编译原理

Compilers

极夜酱

目录

1 有限状态自动机 **1**

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

2 上下文无关语言 **14**

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

3 图灵机 **29**

3.1 图灵机 29

3.2 停机问题 33

4 代码生成 **35**

4.1 类型检查 35

4.2 运行时环境 37

4.3 中间代码生成 39

4.4 代码优化 42

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 Σ 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。
- $\Sigma = \{a, b, \cdots, z\}$：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 $\Sigma = \{a, b, \cdots, z\}$ 上的串。
- (()) 和 (() 是 $\Sigma = \{(,), [,], \{, \}\}$ 上的串。

空串

空串使用 ϵ 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）

Σ^k 用于表示所有在字母表 Σ 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 Σ^* 用于表示所有在字母表 Σ 上能够组成的串的集合。

$$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \cdots = \bigcup_{k \geq 0} \Sigma^k \tag{1.1}$$

正闭包 Σ^+ 则是在 Σ^* 中除了空串以外的所有串的集合。

$$\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \cdots = \bigcup_{k > 0} \Sigma^k \tag{1.2}$$

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，$\Sigma = \{a, b, c, \cdots, z\}$，那么所有英语单词所构成的集合 L 就是字母表 Σ 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）

$$A \cup B = \{x \mid x \in A \text{ or } x \in B\} \tag{1.3}$$

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

连接（concatenation）

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

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

闭包

$$A^* = \{x_1, x_2, \cdots, x_k \mid k \geq 0 \text{ and each } x_i \in A\} \tag{1.5}$$

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

语法和语言与自动机理论密切相关，它们是许多软件实现的基础，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。

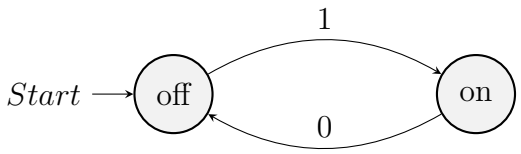


图 1.1: 有限状态机

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
- Σ：字母表
- δ：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：

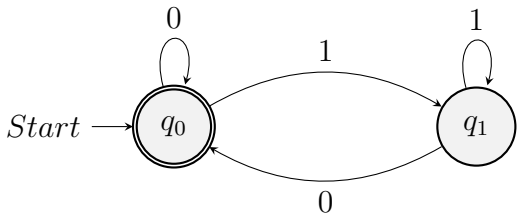


图 1.2: 识别空串或以 0 结尾的串的 DFA

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

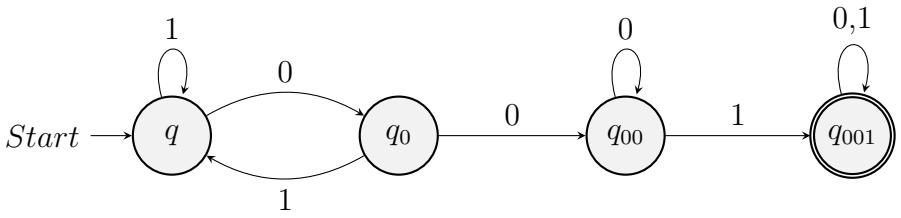
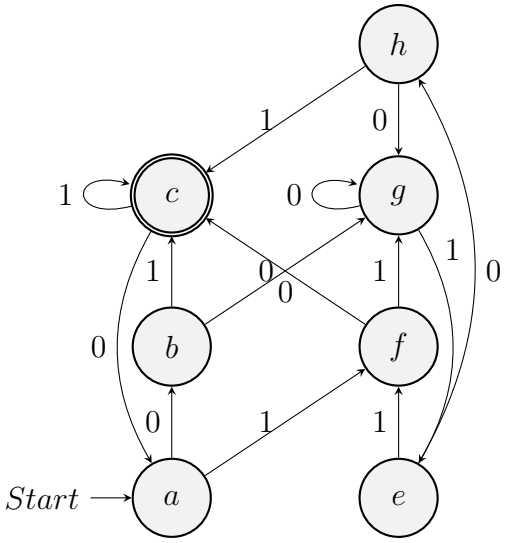


图 1.3: 识别包含子串 001 的串的 DFA

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。

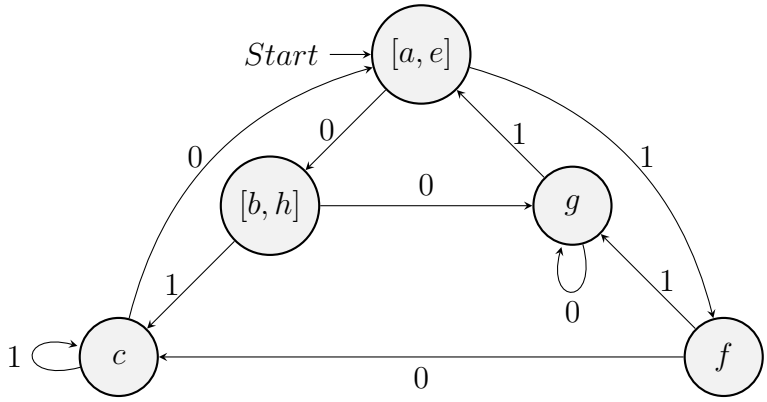


图 1.4: 最小化 DFA

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。

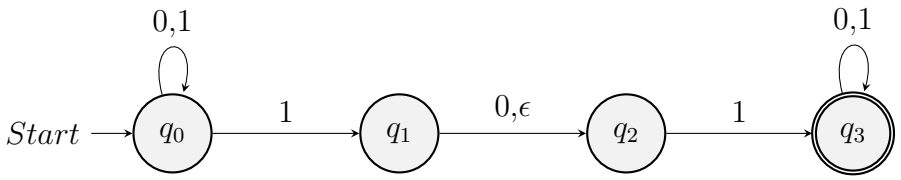


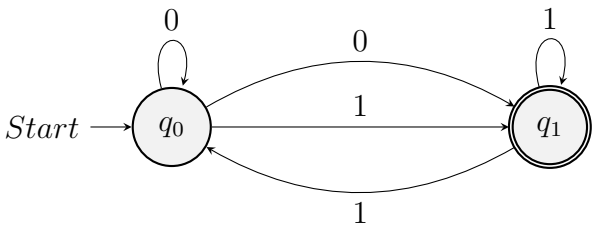
图 1.5: NFA

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。

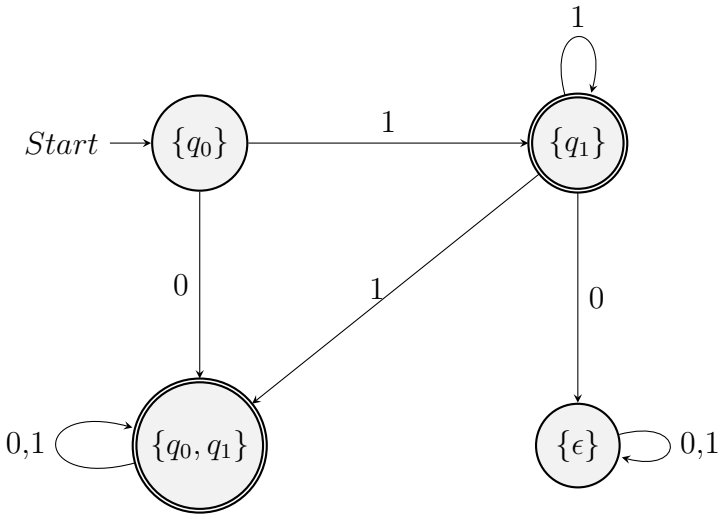


图 1.6: NFA 转换 DFA

1.4.3 ϵ-NFA

ϵ-NFA 允许不消耗输入字符在状态之间转移。

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

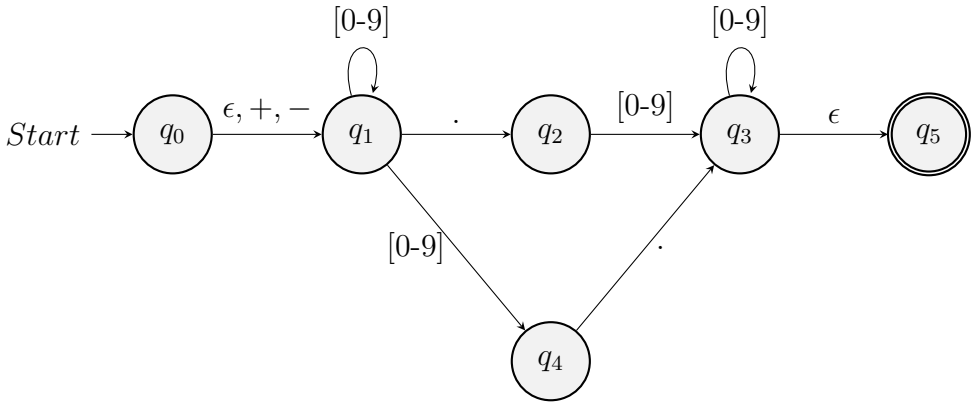


图 1.7: 接受小数的 ϵ-NFA

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（syntex 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 和语法书（syntex 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)

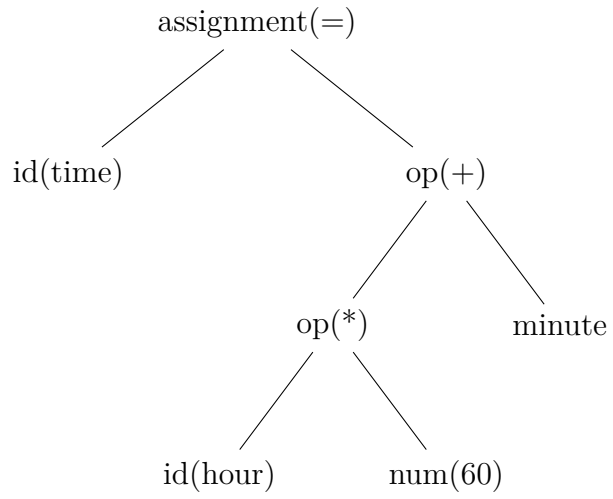


图 1.8: 语法树

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：$a^* = \{\epsilon, a, aa, aaa, \cdots\}$
- 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 $\{a^nb^n \mid n \geq 0\}$ 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

$$A \to \alpha_1$$
$$A \to \alpha_2$$
$$\cdots$$
$$A \to \alpha_k$$

可以被写成 $A \to \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_k$ 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 $E \to E + E \mid E * E \mid (E) \mid id$，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。

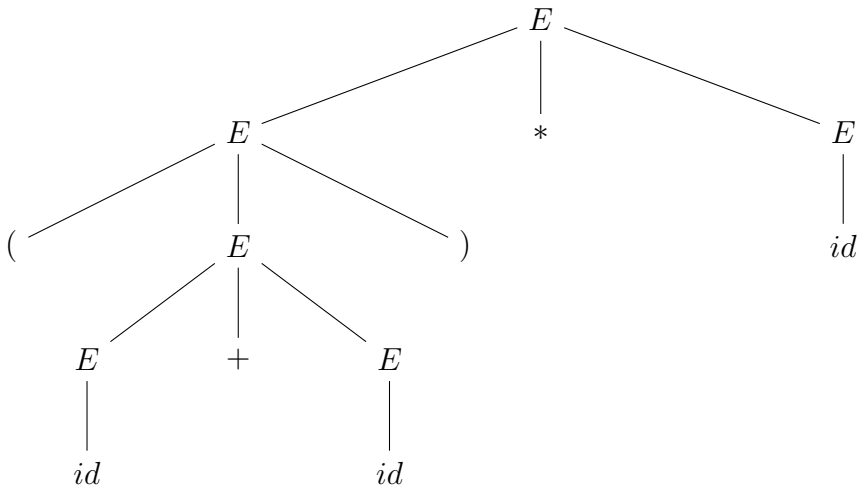


图 2.1: 分析树

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 syntex tree）。

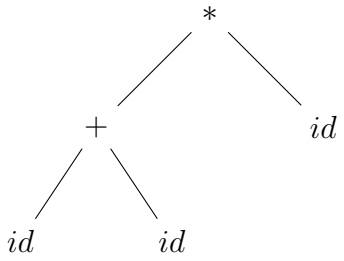


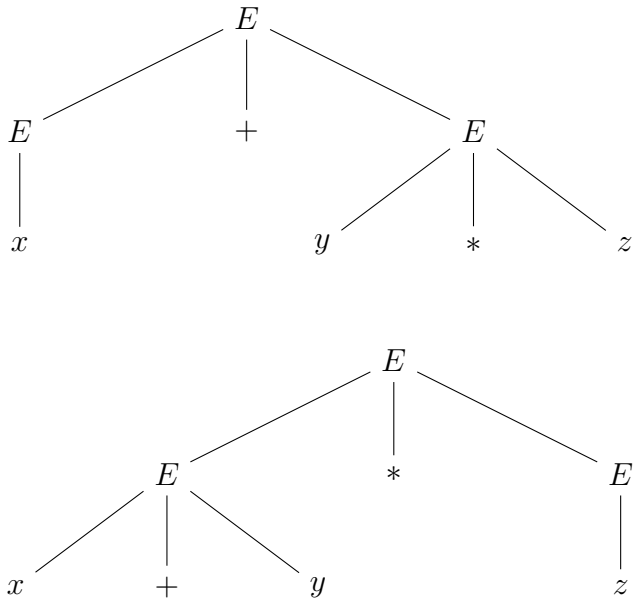
图 2.2: 抽象语法树

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

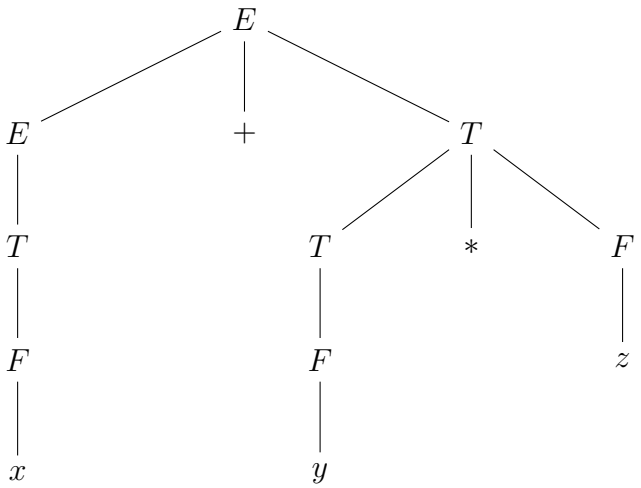


产生二义性的原因在于运算符之间的优先级在文法中并没有体现。消除二义性的办法就是在文法中引入一个中间量。

$$E \to E + T \mid T$$

$$T \to T * F \mid F$$

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



2.2 CNF

2.2.1 上下文无关语言

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

$$S \to aSb$$
$$S \to ab$$

例如根据 S 可以生成生成 aaabbb：

$$\begin{aligned} S &\Rightarrow aSb \\ &\Rightarrow aaSbb \\ &\Rightarrow aaabbb \end{aligned}$$

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

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

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：

$$S_1 \to 0S_1 1 \mid \epsilon$$
$$S_2 \to 1S_2 0 \mid \epsilon$$

只需合并这两个部分，即可得到最终的 CFG：

$$S \to S_1 \mid S_2$$

$$S_1 \to 0S_1 1 \mid \epsilon$$

$$S_2 \to 1S_2 0 \mid \epsilon$$

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. $S \to \epsilon$：开始变元 S 可以为空。
2. $A \to BC$：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. $A \to a$：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 ϵ 规则：消除从变元到空字符的规则。
3. 消除所有 $A \to B$ 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 $A \to BC$ 的规则，需要将 $A \to BCD$ 替换为 $A \to ED$，即添加变元 $E \to BC$。

例如将 CFG 转换为 CNF：

$$S \to ABA$$

$$A \to aA \mid \epsilon$$

$$B \to bB \mid \epsilon$$

消除所有 ϵ 规则

将 $A \to \epsilon$ 的规则，替换到出现 A 的规则中：

$$
\begin{aligned}
S &\to ABA \mid BA \mid AB \mid B \\
A &\to aA \mid a \\
B &\to bB \mid \epsilon
\end{aligned}
$$

将 $B \to \epsilon$ 的规则，替换到出现 B 的规则中：

$$
\begin{aligned}
S &\to ABA \mid BA \mid AB \mid B \mid AA \mid A \\
A &\to aA \mid a \\
B &\to bB \mid b
\end{aligned}
$$

消除所有 $A \to B$ 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：

$$
\begin{aligned}
S &\to ABA \mid BA \mid AB \mid bB \mid b \mid AA \mid aA \mid a \\
A &\to aA \mid a \\
B &\to bB \mid b
\end{aligned}
$$

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

$$
\begin{aligned}
S &\to ABA \mid BA \mid AB \mid bB \mid b \mid AA \mid aA \mid a \\
A &\to aA \mid a \\
B &\to bB \mid b
\end{aligned}
$$

添加变元

为了消除 $A \to BCD$ 这种情况，需要添加新的变元进行替换。

假设 $X \to AB$：

$$S \to XA \mid BA \mid AB \mid bB \mid b \mid AA \mid aA \mid a$$

$$A \to aA \mid a$$

$$B \to bB \mid b$$

$$X \to AB$$

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

假设 $A_1 \to a$、$B_1 \to b$:

$$S \to XA \mid BA \mid AB \mid B_1B \mid b \mid AA \mid A_1A \mid a$$

$$A \to A_1A \mid a$$

$$B \to B_1B \mid b$$

$$X \to AB$$

$$A_1 \to a$$

$$B_1 \to b$$

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。

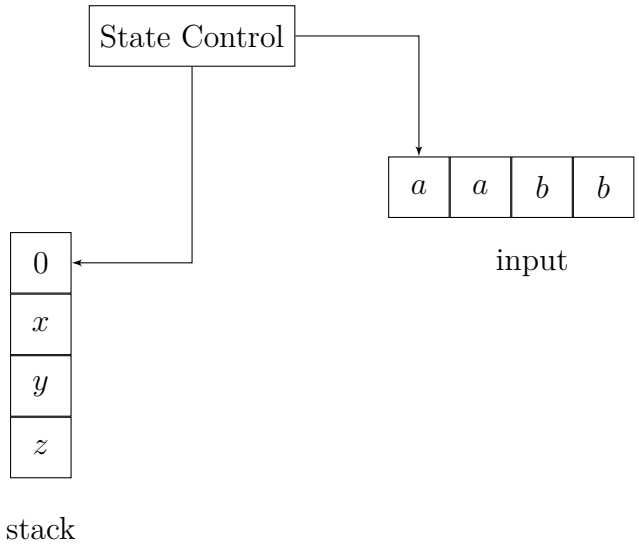


图 2.3: PDA

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

- Q：状态集合
- Σ：输入字母表
- Γ：栈字母表
- δ：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 $a, b \to \epsilon$。

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：

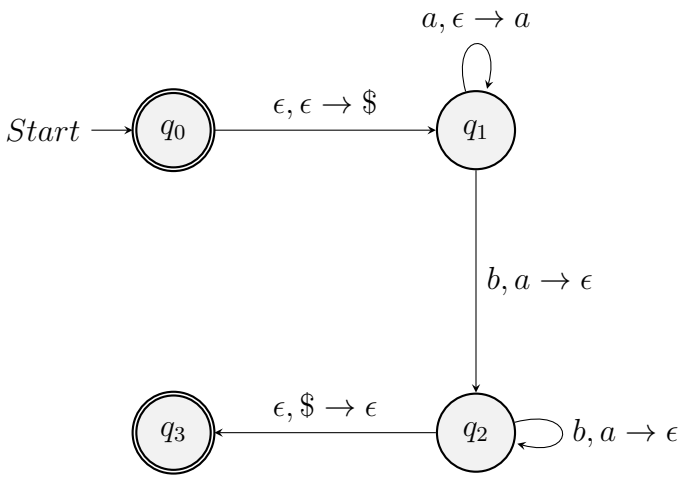
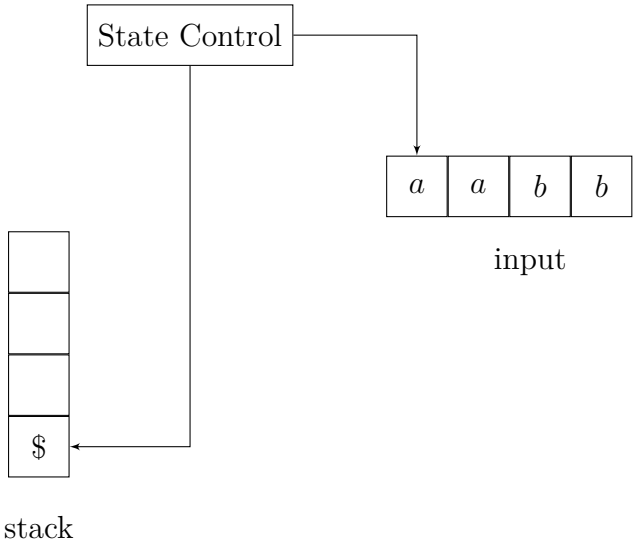
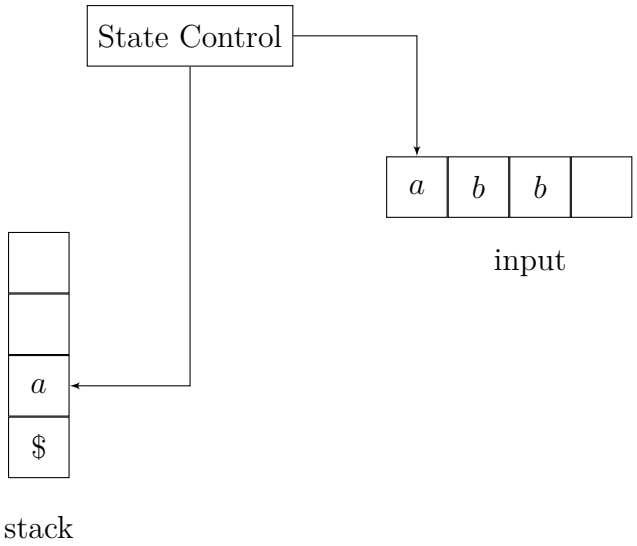
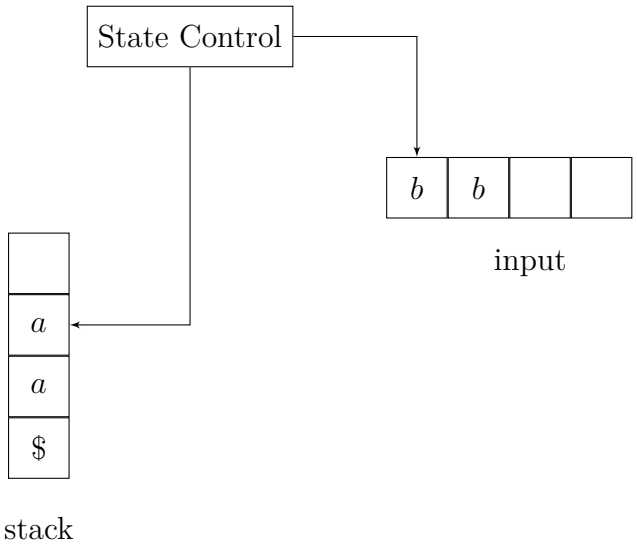


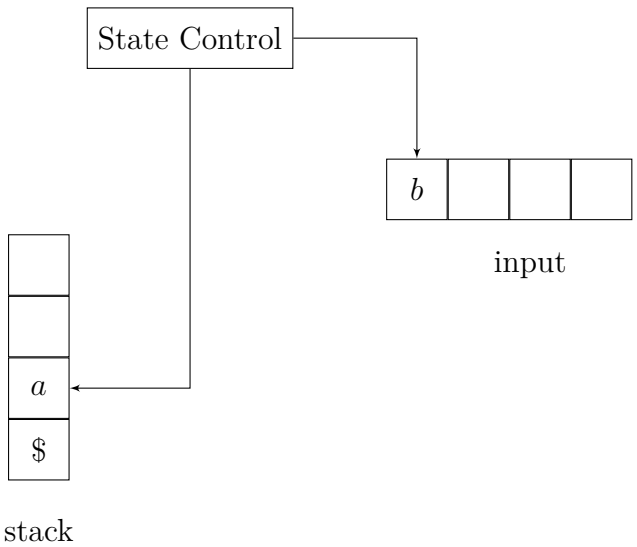
图 2.4: 识别 $\{a^n b^n \mid n \geq 0\}$ 的 PDA

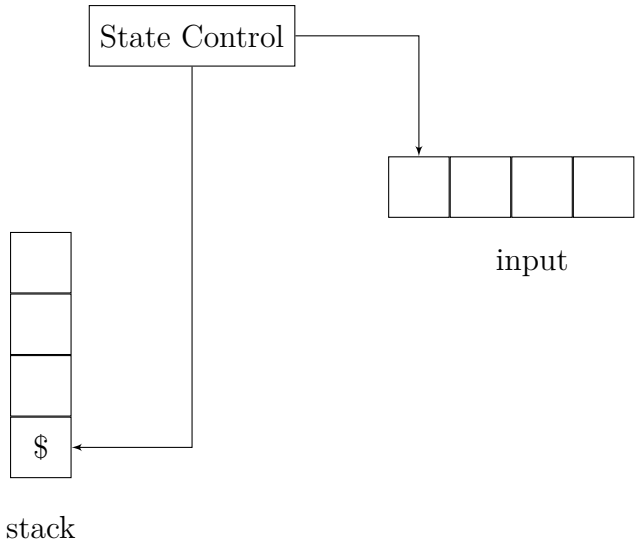
识别串 $aabb$ 的过程如下：

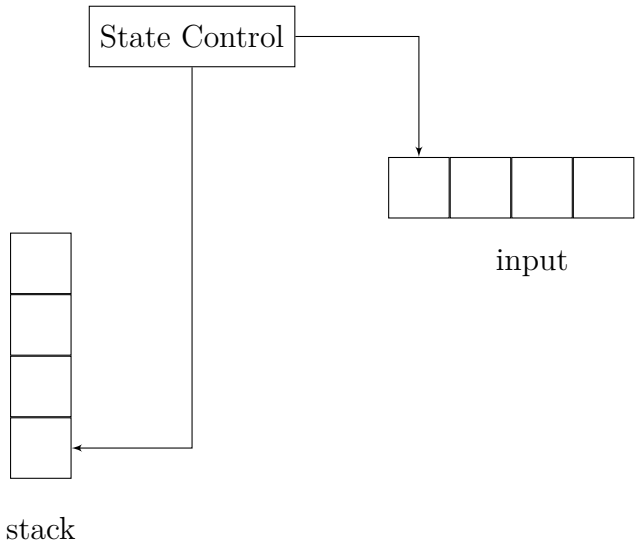



State Control
a
b
b
input
a
$
stack


State Control
b
b
input
a
a
$
stack


State Control
b
input
a
$
stack


State Control
input
$
stack


State Control
input
stack

2.4 预测分析法

2.4.1 LL(1) 文法

例如产生式 $A \to +T \mid -P$，当读到 + 开头的串的时候，可以很直接地判断选择 $A \to +T$ 这个生成式；而读到 − 开头的串的时候，可以直接判断选择 $A \to -P$ 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 $A \to T \mid P$ 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 $T \to a \mid b$，$P \to c \mid d$, 那当串以 a 或 b 开头时，显然需要选择 $A \to T$；而当串以 c 或 d 开头时，就应该选择 $A \to P$ 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 α 推导出的出现在开头的终结字符的集合。

例如：

$$
\begin{aligned}
E &\to TE' \\
E' &\to +TE' \mid \epsilon \\
T &\to FT' \\
T' &\to *FT' \mid \epsilon \\
F &\to (E) \mid id
\end{aligned}
$$

每个终结符的 FIRST 集合都是自己本身。

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

如果 $E \to T$，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：

$$A \to Tb \mid P$$
$$T \to \epsilon \mid a$$
$$P \to c$$

可以得出：

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

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

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

所以，为了特殊处理当一个非终结字符可以推出 ϵ 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 $A \to \alpha B \beta$,那么 $FIRST(B)$ 中除了 ϵ 外所有符号都在 $FOLLOW(B)$ 中。
3. 如果存在 $A \to \alpha B$ 或 $A \to \alpha B \beta$ 且 $FIRST(B)$ 中包含 ϵ,那么 $FOLLOW(A)$ 中的所有符号都在 $FOLLOW(B)$ 中。

Chapter 3 图灵机

3.1 图灵机

3.1.1 艾伦 · 麦席森 · 图灵（Alan Mathison Turing）

在 1900 年的巴黎国际数学大会上，数学家希尔伯特（David Hilbert）提出 23 个重要的数学问题，其中第十个是“随便给一个不确定的方程，能够通过有限步的运算，判断它是否存在整数解?”

如果这个问题的答案是否定的，那么意味着有些问题是无解的。正是因为对这个问题的深刻认识，让图灵认识到计算机的能力存在极限。

后来在 1970 年，前苏联伟大的数学家马季亚谢维奇从数学上解决了希尔伯特的那个问题。也就是说，的确有很多数学问题，根本没有答案，而且这样的问题比有答案的问题还要多得多。

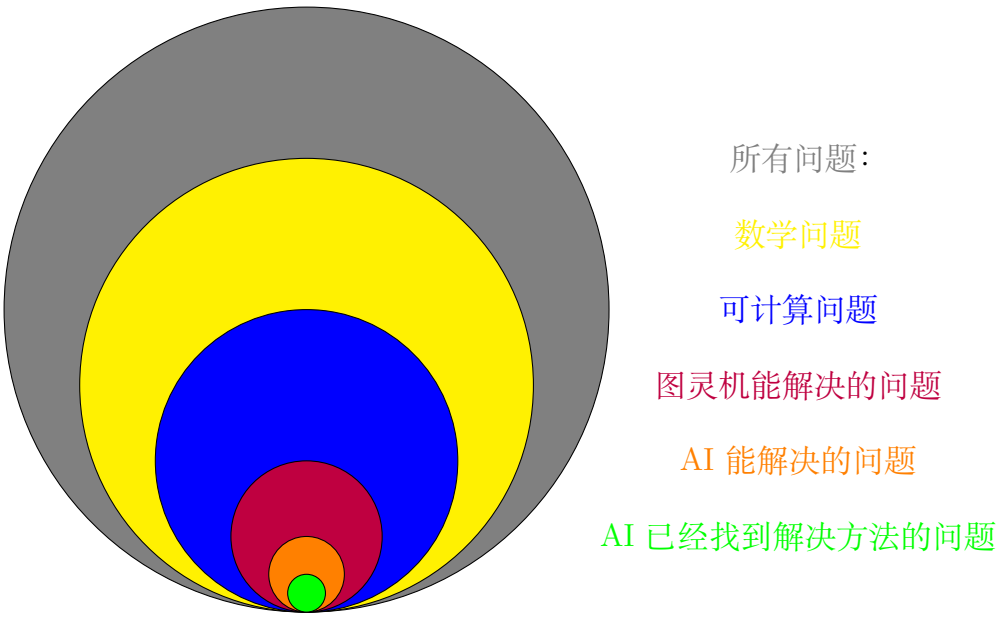


图 3.1: 数学问题的分类

3.1.2 图灵机（Turing Machine）

1935 年，22 岁的图灵写出了 *On computable numbers, with an application to the Entscheidungsproblem*，并从数学和逻辑上定义了著名的图灵机。今天所有的计算机，包括全世界正在设计的新的计算机，从解决问题的能力来讲，都没有超出图灵机的范畴。

图灵机是图灵构想出来的虚拟机器，这个机器的伟大之处在于，它非常简单，但是却可以模拟任何的计算机程序。

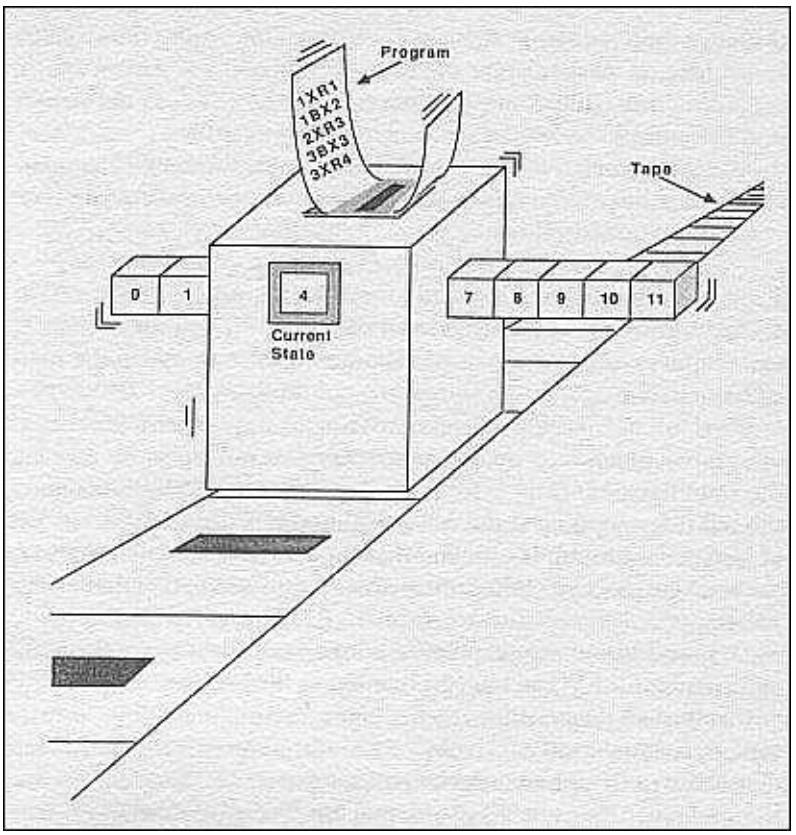


图 3.2: 图灵机

任何能被数字计算机执行的事情，图灵机同样也能够完成。如果你能写出一个算法来解决某个问题，那么同样可以写出一个图灵机程序来解决相同的问题。

图灵机的结构包括：

- 存储带（tape）

  - 双向无限延长
  - 存储带上有一个个小方格，每个小方格里面存储一个符号

- 控制器
  - 可以存储图灵机当前自身的状态
  - 可以改变自身的状态
  - 包含一个读写头（read-write head），可以读、写存储带上方格里面的内容
  - 读写头可以沿着存储带一格一格地左移或者右移。

这里存储带其实就相当于现在计算机的内存，控制器相当于 CPU 和程序代码。

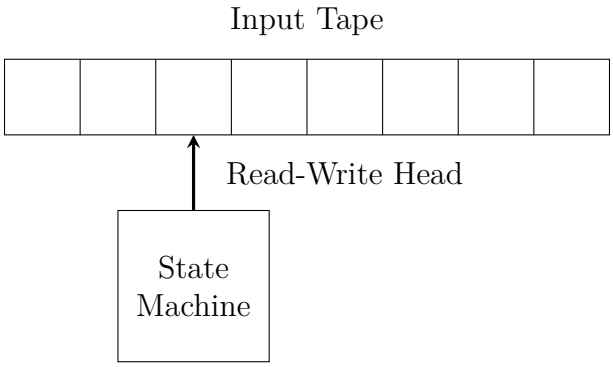
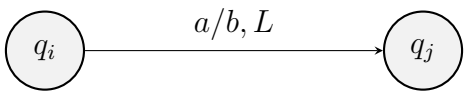


图 3.3: 图灵机

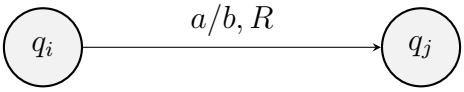
在运行图灵机前，首先需要将输入放入存储带中，然后将图灵机的状态设置为初始状态。开始运行后，图灵机会对存储带上的字符进行读写，当图灵机停止后，结果将会留在存储带上。

δ 是图灵机的状态转换函数。

例如将当前字符 a 替换为 b，并左移一格，可表示为 $\delta(q_i, a) = (q_j, b, L)$。



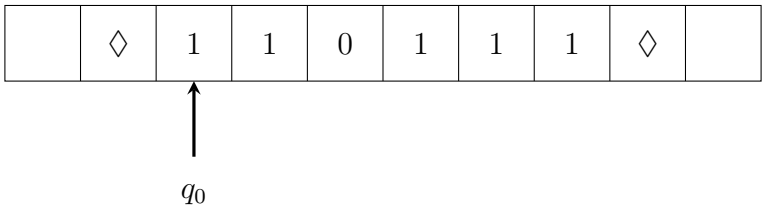
例如将当前字符 a 替换为 b，并右移一格，可表示为 $\delta(q_i, a) = (q_j, b, R)$。



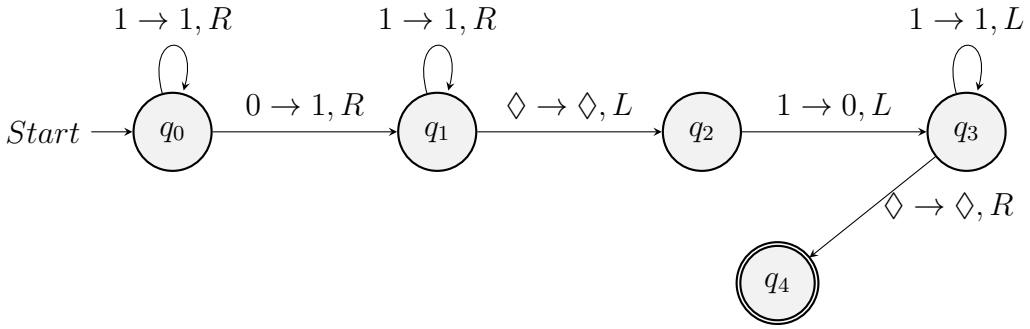
如果 $a = b$，则用 a, L 或 a, R 表示。

例如使用图灵机进行加法运算，为了方便操控图灵机，可将运算数转换为一进制的形式，如 $2 + 3$ 表示为 $11 + 111$。

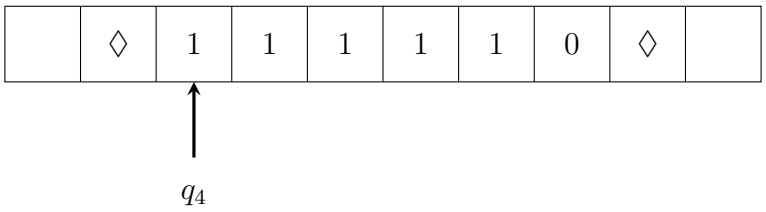
在输入带上放入这两个运算数，中间用 0 作为分割。输入内容的两端放入 $\Diamond$，用于标记开始和结束位置。



用于计算加法的图灵机可表示为：



当图灵机停止后，输入带中的内容即为最终结果。



借助图灵机模拟器http://turingmachine.vassar.edu/EZzvuYfv8可以可视化运行过程。

3.2 停机问题

3.2.1 不可判定性（Undecidability）

世界上存在不可解问题，存在数学和程序都不能抵达的边界。

图灵当年想要证明希尔伯特的可判定性问题，也就是说，是否存在一种通用的机械过程，能够判定任何数学命题的真假。

于是图灵就设计了一种假象的机器，也就是图灵机。他首先证明，图灵机就覆盖了所有的机械过程。如果存在一个问题，图灵机判定不了，那么就说明，不存在这种通用的机械过程，这样就证明了原问题。

然后，图灵就设计了一个问题，确实是图灵机判定不了的，这个问题就是“对于一个输入，让图灵机判定自己是否能够在有限的时间内停下来”。

经过证明，这个问题是图灵机回答不了的，所以原问题得以证否。这个问题被称为停机问题（Halting Problem）。

图灵当时设计这个图灵机，完全只是为了辅助他证明这个问题而已，这个机器是假想的、不存在的。可是后来他又发现，虽然这个机器不能解决所有的问题，但确实能够解决很多问题，而且真的是可以造出来的。

3.2.2 理发师悖论

停机问题类似于理发师悖论。

在一座小岛上有位理发师，有一天他做出一项规定：“他只给岛上所有不自己理发的人理发”。随着这个理发师自己的头发越来越长，他发现自己陷入了一个两难的境地：他该不该给自己理发？

如果他不为自己理发，按照他的规则，他属于自己的服务客户范围，因此可以给

自己理发；如果他选择为自己理发，同样按照规则，他便不属于自己的服务对象，因此他又不能给自己理发。

3.2.3 停机问题

在写代码时，有时会遇到一个程序一直在运行，等了半天毫无反应。但是我们不知道程序是陷入了死循环导致根本不会停止，还是仅仅只是运行时间很久。

如果一个程序能够有限时间内运行完，就认为是可停机的。这个有限时间是个理论的概念，无论是 1 秒还是 200 亿年，只要有终止的时候，就是可停机的。

因此，如果存在一种程序，能够判断一个程序是否可停机，那么就可以解决上面的问题。遗憾的是，不存在这样的一个程序使得其能判断任意程序的停机问题，即停机问题不可判定。

可以利用反证法进行证明。假设存在这样一个程序 H，能够判断任何一个程序是否可以停机，即 H 的判断总是正确的。

那么，我们可以设计一个程序 X，它的功能是让 H 判断 X 本身能否停机，并将结果取反（即如果可以停机则输出不可停机；如果不可停机则输出可停机）。

这样就得到了一个悖论：

1. 如果 H 得出的结果是可停机的，但是 X 输出的结果是不可停机，说明 H 判断错了。
2. 如果 H 得出的结果是不可停机的，但是 X 输出的结果是可停机，说明 H 判断错了。

因此，并不存在 H 这样一个程序，能够判断任意程序的停机问题。

通过动画https://www.youtube.com/watch?v=92WHN-pAFCs可以更直观地理解停机问题。

Chapter 4 代码生成

4.1 类型检查

4.1.1 类型检查（Type Checking）

经过词法分析后，词法分析器将 token 流传递给语法分析器，语法分析器将生成一个语法树。当源代码转换为语法树时，类型检查器起着至关重要的作用。通过查看语法树，您可以判断每种数据类型是否正在处理正确的变量。

类型检查主要是为了判断变量或者参数的实际类型和声明的类型是否匹配。类型检查可以及早地发现类型不匹配的问题，不要等到执行的时候才发现问题。提前发现问题可以降低成本。

每种语言都有自己的语言类型规则集，编译器必须检查源程序是否遵循语言的句法和语义约定。它限制程序员在某些情况下可以使用的类型并将类型分配给值。

编译器需要检查对象的类型并在违反的情况下报告类型错误，并纠正不正确的类型。

如果要由编译器自动完成从一种类型到另一种类型的转换，则称为隐式转换（implicit conversion）。例如整数可以转换为实数，而实数不能转换为整数。而需要由程序员明确指定的类型转换被称为显示类型转换（explicit conversion），也称为强制类型转换。

4.1.2 静态类型检查（Staic Type Checking）

静态类型检查发生在编译期间，它在编译时检查变量的类型，这意味着变量的类型在编译时是已知的。

静态类型检查包括：

- 类型检查：如果将运算符应用于不匹配的操作数，编译器需要报告错误。例如将一个数组变量和函数相加。
- 控制流检查：一些能够导致离开某个控制结构的语句，应该存在能够被转移到的位置。例如 C 中的 break 语句会导致离开离它最近的 while、for 或 switch 结构，如果这样的结构不存在，则会发生错误。
- 唯一性检查：在某些情况下，对象只可以定义一次。例如 Pascal 中标识符必须唯一声明，case 语句中的标签必须是不同的。
- 名称检查：有时同一个名称可能会出现两次或多次。例如 Ada 中循环的名称可能出现在结构的开头和结尾。编译器必须检查两个地方是否使用了相同的名称。

静态类型检查能够有效发现语法错误和错误的名称，例如额外的标点符号或写错了预定义的名称。对于函数而言，可以检查函数参数的类型和数量是否匹配。

4.1.3 动态类型检查（Dynamic Type Checking）

动态类型检查发生在运行时，即当涉及到具体的数据值时才进行类型检查。动态类型检查提供了更宽松、灵活的程序设计环境，在交互式语言中十分有用。

动态类型语言一般是脚本语言，如 Perl、Ruby、Python、PHP、JavaScript 等，可以更快地编写代码，不必每次都指定类型。

但是动态类型检查有着一些缺陷。首先它增加了程序的运行时间，影响了效率。同时动态类型检查错误发现太晚，不能防止运行时产生出错。

4.2 运行时环境

4.2.1 存储分配

编译器在工作过程中，必须为源程序中出现的一些数据对象分配运行时的存储空间。对于那些在编译时刻就可以确定大小的数据对象，可以在编译时刻就为它们分配存储空间，这样的分配策略称为静态存储分配。

如果不能在编译时完全确定数据对象的大小，就要采用动态存储分配的策略。即在编译时仅产生各种必要的信息，而在运行时刻，再动态地分配数据对象的存储空间。这包括了栈式存储分配和堆式存储分配。

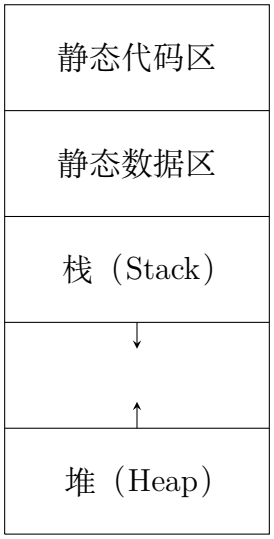


图 4.1: 内存管理

编译程序生成的代码大小通常是固定的，一般存放在代码区。目标程序运行过程中，需要创建和访问的数据对象存放在数据区。

栈区由编译器自动分配和释放，存放函数的参数值、局部变量的值等。

在编译时刻不能确定大小的对象将被分配在堆区，堆区的空间一般由程序员分配和释放。若程序员不释放，程序结束后被操作系统改回收。

4.2.2 活动记录（Activation Record）

编译器通常以过程（或函数、方法）为单位分配存储空间，过程体的每次执行称为该过程的一个活动，过程每执行一次，就为它分配一块连续存储区，用来管理过程一次执行所需的信息。

活动记录中包括：

- 实参
- 返回值
- 控制链：指向调用者的活动记录
- 访问链：用来访问存放于其它活动记录中的非局部数据
- 保存的机器状态
- 局部数据
- 临时变量

过程的存储以栈的形式进行管理，当一个过程被调用时，该过程的活动记录被压入栈；当过程结束时，该活动记录被弹出栈。

过程调用和过程返回都需要执行一些代码来管理活动记录栈，保存或恢复机器状态等。其中，调用过程时需要为一个活动记录在栈中分配空间，并在此记录的字段中填写信息。过程返回需要恢复机器状态，使得在调用结束之后继续执行。

4.3 中间代码生成

4.3.1 中间代码生成（Intermediate Code Generation）

源代码可以直接翻译成目标机器码，但是为什么要将其先翻译成中间代码，再翻译成目标代码呢?

如果没有中间代码，那么每种语言都需要为每种目标机器都设计一个编译器。如果使用中间代码表示（intermediate representation），将会使需要设计的编译器少得多。

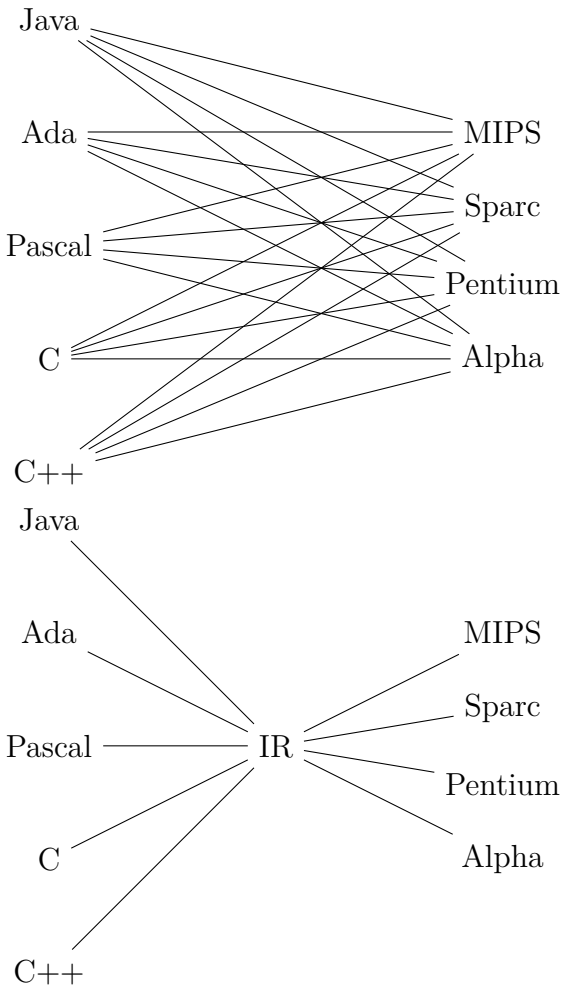


图 4.2: 中间代码

中间代码不依赖目标机的结构，中间语言与具体机器特性无关，一种中间语言可以为生成多种不同型号的目标机的目标代码服务。因此，对于中间语言，要求其

不但与机器无关，而且有利于代码生成。

4.3.2 三地址码（Three Address Code）

三地址码是一种易于生成且易于转换为机器码的中间代码，它最多使用三个地址和一个运算符来表示一个表达式。

算术表达式

例如 $2 * a + (b - 3)$ 用三地址码表示为：

```
t1 = 2 * a
t2 = b - 3
x = t1 + t2
```

数组

对于数组的表达式 $a[i + 1] = a[j * 2] + 3$，用三地址码表示为：

```
t1 = j * 2
t2 = a[t1]
t3 = t2 + 3
t4 = i + 1
a[t4] = t3
```

数组也可以使用地址偏移量的形式表示：

```
t1 = j * 2
t2 = t1 * elem_size(a)
t3 = &a + t2
t4 = *t3

t5 = t4 + 3

t6 = i + 1
```

```
t7 = t6 * elem_size(a)
t8 = &a + t7
*t8 = t5
```

条件

条件语句的三地址码需要使用跳跃（jump）和标签（label）表示。

例如 $if(expr)stmt1elsestmt2$ 表示为：

```
t1 = <code for expr>
if_false t1 goto L1
<code for stmt1>
goto L2
label L1
<code for stmt2>
label L2
```

循环

循环语句的三地址码也需要使用跳跃（jump）和标签（label）表示。

例如 $while(expr)stmt$ 表示为：

```
label L1
t1 = <code for expr>
if_false t1 goto L2
<code for stmt>
goto L1
label L2
```

4.4 代码优化

4.4.1 代码优化（Code Optimization）

代码优化试图通过使中间代码消耗更少的资源（CPU 和内存）来改进中间代码，从而产生运行速度更快的机器代码。

代码优化过程应该确保：

- 优化必须是正确的，它不能以任何方式改变程序的含义。
- 优化应该提高程序的速度和性能。
- 编译时间必须保持合理。

代码优化主要可以从以下几个方面进行：

编译时运算

在编译期间直接计算出表达式的值，例如：

```
1 x = 9.2
2 y = x / 2.3
```

可直接使用 $y = 4.0$ 代替。

消除重复计算表达式

有些表达式可能会被重复地多次计算，可以将第一次计算的结果保存，之后再出现如果重复计算的时候，可直接使用保存的值。

避免保存未使用的变量

一些变量在定义之后没有被使用，代码优化时可以不用保存这些变量的值。

删除不可到达的代码

如果一个代码块或函数永远不会被执行或调用，那么可以将其删除。例如：

```
#define DEBUG 0
if(DEBUG) {
    // code
}
```

强度削弱

把强度大的运算换算成强度小的运算，如乘方变乘法、乘法变加法。

例如：

- 将 x^3 优化为 $x * x * x$
- 将 $2 * x$ 优化为 $x << 2$
- 将 $5 * x$ 优化为 $x << 4 + x$

函数内联（Function Inlining）

函数内联是指将函数调用处的函数体直接插入到函数调用处，从而避免函数调用的开销。

消除尾递归（Tail Recursion Removal）

尾递归是指在一个函数内部，递归调用后直接 return，没有任何多余的指令了。

```
int gcd(int u, int v) {
    if(v == 0) {
        return u;
    }
    return gcd(v, u % v);
}
```

理论上，所有的递归函数都可以写成循环的方式。因为在尾递归中，递归调用是最后一条待执行的语句，于是当这个调用返回时栈帧中并没有其它事情可做，因此也就没有保存栈帧的必要了。

```
1 int gcd(int u, int v) {
2     while(v != 0) {
3         int t1 = v;
4         int t2 = u % v;
5         u = t1;
6         v = t2;
7     }
8     return u;
9 }
```