计算理论讲义

CC98 九思

最后一次修改日期: 2022 年 11 月 30 日

目录

带有 * 的章节是凭我个人志趣额外介绍的.

引言

计算理论旨在研究一些基本的问题: 什么是计算? 什么是算法? 算法能解决所有问题吗? 如果不能, 那算法能解决什么样的问题呢? 我们首先会介绍正则语言与有限自动机, 上下文无关文法与下推自动机, 而我们的讨论重点会放在图灵机以及在该计算模型上定义出的可计算性, 不可判定性等定义上与讨论上.

本文以本校计算理论课程, 即教材《Elements of the Theory of Computation》为主, 本文的初衷是本校计算理论课程的梳理与总结, 因此读者也可以将本文看作自洽的**课程笔记**.

0 预备知识

在离散数学的课程中, 读者应当对朴素的集合 (set), 关系 (relation), 函数 (function) 等概念较为熟悉了, 因此我们直接罗列后文可能会用到 (其实很多用不到...) 的概念与技术:

- 朴素的集合: 定义, 集合间操作, 德摩根律, 简单的序数知识.
- 关系/二元关系: 自反性, 对称性/反对称性, 传递性, 闭包.
- 特殊的关系: 等价关系与集合的分划, 偏序关系与格.
- 证明技术: 鸽笼原理, 数学归纳法, 对角线法.

最后, 我们简单的引入一些下文要用到的定义与术语:

定义 0.1.

- 字母表 *(alphabet):* 有限的符号集合. 例如 $\sum_1 = \{0, 1\}$.
- 串 *(string):* 由符号组成的有限序列. 例如 01001. 串的连接: $\circ$.
- 空串: e, 使得 $we = ew = w, \forall w$.
- 语言 *(language):* 由串构成的集合.
- *Kleene* 闭包 *(Kleene star):* $L^* = L^0 \cup L^1 \ldots$, $\sum^*$ 即为由该字母表内符号组成的所有串的集合, 它显然是一个语言.

串和语言实质上都是在一个字母表下谈论的, 在不引起歧义的情况下我们可以不指明字母表. 如语言 $L = \{a^n b^n \mid n \geq 1\}$ 的符号表为 $\{a, b\}$.

注记 0.1. 注意空集 $\emptyset$, 空串 e, 空串构成的语言 $\{e\}$ 三者的区别.

1 正则语言与有限自动机

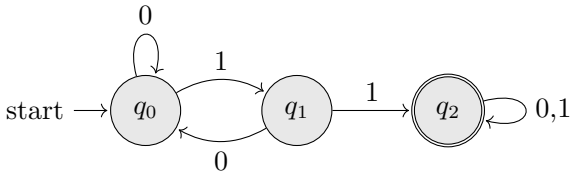
1.1 有限自动机

定义 1.1 (DFA)**.** 确定有限自动机 *(deterministic finite automata,* 简称 *DFA),* 由五元组 $(K, \sum, \delta, s, F)$ 构成*:*

- K 是由状态构成的有限集合.
- $\sum$ 是字母表/字符集.
- $s \in K$ 是起始状态.
- $F \subseteq K$ 是接受状态的集合.
- $\delta : K \times \sum \to K$ 为状态转移函数.

注记 1.1. 注意 δ 是输入一个状态和一个字符, 输出一个新状态的函数. 它不仅保证了每一个状态在每一个读取的字符下都会有一个新状态 *(*可能与原状态相同*),* 同时可保证了这个新状态是唯一的. 所谓确定性即指下一个状态被当前状态和输入**唯一确定**.

我们有时候会混用 "状态机", "有限自动机", "自动机" 等词汇, 它们的意义都是相同的.



(a) 图示

	0	1
q_0	q_0	q_1
q_1	q_0	q_2
q_2	q_2	q_2

(b) 转移函数

图 1: DFA-1

借图1我们给出 DFA 的一个具体例子并介绍 FA 的图表表示. 图表表示的意义就在于直观理解, 所以读者看图就基本能够理解大致意思. 圆圈表示的是状态, 圆圈间的箭头则是状态在输入字符下的转移, 双圆的是终止状态, 而 *start* 指向的状态为起始状态, 读者不难写出五元组的每一项.

下面的定义让我们可以更好的描述一个串在输入 DFA 进行状态转移的整个过程:

定义 1.2. *DFA* 的格局 *(configuration)* 为元素 $(q, w) \in K \times \sum^*$.

我们定义格局间的二元关系 $\vdash_M$ 为*:*

$$(q, w) \vdash_M (q', w') \Leftrightarrow \exists a \in \sum, w = aw', \text{ 并且} \delta(q, a) = q'.$$

我们称 (q, w) 通过一步推得 (q', w').

虽然格局这个词略显奇怪, 但从定义中可以直接看出 $\vdash_M$ 描述的是消耗一个字符进行一次状态转移的一步推导过程. 虽然这个二元关系很有用, 但我们更想要的是可以描述多步推导的过程. 不难发现, 在 $\vdash_M$ 这个二元关系加上传递性就能达到这个目的, 为了后面叙述方便我们再加上自反性, 即零步推导也算推导. 我们把加上自反性和传递闭包的二元关系称为 $\vdash_M^*$, $*$ 的意思和字符中的 Kleene 星号类似, 代表零步或者多步.

在格局和我们定义的二元关系的语言下, 我们可以很容易的形式化描述一个串被自动机接受与否. 串 $w \in \sum^*$ 被 M 接受, 当且仅当存在状态 $q \in F$ 使得 $(s, w) \vdash_M^* (q, e)$. 而所有被 M 所接受的语言, 我们记为 $L(M)$.

在下面这个较为复杂的例子里, 我们给一个简单的格局的计算:

例 1.1. *考虑如图2所示 $L = (ab \cup aba)^*$ 的 DFA, 不妨考虑串 ababa 的接受过程, 我们可以列出格局的变化*

$$(q_0, ababa) \vdash (q_1, baba) \vdash (q_2, aba) \vdash (q_3, ba) \vdash (q_2, a) \vdash (q_3, e).$$

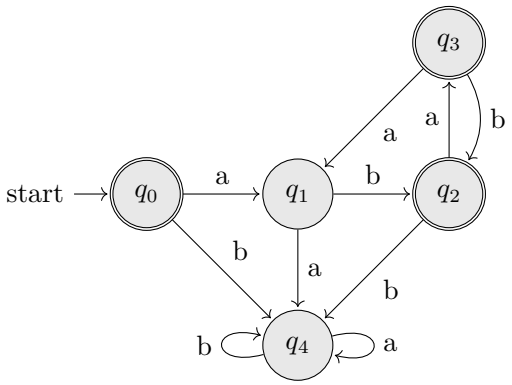


图 2: $(ab \cup aba)^*$ DFA

上面的例子给出了一个格局的推演过程,

定义 1.3 (NFA)**.** *非确定有限自动机 (nondeterministic finite automata, 简称 NFA), 由五元组 $(K, \sum, \Delta, s, F)$ 构成:*

- *K 是由状态构成的有限集合.*
- *$\sum$ 是字母表/字符集.*
- *$s \in K$ 是起始状态.*
- *$F \subseteq K$ 是接受状态的集合.*
- *$\Delta \subseteq K \times (\sum \cup \{e\}) \times K$ 为状态转移**关系**.*

注记 1.2. *这里的 Delta 为状态间的关系, 这也就决定了一个状态在某一字符/输入下可以转移到多个状态, 也说明了不是每一个状态都可以在任一输入下进行转移的.*

在介绍完 DFA 与 NFA 后, 一个自然的问题是它们的计算能力是一样的吗? 所谓计算能力, 粗略来说就是比较能够接受的语言的多少. 形式上来说, 两个 FA M_1 和 M_2 是等价的, 当且仅当 $L(M_1) = L(M_2)$. 直觉上来说, NFA 的计算能力似乎要比 DFA 稍稍强一点, 毕竟它看起来可以不确定能够多一些可能, 但事实却不如此. 接下来引入本章第一个定理:

定理 1.1. *任给一个 DFA, 都存在与其等价的 NFA, 反之亦然.*

证明. ($\Longrightarrow$) 这是显然的, 因为任意一个 DFA 都可以是一个 NFA. 具体来说, 我们只需要把 DFA 定义里的状态转移函数 $\delta : K \times \sum \to K$ 中的任意一项 $(k, s) \mapsto s'$ 变为关系 $(k, s, s') \in \Delta$. 本质上来说, DFA 是 NFA 是因为每一个函数都天然构成一个关系.

($\Longleftarrow$) 对于任一 NFA $M = (K, \sum, \Delta, s, F)$, 我们意图构造 DFA $M' = (K', \sum', \delta, s', F')$, 使得 $L(M) = L(M')$. 为了构造, 我们先引入下面的等价状态的概念, 它主要用来处理 NFA 中的 ϵ/e 转移情况:

$$E(q) = \{p \in K : (q, e) \vdash_M^* (p, e)\}.$$

我们定义 M' 的每一项如下:

1. $K' = 2^K$, 即原来状态的幂集.

2. $\sum' = \sum$, 字母表不变.

3. $s' = E(s)$, 即原始状态的等价集合.

4. $F' = \{Q \mid Q \subseteq K, Q \cap F \neq \emptyset\}$, 即所有与原来的终止状态有交集的集合.

5. 对于任意集合 $Q \subseteq K$ 和 $a \in \sum$, 令

$$\delta(Q, a) = \bigcup \{E(p) \mid q \in Q, (q, a, p) \in \Delta\}$$

对于一个状态 Q, 它是由原状态 K 中的元素构成的集合, 我们将其在输入 a 下转移到的新状态变为 NFA 能转移到的所有状态 (考虑等价状态) 的并.

我们新构造的 M' 显然是一个 DFA, 它使得每一个状态在每一个输入下都进入唯一的新状态. 接下来我们证明它们是等价的, 即 $w \in L(M) \Leftrightarrow w \in L(M')$.

我们断言:

$$(q, w) \vdash_M^* (p, e) \Leftrightarrow (E(q), w) \vdash_{M'}^* (P, e), p \in P$$

有了这个断言, 等价性就是显然的. 因为

$$\begin{aligned} w \in L(M) &\Leftrightarrow (s, w) \vdash_M^* (f, e), f \in F \\ &\Leftrightarrow (E(s), w) \vdash_{M'}^* (Q, e), f \in Q \\ &\Leftrightarrow (s', w) \vdash_{M'}^* (Q, e), Q \in F' \\ &\Leftrightarrow w \in L(M') \end{aligned}$$

我们采用数学归纳法来证明. $|w| = 0$, 即 $w = e$ 时我们只要证明 $(q, e) \vdash_M^* (p, e)$ 等且仅当 $(E(q), e) \vdash_{M'}^* (P, e), p \in P$. 这纯粹是概念的替换, 证明留作联系.

假设 $|w| = k + 1$, 由归纳假设我们知道 $|w| \leq k$ 时断言都是成立的. 很自然可以想到令 $w = va, a \in \sum$. 断言的证明同样分成两步:

1. 假设 $(q, w) \vdash_M^* (p, e)$, 我们知道

$$(q, w) \vdash_M^* (r_1, a) \vdash_M (r_2, e) \vdash_{M'}^* (p, e).$$

我们要证明存在集合 $P \ni p$, 使得 $(E(q), e) \vdash_{M'}^* (P, e)$. 由归纳假设 $(E(q), v) \vdash_{M'} (R_1, e), r_1 \in R_1$. 因为 $(r_1, a) \vdash_M (r_2, e)$, 则存在 $(r_1, a, r_2) \in \Delta$, 所以由我们对于 M' 的构造, $E(r_2) \subseteq \delta'(R_1, a)$. 另一方面 $(r_2, e) \vdash_M^* (p, e)$, 则 $p \in E(r_2)$, 那么 $p \in \delta'(R_1, a)$, 因此 $(R_1, a) \vdash_{M'} (P, e), p \in P$. 故所证成立.

2. 另一边的证明是类似的, 留给感兴趣的读者.

□

注记 1.3. *这一定理的构造事实上也告诉我们该如何把任一 NFA 转换成等价的 DFA, 在具体的计算中我们还会用到这一方法. 因此本定理就不举例阐述了.*

对于构造等价性的证明本质上就是概念的解释和使用 (困难的是构造本身), 读者可以用作对自己概念理解情况的检查.

1.2 正则语言

花开两朵, 各表一枝. 本节我们来到正则语言和正则表达式. 在算术运算中, 我们可以利用符合 + 和 × 来建立如下的表达式:

$$(8+3)*5$$

同样的, 我们也可以使用正则符号来描述语言, 即正则表达式:

$$(0 \cup 1)0^*$$

就像上面的算法表达式的结果为 55 一样, 正则表达式也有其意义. 它与自动机类似, 也是用来识别语言的. $(0 \cup 1)0*$ 的意思是首先要识别一个 0 或 1, 然后要识别零个或多个 1. 不难发现, $0, 00, 100$ 都是可以被该正则表达式识别的, 而 $e, 1$ 则不行.

当然上面只是我们的直观理解, 正则表达式的形式定义如下:

定义 1.4. *在字母表 $\sum$ 下, 以下方式可以构造正则表达式:*

(1) $\emptyset$ *和* $a \in \sum$ *是一个正则表达式.*

(2) 若 α *和* β *是正则表达式,* $(\alpha\beta)$ *也是.*

(3) 若 α *和* β *是正则表达式,* $(\alpha \cup \beta)$ *也是.*

(4) 若 α *是正则表达式,* α^* *也是.*

(5) 除 (1) 至 (4), 没有其他方法可以构造正则表达式.

正则表达式的形式定义实质上定义了其语法, 也就是怎样才能写出合法的正则表达式, 而给定一个正则表达式我们还需要定义其语义. 当然由于我们已经对于 $*, \cup$ 等等记号的意义相当熟悉了, 本文也就不形式化的介绍了. 而所有能被正则表达式识别的串全体, 我们称为**正则语言**.

性质 1.1. *正则表达式基本性质如下:*

- $S \cup R = R \cup S$
- $R(ST) = (RS)T$
- $R(S \cup T) = RS \cup RT, (R \cup S)T = RT \cup ST$
- $\emptyset^* = \{e\}$
- $(R^*S^*)^* = (R \cup S)^*$

读者自证不难.

讲到这里, 相比读者会对上面的内容感到奇怪: 为什么讲着自动机突然开始介绍正则表达式? 那是因为正则语言和自动机能识别的语言 (我们已经证明了 DFA 和 NFA 是等价的) 是相同的. 为了证明它们是等价的, 我们先证明自动机能够识别正则语言.

定理 1.2. *自动机能接受的语言在并, 连接, Kleene 星, 补, 交下封闭.*

证明. 为了简化叙述, 这涉及到一个有限自动机时 (Kleene, 补), 我们记为经典的 $M = (K, \sum, \Delta, s, F)$, 而在涉及到两个有限自动机时, 我们记为 $M_1 = (K_1, \sum, \Delta_1, s_1, F_1), M_2 = (K_2, \sum, \Delta_2, s_2, F_2)$.
而我们新设计的状态机记为 $M' = (K', \sum', \Delta', s', F')$.

1. 并集: 容易想到只要新产生一个状态 s', 然后将 s' 无条件转移到两个状态机的初始状态即可. 即 $K' = K_1 \cup K_2 \cup \{s\}$, $F = F_1 \cup F_2$, $\Delta = \Delta_1 \cup \Delta_2 \cup \{(s, e, s_1), (s, e, s_2)\}$.

   这也说明了正则的有限交还是正则的.

2. 连接: 容易想到只要把前一个状态机的终止状态连接到后一个状态机的起始状态即可. 状态机的设计留给读者.

3. Kleene 星: 本质上只要加一条终止状态到起始状态的回路即可识别多次字符串, 但由于还要识别零次所以要加上一个新的初始状态. 状态机的设计留给读者.

4. 补集: 显然的, 只要令 $F' = K' \setminus F$ 并保持其他元素相同, 就实现了识别原语言的补的功能.

5. 交集: 本质上由于德摩根律我们知道在补集和并集下封闭也就蕴含了交集下的封闭.

□

推论 1.1. 由于空集 $\emptyset$ 和字母表的单个字符 $a \in \sum$ 都显然可以被自动机接受, 而自动机接受的语言在并集, *Kleene* 星号, 和连接下封闭, 故任一正则语言都可以被一个自动机识别.

注记 1.4. 这种构造证明的方式实质上告诉我们该如何把正则表达式转换成 *DFA*, 在下面的算法中我们会进一步看到这一点.

定理 1.3. 若一个语言能被某个自动机接受, 那么它能被一正则表达式识别.

证明. 令 $M = (K, \sum, \Delta, s, F)$ 为有限自动机. 我们的目标是构造正则表达式 R 使得 $L(R) = L(M)$. 构造的核心想法是将原语言分成有限个简单语言, 逐一构造, 再将他们并起来.(正则语言的并还是正则语言) 记 $K = \{q_1, \ldots, q_n\}, s = q_1$.

我们定义 $R(i, j, k)$ 是所有在状态机 M 下从 q_i 到 q_j, 且不经过下标大于等于 $k+1$ 到中间状态的串的集合. 注意 i, j 可以大于 k. 由于状态只有 k 个, 所以当 $k = n$ 时, 立即有

$$R(i, j, k) = \{w \in \sum^* \mid (q_i, w) \vdash_M^* (q_j, e)\}$$

因此

$$L(M) = \bigcup \{R(1, j, n) \mid q_j \in F\}$$

即能走到各个终止状态的串的集合, 我们要证明每个 $R(i, j, k)$ 都是正则的.

我们对 k 进行归纳. $k = 0$ 时, $R(i, j, 0)$ 即位一步能到达的情况, 即 $(q_i, a, q_j) \in \Delta$, 在 $i = j$ 时加上 $\{e\}$, 这些串都是有限的, 因此是正则的. (为什么?)

进一步, 若 $R(i, j, k-1)$ 都是正则的, 我们可以将这些组成 $R(i, j, k)$, 即

$$R(i, j, k) = R(i, j, k-1) \cup R(i, j, k-1)R(k, k, k-1)^* R(k, j, k-1).$$

因为正则语言在并, Kleene 星号和连接下都是封闭的, 可推得 $R(i, j, k)$ 也是正则语言, 也就完成了证明. □

注记 1.5. 上述证明完全来自课本, 推荐有时间的同学将课本完整看一遍, 定会受益匪浅.

推论 1.2. 由推论*1.1*, 定理*1.3*我们知道, 自动机和正则语言是等价的. 具体来说, 一个语言是正则的**当且仅当**它能被某个自动机接受.

算法 1. 如图*5*所示, 我们通过一个具体的例子来展示自然语言到 *DFA* 的全过程操作, 我们选择的语言是在字母表 $\sum = \{a, b\}$ 下以串 ab 结尾的所有串.

1. 自然语言 $\to$ 正则表达式. 这一点并没有什么统一的方式. 这个例子是比较简单的, 很容易就能写出 $(a \cup b)^*ab$ 的答案. 当然这问题困难的情况下这一步其实是最难的, 很多时候只能靠感觉了.

2. 正则表达式 $\to$ *NFA*. 回顾定理1.2我们知道该如何构造. 结果如图3所示.

3. *NFA* $\to$ *DFA*, 这也由定理1.1直接可构造出来. 注意在实际操作中我们一般不会将所有可能的状态都表示出来 (即原来 *NFA* 状态的幂次), 而是从初始状态开始, 按照取 ϵ 闭包, 再加上状态转移的方式构造新的状态. 这样就可以不用构造那些到达不了的状态.

   具体在这个例子中, 我们知道 0 的闭包是 $A = \{0, 1, 2, 4, 7\}$, 我们就直接考虑

   $$A = \{0, 1, 2, 4, 7\} \xrightarrow{a} \{1, 2, 3, 4, 6, 7, 8\} = B$$

   如此类推. 最终结果如图4a所示.

4. 最后我们考虑如何使得 *DFA* 状态最小. 其核心思想是把原有状态化为一些不相交的子集, 使得任何两个不同子集的状态是可区分的. 由于这并不是计算理论关注的核心内容, 我们仅将答案列出供读者参考, 如图4b.

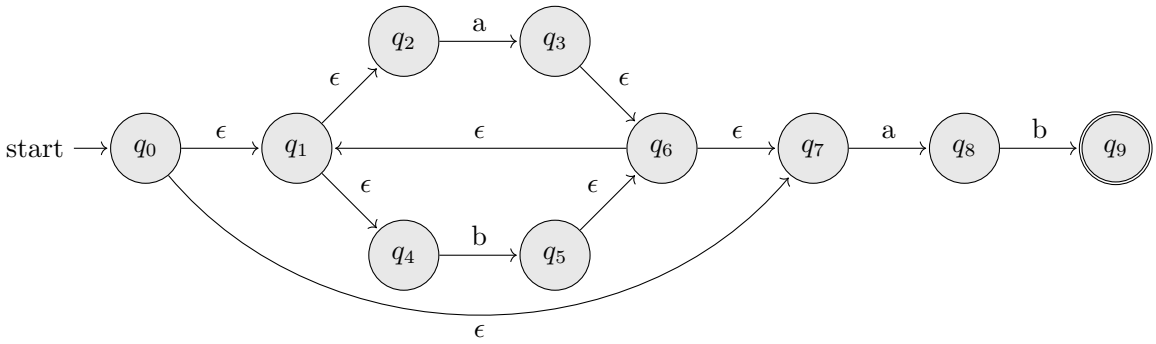
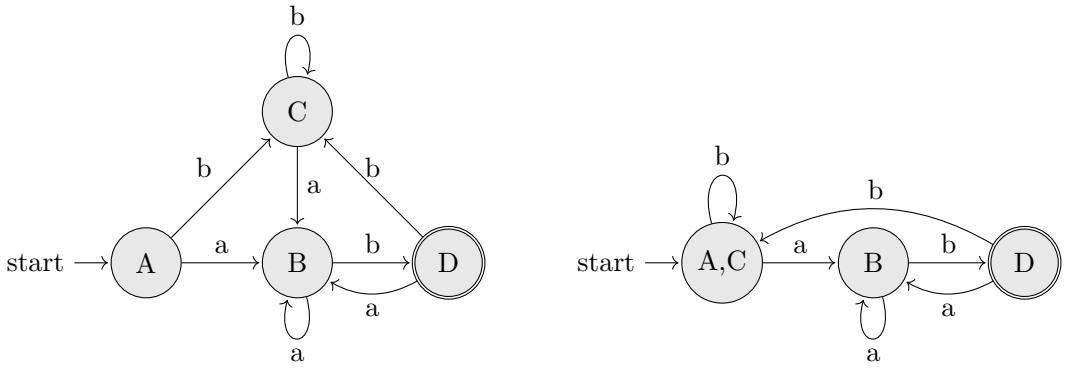


图 3: $(a \cup b)^*ab$ NFA



(a) $(a \cup b)^*ab$ DFA

(b) $(a \cup b)^*ab$ 最小化 DFA

注记 1.6. 事实上, 今天我们实际使用的正则表达式要比本文介绍的强的多的多, 如下面的正则表达式能判定一个字符串是否恰好由非素数个 1 组成:

$$/\,\hat{}\,1?\$|\hat{}\,(11+)1+\$/$$

为了支持现代正则表达式的 "邪恶" 特性, 现代的引擎并不是本文这样构造 *DFA* 的识别方式了, 而是一种基于搜索的方式. 感兴趣的同学推荐论文《*Programming techniques: Regular expression search algorithm*》.

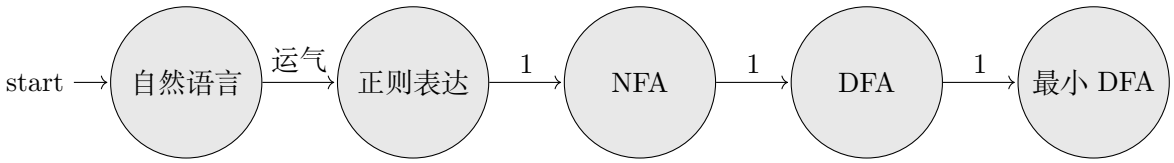


图 5: 自然语言到最小 DFA 全过程

1.3 非正则语言

在明晓了正则语言, 或者说 DFA/NFA 能接受的语言后, 一个自然的问题是什么语言是它不能接受的呢? 有限状态机的每一状态跳转都只和当前状态以及当前输入字符相关, 那么有限状态机实质上并不能记住过去经历的状态, 所以我们可以期待的是: 在数据结构课程学过的一些需要辅助数据结构如栈来解决的问题, 有限状态机是无法解决的. 这其中最为经典的就是括号匹配问题. 而为了证明这一点, 我们引出一个重要的定理:

定理 1.4 (Pumping Theorem-1)**.** *令 L 为正则语言, 则存在 $n \in \mathbb{N}^+$ 使得任一串长度大于等于 n 的串 $w \in L$ 都可以写作 $w = xyz$ 并满足:*

- $y \neq e$ • $|xy| \leq n$ • $xy^iz \in L, i \in \mathbb{N}$

证明. 本质上来说, 正则语言与有限状态机是等价的. 由于有限自动机的状态是有限的, 不妨假设为 n, 那么所有长度大于等于 n 的串都会回到过去曾经到过的状态 (鸽笼原理), 这一重复的过程也就构成了定理中的 y^i. 证明细节留给读者. □

一开始看到该定理的读者可能会有些疑惑, Pumping Theorem 本质上是叙述了正则语言的一个重要性质, 那怎么通过该定理证明一个语言不是正则语言呢? 答案是逆否命题. 正则语言一定有该循环性质, 那么没有该循环性质的语言一定不是正则语言. 我们通过一个例子, 展示利用该定理和反证法证明一个语言不是正则语言的一般流程:

例 1.2. *目标: 证明 $L = \{a^ib^i \mid i \geq 0\}$ 不是正则语言.*

1. *假设 L 是正则语言. 那么 L 满足 Pumping Theorem 的前置条件.*
2. *存在 $n \in \mathbb{N}^+$, 使得任一串 $w, |w| \geq n$ 都可以写作 $w = xyz$ 并满足相应条件.*
3. *我们选择 $w = a^nb^n$, 显然 $|w| \geq n$, 故 $w = xyz$ 且 $y \neq e, |xy| \leq n$, 不难发现 x, y 都只能由重复的 a 构成, 即 $x = a^p, y = a^q, q > 0$.*
4. *我们知道 $xy^iz \in L$, 则只需取 $i = 0$, 我们得到 $xz \in L$. 而由上一点我们知道 $xz = a^{n-q}b^n \notin w$, 矛盾!*
5. *故 L 不是正则语言.*

当然, 我们同样可以利用 Pumping Theorem 来证明匹配括号不是正则语言. 令 w 为由 n 个开括号和 n 个闭括号组成的串, 即 $w = (((\ldots)))$, 剩下的与上例同理. 如何根据给定 n 选择一个合适的长度大于 n 的串是利用 Pumping Theorem 证明的关键.

最后我们通过一个例子说明并不是只有正则语言才满足 Pumping Theorem, 该例子来自 StackExchange.

例 1.3. *$L = \{\$a^nb^n \mid n \geq 1\} \cup \{\$^kw \mid k \neq 1, w \in \{a, b\}^*\}$. L 的第二部分显然是正则的, 而第一部分不是正则的, 而两部分之间并没有交集, 因此 L 不是正则语言.*

下证 L 满足 Pumping Theorem, 我们只需考虑 $= \$a^kb^k, k \geq 1$. 只要选择 $x = e, y = \$, z = a^kb^k$, 我们知道 $xy^iz = \$^ia^kb^k \in L$. 由此 L 满足 Pumping Theorem.

2 上下文无关文法与下推自动机

回顾上一章我们讲过正则语言无法识别匹配括号, 这说明正则语言的计算能力还是比较一般的. 本章我们将介绍上下文无关语言, 以及和其对应的下推自动机.

2.1 上下文无关文法与语法树

开宗明义, 我们直接列出定义:

定义 2.1 (CFG)**.** *上下文无关文法 (context-free grammar, CFG) 由四元组 $G = (V, \sum, R, S)$:*

- V 是字母表.
- $\sum \subseteq V$ 是**终结符**.
- $S \in V - \sum$ 是**起始符**.
- $R \subseteq (V - \sum) \times V^*$ 是规则.

这里比较有趣的一项显然是规则, 其表达的含义是一个非终结符可以推导成许多其他字符.

和前面一样, 当我们定义了一个文法概念后, 就要想着怎么形式化的表示推导, 最终表示识别的语言:

定义 2.2.

- 所谓一步推导, 就是将串里面的一个非终结符根据对应的规则进行替换. 对于串 $u, v \in V^*$, 我们称 u 在文法 G 一步推导到 v, 记为 $u \Rightarrow_G v$, 若 $u = xAy, v = xBy, A \to_G B$.
- 类似的, 我们该兴趣的是整个推导过程, 即多步推导, 用符号 $\Rightarrow_G^*$ 表示, 为一步推导自反 + 传递闭包.

同样的, 记所有由语法 G 生成的所有语言为 $L(G)$, 即

$$L(G) = \{w \in \sum^* \mid S \Rightarrow_G^* w\}.$$

若该文法是上下文无关文法, 那么我们称该语言为**上下文无关语言**, 简称 CFL.

这里要注意的是, 如果推导不会结束, 例如规则只有 $S \to S$. 这是符合定义的, 那么它生成的语言是什么呢? 由于不会结束, 所有没有任何一个串可以由该文法推导出来, 故其生成的语言为空集.

有了上下文无关文法, 我们就可以解决上一章提到的有限自动机所不能解决的匹配括号问题了:

例 2.1 (括号匹配)**.** $G = (V, \sum, R, S)$, 其中 $V = \{S, (,)\}, \sum = \{(,)\}, R = \{S \to e, S \to SS, S \to (S)\}$. 从规则中不难看出括号匹配的核心是要么生成并排的括号 *()()*, 要么生成嵌套的括号 *(())*.

具体到一个例子 *(()())*, 我们可以用下述推导

$$S \Rightarrow (S) \Rightarrow (SS) \Rightarrow ((S)S) \Rightarrow (()S) \Rightarrow (()(S)) \Rightarrow (()()).$$

性质 2.1. 所有的正则语言都是上下文无关语言.

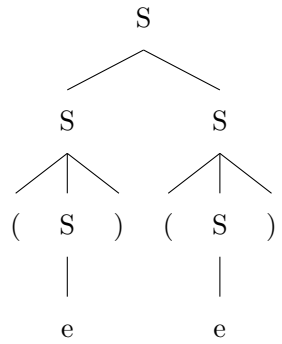
证明. 只需明 CFL 在并集, 连接, Kleene 星号下封闭, 即构造对应的 CFG 的即可. 并集只需要 $S \to S_1, S \to S_2$, 连接只需要 $S \to S_1 S_2$, Kleene 星号只需要 $S \to SS', S \to e$, 细节留给读者. □

CFG 还是有其独特有趣之处, 比如我们看下面两个括号匹配的推导过程:

$$S \Rightarrow SS \Rightarrow (S)S \Rightarrow ()S \Rightarrow ()(S) \Rightarrow ()() \tag{2.1}$$

$$S \Rightarrow SS \Rightarrow S(S) \Rightarrow (S)(S) \Rightarrow (S)() \Rightarrow ()() \tag{2.2}$$

如果我们仔细观察, 我们会发现这两个推导在**本质上**是一样的, 只不过顺序不同罢了. 一个先换了左边的 S, 另一个先换了右边的 S. 那么我们怎么来描述这一点呢? 这就引出了语法树 (parsing tree) 的概念.



本文中并不打算给出语法树的形式定义, 因为不太有必要. 从上面的语法树中可以看出, 语法树就是利用树的结构把推导过程表示了出来. 有趣的是 (2.1), (2.2) 这两个看似不同的推导, 其语法树是完全一致的.

语法树实质上是对推导的一个深刻的洞察, 它其实是一个等价类. 那些本质上相同的推导, 在语法树这个等价类下就是相同的元素. 容易满足, 这个关系满足自反性, 对称性与传递性. 用更数学的语言来说, 语法树是所有推导**模**顺序的结果.

既然语法树实质上是等价类, 我们只要保证每次从一个等价类里拿出的元素并不与之前的等价类相同即可, 除了画出语法树我们还有另外的方法: 最左推导与最右推导. 顾名思义, 最左推导指的就是每次都优先替换最左边的最左边的非终结符. 按照上面的例子, 即为:

$$S \Rightarrow SS \Rightarrow (S)S \Rightarrow ()S \Rightarrow ()(S) \Rightarrow ()()$$

可以发现, 这实质上就是固定了语法树的生成. 也就是说, 两个推导的语法树相同, 它们的最左推导也就相同, 最右推导亦是如此.

当然, 有些串的推导是可以存在两个不同的语法树但有相同的推导结果的, 我们称这样的情况为歧义, 因为这样导致了给定一个串, 我们可以画出不同的语法树, 也就对应了不同的计算方式:

定义 2.3. *一个文法是有**歧义**的, 若对于一个串存在两个不同的语法树.*

在实践中, 我们一般通过说明某个生成的串的最左推导不唯一来证明该文法存在歧义, 因为我们已经知道了最左推导和语法树是一一对应的.

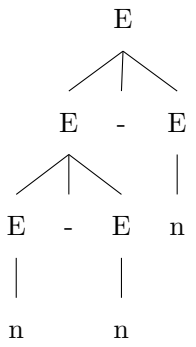
例 2.2. *很容易发现, $S \to S+S, S \to a$, 在 $a+a+a$ 下是有歧义的.*

2.2 歧义的消除 *

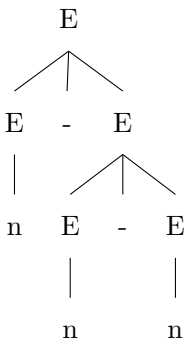
例 2.3 (算术表达式)**.** *最经典也是最重要的歧义出现在算术表达式中, 我们首先列出一个经典的描述的上下文无关文法:*

(i) $V = \{+, -, \times, n, E\}$.

(ii) $\sum = \{+, -, \times, n\}$.

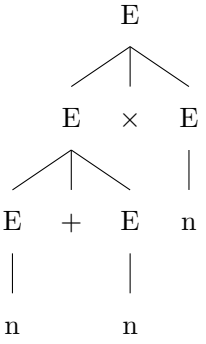


(a) 左结合

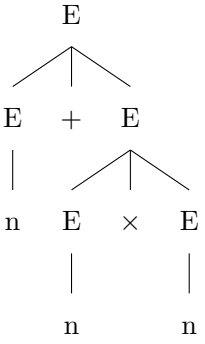


(b) 右结合

图 6: 结合性歧义



(a) 错误的优先级



(b) 正确的优先级

图 7: 优先级歧义

(iii) 起始符即为 E.

(iv)

$$E \to E+E, E \to E-E, E \to E \times E, E \to n.$$

可以看到, 上面就是一个简单的算术表达式. 而上面的文法存在两个严重的歧义问题:

1. 结合性: 虽然加法和乘法都有结合律, 但减法却没有, 即 $a-(b-c) \neq (a-b)-c$. 所以我们在解析四则运算过程中应当按照左结合性, 但很显然, 上述语法并不满足, 对于 $n-n-n$ 我们有两个语法树, 如图6所示. 可以看到, 我们期待的结果是左结合的, 但该文法并不能避免右结合的出现.

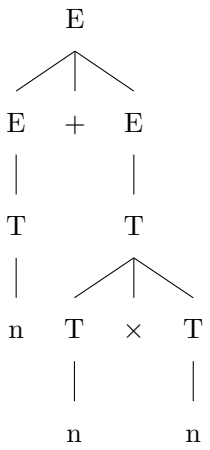
2. 优先级: 我们知道, 乘法的优先级是高于加法的, 这意味着 $3+4*5$ 应当解析为 $3+(4*5)$ 而不是 $(3+4)*5$, 但如果只是上述文法同样会出现歧义, 如图7所示.

那么怎么解决这个问题呢? CFG 自己是没有办法获取优先级和结合性的信息的, 我们只能通过修改语法但不改变语言来解决这个问题.

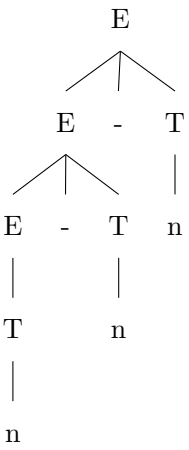
为了解决优先级的问题, 我们新增加一个非终结符 T, 将规则改造为

$$E \to E+E, E \to E-E, E \to T, T \to T \times T, T \to n.$$

我们实质上把加法, 减法和乘法隔绝开了, 其效果是加减法更靠近 E, 就更快以它们作为串的分割, 优先级也就越低. 通过这样的改造, 现在的语法树就没有优先级的问题了, 如图8a所示.



(a) 解决优先级



(b) 解决结合性

图 8: 修改后的文法

为了解决结合性的问题, 我们进一步修改文法, 而我们的修改也很简单, 如图8b所示:

$$E \to E+T, E \to E-T, E \to T, T \to T \times T, T \to n.$$

注记 2.1. *以上内容在课程设计中属于编译原理语法分析, 写在这里供有兴趣的读者提前查阅.*

2.3 下推自动机与等价性

定义 2.4 (PDA)**.** *下推自动机 (pushdown automata, 简称 DFA), 由六元组 $(K, \sum, \Gamma, \delta, s, F)$ 构成:*

- K *是由状态构成的有限集合.*
- $\sum$ *是字母表/字符集 (输入).*
- Γ *是字母表 (栈顶).*
- $s \in K$ *是起始状态.*
- $F \subseteq K$ *是接受状态的集合.*
- $\delta : (K \times (\sum \cup \{e\}) \times \Gamma^*) \times (K \times \Gamma^*)$ *为状态转移关系.*

初看 PDA 的定义尤其是状态转移关系 δ 很容易让人有所惧怕, 但实际上我们一项一项拆解来看. 前面相对于 NFA 只是多了栈上的字符, 而状态转移实际上就是当前状态, 和输入的字符 (加上无条件转移 e), 再加上栈上的**某些元素**, 完成状态的转移, 以及栈上元素的变化.

类似的, PDA 的格局也就是加上了栈的状态, 即属于 $K \times \sum^* \times \Gamma^*$ 的元素. 而此时格局当推演也要额外的考虑栈的变化, 不再赘述.

而 PDA 接受某个串的定义也有所不同. 串 $w \in \sum^*$ 能被 PDA 接受, 当且仅当 $(s, w, e) \vdash_M^* (p, e, e), p \in F$. 注意此时栈顶的元素也须为空.

例 2.4. *我们构造一个 PDA 来识别 $L = \{wcw^R \mid w \in \{a, b\}^*\}$. $K = \{s, f\}, \sum\{a, b, c\}, \Gamma =$*

$\{a,b\}, F=\{f\}$.

$$((s,a,e),(s,a))$$
$$((s,b,e),(s,b))$$
$$((s,c,e),(f,e))$$
$$((f,a,a),(f,e))$$
$$((f,b,b),(f,e))$$

而如果要识别的语言是 $L=\{ww^R \mid w\in\{a,b\}^*\}$ 的话, 我们只要简单的修改字母表, 以及将 *((s, c, e),(f, e))* 改为 *((s,e, e),(f, e))* 即可.

熟悉上一章的读者应该已经意识到了我们接下来要证明 PDA 与 CFG 是等价的, 同样的, 这个证明的一个方向是容易的, 另一个方向则稍显繁琐.

定理 2.1. *对于每一个上下文无关语言, 都存在一个 PDA 可以接受它.*

证明. 对于上下文无关文法 $G=(V,\sum,R,S)$, 我们直接构造 PDA M. 它只有两个状态, p 为起始状态, q 为终止状态, 栈上的字母表与 G 的字母表相同, 而最重要的是状态转移关系:

1) $((p,e,e),(q,S))$

2) 对于任一规则 $A\to x\in R$, $((q,e,A),(q,x))$

3) $((q,a,a),(q,e)),\forall a\in\sum$.

可以看到, 这个 PDA 的构造相当直接, 先直接转移到终止状态并在栈顶放一个起始符, 然后选择一个合适的规则生成 (非确定性), 如果有生成对应的非终结符就消掉, 最后如果能全部消掉就说说明推导是成功的.

构造正确性的证明略, 其实质上对应了一个最左推导. □

另一边, 为了证明每一个被 PDA 接受的语言都是 CFL, 我们需要一个引理先简化 PDA.

引理 2.1. *如果一个语言被 PDA 所接受, 那么它也能被简化的 PDA 所接受.*

所谓简化的 PDA, 是指转移关系 $((q,a,\beta),(p,\gamma))$ 在 q 不是初始状态时, $\beta\in\Gamma$ 且 $|\gamma|\le 2$.

定理 2.2. *如果一个语言被 PDA 接受, 那它是 CFL.*

推论 2.1. *由定理2.1和定理2.2我们知道 PDA 和 CFG 是等价的, 即任给一个 PDA, 都有一个 CFG 可以接受同样的语言, 反之亦然.*

注记 2.2. *上述证明如果有读者感兴趣的话可自行参阅教材, 本文略. (其实是我懒)*

在快速介绍完 PDA 与 CFL 等价后, 我们来证明一个封闭性的定理:

定理 2.3. *上下文无关语言与正则语言的交还是上下文无关语言.*

证明. 令 L 为 CFL, R 为 RL. $L=L(M_1)$, $M_1=(K_1,\sum,\Gamma_1\Delta_1,s_1,F_1$, R $=L(M_2)$, $M_2=(K_2,\sum,\delta,s_2,F_2)$. 我们的想法是**并行**模拟 M_1,M_2, 当它们都接受的时候我们才接受.

具体来说, 我们构造 $M=(K,\sum,\Gamma,\Delta,s,F)$, 其中

1. $K=K_1\times K_2$.
2. $\Gamma=\Gamma_1$.
3. $s=(s_1,s_2)$.

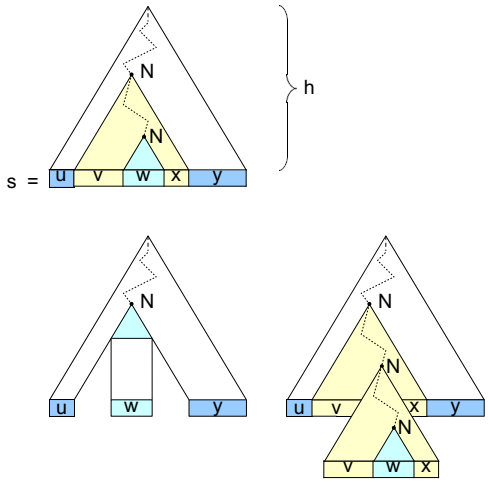


图 9: 维基百科: Pumping Theorem

4. $F = F_1 \times F_2$.

5. Δ 的定义是核心. 对于 PDA M_1 中的 $((q_1, a, \beta), (p_1, \gamma)) \in \Delta_1$, 和每一个 $q_2 \in K_2$, 我们构造 $(((q_1, q_2), a, \beta), ((p_1, \delta(q_2, a), \gamma)))$; 而对于空转移的情况, 即 $((q_1, e, \beta), (p_1, \gamma)) \in \Delta_1$, 我们添加 $(((q_1, q_2), e, \beta), ((p_1, q_2), \gamma)))$.

不难看出 $w \in L(M)$ 当且仅当 $w \in L(M_1) \cap L(M_2)$. □

2.4 非上下文无关语言

最后还是和上一章类似, 我们来考虑哪些语言是 CFG 无法表示的. 正如 DFA 的灵感来源于状态数量是有限的所以必然会有循环的出现, CFL 的灵感来源于语法树上的终结符, 非终结符, 推导亦是有限的. 也就有了下面的定理:

定理 2.4 (Pumping Theorem-2)**.** *令 $G = (V, \sum, R, S)$ 为 CFG. 那么存在 $n \in \mathbb{N}^+$, 使得任意 $w \in L(G), |w| > n$ 都可以写成 $w = uvxyz$, 满足:*

- $|vy| \geq 1$.
- $uv^i xy^i z \in L(G), \forall i \geq 0$.

图 9道尽一切.

例 2.5. $= \{a^n b^n c^n \mid n \geq 0\}$ *不是 CFL.*

(i) 假设 L 是 CFL, 那么其满足 Pumping Theorem.

(ii) 存在 n, 使得任一 $|w| > n, w \in L$, 都能写成 $w = uvxyz$, 且 $vy \neq e, uv^i xy^i z \in L$.

(iii) 令 $w = a^n b^n c^n$, 我们考虑 v, y 可能是什么. 若它们包含 a, b, c 三个字母, 那么至少有一个包含两种字母, 不妨设为 v. 那么 uv^2xy^2z 的顺序就会有问题, 不满足 a, b, c 的次序.

(iv) 若它们不包含所有字母, 那么显然 uv^2xy^2z 的 a, b, c 数目就不可能相同了. 矛盾!

有了这个重要的例子, 我们可以说明 CFL 在交和补下并不是封闭. 令 $L_1 = \{a^n b^n c^m \mid m, n \geq 0\}, L_2 = \{a^m b^n c^n \mid m, n \geq 0\}$, 这两个显然都是 CFL, 而它们的交集则是我们上文刚证明的 $a^n b^n c^n$, 不封闭. 有了交集下的反例, 补下不封闭是显然的. (德摩根律).

3 图灵机

到目前为止我们所作的一切其实都只能算是热身, 我们之前所讨论的计算模型虽然有助于我们理解和提供例子, 但就其能力而言, 它们并不能完成我们所期待的计算机能完成的一些很简单的事情, 从前面的例子中我们也看到了这一点. 本章开始我们要介绍的图灵机 (Turing machine) 才是真正意义上的计算模型.

3.1 图灵机的定义与计算

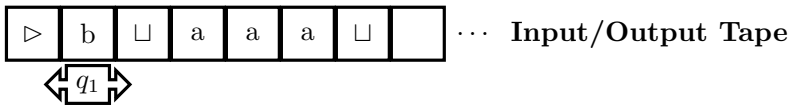


图 10: 图灵机

定义 3.1. *图灵机是一个五元组 $K, \sum, \delta, s, H$, 其中:*

- *K 是有限状态集.*
- *$\sum$ 是字母表, 我们要求 $\sum$ 必须要包含 $\sqcup$ (空符号) 和 $\triangleright$, 且不能包含 $\rightarrow$ 和 $\leftarrow$.*
- *$s \in K$ 是起始状态.*
- *$H \subseteq K$ 是终止状态的集合*
- *$\delta : (K - H) \times \sum \rightarrow K \times (\sum \cup \{\rightarrow, \leftarrow\})$ 是转移**函数**, 且满足:*
  1. *$\forall q \in K - H$, 若 $\delta(q, \triangleright) = (p, b)$, 则 $b = \rightarrow$.*
  2. *$\forall q \in K - H$ 且 $a \in \sum$, 若 $\delta(q, a) = (p, b)$, 则 $b \neq \triangleright$.*

注记 3.1. *图灵机的强大之处来自于它可以写磁带, 这是之前的计算模型都不具备的. 当然如果不能回头 (左右移动带头), 能写磁带也就没有太大意义了, 所以图灵机自然也要能够两头移动.*

同样, 让我们来看两个简单的例子感受一下图灵机是怎么工作的.

例 3.1.

1. *如图12所示, 我们从初始状态 q_0 开始, 碰到 a 进写成空并进入 q_1, 这时此然会右移然后回到状态 q_0, 直到看到一个初始即为 的字符就终止.*
2. *如图11所示. 如果磁带上的字符为 a 的话, 该图灵机就会一直右移左移永不停止. 这个例子告诉我们图灵机是可以不终止的, 这也就引出了极多有趣的问题.*

同样的, 为了形式化图灵机操作的过程, 我们也须引入格局的概念:

定义 3.2. *图灵机 $M = (K, \sum, \delta, s, H)$ 的格局是 $K \times \triangleright \sum^* \times (\sum^* (\sum - \{\sqcup\}) \cup \{e\})$ 的一个元素.*

简单来说, 图灵机的格局是一个由状态, 当前及之前符号, 之后符号构成的三元组. 我们用 $(q, w\underline{a}u)$ 来代替格局 (q, wa, u), 下划线代表了当前正在磁带指向的字符.

有了格局的定义, 和往常一样我们来讲图灵机的推导.

图灵机由五元组 $(K, \sum, \delta, s, H)$ 构成:

- $K = \{q_0, h\}$.
- $\sum = \{a, \sqcup, \triangleright\}$.
- $s = q_0$.
- $H = \{h\}$
- δ 如右图所示.

$q,$	σ	$\delta(q, \sigma)$
q_0	a	$(q_0, \leftarrow)$
q_0	$\sqcup$	$(h, \sqcup)$
q_0	$\triangleright$	$(q_0, \rightarrow)$

图 11: 图灵机例 2

图灵机由五元组 $(K, \sum, \delta, s, H)$ 构成:

- $K = \{q_0, q_1, h\}$.
- $\sum = \{a, \sqcup, \triangleright\}$.
- $s = q_0$.
- $H = \{h\}$
- δ 如右图所示.

q	σ	$\delta(q, \sigma)$
q_0	a	$(q_1, \sqcup)$
q_0	$\sqcup$	$(h, \sqcup)$
q_0	$\triangleright$	$(q_0, \rightarrow)$
q_1	a	(q_0, a)
q_1	$\sqcup$	$(q_0, \rightarrow)$
q_1	$\triangleright$	$(q_1, \rightarrow)$

图 12: 图灵机例 1

定义 3.3. 令 $M = (K, \sum, \delta, s, H)$ 为图灵机, $(q_1, w_1\underline{a_1}u_1)$ 和 $(q_2, w_2\underline{a_2}u_2)$ 为两个格局, 那么我们称推导

$$(q_1, w_1\underline{a_1}u_1) \vdash_M (q_2, w_2\underline{a_2}u_2)$$

当且仅当存在 $b \in sum \cup \{\leftarrow, \rightarrow\}$, 满足 $\delta(q_1, a_1) = (q_2, b)$ 且下列情况之一成立

1. $b \in \sum, w_1 = w_2, u_1 = u_2, a_2 = b$, 即最标准的图灵机写操作.
2. $b = \leftarrow, w_1 = w_2a_2$, 那么 $u_2 = a_1u_1$ 或者 $u_2 = e$.
3. $b = \rightarrow, w_2 = w_1a_1$, 那么 $u_1 = a_2u_2$ 或者 $u_1 = u_2 = e$.

而一个图灵机的计算即为格局的推导:

$$C_0 \vdash_M C_1 \vdash_M C_2 \vdash_M \cdots \vdash_M C_n.$$

图灵机的一些记号 (Notation) 我们略过, 因为这些图不太好画而内容也不太有趣, 有兴趣的读者可自行参阅原书. 我们在此直接给出一个实际例子:

例 3.2 (Copy Machine)**.** 拷贝机可以将 $\sqcup w\sqcup$ 转化为 $\sqcup w \sqcup w\sqcup$, 如图*13*所示. 解释起来也是简单的, 我们不过是每次往右找一个字符, 先写成空, 然后找到第二个空的地方, 写下该字符, 再回到原来写成空的地方重新写下即可. 注意这里我们要先写成空的原因是为了回来的时候能够定位.

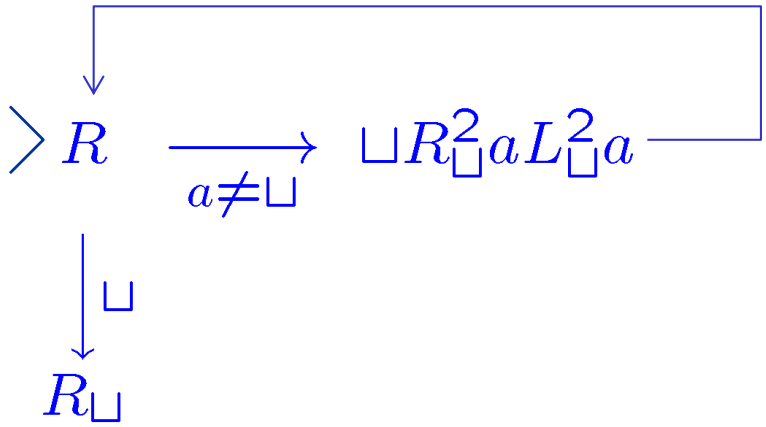


图 13: 拷贝机

定义 3.4. 令图灵机终止状态只包括 y, n, 其中如果能推导得到状态 y 便接受该串, 得到状态 n 便拒绝该串.

图灵机 M 决定了语言 L, 若:

- $w \in L$ 时, M 接受 w
- $w \notin L$ 时, M 拒绝 w.

注意如果要判定一个语言的话无论是接受还是拒绝图灵机都应该终止, 这是十分重要的, 也是之前的计算模型所不用考虑的问题.

而我们称 M 半判定语言 L, 当 $w \in L$ 当且仅当 M 在输入 w 上会停机. 注意这个时候我们不知道当 $w \notin L$ 时候当情况.

我们称语言 L 是递归的 *(recursive)*, 当存在图灵机决定 L; 称为递归可枚举的, 当存在图灵机半判定 L.

不难看出, 递归语言在并, 交, 补下都是封闭的. 递归语言一定是递归可枚举语言.

注记 3.2. 为什么要使用 "递归" 这个词呢? 因为我们后面会看到, 这与另一套由哥德尔发现的有关逻辑与递归函数的理论实质是一回事情.

定义 3.5 (计算函数)**.** 令 $M = (K, \sum, \delta, s, H)$ 是一个图灵机. 令 $\sum_0 \subseteq \sum - \{\triangleright, \sqcup\}$ 为一个字母表且 $w \in \sum_0^*$.

假设 M 在输入 w 终止, 即 $(s, \triangleright \sqcup w) \vdash_M^* (h, \triangleright \sqcup y)$, 我们称 y 是 M 在输入 w 上的输出, 记为 $M(m)$.

如此, 我们就可以定义 M **计算** 函数 f, 即

$$M(w) = f(w)$$

我们称函数 f 是递归的, 若存在图灵机 M 计算 f.

3.2 文法

前面我们介绍了上下文无关文法, 其核心是由一个非终结符进行推导, 即 $S \to A \ldots$, 其中的上下文无关指的就是只存在一个 S. 那么我们自然可以想到能不能定义 $SAB \to C \ldots D$ 这样的规则, 即替换与上下文有关而不仅仅是一个非终结符. 这样, 我们就有了任意文法的定义:

定义 3.6. 一个文法 *(grammar)*, 或称为无限制文法 G, 由四元组 $(V, \sum, R, S)$ 构成:

- V 是字母表.

- $\sum \subseteq V$ 是终结符.
- $S \in V - \sum$ 是起始符.
- R 是规则, 即含于 $(V^*(V - \sum)V^*) \times V^*$ 的有限子集.

规则看似有些复杂, 实质上只是要求左边至少要有一个非终结符.

关于推导, 生成的语言文法与上下文无关文法完全相同, 显然上下文无关文法亦是文法.

例 3.3. 回忆一下上下文无关文法无法生成 $\{a^n b^n c^n \mid n \geq 1\}$. 我们就以该语言举例:

- $\sum = \{a, b, c\}$.
- $V = \{a, b, c, S, A, B, C, T_a, T_b, T_c\}$.
- $R = \{S \to ABCS, S \to T_c$
  (因此 $S \Rightarrow (ABC)^n T_c$)
  $B\ A \to AB, CA \to AC, CB \to BC$
  (可以任意交换顺序)
  $CT_c \to T_c c, CT_c \to T_b c$
  $BT_b \to T_b b, BT_b \to T_a b, AT_a \to T_a a, T_a \to e\}$

上述文法保证了我们生成的串一定是形如 $a^n b^n c^n$. (为什么?).

定理 3.1. 一个语言由文法生成当且仅当它是递归可枚举的.

3.3 数值函数

数值函数同样是一个有趣的概念, 它实质上就是一些从整数 (整数的积) 到整数的映射.

定义 3.7. 我们首先要定义一些 $\mathbb{N}^k \to \mathbb{N}$ 的**基本函数**:

1. k 元零函数定义为 $zero_k(n_1, \ldots, n_k) = 0$.
2. 第 j 个 k 元恒等函数定义为 $id_{k,j}(n_1, \ldots, n_k) = n_j$.
3. 后继函数定义为 $succ(n) = n + 1$.

我们另外定义一些合成函数的方法.

1. 利用 g 和 h_i 函数复合:

$$f(n_1, \ldots, n_l) = g(h_1(n_1, \ldots, n_l), \ldots, h_k(n_1, \ldots, n_l))$$

2. 利用 g 和 h 递归定义:

$$f(n_1, \ldots, n_k, 0) = g(n_1, \ldots, n_k) \tag{3.1}$$
$$f(n_1, \ldots, n_k, m+1) = h(n_1, \ldots, n_k, m, f(n_1, \ldots, n_k, m)) \tag{3.2}$$

上面定义的基本函数和通过基本函数进行复合和递归能得到的函数全体, 我们称为 **原始递归函数**.

有许许多多的函数都是原始递归函数, 让我们来看:

例 3.4. 以下函数是递归函数:

1. $plus2(n) = n + 2$.

2. $plus(m, n) = m + n$.

3. $mult(m, n) = m \cdot n$.

4. 前驱函数, 即 $pred(0) = 0, pred(n + 1) = n$.

5. $m \sim n = \max\{m - n, 0\}$

$$m \sim 0 = m$$
$$m \sim (n + 1) = \text{pred}(m \sim n)$$

6. $iszero(n) : iszero(0) = 1, iszero(m + 1) = 0$.

7. $greater - than - or - equal(m, n) = iszero(n, m)$.

8. $less - than(m, n)$.

9. 原始递归的与, 或, 非.

$$\neg p(m) = 1 \sim p(m)$$
$$p(m, n) \vee q(m, n) = 1 \sim iszero(p(m, n) + q(m, n))$$
$$p(m, n) \wedge q(m, n) = 1 \sim iszero(p(m, n) \cdot q(m, n))$$

10. 原始递归的选择

$$f(n_1, \ldots, n_k) = \begin{cases} g(n_1, \ldots, n_k), & 若p(n_1, \ldots, n_k) \\ h(n_1, \ldots, n_k), & 否则 \end{cases}$$

11. $prime(x)$, 即判断一个数是不是素数.

不难发现原始递归函数一定是可枚举的, 因为每一个原始递归函数都是由基本函数构造而成的. 因此我们可以按照词典序将其列出:

$$f_0, f_1, f_2, \ldots$$

现在我们定义 $g(n) = f_n(n) + 1$.

不难发现, $g(n)$ 是可计算的, 但却不是原始递归的. 那么我们需要什么操作才能对起进行扩充呢? 这就引出下一个概念:

定义 3.8. 令 g 为 $k+1$ 元函数. 我们称 g 的极小化是 k 元函数 f:

$$f(n_1, \ldots, n_k) = \begin{cases} m \text{ 存在的最小m 使得} g(n_1, \ldots, n_k, m) = 1 \\ 0 \text{ 否则} \end{cases}$$

我们记为 $\mu m[g(n_1, \ldots, n_k, m)) = 1]$.

极小化的引入本质上是给了我们对于一个可能不会终止的问题的一个确定性答案, 因为如果存在这样的 m, 我们保证可以将其枚举出来, 但是如果不存在的话什么都不会发生, 我们会一直计算下去.

而如果这个过程是一定能够终止的, 我们就称这个函数是**可极小化的**.

例 3.5. $\log(m, n) = \left\lceil \log_{(m+2)}(n + 1) \right\rceil$ 可定义为 $\mu p[greater - than - or - equal((m+1)^p, n+1)]$

定义 3.9. 函数是 μ-可递归的, 若它可以由基本函数经过复合, 递归, 和极小化后生成的.

定理 3.2. 函数 $f:\mathbb{N}^k\to\mathbb{N}$ 是 μ-递归的, 当且仅当它是递归的.

证明. ($\Rightarrow$) 若函数是 μ-递归的, 那么它是图灵可计算的. 这并不困难, 因为根据定义, 我们只需要按照基本函数的相应规则计算即可, 可极小化保证了这个过程是会终止的.

($\Leftarrow$) 假设图灵机 $M=(K,\sum,\delta,s,\{h\})$ 计算函数 $f:\mathbb{N}\to\mathbb{N}$, 我们要证明 f 是 μ-递归的. 令 $b=|\sum|+|K|$, 然后我们构造映射 $E:\sum\cup K\to\{0,1,\ldots,b-1\}$, 使得 $E(0)=0,E(1)=1$. 使用这个映射, 我们可以用 b 进制表示 M 的格局. 格局 $q,a_1a_2\ldots\underline{a_k}\ldots a_n$ 可以唯一编码为

$$\mathbf{E}(a_1)\,b^n+\mathbf{E}(a_2)\,b^{n-1}+\cdots+\mathbf{E}(a_k)\,b^{n-k+1}+\mathbf{E}(q)b^{n-k}+\mathbf{E}(a_{k+1})\,b^{n-k-1}+\cdots+\mathbf{E}(a_n)$$

我们构造 μ-递归函数

$$f(n)=num(output(last(comp(n))))$$

其各函数意义与定义如下:

1. *comp* 是表示一列从初始状态格局到最后停机的格局列.
   $\mathrm{comp}(n)=\mu m[i\,\mathrm{iscomp}(m,n)\text{ and halted }(\mathrm{last}(m))]$
2. *last* 即为从上面的格局序列中找到最后一个格局.
   $\mathrm{lastpos}(n)=\mu m[\text{ equal }(\mathrm{digit}(m,n,b),\mathbb{E}(\triangleright))\text{ or equal }(m,n)]$
3. *output* 扔掉 $\triangleright\sqcup hw$ 中前面三个, 只需要取余数即可. $\mathrm{output}(n)=\mathrm{rem}(n,b\uparrow\log(b\sim 2,n\sim 1)\sim 2)$.
4. *num* 把一个数字转换成二进制的串. $\mathrm{num}(n)=\mathrm{digit}(1,n,b)\cdot 2+\mathrm{digit}(2,n,b)\cdot 2\uparrow 2+\cdots$.

事实上我们还没有完全讲清楚定义的细节, 这里略. □

4 不可判定性

终于我们来到了本课程最抽象的一章, 不可判定性. 在本章, 我们将会证明图灵机/算法能解决的问题很少, 而有相当多的问题都是算法解决不了的. 让我们从停机问题开始说起.

停机问题: 判断一个程序在给定输入上是否会终止.

```
bool Halt(p); // 假设存在停机问题判断算法

void Evil() // 我们可以构造出一个邪恶的程序
{
    if (!Halt(Evil)) return;
    else while(1);
}
```

Halt(Evil) 的返回值是什么?

- 如果为真, 则 Evil 不停机, 矛盾.
- 如果为假, 则 Evil 停机, 矛盾.

本章我们会在图灵机的范畴下严格的讨论停机问题, 证明大量的语言都是不可判定的.

4.1 广义图灵机

想要研究图灵机的性质和行为, 一个重要的方法是要对图灵机进行编码, 让 “图灵机” 本身也能作为输入, 但一个问题是图灵机可以使用的状态和字母表都是没有限制的, 该怎么表示一个通用的图灵机呢?

答案也很简单, 把对图灵机编码方式也作为图灵机的一部分, 下面的图道尽一切.

state	symbol	δ
s	a	$(q, \sqcup)$
s	$\sqcup$	$(h, \sqcup)$
s	$\triangleright$	$(s, \rightarrow)$
q	a	(s, a)
q	$\sqcup$	$(s, \rightarrow)$
q	$\triangleright$	$(q, \rightarrow)$

可以编码为

state/symbol	representation
s	$q00$
q	$q01$
h	$q11$
$\sqcup$	$a000$
$\triangleright$	$a001$
$\leftarrow$	$a010$
$\rightarrow$	$a011$
a	$a100$

$$\begin{aligned} \text{“}M'' = &(q00, a100, q01, a000), (q00, a000, q11, a000), \\ &(q00, a001, q00, a011), (q01, a100, q00, a011)) \\ &(q01, a000, q00, a011), (q01, a001, q01, a011) \end{aligned}$$

这也就引出了广义图灵机, universal Turing machine: UTM 对于给定的图灵机 M 和输入 w(编码后), 满足性质:

$$U(\text{``}M''\text{``}w'') = \text{``}M(w).''$$

经过我们前面的编码, 这样的图灵机的构造并不困难, 只需要一步一步的在编码好的图灵机上模拟过程即可.

有了 UTM, 我们可以来形式化的描述停机问题了:

定理 4.1. *令 $H = \{\text{``}M''\text{``}w'' \mid M$ 在输入 w 上终止$\}$ 那么语言 H 不是递归的, 因此递归语言是递归可枚举语言的真子集.*

证明. 递归可枚举是显然的, 因为我们 UTM 已经可以半判定 H 了.

仿照我们在通俗的停机问题中把程序的输入喂给自己一样, 我们也要把图灵机作为输入交给自己判断. 而由于我们已经知道该怎么对图灵机进行编码了, 这是容易的. 假设 H 是递归语言, 我们知道 H1 = { "M"| M 在输入 "M" 下停机 } 也是显然递归的. 而我们已经证明了递归语言的补还是递归语言, 因此 H2 = { "M"| M 在输入 "M" 下不停机 } 也是递归的.

矛盾已经出现了, H2 甚至不是递归可枚举语言. 假若存在图灵机 M^* 半判定 H2, 那么 ``$M^*$'' 属于 H2 吗? 如果属于, 说明其自己作为输入不停机, 但是 $M^*$ 是半判定 H2 的, 说明如果串属于 H2, 那就一定会停机, 那么串是 ``M^*'' 自己也一样, 这是矛盾的! ``M^*'' 不属于 H2 时矛盾是类似的.

综上所述, H 不是递归语言. □

TMs	"M_1''	"M_2''	"M_3''	"M_4''	...	"D"
M_1	**接受**	拒绝	接受	拒绝	...	
M_2	拒绝	**拒绝**	拒绝	拒绝		
M_3	接受	接受	**接受**	接受	...	
M_4	拒绝	拒绝	接受	**接受**		
⋮			⋮			
D	**拒绝**	**接受**	**拒绝**	**拒绝**		**矛盾!**

表 1: 对角线证明

注记 4.1. *语言的个数是不可数的 (2^{Σ^*}), 而图灵机的个数是可数的 (将图灵机编码成字符串), 因此绝大多数语言都是不可判定的, 即不存在图灵机判定它 (核心是要保证停机), 也就没有我们所谓的算法.*

4.2 规约与里斯定理

定义 4.1 (规约)**.** *令 $L_1, L_2 \subseteq \sum^*$ 为两个语言, 一个从 L_1 到 L_2 的***规约*** 是一个递归函数 $\tau : \sum^* \to \sum^*$ 使得 $x \in L_1$ 当且仅当 $\tau(x) \in L_2$.*

A 可以规约到 B, 说明一个对于 B 的解答就可以解答 A, 这告诉我们两件事:

- 如果 B 是简单的, 那么 A 也是简单的.
- 如果 A 是困难的, 那么 B 也是困难的.

定义 4.2. *我们称图灵机 M 枚举语言 L 当且仅当对于一些 M 中固定的状态 q,*

$$L = \{w : (s, \triangleright\sqcup) \vdash_M^* (q, \triangleright \sqcup w)\}$$

这样的语言称为图灵可枚举的.

也就是说, M 从空纸带开始枚举 L, 周期性的经过 q 状态, 然后每次提取生成的所有串. 注意 L 中的元素可以重复出现, 也没有顺序要求.

定理 4.2. 一个语言是递归可枚举的, 当且仅当它是图灵可枚举的.

定义 4.3. 令 M 是图灵机, 图灵可枚举语言 L. 我们称 M 按词典序枚举, 若枚举出现的字符是按词典序排列的.

定理 4.3. 一个语言是递归的, 当且仅当它是词典序下图灵可枚举的.

这两个概念我个人认为没有太大的意思, 因此定理的证明也就略过了. 我们来到最后一个重要的定理, 它实际上告诉了我们一件非常重要的事情: 所有非平凡的性质都是不可判定的.

定理 4.4 (Rice's Theorem)**.** 令 $\mathcal{C}$ 为整个递归可枚举语言类的一个非空真子集, 那么下列问题是不可判定的:

给定图灵机 M, $L(M) \in \mathcal{C}$?

证明. 不妨假设 $\emptyset \notin \mathcal{C}$, 我们知道存在语言 $L \in \mathcal{C}$ 被图灵机 M_L 半判定.

我们可以把停机问题规约到判定被给定图灵机半判定的语言是否属于 $\mathcal{C}$ 上. 假设我们给定一个图灵机 M, 输入 w, 我们想要判定 M 是否在 w 上停机. 为了解决这个问题, 我们可以构造图灵机 $T_{M,w}$, 使得被 $T_{M,w}$ 半判定的语言要么是 L, 要么是空语言. 在输入 x 上, $T_{M,w}$ 首先模拟 UTM 在输入 "M""w". 若果它停机, $T_{M,w}$ 继续模仿 M_L 在输入 x 上.

我们断言 $T_{M,w}$ 半判定的语言属于 $\mathcal{C}$ 当且仅当 M 在 w 上停机. 这实质上是停机问题到我们上面所说的问题到规约. □

4.3 自动机的判定性问题

有了上面的准备, 我们可以回答一些自动机的判定问题, 尤其是不可判定的:

定理 4.5 (自动机的判定性问题)**.**

1. $A_{DFA} = \{$ "B""w" | B 是 DFA 且 B 接受 w $\}$: 可判定
2. $A_{NFA} = \{$ "B""w" | B 是 NFA 且 B 接受 w $\}$: 可判定
3. $E_{DFA} = \{$ "B" | B 是 DFA 且 $L(B) = \emptyset$ $\}$: 可判定
4. $EQ_{DFA} = \{$ "A""B" | A, B 是 DFA 且 $L(A) = L(B)$ $\}$: 可判定
5. $A_{CFG} = \{$ "G""w" | G 是 CFG 且 G 接受 w $\}$: 可判定
6. $E_{CFG} = \{$ "G" | G 是 CFG 且 $L(B) = \emptyset$ $\}$: 可判定
7. $ALL_{CFG} = \{$ "G" | G 是 CFG 且 $L(G) = \sum^*\}$: **不可判定**
8. $EQ_{CFG} = \{$ "A""B" | A, B 是 CFG 且 $L(A) = L(B)$ $\}$: **不可判定**
9. $A_{TM} = \{$ "M""w" | M 是 TM 且 M 接受 w $\}$: **可识别不可判定**

证明. 1, 2, 3 的证明留给读者. 利用 3 可得证 4, 即语言 C 使得 $L(C) = (L(A) \cap \overline{L(B)}) \cup (\overline{L(A)} \cap L(B))$, 由正则语言在补交并下封闭我们知道 C 也是正则语言, 则 C 存在 DFA, 将该语言输入 E_{DFA} 即可.

5, 6 的证明同样留给读者, 注意 5 的证明需要在一个化简的 CFG 上进行从而保证终止, 7 的证明是下一节的内容. 而证明了 7 是不可判定之后, 很自然可以发现 8 也是不可判定的, 因为 $\sum^*$ 是 CFG, 所以如果存在图灵机可以判定 8, 显然有图灵机可以判定 7 (这其实就是我们前面说的规约), 由此 8 是不可判定的. 9 是我们本章一开始就已经证明. □

4.4 Computation History Method*

计算历史方法是一类重要的证明某些语言的不可判定性的方法.

定义 4.4. *假设有图灵机 M 和输入 w. 一个被接受的计算历史即位一串格局 $C_1, C_2, \ldots, C_l$, 其中 C_1 是起始格局, C_l 是处于接受状态的格局, 而且每一个格局的演变都是合法的.*

记得我们上一节要证明的 $ALL_{\text{CFG}} = \{$ "G" | G 是 CFG 且 L(G) $= \sum^*\}$ 是不可判定问题, 现在我们就使用本节的计算历史方法来证明这一点:

例 4.1. *我们假设存在图灵机可以判定 ALL_{CFG}, 那么我们就可以利用类似规约不过是在计算历史上的方法来构造可以判定停机问题的图灵机. 对于图灵机 M 和输入 w, 我们构造 CFG G, 当且仅当 M 不接受 w 时生成所有串, 那么如果 M 接受 w, G 一定会生成不了某些特定的串, 那个串就是 M 在 w 上的计算历史. 即 G 生成那些不是 M 在 w 上的可接受计算历史的串. 一个接受的计算历史表示为 $\#C_1\#C_2 \ldots \#C_l\#$.*

为了构造这样的 CFG, 我们选择使用 PDA, 首先检测输入是不是初始格局, 如果不是就接受. 其次检测输入的最后是不是接受格局, 如果不是就接受. 最后我们要判断的是某一次格局的推导是不是合法的, 如果不是同样也要接受. 而 PDA 的非确定性就在这里起作用, 判断两个相邻的格局是否是合法就来源于图灵机的转换函数, 这并不困难.

由此, 我们说明了如果可以判断 CFG 会不会生成所有串, 就能判定停机问题, 这就把停机问题规约到了 ALL_{CFG}, 也就完成了证明.

最后我们介绍一个简单的不可判定问题, 相对于我们前面所举的诸如停机问题这种看起来很奇怪的例子, 这个问题十分的简单, 而且和自动机的语言没有什么关系, 也就是著名的 Post Correspondence Problem, 即 PCP.

我们从一个多米诺骨牌讲起, 下面是一列多米诺骨牌, 注意这个骨牌的独特之处是它的上下都有一些串, 上面的串是 t_i, 下面的串是 k_i.

$$P = \left\{ \left[\begin{array}{c} t_1 \\ b_k \end{array} \right], \left[\begin{array}{c} t_2 \\ b_2 \end{array} \right], \ldots, \left[\begin{array}{c} t_k \\ b_k \end{array} \right] \right\}$$

在给定了一组骨牌之后, 我们可以挑出任意多个拼接在一起, 如果上面的串连接起来和下面的串连接起来恰好相同, 我们就称为这是一个匹配.

例 4.2. *我们举一个有匹配的例子和一个没有匹配的例子:*

1. 有匹配:

$$P = \left\{ \left[\begin{array}{c} ab \\ aba \end{array} \right], \left[\begin{array}{c} aa \\ aba \end{array} \right], \left[\begin{array}{c} ba \\ aa \end{array} \right], \left[\begin{array}{c} abab \\ b \end{array} \right] \right\}$$

记这些骨牌为 1, 2, 3, 4. 而 12324 记为一个匹配.

2. 无匹配:

$$P_1 = \left\{ \left[\begin{array}{c} aa \\ aaba \end{array} \right], \left[\begin{array}{c} ba \\ ab \end{array} \right], \left[\begin{array}{c} ab \\ ba \end{array} \right] \right\}$$

那么一个自然的问题是, 给定骨牌之后, 我们能不能判断存不存在匹配呢? 即 PCP $= \{$ "P": P 是一个 PCP, 且 P 存在匹配 $\}$. 神奇的事, 这个看起来很简单的问题实际上是不可判定的:

定理 4.6. *PCP 是不可判定的.*

证明. 我们概述证明. 十分感兴趣的读者可以参照 Michael Sipser 的《Introduction to the theory of computation》(third edition). 本质上我们也是通过计算历史把这个问题规约到停机问题. 我们将证明任给一个图灵机 M 和输入 w, 我们可以构造一个 PCP 实例, 其中的匹配代表了 M 对 m 的接受计算历史. 如果该实例可以判定, 那么停机问题就也可以判定. 实际操作中, 我们要构造的骨牌都是由图灵的格局构成的, 骨牌的匹配就由计算历史的模拟进行. □

5 结语

所有内容都已经结束了, 回首我们学习到的 DFA/NFA, CFG/PDA, TM 与不可判定性, 我们完成了计算理论的基础内容. 最后我们对我认为重要的概念做一个梳理和总结:

1. 字母表: 有限的字符集合.
   串: 由字母表里的字符构造的有限序列.
   语言: 串的有限/无限集合.

2. 自动机的确定性/非确定性与状态转移函数/状态转移关系的对应.

3. 串被自动机**接受**, 当且仅当存在一个推导使得自动机落在终止状态且串被消耗完. 所有被该自动机接受的串的集合, 称为被该自动机接受的语言.

4. 自动机与自动机 (DFA, NFA), 自动机与语法 (PDA, CFG) 是**等价**的, 意思是说: 任给一个自动机的实例, 它有对应的接受的语言, 都能找到一个语法的实例, 它恰好能生成这个语言, 反之亦然.

5. Pumping Theorem 是正则语言/上下文无关语言的性质, 因此由逆否命题我们可以通过证明 Pumping Theorem 在某一语言上不成立来证明这个语言不是正则语言/上下文无关语言.

6. 上下文无关文法的歧义由同一个串的多个语法树/最左推导得到. 语法树可以看作推导的等价类.

7. 图灵机接受/拒绝一个串当且仅当在这个串上可以推导到接受格局/拒绝格局, 注意图灵机有可能不停机也就是永远无法进入接受/拒绝格局.
   图灵机**判定**一个语言当且仅当串属于语言时, 接受; 串不属于语言, 拒绝.
   图灵机**半判定**一个语言当且仅当串属于该语言时, 图灵机停机.

8. 一个语言称为**递归语言**当且仅当存在图灵机判定该语言.
   一个语言称为**递归可枚举**当且仅当存在图灵机半判定该语言.

9. 原始递归函数由基础函数 (零元函数, 等价函数, 后继函数) 经过复合和递归能得到的所有函数组成. 注意 μ-递归才是递归的, 即存在图灵机判定.

10. A 规约到 B 说明到是 A 比 B 难, 即如果 B 不可判定, 那么 A 不可判定. 证明某一个语言不可判定的经典方法是规约到停机问题.

11. 图灵可枚举等价于递归可枚举; 图灵词典序可枚举等价于递归.

下面列出一些我认为必须要掌握的方法/技术:

1. 正则语言转 NFA 转 DFA.

2. CFG 转 PDA.

语言类型	封闭性
正则语言	并, 交, 补, 连接, Kleene 星号
上下文无关语言	并, 连接, Kleene 星号
递归可枚举语言	并, 交, 连接, Kleene 星号
递归语言	并, 交, 补, 连接, Kleene 星号

表 2: 语言封闭性

文法类型	语言类型	自动机
正则文法	正则语言	有限状态自动机
上下文无关文法	上下文无关语言	下推自动机
上下文有关文法	上下文有关语言	线性有界自动机
任意文法	递归可枚举语言	图灵机

表 3: 乔姆斯基文法分类

3. 利用 Pumping Theorem 证明某些语言不是正则语言/上下文无关语言.

4. 证明某些函数是原始递归的, 即利用原始递归定义将其构造出.

5. 利用规约到停机问题证明某些问题是不可判定的.