



# 编译原理

Compilers

极夜酱

# 目录

<b>1</b>	<b>有限状态自动机</b>	<b>1</b>
1.1	字母表 . . . . .	1
1.2	语言 . . . . .	4
1.3	DFA . . . . .	6
1.4	NFA . . . . .	9
1.5	正则表达式 . . . . .	11
<b>2</b>	<b>上下文无关语言</b>	<b>14</b>
2.1	上下文无关文法 . . . . .	14
2.2	CNF . . . . .	18
2.3	PDA . . . . .	22
2.4	预测分析法 . . . . .	26

# Chapter 1 有限状态自动机

## 1.1 字母表

### 1.1.1 字母表 (Alphabet)

字母表是一个非空的有限集合，一般用  $\Sigma$  表示，集合中的元素被称为符号/字符 (symbol)。

例如：

- $\Sigma = \{0, 1\}$ ：二进制数集合。
- $\Sigma = \{a, b, \dots, z\}$ ：小写字母集合。
- $\Sigma = \{ (, ), [, ], \{, \} \}$ ：括号集合。

### 1.1.2 串 (String)

串是一个由字母表中的字符组成的有限序列。

例如：

- 0011 和 11 是  $\Sigma = \{0, 1\}$  上的串。
- abc 和 bbb 是  $\Sigma = \{a, b, \dots, z\}$  上的串。
- $()()$  和  $(())$  是  $\Sigma = \{ (, ), [, ], \{, \} \}$  上的串。

#### 空串

空串使用  $\epsilon$  表示。

## 串的长度

- $|0010| = 4$
- $|aa| = 2$
- $|\epsilon| = 0$

## 前缀 (prefix)

- aa 是 aaabc 的前缀
- aaab 是 aaabc 的前缀
- aaabc 是 aaabc 的前缀

## 后缀 (suffix)

- bc 是 aaabc 的后缀
- abc 是 aaabc 的后缀
- aaabc 是 aaabc 的后缀

## 子串 (substring)

- ab 是 aaabc 的子串
- aaa 是 aaabc 的子串
- aaabc 是 aaabc 的子串

## 连接 (concatenation)

当  $\omega = abd$ ,  $\alpha = ce$ , 那么  $\omega\alpha = abdce$ 。

### 指数 (exponentiation)

当  $\omega = abd$ , 那么  $\omega^3 = abdabdabd$ ,  $\omega^0 = \epsilon$ 。

### 反转 (reversal)

当  $\omega = abd$ , 那么  $\omega^R = dba$ 。

### 1.1.3 克林闭包 (Kleene Closure)

$\Sigma^k$  用于表示所有在字母表  $\Sigma$  上的长度为  $k$  的串的集合。

例如,  $\Sigma = \{a, b\}$ , 那么  $\Sigma^2 = \{ab, ba, aa, bb\}$ ,  $\Sigma^0 = \{\epsilon\}$ 。

克林闭包  $\Sigma^*$  用于表示所有在字母表  $\Sigma$  上能够组成的串的集合。

$$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \dots = \bigcup_{k \geq 0} \Sigma^k \quad (1.1)$$

正闭包  $\Sigma^+$  则是在  $\Sigma^*$  中除了空串以外的所有串的集合。

$$\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \dots = \bigcup_{k > 0} \Sigma^k \quad (1.2)$$

## 1.2 语言

### 1.2.1 语言 (Language)

语言是一个字母表中所构成串的集合。

例如,  $\Sigma = \{a, b, c, \dots, z\}$ , 那么所有英语单词所构成的集合  $L$  就是字母表  $\Sigma$  上的语言。

假设  $A = \{good, bad\}$  和  $B = \{boy, girl\}$  是两个语言, 语言之间可以进行以下操作。

#### 并集 (union)

$$A \cup B = \{x \mid x \in A \text{ or } x \in B\} \quad (1.3)$$

$$A \cup B = \{good, bad, boy, girl\}$$

#### 连接 (concatenation)

$$A \circ B = \{xy \mid x \in A \text{ or } y \in B\} \quad (1.4)$$

$$A \circ B = \{goodboy, goodgirl, badboy, badgirl\}$$

#### 闭包

$$A^* = \{x_1, x_2, \dots, x_k \mid k \geq 0 \text{ and each } x_i \in A\} \quad (1.5)$$

$$A^* = \{\epsilon, good, bad, goodgood, goodbad, badgood, badbad, goodgoodgood, goodgoodbad, \dots\}$$

语法和语言与自动机理论密切相关, 它们是很多软件实现的基础, 例如编译器/解释器、文本编辑器、文本搜索、系统验证等。

在自动机理论中, 要处理的问题就是判断一个给定的串是否属于某个语言。

例如:

- $0^*10^*$ : 只包含一个 1 的串的集合。
- $\Sigma^*1\Sigma^*$ : 至少有一个 1 的串的集合。
- $\Sigma^*001\Sigma^*$ : 包含子串 001 的串的集合。
- $(\Sigma\Sigma)^*$ : 长度为偶数的串的集合。
- $(\Sigma\Sigma\Sigma)^*$ : 长度为 3 的倍数的串的集合。

## 1.3 DFA

### 1.3.1 DFA (Deterministic Finite Automata)

有限状态机 (FSM, Finite State Machine) 用于决定程序当前状态和状态间的切换, 状态机最终只能指向一个结果。

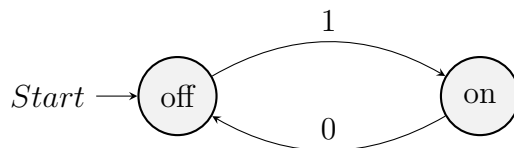


图 1.1: 有限状态机

确定性有限状态自动机 DFA 由一个五元组  $(Q, \Sigma, \delta, q_0, F)$  表示, 其中

- $Q$ : 状态集合
- $\Sigma$ : 字母表
- $\delta$ : 状态转移函数 (transition function)
- $q_0$ : 初始状态
- $F$ : 终结状态集合

例如 DFA 可以用来识别空串或者以 0 结尾的串:

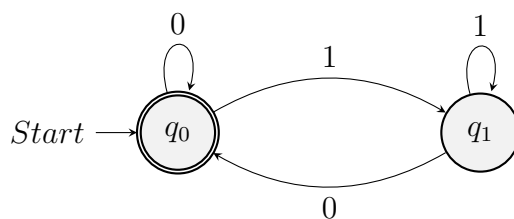


图 1.2: 识别空串或以 0 结尾的串的 DFA

其中  $Q = \{q_0, q_1\}$ ,  $\Sigma = \{0, 1\}$ ,  $q_0$  为初始状态,  $F = \{q_0\}$ ,  $\delta$  为



状态	输入	
	0	1
$q_0$	$q_0$	$q_1$
$q_1$	$q_0$	$q_1$

能够被有限自动机接受的语言被称为正则语言 (regular language)。

例如，构建一个能够识别所有包含子串 001 的串的 DFA：

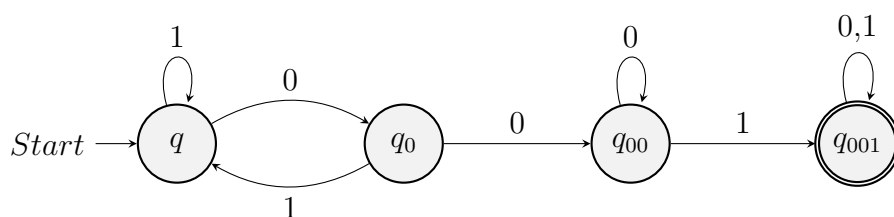
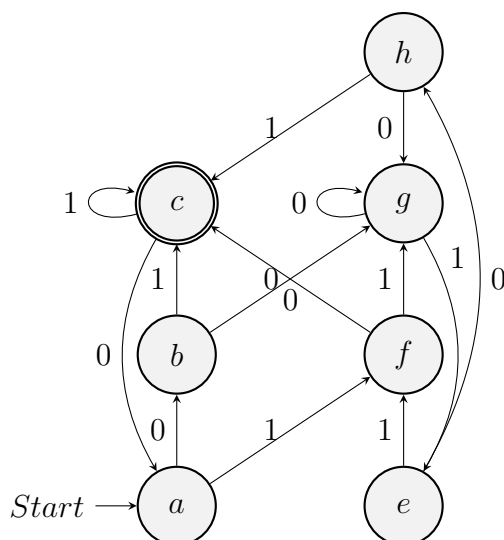


图 1.3: 识别包含子串 001 的串的 DFA

### 1.3.2 最小化 DFA

有限状态机的最小化，即将一个有限状态机转换为一个更小的有限状态机，使得状态的数目最少。

对于两个状态，如果它们之间的转移函数相同，则这两个状态可以合并为一个状态。



在这个 DFA 中，状态  $b$  和  $h$  是等价的，当接收 0 时都转移到状态  $g$ ，当接收 1 时都转移到状态  $c$ 。同时状态  $a$  和  $e$  也是等价的，状态  $a$  接收 0 转移到状态  $b$ ，状态  $e$  接收 0 转移到状态  $h$ ，状态  $a$  和  $e$  接收 1 时都转移到状态  $f$ 。

因此，状态  $b$  和  $h$  以及状态  $a$  和  $e$  可以进行合并。

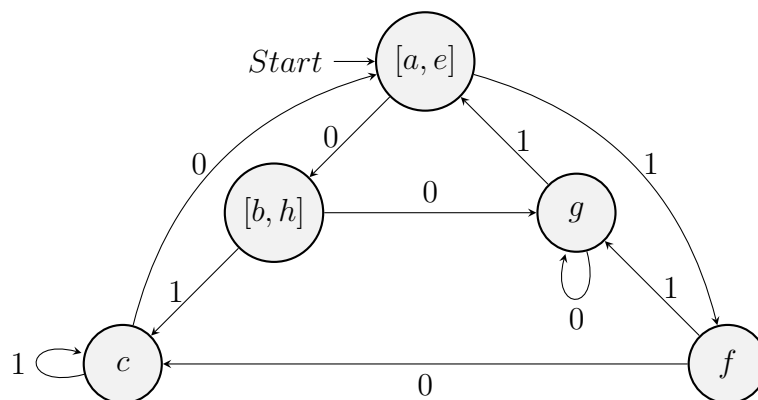


图 1.4: 最小化 DFA

## 1.4 NFA

### 1.4.1 NFA (Non-deterministic Finite Automata)

在 DFA 中，每个状态的下一个状态都是唯一确定的，但是非确定性有限状态自动机 NFA 可能会存在多个下一状态。

例如在这个 NFA 中，状态  $q_0$  存在两个接收 1 的箭头，而状态  $q_1$  没有接收 1 的箭头。

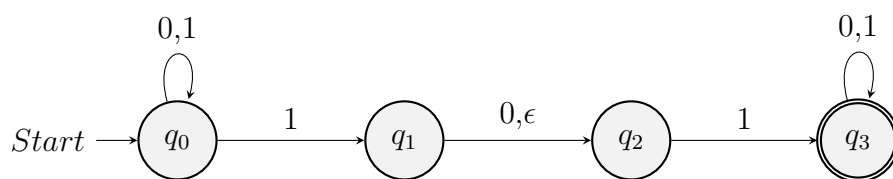


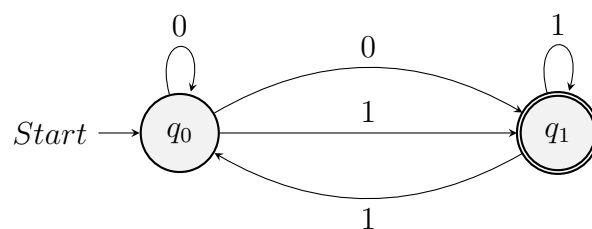
图 1.5: NFA

因此，在 NFA 中，每个状态允许对相同输入存在 0 个、1 个或多个转移的状态。如果存在一条能够到达终结状态的路径，那么就称当前的输入是被 NFA 接受的。

### 1.4.2 DFA 与 NFA 的转换

NFA 并不比 DFA 更加强大，理论证明 NFA 与 DFA 是等价的。

例如将一个 NFA 转换为 DFA：



构建一个与 NFA 等价的 DFA，只需将 NFA 中转换到的状态集合作为 DFA 中的一个状态即可。

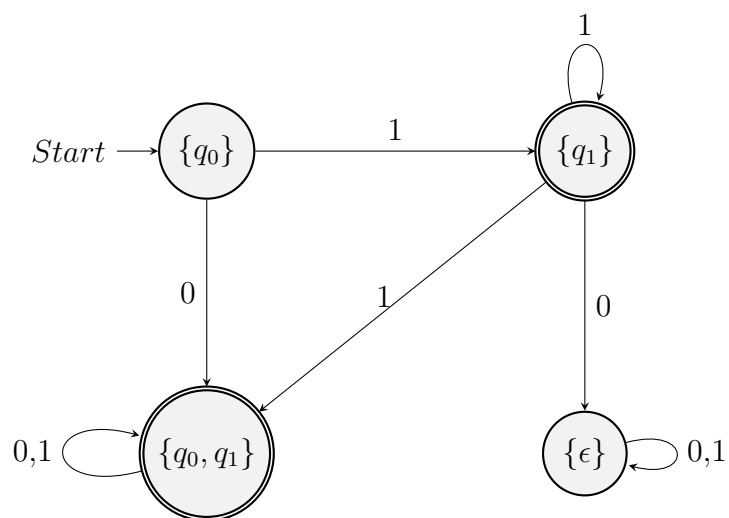


图 1.6: NFA 转换 DFA

### 1.4.3 $\epsilon$ -NFA

$\epsilon$ -NFA 允许不消耗输入字符在状态之间转移。

例如以下  $\epsilon$ -NFA 能够接受小数，如 +3.14、-0.12、.71、2. 等。

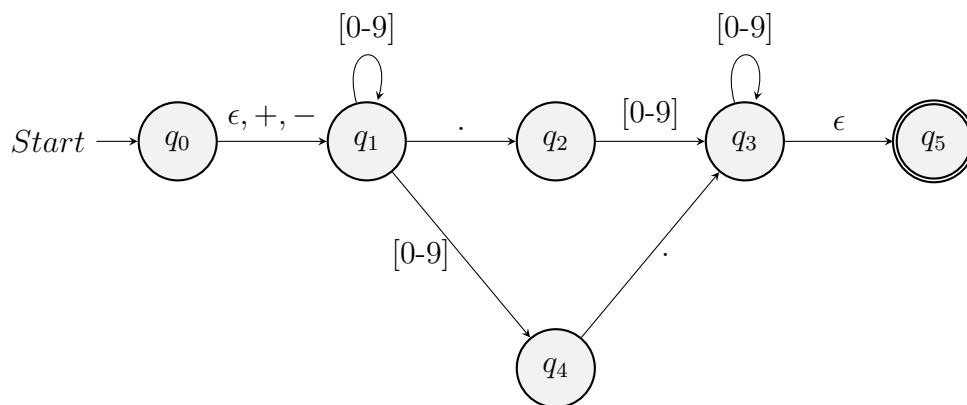


图 1.7: 接受小数的  $\epsilon$ -NFA

## 1.5 正则表达式

### 1.5.1 编译器 (Compiler)

编译器是一种特殊的程序，可以将一种编程语言的源代码翻译成机器码、字节码或另一种编程语言。

编译器包含以下阶段：

1. 词法分析器 (lexical analyzer)
2. 语法分析器 (syntax analyzer)
3. 语义分析器 (semantic analyzer)
4. 中间代码生成器 (intermediate code generator)
5. 代码优化器 (code optimizer)
6. 代码生成器 (code generator)

### 1.5.2 词法分析

词法分析是编译器的第一步，它的主要任务是读取源代码，并生成能够被解析器 (parser) 进行语法分析的 tokens 和语法树 (syntax tree)。

例如 `time = hour * 60 + minute`，经过词法分析后，将会得到：

- `id(time)`
- `assignment(=)`
- `id(hour)`
- `op(*)`
- `num(60)`
- `op(+)`

- `id(minute)`

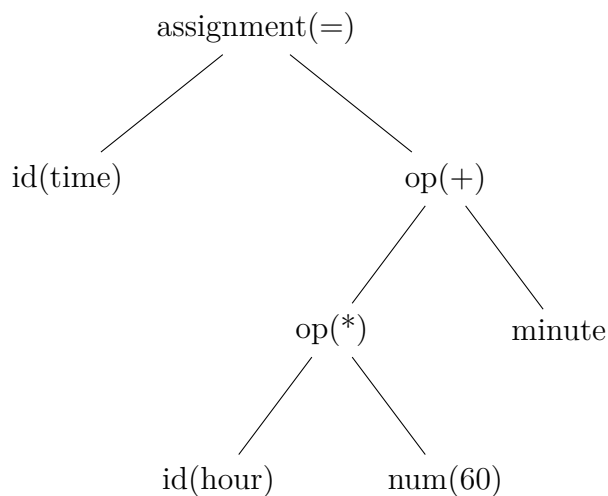


图 1.8: 语法树

### 1.5.3 正则表达式 (Regex, Regular Expression)

正则表达式描述了字符串匹配的模式 (pattern)，可以用来检查一个串是否包含某个子串、替换子串、或提取符合条件的子串。像 `grep`、`vi`、`python`、`lex` 等工具都支持正则表达式的使用。

例如用于匹配一个合法的变量名的正则表达式为 `[a-zA-Z_][a-zA-Z0-9_]*`。即变量名只能由字母或下划线开头，后面可以是任意多个字母、数字或下划线。

正则表达式支持以下操作：

- 连接： $ab$  或  $a \cdot b$
- 选择： $a \mid b$
- 克林闭包： $a^* = \{\epsilon, a, aa, aaa, \dots\}$
- 匹配至少 1 次： $a^+ = aa^*$
- 匹配 0 次或 1 次： $a? = a \mid \epsilon$
- 匹配任意字符： $.$

- 补集:  $(a \mid b)$

其中克林闭包运算的优先级最高，其次是连接，最后是选择。

例如  $(a \mid b)^*aa(a \mid b)^*$  用于匹配包含连续的 a 的串， $b^*(abb^*)^*(a \mid \epsilon)$  用于匹配没有连续的 a 的串。

然而  $\{a^n b^n \mid n \geq 0\}$  却不是正则语言，因为它无法用有限个状态来验证 a 和 b 的出现次数是相等的。

# Chapter 2 上下文无关语言

## 2.1 上下文无关文法

### 2.1.1 上下文无关文法 (CFG, Context Free Grammar)

CFG 能够描述某些具有递归结构的特征，它有足够强的语言表达力来表示大多数编程语言的语法。

CFG 由一个四元组  $(V, T, P, S)$  表示：

- $V$ ：变元 (variable) / 非终结符 (non-terminal) 集合，用大写字母表示。
- $T$ ：终结符 (terminal) 集合，用小写字母表示。
- $P$ ：产生式 (production) 集合。
- $S$ ：开始符号。

一个文法由一组替换规则产生。产生式集合

$$A \rightarrow \alpha_1$$

$$A \rightarrow \alpha_2$$

...

$$A \rightarrow \alpha_k$$

可以被写成  $A \rightarrow \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_k$  的形式。

### 2.1.2 推导 (Derivation)

推导用于确定符合文法规则的串的集合，即用来确定一个语言。



推导从开始符号开始，通过产生式进行替换，得到最终结果。

例如  $E \rightarrow E + E \mid E * E \mid (E) \mid id$ ，由开始符号  $E$  可以推导出  $(id + id) * id$ 。

$$\begin{aligned} E &\Rightarrow E * E \\ &\Rightarrow (E) * E \\ &\Rightarrow (E) * id \\ &\Rightarrow (E + E) * id \\ &\Rightarrow (E + id) * id \\ &\Rightarrow (id + id) * id \end{aligned}$$

解析树（parse tree）是描述推导的一种直观方法。

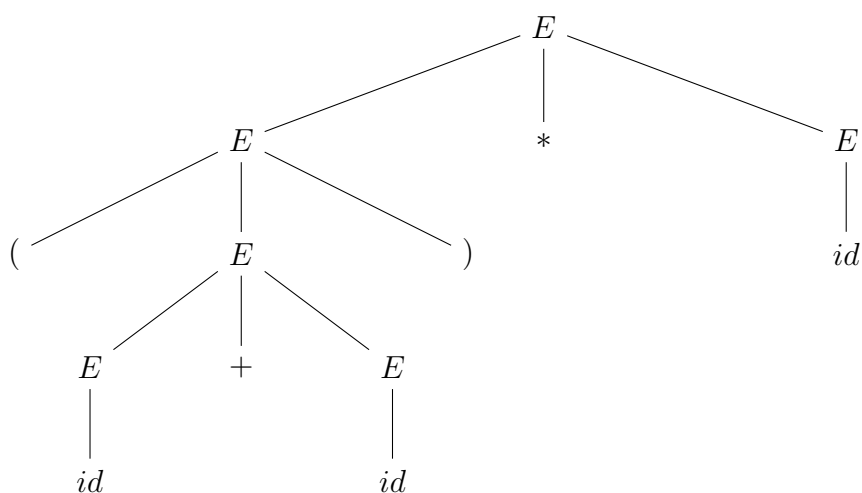


图 2.1: 分析树

如果只关注语义分析和代码生成所需的信息，可以将分析树简化为一棵抽象语法树（abstract syntax tree）。

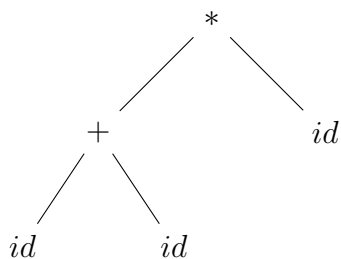


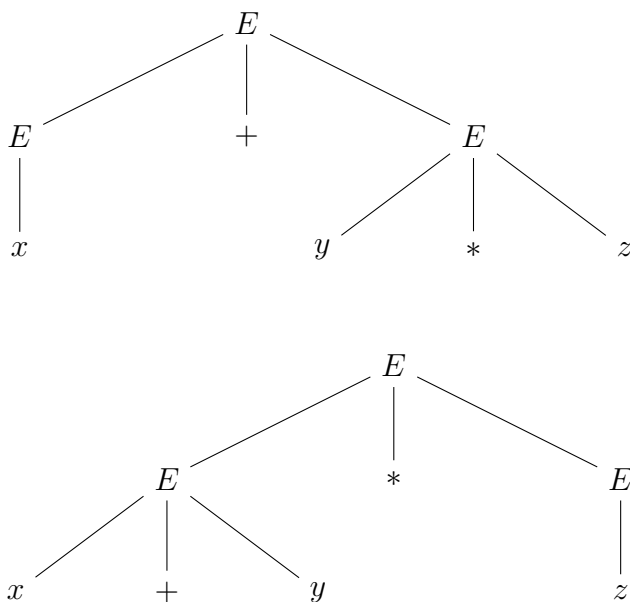
图 2.2: 抽象语法树

### 2.1.3 二义性 (Ambiguity)

在推导的过程中涉及到同级别表达式的替换, 因此按顺序可以分为最左推导 (left-most derivation) 和最右推导 (rightmost derivation)。

文法的二义性, 是指对于符合文法规则的同一个句子, 存在两种可能的分析树。

例如  $E \rightarrow E + E \mid E * E \mid (E) \mid x \mid y \mid z$ , 使用最左推导会对  $x + y * z$  产生两个不同的分析树。

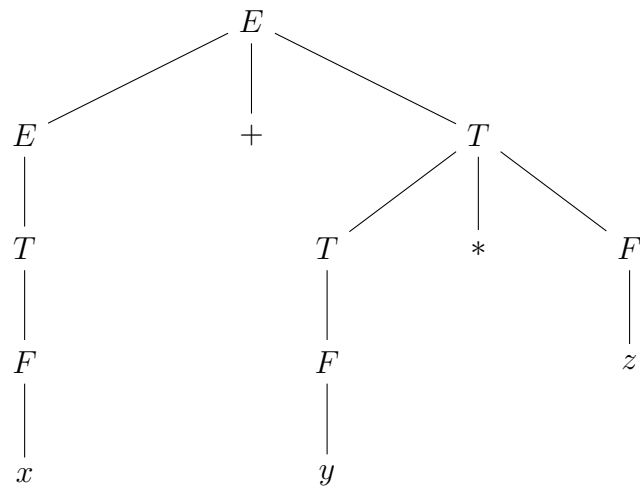


产生二义性的原因在于运算符之间的优先级在文法中并没有体现。消除二义性的办法就是在文法中引入一个中间量。

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow id \mid (E)$$



## 2.2 CNF

### 2.2.1 上下文无关语言

CFG 可以用来表示语言  $\{a^n b^n \mid n \geq 0\}$ :

$$S \rightarrow aSb$$

$$S \rightarrow ab$$

例如根据  $S$  可以生成生成 aaabbb:

$$S \Rightarrow aSb$$

$$\Rightarrow aaSbb$$

$$\Rightarrow aaabbb$$

CFG 好还可以用于表示  $a$  和  $b$  出现相等次数的语言, 例如 babaab:

$$S \rightarrow aB \mid bA$$

$$A \rightarrow a \mid aS \mid bAA$$

$$B \rightarrow b \mid bS \mid aBB$$

设计 CFG 需要一定的创造力, 大部分复杂的 CFG 可以由多个简单的 CFG 并集组成。

例如设计一个能够表示语言  $\{0^n 1^n \mid n \geq 0\} \cup \{1^n 0^n \mid n \geq 0\}$  的 CFG。

这两个部分可以分别表示为:

$$S_1 \rightarrow 0S_11 \mid \epsilon$$

$$S_2 \rightarrow 1S_20 \mid \epsilon$$

只需合并这两个部分, 即可得到最终的 CFG:

$$S \rightarrow S_1 \mid S_2$$

$$S_1 \rightarrow 0S_11 \mid \epsilon$$

$$S_2 \rightarrow 1S_20 \mid \epsilon$$

### 2.2.2 乔姆斯基范式 (CNF, Chomsky Normal Form)

CNF 在保留相同语言的同时对语法规则施加了一些限制，好处是可以避免解析过程中的歧义问题，另一个好处就是为解析的复杂度提供了一个上限。

CNF 规定每条 CFG 的每一条规则都必须满足：

1.  $S \rightarrow \epsilon$ ：开始变元  $S$  可以为空。
2.  $A \rightarrow BC$ ：单个变元可以推导出两个变元，其中  $B$ 、 $C$  不能为开始变元。
3.  $A \rightarrow a$ ：单个变元可以被终结符替换。
4. 不能出现单个变元推导出单个变元。

将 CFG 转换为 CNF 的步骤为：

1. 添加新的开始变元：确保开始变元始终在规则的左侧。
2. 消除所有  $\epsilon$  规则：消除从变元到空字符的规则。
3. 消除所有  $A \rightarrow B$  规则：消除单个变元到单个变元的规则。
4. 添加变元：为了满足  $A \rightarrow BC$  的规则，需要将  $A \rightarrow BCD$  替换为  $A \rightarrow ED$ ，即添加变元  $E \rightarrow BC$ 。

例如将 CFG 转换为 CNF：

$$S \rightarrow ABA$$

$$A \rightarrow aA \mid \epsilon$$

$$B \rightarrow bB \mid \epsilon$$

### 消除所有 $\epsilon$ 规则

将  $A \rightarrow \epsilon$  的规则，替换到出现  $A$  的规则中：

$$S \rightarrow ABA \mid \textcolor{red}{BA} \mid \textcolor{red}{AB} \mid B$$

$$A \rightarrow aA \mid \textcolor{red}{a}$$

$$B \rightarrow bB \mid \epsilon$$

将  $B \rightarrow \epsilon$  的规则，替换到出现  $B$  的规则中：

$$S \rightarrow ABA \mid BA \mid AB \mid B \mid \textcolor{red}{AA} \mid \textcolor{red}{A}$$

$$A \rightarrow aA \mid a$$

$$B \rightarrow bB \mid \textcolor{red}{b}$$

### 消除所有 $A \rightarrow B$ 规则

在  $S$  中出现了单个变元到单个变元的情况，将这些规则进一步替换：

$$S \rightarrow ABA \mid BA \mid AB \mid \textcolor{red}{bB} \mid \textcolor{red}{b} \mid AA \mid \textcolor{red}{aA} \mid \textcolor{red}{a}$$

$$A \rightarrow aA \mid a$$

$$B \rightarrow bB \mid b$$

目前， $S \rightarrow BA$ 、 $S \rightarrow AA$ 、 $S \rightarrow AB$ 、 $S \rightarrow a$ 、 $S \rightarrow b$ 、 $A \rightarrow a$ 、 $B \rightarrow b$  这些规则已经满足了 CNF 的要求：

$$S \rightarrow ABA \mid \textcolor{teal}{BA} \mid \textcolor{teal}{AB} \mid bB \mid \textcolor{teal}{b} \mid \textcolor{teal}{AA} \mid aA \mid a$$

$$A \rightarrow aA \mid \textcolor{teal}{a}$$

$$B \rightarrow bB \mid \textcolor{teal}{b}$$

### 添加变元

为了消除  $A \rightarrow BCD$  这种情况，需要添加新的变元进行替换。

假设  $X \rightarrow AB$ ：

$$S \rightarrow \textcolor{red}{X}A \mid \textcolor{green}{B}A \mid \textcolor{green}{A}B \mid bB \mid \textcolor{green}{b} \mid \textcolor{green}{A}A \mid aA \mid a$$

$$A \rightarrow aA \mid \textcolor{green}{a}$$

$$B \rightarrow bB \mid \textcolor{green}{b}$$

$$\textcolor{red}{X} \rightarrow \textcolor{red}{A}B$$

同时为了满足 CNF 规则中  $A \rightarrow BC$  的要求，需要对如  $A \rightarrow aA$  这样的规则进行替换。

假设  $A_1 \rightarrow a$ 、 $B_1 \rightarrow b$ ：

$$S \rightarrow \textcolor{green}{X}A \mid \textcolor{green}{B}A \mid \textcolor{green}{A}B \mid \textcolor{red}{B}_1B \mid \textcolor{green}{b} \mid \textcolor{green}{A}A \mid \textcolor{red}{A}_1A \mid a$$

$$A \rightarrow \textcolor{red}{A}_1A \mid \textcolor{green}{a}$$

$$B \rightarrow \textcolor{red}{B}_1B \mid \textcolor{green}{b}$$

$$X \rightarrow \textcolor{green}{A}B$$

$$\textcolor{red}{A}_1 \rightarrow \textcolor{green}{a}$$

$$\textcolor{red}{B}_1 \rightarrow \textcolor{green}{b}$$

这样就完成了 CFG 到 CNF 的转换，语法中的每条规则都满足了 CNF 的要求。

## 2.3 PDA

### 2.3.1 下推自动机 (PDA, Pushdown Automata)

DFA 和 NFA 由于受限于存储空间的问题，不能识别类似于  $\{a^n b^n \mid n \geq 0\}$  这种语言。PDA 通过一个栈 (stack) 解决了这个问题。PDA 与 CFG 的功能的等价的。

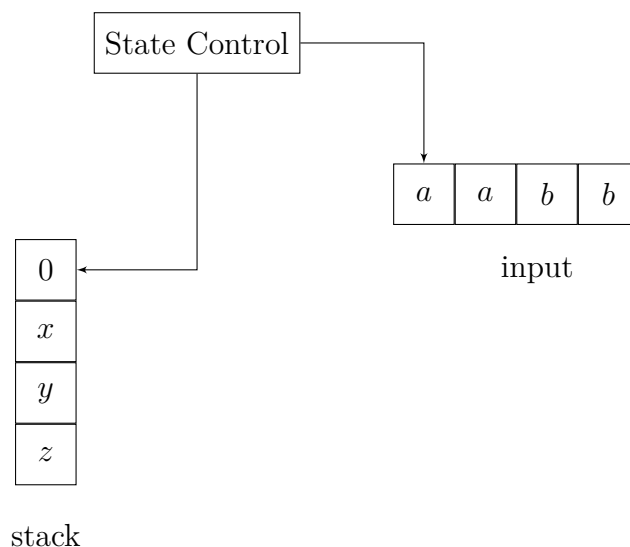


图 2.3: PDA

PDA 由一个六元组  $(Q, \Sigma, \Gamma, \delta, q_0, F)$  表示：

- $Q$ : 状态集合
- $\Sigma$ : 输入字母表
- $\Gamma$ : 栈字母表
- $\delta$ : 状态转移函数
- $q_0$ : 初始状态
- $F$ : 终结状态集合

例如状态转移函数  $\delta(q_1, a, b) = \{(q_2, \epsilon)\}$  表示，在状态  $q_1$  时，如果输入字符为  $a$ ，并且栈顶元素为  $b$ ，那么就将  $a$  消耗掉，并将  $b$  出栈，进入状态  $q_2$ 。在 PDA 中



可表示为  $a, b \rightarrow \epsilon$ 。

例如状态转移函数  $\delta(q_3, \epsilon, b) = \{(q_4, a), (q_5, b)\}$  表示，在状态  $q_3$  时，如果输入字符为空，并且栈顶元素为  $b$ ，那么有两种选择：

1. 使用  $a$  代替栈顶元素  $b$ ，并进入状态  $q_4$ 。
2. 栈保持原样 ( $b$  为栈顶)，并进入状态  $q_5$ 。

能够识别  $\{a^n b^n \mid n \geq 0\}$  的 PDA 如下：

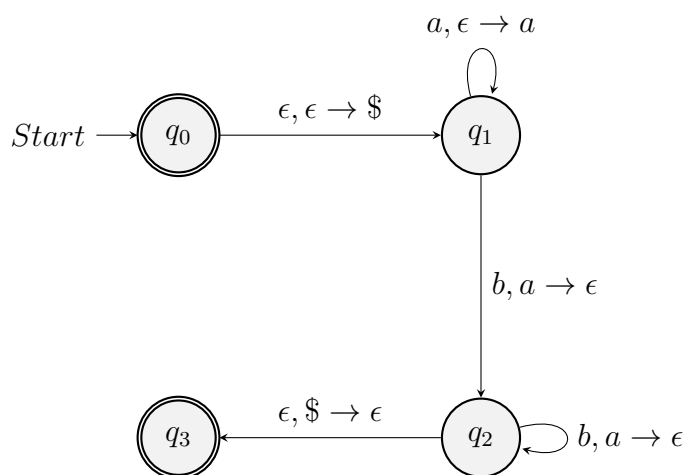
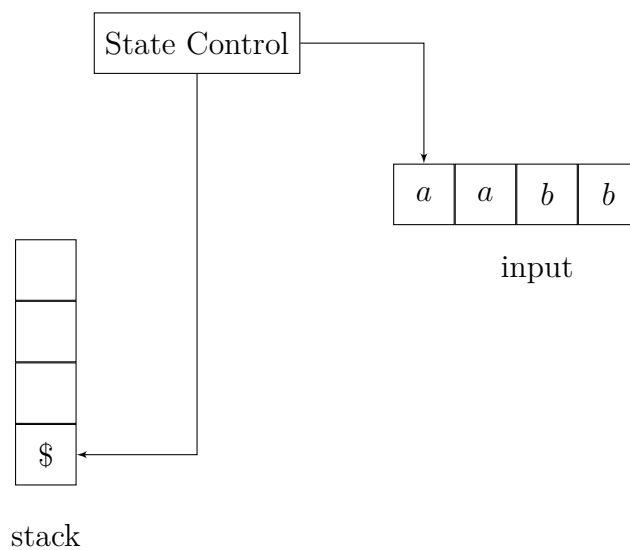
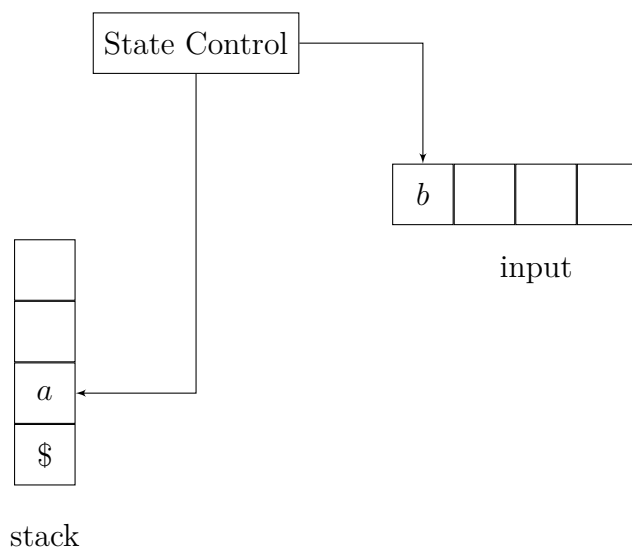
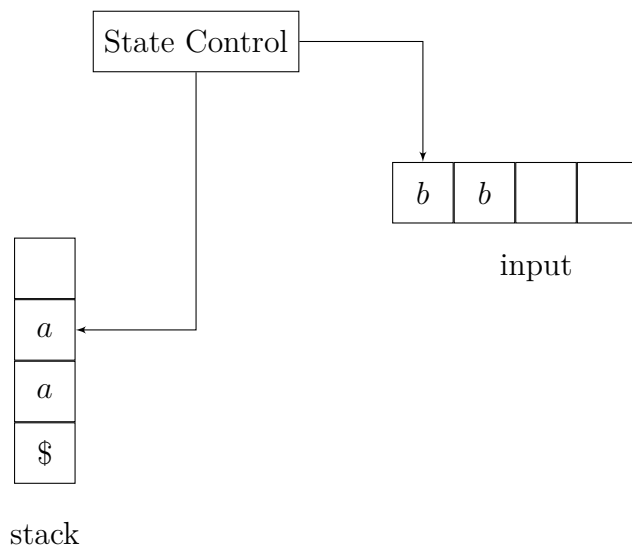
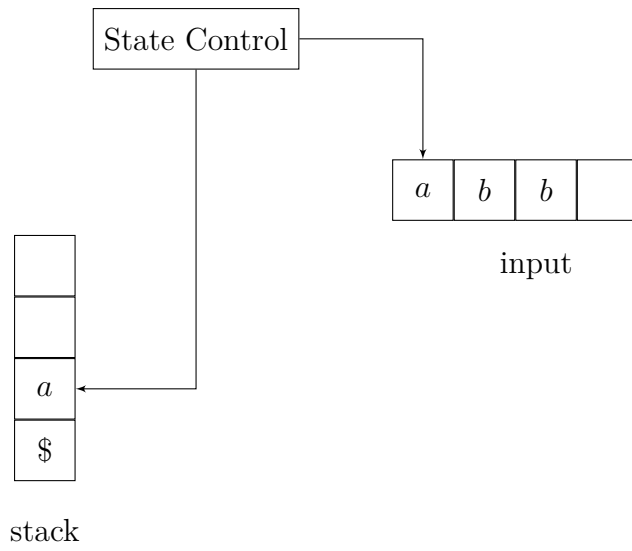
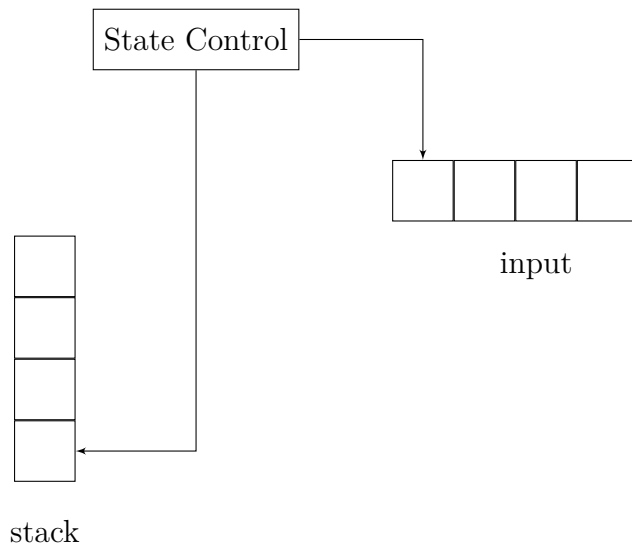
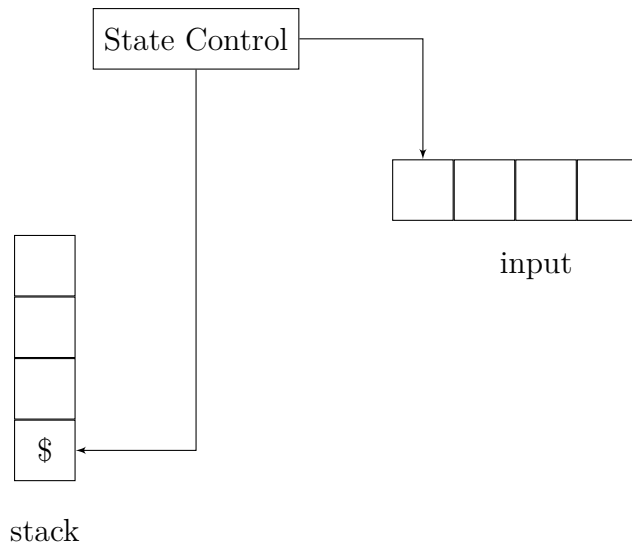


图 2.4: 识别  $\{a^n b^n \mid n \geq 0\}$  的 PDA

识别串  $aabb$  的过程如下：







## 2.4 预测分析法

### 2.4.1 LL(1) 文法

例如产生式  $A \rightarrow +T \mid -P$ ，当读到  $+$  开头的串的时候，可以很直接地判断选择  $A \rightarrow +T$  这个生成式；而读到  $-$  开头的串的时候，可以直接判断选择  $A \rightarrow -P$  这个生成式。

像这种根据第一个 token 就能得出选择哪个生成式的情况，就叫做预测分析法。

LL(1) 文法表示只查看后面 1 个符号，来判断需要选择的生成式。LL(k) 文法则表示根据后 k 个符号来决定生成式。

但是，如果文法是类似于  $A \rightarrow T \mid P$  这样都以非终结符开头的话，一眼就很难判断。因此就需要知道， $T$  是如何展开的。如果  $T \rightarrow a \mid b$ ， $P \rightarrow c \mid d$ ，那当串以  $a$  或  $b$  开头时，显然需要选择  $A \rightarrow T$ ；而当串以  $c$  或  $d$  开头时，就应该选择  $A \rightarrow P$  这个生成式。

### 2.4.2 FIRST

为了能够预测下一个生成式，就需要知道每个生成式能够产生的开始符号的集合，称为 FIRST 集合。 $FIRST(\alpha)$  是一个记录所有能够由  $\alpha$  推导出的出现在开头的终结字符的集合。

例如：

$$E \rightarrow TE'$$

$$E' \rightarrow +TE' \mid \epsilon$$

$$T \rightarrow FT'$$

$$T' \rightarrow *FT' \mid \epsilon$$

$$F \rightarrow (E) \mid id$$

每个终结符的  $FIRST$  集合都是自己本身。

$$FIRST(id) = \{id\}$$

$$FIRST(*) = \{*\}$$

$$FIRST(+) = \{+\}$$

$$FIRST(( ) = \{( \}$$

$$FIRST( ) ) = \{ ) \}$$

如果  $E \rightarrow T$ , 则应该把  $FIRST(T)$  也加入到  $FIRST(E)$  中。

$$FIRST(E') = \{+, \epsilon\}$$

$$FIRST(T') = \{*, \epsilon\}$$

$$FIRST(F) = \{(, id\}$$

$$FIRST(T) = FIRST(F) = \{(, id\}$$

$$FIRST(E) = FIRST(T) = \{(, id\}$$

### 2.4.3 FOLLOW

仅有  $FIRST$  集合还不够, 例如:

$$A \rightarrow Tb \mid P$$

$$T \rightarrow \epsilon \mid a$$

$$P \rightarrow c$$

可以得出:

$$FIRST(T) = \{\epsilon, a\}$$

$$FIRST(P) = \{c\}$$

当遇到  $a$  开头的串时, 应该选择  $A \rightarrow Tb$ ; 当遇到  $c$  开头的串时, 应该选择  $A \rightarrow P$ 。

但其实, 由于  $\epsilon$  在  $FIRST(T)$  中, 所以当遇到  $b$  开头的串时, 也应该选择  $A \rightarrow Tb$ 。

所以，为了特殊处理当一个非终结字符可以推出  $\epsilon$  的情况，就需要知道它后面紧跟的是什么终结字符，这样的集合被称为 FOLLOW 集合。

FOLLOW 集合的规则：

1. 当  $S$  为开始符号时，将结束标记  $\$$  添加到  $FOLLOW(S)$  中。
2. 如果存在  $A \rightarrow \alpha B \beta$ , 那么  $FIRST(B)$  中除了  $\epsilon$  外所有符号都在  $FOLLOW(B)$  中。
3. 如果存在  $A \rightarrow \alpha B$  或  $A \rightarrow \alpha B \beta$  且  $FIRST(B)$  中包含  $\epsilon$ , 那么  $FOLLOW(A)$  中的所有符号都在  $FOLLOW(B)$  中。