存在产生式 $B \to \gamma$ ，则对于 $FIRST(\beta a)$ 中的所有 b ，将 $[B \to \cdot\gamma, b]$ 加入 closure

为了理解新的操作，考虑

$$S \underset{\text{rm}}{\overset{*}{\Rightarrow}} \delta Aax \underset{\text{rm}}{\Rightarrow} \delta\alpha B\beta ax$$

其中 $\gamma = \delta\alpha$，假设 βax 推导出终结符串 by，即考虑产生式 $B \to \eta$

$$S \underset{\text{rm}}{\overset{*}{\Rightarrow}} \gamma Bby \underset{\text{rm}}{\Rightarrow} \gamma\eta by$$

这说明 $[B \to \cdot\eta, b]$ 是 γ 的有效项。b 可能是从 β 推导出的第一个终结符，也可能是 β 推导出 ϵ，从而 $b = a$，但显然 b 不可能从 x 推导出，故 b 可能是 $FIRST(\beta a)$ 的任意符号

GOTO 的构造不变，向前看符号不影响 GOTO

构造项集族的方法同理，只需将初始项集变为 $CLOSURE([S' \to \cdot S, \$])$

规范 LR(1) 语法分析表

规范 LR(1) 的分析表构造与 SLR 大同小异

1. 构造增广文法 G' 的规范 LR(1) 项集族 $C = \{I_0, I_1, \dots I_n\}$
2. 根据项集 I_i 得到状态 i，ACTION 根据以下原则确定
    1. 如果 $[A \to \alpha \cdot a\beta, b]$ 在 I_i 并且有 $GOTO(I_i, a) = I_j$，则令 $ACTION[i, a]$ 为移入 j，这里的 a 必须是终结符
    2. 如果 $[A \to \alpha\cdot, a]$ 在 I_i，并且 $A \neq S'$，则将 $ACTION[i, a]$ 设为归约 $A \to \alpha$
    3. 如果 $[S' \to S\cdot, \$]$ 在 I_i，则将 $ACTION[i, \$]$ 设为接受

    如果在上述过程中出现冲突，说明文法不是 SLR(1) 的，不能生成分析器
3. GOTO 的确定为如果 $GOTO(I_i, A) = I_j$，则 $GOTO[i, A] = j$
4. 上述过程后未定义的条目都为报错
5. 初始状态即 $[S' \to \cdot S, \$]$ 所在状态

算法生成的表称为规范 LR(1) 分析表，若不存在冲突条目，则称给定文法为 LR(1) 文法

LALR 语法分析表

LR(1) 的分析能力强于 SLR(1)，但是状态也很多，为此产生了 LALR

LALR 状态和 SLR 一样多，且能力强于 SLR

核心（core）：是指一个项集的第一分量的集合

LALR 的思想是将拥有相同核心的 LR(1) 项集合并为一个项集。由于 GOTO 的行为仅根据核心而定，故合并时也可合并 GOTO 函数的目标

合并项集不会引入新的移入/归约冲突，但是会引入新的归约归约冲突

> 不可能引入新的移入/归约冲突：
>
> 如果有移入归约冲突，即合并后有项 $[A \to \alpha\cdot, a], [B \to \beta \cdot a\gamma, b]$，这说明合并前必有一个集合有 $[A \to \alpha\cdot, a]$ 且每个集合都有某个项第一分量为 $B \to \beta \cdot a\gamma$，则合并前就有移入/归约冲突
>
> 可能引入新的归约归约冲突：
>
> 考虑 $\{[A \to c\cdot, d], [B \to c\cdot, e]\}$ 和 $\{[A \to c\cdot, e], [B \to c\cdot, d]\}$，合并后会有归约/归约冲突

在引入新的冲突这一点可看出 LALR 能力弱于规范 LR(1)

LALR 朴素的构造方法即将状态合并后计算 ACTION 和 GOTO

高效的方法即

- 只用内核项表示 LR(0) 或 LR(1) 项集，即只用 $[S' \to \cdot S]$ 或 $[S' \to \cdot S, \$]$ 以及点不在产生式体左侧的项表示项集
- 使用“传播和自发生成”的过程生成向前看符号，根据 LR(0) 内核生成 LALR(1) 内核
- 根据 LALR(1) 内核，使用 CLOSURE 函数对内核求 closure，将其当作规范 LR(1) 项集，求 ACTION 和 GOTO 条目

> 根据 LR(0) 内核项创建 LALR(1) 项集。在两种情况下，向前看符号 b 可以添加到某个 LALR(1) 项集 J 中的 LR(0) 项 $B \to \gamma \cdot \delta$ 上
>
> - 存在一个包含内核项 $[A \to \alpha \cdot \beta, a]$ 的项集 I 并且 $J = GOTO(I, X)$。此时无论 a 取何值，按照上文的 CLOSURE 算法构造 $GOTO(CLOSURE(\{[A \to \alpha \cdot \beta, a]\}), X)$，其结果总是包含 $[B \to \gamma \cdot \delta, b]$。这个 b 被称为自发生的。特殊的，$ 对 $[S' \to \cdot S]$ 是自发生的
> - 与上述情况相同，但 $a = b$，这种情况下 b 是传播的
>
> 具体可参见书 P 174-175

语法错误的恢复

有两种基本的方法：恐慌模式和短语层次的恢复

恐慌模式

基本思想：语法分析器忽略输入中的一些符号，直到出现由设计者选定的某个同步词法单元

语法错误的出现表示在输入的前缀已经不可能找到与某个非终结符对应的串，此时可以跳过当前的结构，假装已经找到了该结构，然后继续分析。同步词法单元就是结构结束的标志

短语层次的恢复

基本思想：在预测语法分析表的空白条目中插入错误处理例程的函数指针，例程可以改变、插入或删除输入中的符号；发出适当的错误消息