目录

第一章　DFA 和 NFA

1.1　DFA

1.1.1　What is DFA

什么是自动机? 简单来说, 自动机有很多状态, 在这些状态下面, 自动机会接收一些输入, 接收了一些输入之后, 状态将会发生变化. 比如说我们有一个最简单的自动机: 开关. 假设我们有一个开关, 默认是关着的, 就是说其状态是**关的**. 随后我们输入 **On**, 接收了输入之后, 其状态发生变化, 状态变为**开的**. 类似的, 后者状态下, 接受了输入 **Off**, 状态再次变为 **关的**.

这就是一个简单的状态机.

DFA 的定义

A DFA is a five tuple: $\{ Q, \Sigma, \delta, q_0, F \}$. Here we list the definition of the things above.

1. Q is the set of the states.
2. Σ is the alphabet.
3. δ is transtion function
4. q_0 is in the Q. He is the **initial** state.
5. F is the set of the final states (you may call them the accepting states).

Why we need the seemed so complex definition? Ok, my friend, it is not complex. What a sad story. Anyway, the definition tells all the information we need so far.

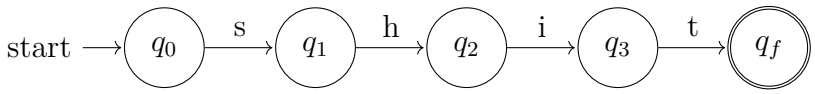

图 1.1: 一个接收 shit 的 DFA

Now lets see a simple example to check for what is an automaton. (Oh, right, do you know that the single form of the word 'automata' is actually 'automaton'. Cool.)

An Example of DFA

我们有一个简单的 DFA, 他能够识别字符串 'shit'. 这个 DFA, 用图 1.1[1]表示. 这里涉及到了 DFA 的表示方法: 我们有两种方法来表示一个 DFA.

1. 使用 diagram
2. 使用 δ 函数的表格表示.

来看我们这个图, start 表示的是初始状态, 随后, 是箭头, 箭头上面的字符是便是说, "如果接受到了这个字符, 那么状态转移到…" 的这个意思. 被双线包围的状态便是 F 之中的状态. 但是这个图还是有问题, 如果说我们在一开始接收到了一个不是 s 的字符呢? 这个时候状态将会走到哪里? 我们这个时候说, 这 DFA 直接 die 了, 他将处于一个死掉的状态, 永远不能够从其中出来, 要构造这个状态也是简单的, 我们只需要说, 在这个状态之下, 无论接收什么字符, 其都将保持在当前这个状态. 死得很彻底.

这里是一个简单化地表示, 我们尽量在不引入歧义得情况下, 进行图的设计. Ok, 让我们继续.

我们来看看这个 DFA, 其是能接受字符串, 这些字符串是由 $0, 1$ 组成的, 并且, 对于这些字符串, 存在一个子字符串, 其为 01. 是不是看起来很简单? 为了构建这个 DFA, 我们需要问出几个问题.

1. 如果已经接到了 01, 无论后续是什么字符, 我们接收.
2. 如果没有接到 01, 且上一个字符为 0, 那么收到 1 的话, 我们就看到了 01, 也就是转到 (1).

[1]Hey, You can import the tikz library named automata to draw automata in a simplest way.

3. 如果没有接到 01, 且上一个字符为 1, 那么收到 0 的话, 我们就得转到 (2).

实际上我们可以看出这里有需要有三种状态, 分别对应上面三种情况. 基本上, 构建一个 DFA 就是一个劲地分类讨论, 对于每一种类, 都赋予一个状态.

An eXtension function of δ

Anyway, let us go on. 我们使用一个 extention function of δ 来更好的表示一个 DFA 能够接收什么东西. 下面这个定义方法是递归的.

设我们能够诱导出一个函数 $\hat{\delta}$: $Q \times \Sigma^*$ 也就是说, 其表示, 状态 q 之下, 接受了字符串 w 之后, 其所在的状态.

Basic: $|w| = 1$, let $w = a$.

$$\hat{\delta}(q, w) = \delta(q, a)$$

Induction: if $|w| \geq 1$, let $w = xa$, where a is a symbol:

$$\hat{\delta}(q, w) = \hat{\delta}(q, xa) = \delta\big(\hat{\delta}(q, x), a\big)$$

上面就是 $\hat{\delta}$ 的定义方法, 我们需要证明, $\hat{\delta}$ 确实满足其定义:

证明: 若在状态 q 之下, 接受了 w, 则其状态变为 $\hat{\delta}(q, w)$.

证明是简单的, 需要对 w 的长度上用 induction. 好了, 证明之后呢? 我们为什么需要 $\hat{\delta}$? 仅仅是为了表示方便而已.

Example 1.1.1. 我们设计一个 DFA A, A 满足

$$L(A) = \{\, w \mid w \text{ 有偶数个 1's, 有偶数个 0's} \,\}$$

构造是简单的, 但是图画起来是麻烦的. 但是我还是画出来了, 见图 1.2. 读者可以自行验证其正确性.

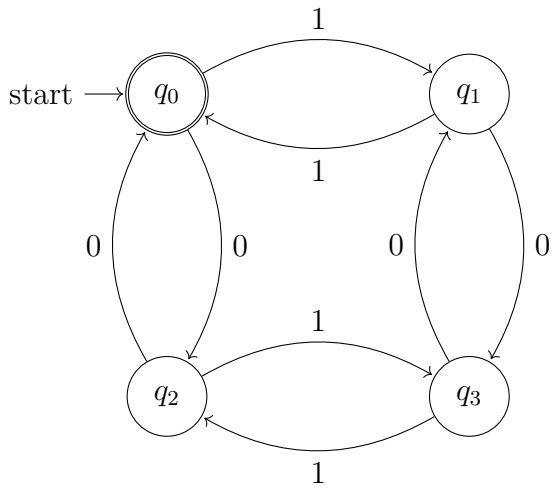

图 1.2: 接收偶数个 1, 偶数个 0 的 string 的 DFA

1.1.2 The language of a DFA

DFA 的 language 指的是其接收的 string 的全体. 那么,

$$L(A) = \{\, w \mid \hat{\delta}(q_0, w) \in F \,\}$$

其中 A 是一个 DFA. 其实本节并没有什么知识点, 建议查看书本上的对应章节, 以查看其上记录的那些例子.

1.2 Nondeterministic Finite State Automata

1.2.1 The Definition of the NFA

Nondeterministic 的意思是说, 这个 automata 的所处于的状态并不是确定的, 其能够同时处于多个状态. 最初的时候处于 start state. 为什么之后会处于多个状态呢? 这是因为, 在某一个状态, 接收了一个字符, 其跳转到了多个状态. 比如说:

$$\delta(q, a) = \{\, q_1, q_2 \,\}$$

在状态 q 之下, 接受了 a 就跳转到了两个状态 q_1, q_2. 我们的 δ 函数的定义就发生了改变:

$$\delta : Q \times \Sigma \to P(Q)$$

其中 $P(Q)$ 是 Q 的幂集. 随后, 我们温习一下 DFA 的定义:

Definition 1.2.1 (DFA)**.** DFA 是一个 five–tuple:

$$A = (Q, \Sigma, \delta, q_0, F)$$

其中 Q, Σ, δ, q_0, F 的定义都是我们熟知的. NFA 也是如此.

$$N = (Q, \Sigma, \delta, q_0, F)$$

只不过其中 δ 的定义稍有不同.

1.2.2 eXtended function of δ

和 DFA 类似的, 我们也有 $\hat{\delta}$ 的定义:

Basic: $\hat{\delta}(q, a) = \delta(q, a)$

Induction: let $w = xa$, $\hat{\delta}(q_0, x) = \{ p_1, \cdots, p_k \}$

$$\hat{\delta}(q_0, w) = \hat{\delta}(\{ p_1, \ldots, p_k \}, a) = \bigcup_{i=1}^{k} \delta(p_i, a)$$

是的, 没什么, 但是等到我们引入了 ϵ-transition 的时候, 我们还会对 $\hat{\delta}$ 进行修正.

有了 $\hat{\delta}$ 之后, 我们能够定义 NFA 的 language. 设 N 是一个 NFA, 那么我们有

$$L(N) = \{ w \mid \hat{\delta}(q_0, w) \cap F \neq \varnothing \}$$

1.2.3 The Equivalence of NFA and DFA

我们将证明 DFA 和 NFA 实际上是等价的, 那么为什么需要使用 NFA 呢? 其实很简单, 就是 NFA 能够将很多表述简化, 某些问题用 NFA 写出来就是要简单一些, 尽管说 NFA 和 DFA 的能力是一样的.

Theorem 1.2.2 (DFA = NFA)**.** DFA 和 NFA 之间是等价的.

两者之间的等价性是这么描述的: 对于一个 DFA A, 存在一个 NFA N, 使得 $L(A) = L(N)$; 并且对于一个 NFA N, 存在一个 DFA A, 使得 $L(N) = L(A)$. 于是说, 我们要证明这个等价性就需要进行两方面的证明:

1. 给定一个 DFA A, 构造一个 NFA N, 满足 $L(A) = L(N)$

2. 给定一个 NFA N, 构造一个 DFA A, 满足 $L(N) = L(A)$.

其中 (1) 的构造是显然的, 因为很简单, DFA 可以视为一个特殊的 NFA. 对于 (2) 的构造, 我们使用的子集构造法: 将 $Q' = P(Q)$, 于是我们就能够将 "多个状态" 视为 "单个状态". 比如说 $\{p_1, p_2, \ldots, p_k\}$ 写为 $q_{1\ldots k}$. 因为 $P(Q)$ 也是有限的, 所有从这一个角度出发, NFA 和 DFA 其实没有什么区别.

Proof. (1) 的构造是显然的.
(2) 的构造也是简单的. 我们将 $\{p_1, \ldots, p_k\}$ 写为 $q_{1\ldots k}$. 若是 $\bigcup_{i=1}^{k} \delta(p_i, a) = \{p_{a_1}, \ldots, p_{a_m}\}$, 那么有 $\delta'(q_{1\ldots k}, a) = q_{a_1 \ldots a_m}$. 或者说你不想引入 $q_{1\ldots k}$ 的符号, 因为反而会看着挺麻烦, 那么你直接说 $\delta'(\{p_1, \ldots p_k\}, a) = \{p_{a_1}, \ldots, p_{a_m}\}$ 也是可以的. □

第二章　正则语言和正则表达式

2.1　正则表达式的定义

正则表达式　的递归定义.

Definition 2.1.1. 分为基础部分, 归纳部分. 基础部分:

1 空集是一个正则表达式, 匹配空语言.

2 ϵ 是一个正则表达式, 匹配 $\{\epsilon\}$

3 对于任意一个符号 a , 其也是一个正则表达式.

归纳部分: E_1, E_2 都是正则表达式, 那么 $E_1 + E_2$ 也是正则表达式, $+$ 表示或; E_1E_2 也是正则表达式; E_1^*, E_2^* 也是正则表达式; 括号用来表示运算的顺序.

2.1.1　语言之间的运算

我们给定了语言之间的运算, 实际上就是集合之间的运算, 我们根据这个规定, 能够对正则表达式进行运算.

语言之间的运算　给定两个语言 L, M, 语言是字符串的集合. 语言之间有乘法, 幂, 闭包等运算. 注意到乘法仅仅是一个称呼.

Definition 2.1.2 (运算)**.** 下面列出四种运算.

$$L \cdot M \equiv \{w \mid w = xy, x \in L, y \in M\}$$
$$L + M \equiv L \cup M$$
$$L^2 = L \cdot L \text{ 特别地}, L^0 = \{\epsilon\}$$
$$L^* = \bigcup_{i=0}^{\infty} L^i$$

运算和运算的优先级 对于三种运算, 我们可以将闭包看为是一个幂, 那么这些的运算顺序就如同我们平常的运算一样: 1. 考虑幂; 2. 考虑乘积; 3. 考虑加法.

Example 2.1.3. 下面正则表达式的语言是什么?

$$1 + 01^* \tag{2.1}$$

Example 2.1.4.

$$(a + b)^*(a + bb) \tag{2.2}$$

a, b 组成的, 以 a 或者 bb 结尾的字符串.

2.2 正则表达式的性质

2.2.1 正则表达式和 DFA 的等价性: 根据 DFA 构造正则表达式

正如我们前面已经了解到的那样, 正则表达式和 DFA 的描述能力实际上是一样的, 也就是说 $L(E) = L(D)$, 我们将证明这种等价性. 我们分为两个方向, 给定一个正则表达式, 构造一个 DFA , 和给定一个 DFA 构造一个正则表达式. 有两种方法: 1. 递归法; 2. 状态消除法. 我们接下来就讲讲这两种方法.

递归法 递归法是一种算法, 其实是动态规划的思想. 给 DFA 进行编号 (从 1 开始编号, 注意这点), 定义 $R_{i,j}^k$, 其表示的是, 由 i 到 j 的, 中间经过的状态的编号不超过 k 的, 一个字符串的集合.

考虑其初始状态, $k = 0$ 的时候, 分 $i = j, i \neq j$ 讨论. 对于 $i = j$ 的时候, $R_{i,j}^k$ 为 i 到 i 状态的字符的闭包. 对于 $i \neq j$, 则, 考虑状态 i 到 j 的字符.

其后, 开始遍历, 对于 $k > 1$, 我们有

$$R_{i,j}^{(k)} = R_{i,j}^{(k-1)} + R_{i,k}^{(k-1)}(R_{k,k}^{(k-1)})^* R_{k,j}^{(k-1)} \tag{2.3}$$

其中 $R_{i,j}^{(k-1)}$ 是指不经过 k 的路径[1]. 我们能够看出, 这个递归方法是眼熟的, 这不是 tm 的动态规划吗? 随后就好理解了. 甚至说, 我们能够写出对应的代码.

故, 我们和 DFA 等价的正则表达式是

$$\sum_{j \in F} R_{1,j}^{(n)} \tag{2.4}$$

Example 2.2.1. 见 ppt , 流程还是比较麻烦的.

状态消除法 我超, 这个有点难搞, 不会画图, 反正也不是很难, 要不就直接看 ppt 得了. 根据几个规则在图上化简, 看起来是这个学校喜欢的东西.

[1]路径实际上是一个字符串!

2.2.2 正则表达式和 DFA 的等价性: 根据正则表达式构造 DFA

我们根据正则表达式的定义出发, 可以对正则表达式构造规则, 建立起 DFA 的构造规则. 具体的符号还是见 ppt, 这里画不了图.

Example 2.2.2. 将 $0(0+1)^*$ 转化为 ϵ – NFA.

2.3 正则表达式的代数性质

比较简单, 自己看吧!

第三章　正则语言的性质

我们这边的题型是如何判定一个语言不是正则的. 既然他这么问了, 那么他大概率不是正则的.

3.1　泵引理和泵引理的证明

所用到的方法便是下面介绍的泵引理. 需要注意到, 符合泵引理的语言不一定是正则的, 但不满足的一定不是正则的, 也就是说, 泵引理实际上是正则的必要条件.

鸽巢原理　原理应该很简单, 之前在其他地方也有涉及[1], 其实是一个挺有意思的定理.

泵引理　泵的意思是, 正则语言之中的有些语句之中某些部分, 能够被泵出来. 若是 $w = xyz$ 是满足这个条件的, 就有 y 可以被泵出来, 也就是 $w^i = xy^iz$, $i = 1, 2, \ldots$, 也有 $w \in L$. 所以形象地称呼其为泵引理.

Theorem 3.1.1. 对于正则语言 L, 存在 N 使得对于长度大于等于 N 的语句, i.e. $w \mid |w| \geq N$, 可以分为三个部分 xyz, i.e. $w = xyz$, 并且满足:

1. $y \neq \epsilon$, 这是说泵出的部分不能为空

2. $|xy| \leq N$, 这描述了 y 的位置.

3. $xy^*z \in L$, 这是说 y 可以被泵.

[1]比如说, 线性代数之中有一题: 证明对于一个 n 阶方阵, 存在 $k \leq n$ 使得 $r(A^k) = r(A^{k+1}) = r(A^{k+2}) = \ldots$

证明 证明需要用到前面的定理.

考虑这个语言对应的 DFA. 并且给定了 DFA 的状态编号. 并且设 N 为状态的数目.

设 $w = a_1a_2a_3\dots a_m$, $m \geq N$, $q_i = \hat{\delta}(q_0, a_1\dots a_i)$. q_i 代表的是, DFA 接收到 w 的前 i 字母的时候所处于的状态. 这个证明的重点在于: 使用鸽巢原理知道, 至少存在一对 i, j, $i \neq j$, $0 \leq i, j \leq N$, 使得 $q_i = q_j$[2].

我们设 (q_n, q_m) 代表状态 q_n 到 q_m 需要的字符串集合, viz., $w \in (q_n, q_m) \iff \hat{\delta}(q_n, w) = q_m$, 那么我们有

$$(q_n, q_m) = (q_n, q_n)^*(q_n, q_m)$$

那么我们知道 w 之中 i, j 之间的字符串 $w_{i,j} \in (q_i, q_i)$, 那么

$$w^*_{i+1,j}w_{j+1,m} \in (q_i, q_m)$$

自然有 $w_{1,i}w^*_{i+1,j}w_{j+1,m} \in L = (q_0, q_m)$. 于是自然地, $|w_{i+1,j}| \neq 0$, 且 $|w_{1,j}| \leq N$.

通过泵引理证明某些语言并不是正则的. 我们称 w 是满足泵引理的, 如果存在对 w 的划分 xyz, 都有 $xy^*z \in L$. 那么我们可以重述泵引理.

$$L \text{ is regular} \implies \exists N \in \mathbb{N} \forall w \in L\ , |w| \geq N(w \text{ 满足泵引理})$$

取其逆否命题, 就有

$$\neg(\exists N \in \mathbb{N} \forall w \in L\ , |w| \geq N(w \text{ 满足泵引理})) \implies L \text{ is not regular}$$

但是我们有

$$\begin{aligned}&\neg(\exists N \in \mathbb{N} \forall w \in L\ , |w| \geq N(w \text{ 满足泵引理}))\\ =&\forall N \in \mathbb{N}, \exists w \in L(\forall x, y, z(xyz = w \to (\exists i \in N, xy^iz \notin L)))\end{aligned}$$

如果说我们证明了这个命题, 那么我们就能够证明 L 不是正则的.

Example 3.1.2. 证明 $L_{01} = \{0^n1^n \mid n \geq 0\}$ 不是正则的.

Example 3.1.3. 证明 $L = \{0^i1^j \mid i > j\}$ 不是正则的.

Example 3.1.4. 证明 $L = \{a^{n!} \mid n \geq 0\}$ 不是正则的.

[2]当 $i = N$ 的时候, 经过的状态已经达到了 $N + 1$ 个.

3.2 运算的封闭性

正则语言再某些运算之后得到的新语言仍然是正则的. 称正则语言再这些运算下封闭.

并补交差反转同态逆同态

3.3 判定正则语言

三个判定问题:

w 是否属于描述的语言

语言是否为空是否为无穷的.

语言的等价性问题.

对于非空的, 检查全部长度小于 n 串.

是否为无穷, 检查全部长度有 n 到 $2n-1$ 的串.

3.4 DFA 的最小化

最小化

将冗余的状态消除

$\hat{\delta}(p,w) \in F \iff \hat{\delta}(q,w) \in F$

不等价

对于两个状态 p,q , 存在一个 w 使得, $\hat{\delta}(p,w), \hat{\delta}(q,w)$ 其中一个属于 F, 而另一个不是.

3.4.1 填表算法:

1. 基础: $p \in F, q \notin F$ then p,q 不等价.

2. 对于 $a \in \Sigma$ if $r = \delta(p,a), s = \delta(q,a)$ 不等价, 那么 p,q 不等价.

和那个数字逻辑里面的差不多. 但是严谨一点.

我们这里有四个步骤.

1 首先考虑基础, 初始化

2 其次考虑经过 0 到达终态和非终态的状态对. 就是说, 将所有状态分为三类: 1. 输入为 0 不发生状态转移的; 2. 输入为 0 到达终态的; 3. 不能到达终态的. 考虑后面两类, $\{D, F\} \times \{A, B, E, G, H\}$, 其中 $\times$ 是一个直积.

3 类似的, 考虑经过 1 到达终态和非终态的状态对.

4 最后进行对所有空余进行验证, 卧槽, 真几把麻烦.

Remark 3.4.1. 对于状态对, 比如说 (A, E), 考虑 $(\delta(A, 0), \delta(E, 0) = (B, H))$ 这是说, (A, E) 不等价取决于 (B, H) 是否等价.

考虑 (B, H) , $(\delta(B, 0), \delta(H, 0)) = (G, G)$, $(\delta(B, 1), \delta(H, 1)) = (C, C)$ 终态相等, 也是等价的标志. 那么 B, H 等价.

于是 A, E 也是等价的.

我们进行总结, 如果说有两对, 或者以上的状态对, 其等价性相互依赖, 那么可以断言这些状态对等价. □

3.5 Regular Expressions in Unix

第四章　上下文无关文法

Contents:

4.1　上下文无关文法

例 3.1.2 告诉我们 $L = \{ 0^n 1^n \mid n \in \mathbb{N} \}$ 并不是一个正则语言. 但其为一个**上下文无关语言**. 并且很有意思的是 $L = \{ w \mid w$ 是正则表达式 $\}$ 并不是一个正则语言, 也就是说, 正则语言, 并不能描述所有的正则语言.

Example 4.1.1 (回文)**.** if $w \in \Sigma^*, w = w^R$, then w 为回文

Example 4.1.2 (回文语言)**.** if $L = \{w \in \Sigma^* \mid w = w^R\}$ then L 是回文语言.

我们可以使用递归方法定义回文语言, 正如我们定义一阶逻辑之中的命题一样:

1. 首先 $\epsilon, 0, 1$ 都是回文.

2. if w 是回文, 那么 $0w0, 1w1$ 都是回文.

Definition 4.1.3 (文法)**.** 文法 G 是一个 $4p$ 结构, $G = (V, T, P, S)$. 其中 V for variable, T for terminator, P for Production, S for start.

1. $P \rightarrow \epsilon$
2. $P \rightarrow 0$
3. $P \rightarrow 1$
4. $P \rightarrow 0P0$
5. $P \rightarrow 1P1$

图 4.1: A context free grammar for palindromes

1 V 是变量的集合.

2 T 称为 terminators.

3 P 是生产规则, 形式为 $A \rightarrow \alpha \mid \beta$, 其中 $\alpha, \beta \in (V \cup T)^*$

4 $S \in V$, 表示的是初始变量.

Example 4.1.4 $(0,1$ 组成的回文$)$**.** $G = (V, T, P, S)$, 其中 $V = \{A\}$, $T = \{0, 1\}$, $P = A \rightarrow \epsilon \mid 0 \mid 1 \mid 0A0 \mid 1A1$, $S = A$

Example 4.1.5 (0^n1^n)**.** $L = \{0^n1^n \mid n \in \mathbb{N}\}$ 和上面一个完全类似, 这里就懒得说了

Example 4.1.6. 算术表达式, 比如说 $a + b$, $a * b + a$, 或者是有 0 或者 1 的式子, 比如说 $1 + a * 0$ 等等. 这些式子的语言就不是正则的. 在文法的构造之中用到了两个变量 E, I. 其中 E for Expression; I.

4.1.1 归约和派生

inference 规约
derivation 派生
多步派生
左派生和右派生

Inference is the construction of a string using productions of the grammar. It is a construction **from body to head**. Which is to say that we are given some strings that are known to be the string in the grammar, and then use them to construct another string that is in the grammar.

Meanwhile, derivation is the construction **from head to body**, from the initial variable, we use productions in the grammar to substitute the variable with strings in $(V \cup T)^*$.

Definition 4.1.7 (派生)**.** $\alpha, \beta, \gamma \in (V \cup T)^*$, if $A \to \gamma$, then $\alpha A \beta \underset{G}{\Rightarrow} \alpha\gamma\beta$.

Definition 4.1.8 (文法派生的句型)**.** $\alpha \in (V \cup T)^*$, α 是变元可能派生出的语句. 比如说 $A \to \alpha$, 那么 α 就是一个句型, 而后如果 α 继续派生的得到了 β, 那么 β 也是一个句型.

Definition 4.1.9 (归约)**.** 以算术表达式为例, 如果说 w 是 I, 那么 aw, bw, $0w$, $1w$ 也是 I, 如果说 w', w'' 是 E, 那么 $w' \cdot w''$, $w' + w''$, (w') 都是 E. 这种自底向上的构造法就是归约.

Example 4.1.10 (派生的例子)**.**

Definition 4.1.11 (多步派生)**.**

$$\alpha_1 \overset{i}{\underset{G}{\Rightarrow}} \alpha_{1+i} \tag{4.1}$$

对于一般的多步派生, 记为 $\overset{*}{\underset{G}{\Rightarrow}}$. 特别地, 还有零步派生 $\alpha \overset{*}{\Rightarrow} \alpha$.

Definition 4.1.12 (最左派生, 最右派生)**.** 对于一个句型 α, 我们考虑只对最左 (右) 的变量进行派生, 这就是最左 (右) 派生. 记为 $\underset{lm}{\Rightarrow}$ ($\underset{rm}{\Rightarrow}$). 为了取消歧义.

Theorem 4.1.13 (派生的**等价性**)**.** 对于任意一个派生, 都存在一个最左 (右) 派生与其对应

4.1.2 文法的语言

$L = \{ w \mid 0, 1 \text{ 数量相等} \}$
$L = \{ a^n, b^n \mid n \in \mathbb{N} \}$

Definition 4.1.14 (语言)**.**

$$L(G) = \{w \mid w \in T^*, S \overset{*}{\underset{G}{\Rightarrow}} w\} \tag{4.2}$$

Remark 4.1.15 (为什么称为是上下文无关文法). 我们参考 $\alpha A \beta \underset{G}{\Rightarrow} \alpha\gamma\beta$, 这个派生是否成立, 是跟 α, β 无关的, 于是称为上下文无关文法.

Definition 4.1.16 (等价性)**.** G_1, G_2 满足 $L(G_1) = L(G_2)$, 则 G_1 等价于 G_2

Definition 4.1.17 (句型, 左句型, 右句型)**.** G 生成的句型定义如下:

$$\{w \mid w \in (V \cup T)^*, S \overset{*}{\underset{G}{\Rightarrow}} w\} \tag{4.3}$$

也定义了左句型, 右句型.

Example 4.1.18. $L = \{a^n b^n \mid n \geq 1\}$

Example 4.1.19. $G = \{\}$

Example 4.1.20. $L = \{w \mid 0,1$ 数量相等$\}$, G $= (V, T, P, S)$, 其中 $V = \{S\}, T = \{0, 1\}$, P 是下面这个:

$$S \to 0S1S \mid 1S0S \mid \epsilon$$

如何证明其对应的语言确实是 L? 我们需要验证, 对于 L 之中的任意一个字符串 w 确实能够如此划分, 使得 $w = 0w'1w''$, 其中 w', w'' 都是 L 的. 使用分类讨论即可证明这一点.

Example 4.1.21 (算术表达式)**.**

4.2 语法分析树

sparse tree 分析树
Pump Lemma 泵引理
Sparse Tree 和 inference 和 derivation 之间的等价性

语法分析树是表示派生过程的一个树. 此后在 Context free language 的泵引理的证明之中要用到.

Definition 4.2.1 (分析树)**.** 定义为

1 每个内节点是边缘符号

2 叶子节点是 $V \cup T \cup \{\epsilon\}$ 之中的符号

3 如果说内节点的标记是 A, 那么其子节点从左到右分别为

$$X_1, X_2, \ldots, X_n$$

那么 $A \to X_1 X_2 \ldots X_n$ 是一个产生式. □

如果说, 根节点是初始符号 s, 叶子节点是终结符, 那么该树是完成的, 该树的产物属于 $L(G)$.

Definition 4.2.2 (subtree)**.** 在树 T 之中以内节点 A 为根节点的 subtree 为 T 的 subtree

Theorem 4.2.3 (the equivalence of tree)**.** 一棵树和 $L(G)$ 之中的一个语句等价.

1. G 存在 A 为根节点的树, 其产物为 α, then $A \overset{*}{\Rightarrow} \alpha$
2. 如果说 $A \overset{*}{\Rightarrow} \alpha$ then 存在 A 为根的树, 其产物为 α

证明. □

1. $P \rightarrow \epsilon$
2. $P \rightarrow 0$
3. $P \rightarrow 1$
4. $P \rightarrow 0P0$
5. $P \rightarrow 1P1$

图 4.2: A context free grammar for palindromes

4.3 文法和语言的歧义性

歧义性
歧义性和算法

算术表达式的语言中 L_{exp}, 比如说 $w = 1 * 1 + 1$, 有两个生成树. 此为**歧义**.

Example 4.3.1 (表示加法乘法的优先级)**.** Fig 4.2 展示了 G_{exp} 的产生式, G_{exp} 是歧义的, 因为存在一个语句有两个 Sparse Tree.

Definition 4.3.2 (固有歧义)**.** 对于一些语言, 其对应的文法都是有歧义的.

Example 4.3.3. 对于 $L = \{a^i b^j c^k \mid i = j \text{ or } j = k\}$, L 是固有歧义的.

需要知道的是, 固有歧义的证明非常繁琐, 甚至很难理解. 并且, 并不存在算法能够验证任意一个语言是固有歧义的.

4.4 文法的化简和设计

4.4.1 化简

Useless
ϵ-productions
unit productions

1 消除无用符号

2 消除 ϵ 产生式 (ϵ production)

3 消除单元产生式 (unit production) :$A \rightarrow B$

无用符号的消除

我们将一些无用的符号进行化简, 分为两类, 一类是不可达的, 一类是不可产生的, 这两种符号都是没有用的, 这是说, 对于一个语句的派生过程, 这些符号是不会出现的. 所以说是无用符号. 下面给出定义.

Definition 4.4.1 (Useful)**.** 对于 $G = (V, T, P, S)$, 符号 $X \in (V \cup T)$.

1. 如果 $G \overset{*}{\Rightarrow} \alpha X \beta$ then X is **reachable**.
2. if $\alpha X \beta \overset{*}{\Rightarrow} w \in T^*$ then X is **generating**.
3. If X is reachable and generating, then X is **useful**. X is useless otherwise.

Remark 4.4.2. T is generating, while S is reachable. You can easily prove it.

Example 4.4.3. 对于 $P = \{S \to AB \mid a, A \to b\}$ 可达的符号机和产生的符号分别为什么?

证明. $T = \{a, b\}, A, S$ 是产生的, S, A, B, a, b 是可达的, □

Remark 4.4.4. 能够看出, 我们要找到产生的, 则需要**从后往前找**, 若是要找到可达的, 则需要**从前往后找**.

ϵ 产生式

如果说 $A \Rightarrow \epsilon$, 称 A 是可空的. ϵ 产生式除了贡献 L 中的 ϵ 之外没有任何作用, 于是说, 当 L 之中没有 ϵ 的时候, 我们就可以将 G 之中的 ϵ 产生式消除掉. 这 (大概) 就是原理.

Definition 4.4.5 (可空变元)**.**

Example 4.4.6. 消除 $G = (\{S, A, B\}, \{a, b\}, P, S)$ 的 ϵ 产生式.

$$\begin{aligned} P = \{&S \to AB, \\ &A \to AaA \mid \epsilon \\ &B \to BbB \mid \epsilon\} \end{aligned} \tag{4.4}$$

Example 4.4.7. $G_2 = (\{S, A, B, C\}, \{a, b\}, P, S)$

$$\begin{aligned} P = \{&S \to ABC, \\ &A \to aA \mid \epsilon, \\ &B \to bB \mid \epsilon, \\ &C \to \epsilon\} \end{aligned} \tag{4.5}$$

单元产生式的消除

If $A \overset{*}{\Rightarrow} B$ then A, B are equivalent. You may view A, B as the same variable and merge the relative productions.

Example 4.4.8.

$$\begin{aligned} P = \{&S \to A \mid B \mid 0S1, \\ &A \to 0A \mid 0, \\ &B \to 1B \mid 1\} \end{aligned} \tag{4.6}$$

证明. It is easy to note that S, A, B are equivalent. Then $P = \{S \to 0S1 \mid 0S \mid 0 \mid 1S \mid 1\}$ □

化简步骤

1 消除 ϵ 产生式

2 reduce the unit productions

3 Reduce the not generating productions

4 Reduce the not reachable productions

第五章　两种范式和下推自动机的定义

5.1　两种范式

CNF
GNF

范式有两种范式, 一种是乔姆斯基范式 (CNF), 另一种是格雷巴赫范式 (GNF).

Definition 5.1.1 (CNF). 每一个不包含 ϵ 的 G 的产生式都能够写为下面的形式:

$$A \to BC \quad A \to a$$

其中 A, B, C 是变量, 然后 a 是终结符.

证明. 能够知道这个定理实际上是显然的. 步骤为:

1. 将产生式的 terminate 消除
2. 让产生式一次只多一个变量.

具体流程为, 给定一个产生式

$$A \to X_1X_2X_3 \dots X_m$$

其中 $X_i \in (V \cup T)$. 我们对其中的终结符进行这样的处理: X_i 换为 V_i, V_i 是新引入的变量. 且有产生式 $V_i \to X_i$. 这就将 $A \to X_1 \dots X_m$ 的终结符处理了.

我们将表达式 $A \to X_i \dots$ 写为 $A \to V_1 \dots V_m$, V_i 是变量. 我们再次引入变量 B_i, 将 $A \to V_1 \dots V_m$, 拆解为多个产生式, 就是 $A \to V_1B_1$, $B_1 \to V_2B_2$…于是, $A \to X_1X_2X_3 \dots X_m$ 就被被拆分为了多个产生式, 并且他们是属于乔姆斯基范式的. □

Example 5.1.2. 给定 CFG[1] G, P 为

$$S \to bA \mid aB, \quad A \to bAA \mid aS \mid a, \quad B \to aBB \mid bS \mid b$$

流程为

1. 对于产生式之中有终结符的, 进行变量的替换, 比如说 $A \to bA$ 我们引入新的变量 C_b, 将这个式子换为

$$A \to C_bA, \quad C_b \to b$$

2. 对于形如 $A \to B_1B_2 \dots B_m$ 的式子, 引入中间变量.

Example 5.1.3. 将 G 的产生式进行处理, 使得其为乔姆斯基范式.

$$S \to AS \mid BABC \quad A \to A1 \mid 0A1 \mid 01 \quad B \to 0B \mid 0 \quad C \to 1C \mid 1$$

Definition 5.1.4 (格雷巴赫范式)**.** 每个产生式的形式为 $A \to a\alpha$ 其中 a 是一个终结符, 然后 α 是变元的串, 也就是 $\alpha \in V^*$. 简写为 GNF.

Example 5.1.5. 将 $S \to AB, A \to AaA \mid bB \mid b, B \to b$ 转化为 GNF.

虽然说, CNF 的转化是不要求掌握的.

就尼玛讲完了!

5.2 下推自动机

Contents:

1. 接受的语言
2. 其和文法的等价性
3. 确定型下推自动机.

[1]CFG stands for context free grammar. 只是有点忘了

5.2.1 下推自动机的定义

Definition 5.2.1 (下推自动机)**.** 下推自动机 (PDA)[2]. 一个 PDA G 是一个七元组. PDA 相当于一个 NFA 加上一个栈. 进行每一次状态转移的时候, 会将栈顶元素弹出, 然后压入某些字符串. 初始阶段, 栈里面有一个初始元素 Z_0. 比如说我们接受了 0, 而后弹出 Z_0, 压入 $0Z_0$. 对于一个字符串, 压入顺序是**从后往前的**.

Example 5.2.2. 设计识别 $L_{01} = \{0^n 1^n \mid n \geq 1\}$ 的 PDA

Example 5.2.3. 设计识别 $L_{ww^R} = \{ww^R \mid w \in (0+1)^*\}$

[2]PDA for push down automata