本笔记是从上海交通大学软件学院的离散数学课程整理而来，仅供学习使用，如果侵犯任何个体或组织的权益，请联系我进行删除。

Tips:标蓝的 part 为 quiz 考点而非期末考试考点

一．命题逻辑

命题逻辑讨论的是多个命题连接而成的复合命题的规律性。

命题是一个非真即假（不可兼）的陈述句。

Eg.哥德巴赫猜想是命题，其是真命题还是假命题，目前不知道；“这句话是错的”不是命题，因为它是一个悖论。

当 P 表示任意命题时，P 就称为**命题变项**（变元）。

简单命题（原子命题），就是不包含任何的与、或、非等连接词的命题。

蕴涵词：$P \to Q = \neg P \vee Q$

双条件词：$P \leftrightarrow Q = (P \to Q) \wedge (Q \to P) = (P \wedge Q) \vee (\neg P \wedge \neg Q)$

当命题 A，依赖于 n 个命题变元时，由这 n 个命题变元到 A 的真值表就有 2^n行，每一行对应这 n 个命题变元的一组真值，A 有 2^n 个解释。

对于由 n 个命题变元构成的命题公式，可能的真值表有 2^{2^n} 种，因为确定 n 个命题变元的值有 2^n 中情况，只要修改任意一行的 A 的真值情况，又是一种新的真值表，故 2^n 行会产生 2^{2^n} 不同情况的真值表。

命题逻辑的合式公式(Wff)的定义：

1. 简单命题是 Wff
2. 如果 A 是，那么 $(\neg A)$ 也是。
3. 如果 A,B 是，那么 $(A \wedge B)$，$(A \vee B)$，$(A \to B)$，$(A \leftrightarrow B)$ 也是。*注意这些括号都是需要加上的，判定表达式是否为合式公式的代码中，严格需要合法括号的辅助，不允许多加或者漏掉括号。
4. 对规则 1,2,3 进行有限次组合的都是合式公式。

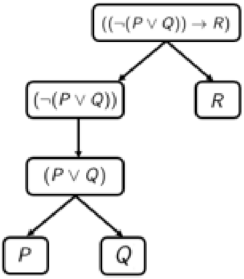
Wff 的代码判定：SAT Solver Notes

目的：给定一个表达式，输出 True（是合式公式）或者 False（不是合式公式）。

步骤：

1. 构建树的根节点，把表达式放入。
2. 如果树的所有节点都放置了一个原子命题，返回 True。
3. 选择一个叶子节点，其中放置的不是原子命题，而是一个表达式 f。
4. 如果 f 没有被括号括起来，则不合法返回 False。
5. 如果 f= $(\neg g)$，那么新开一个儿子节点存放表达式 g。转步骤 2.

6. 令 f=(F)，从左到右扫描 F，直到扫描到一个非空字符串 A，使得 A 的左括号数量等于右括号数量。如果没有则返回 False。
7. 如果 f=(A ⊙ B),⊙ ∈ {→, ↔, ∨, ∧}，那么给 f 所在的叶子节点新加两个子节点，一个存放 A，一个存放 B。转步骤 2。
7. 返回 False。



Eg:

波兰表达式（前缀式）P12

几个重点记忆的公式：

蕴含没有结合律：$(P \to Q) \to R \neq P \to (Q \to R)$

分配律：把外面的符号分配进去 $\begin{aligned} P \vee (Q \wedge R) &= (P \vee Q) \wedge (P \vee R) \\ P \wedge (Q \vee R) &= (P \wedge Q) \vee (P \wedge R) \end{aligned}$

摩根律及扩展：$\begin{aligned} &\neg(P \vee Q) = \neg P \wedge \neg Q \\ &\neg(P \wedge Q) = \neg P \vee \neg Q \\ &\neg(P \to Q) = P \wedge \neg Q \\ &\neg(P \leftrightarrow Q) = (\neg P) \leftrightarrow Q = P \leftrightarrow (\neg Q) = (\neg P \wedge Q) \vee (P \wedge \neg Q) \end{aligned}$

等值公式：

$P \to Q = \neg Q \to \neg P$，逆否定理等价于原定理。

$P \to (Q \to R) = (P \wedge Q) \to R$,两个前提的合取作为总的前提。

$P \to (Q \to R) = Q \to (P \to R)$,前提条件 P,Q 可以交换次序。

$(P \to R) \wedge (Q \to R) = (P \vee Q) \to R$,P 和 Q 都分别是 R 的前提，那么可以说 P 或 Q 是 R 的前提。

$P \leftrightarrow Q = (P \wedge Q) \vee (\neg P \wedge \neg Q)$,P 等价于 Q 等值于 P 和 Q 同时成立或者 P 和 Q 同时不成立

$P \leftrightarrow Q = (P \vee \neg Q) \wedge (\neg P \vee Q)$，从令 P 不等价于 Q 的角度来说，要么 P=T,Q=F，要么 P=F,Q=T，这两种情况一定会导致 $P \vee \neg Q$ 或者 $\neg P \vee Q$ 中有一个为 F，导致 $P \leftrightarrow Q = F$

$P \leftrightarrow Q = (P \to Q) \wedge (Q \to P)$，P 等价于 Q 成立，意味着正定理和逆定理同时成立。

置换规则：公式 A 的子公式置换后，A 化为公式 B，必有 A=B。

联结词的完备集：$\{\neg,\wedge,\vee\},\{\neg,\wedge\},\{\neg,\vee\},\{\neg,\rightarrow\},\{\uparrow\},\{\downarrow\}$

对偶式：待复习

范式：

P 和$\neg P$统称文字。文字的合取成为合取式（合取子句）。文字的析取成为析取式（析取子句）。

析取范式：形如$A_1 \vee A_2 \vee \cdots A_n$，也就是把一堆合取式析取起来。

合取范式：形如$A_1 \wedge A_2 \wedge \cdots A_n$，也就是把一堆析取式合取起来。

范式定理：任一命题公式都存在等值的合取范式和析取范式。

求取一个公式的范式：P25

1. 消去联结词$\rightarrow$和$\leftrightarrow$
2. 重复使用摩根律和双重否定律，把否定词内移到命题变项上。
3. 重复使用分配律。一定要搞清楚什么该出来，什么该进去。

极小项：

对 n 个命题变项$P_1,\cdots,P_n$来说，所组成的公式$Q_1 \wedge Q_2 \wedge \cdots \wedge Q_n$，其中$Q_i = P_i$或$\neg P_i$，则称$Q_1 \wedge Q_2 \wedge \cdots \wedge Q_n$为极小项，以$m_{index}$，其中 index 以每一位$Q_i$的取法作为二进制的一位编码：$Q_i = P_i$则该位取 1，$Q_i = \neg P_i$则该位取 0.

同理$Q_1 \vee Q_2 \vee \cdots \vee Q_n$为**极大项**，以$M_{index}$表示。

极大项和极小项的性质：

恰由2^n个极小项的析取构成的公式，必为重言式。

恰由2^n个极大项的合取构成的公式，必为矛盾式。

由极小项构成的范式成为**主析取范式（PDNF,principle disjunction normal form）**。任一含 n 个命题变项的公式都存在唯一等值的主析取范式。

由极大项构成的范式成为**主合取范式（PCNF,principle conjunction normal form）**。任一含 n 个命题变项的公式都存在唯一等值的主合取范式。

得到主析取范式：利用真值表列写公式，或者将析取范式中的合取式填满命题变项的方法。

(1) 利用 T 来列写主析取范式(P20)。有 A=$(\cdot\wedge\cdot)_1 \vee (\cdot\wedge\cdot)_2 \vee \cdots \vee (\cdot\wedge\cdot)_n$的形式，其中 n 为使 A 取值为 T 的成真指派行数。把 A 的成真条件(行)中的变元取值以合取的形式列写最后析取在一起，即可得到 A 的主析取范式。

(2) 利用填满命题变项法，把$P \rightarrow Q$的析取范式转化为主析取范式。

已知$P \rightarrow Q = \neg P \vee Q$，需要在每一项中添加出其他项，构成极小项。则有：

$$\neg P = \neg P \wedge (Q \vee \neg Q) = (\neg P \wedge Q) \vee (\neg P \wedge \neg Q)$$
$$Q = Q \wedge (P \vee \neg P) = (Q \wedge P) \vee (Q \wedge \neg P)$$
,从而

$$P \to Q = \neg P \wedge Q = (\neg P \wedge Q) \vee (\neg P \wedge \neg Q) \vee (Q \wedge P) \vee (Q \wedge \neg P)$$
$$= (\neg P \wedge \neg Q) \vee (\neg P \wedge Q) \vee (Q \wedge P) = m_0 \vee m_1 \vee m_3 = \vee_{0;1;3}$$

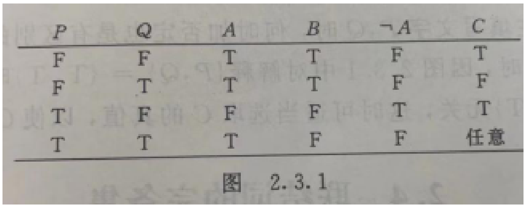
得到主合取范式：利用真值表列写公式，或者将合取范式中的析取式填满命题变项的方法。

(1) 利用 F 来列写主合取范式(P20)。有 A=$(\cdot\vee\cdot)_1 \wedge (\cdot\vee\cdot)_2 \wedge \cdots \wedge (\cdot\vee\cdot)_n$ 的形式，其中 n 为使 A 取值为 F 的成假指派行数。把 A 的成假条件(行)中的变元取值以析取的形式列写最后合取在一起，即可得到 A 的主合取范式。

(2) 利用填满命题变项法，把 $P \wedge Q$ 转化为主合取范式。

$$P \wedge Q = (P \vee (Q \wedge \neg Q)) \wedge (Q \vee (P \wedge \neg P))$$
$$= (P \vee Q) \wedge (P \vee \neg Q) \wedge (Q \vee P) \wedge (Q \vee \neg P)$$
$$= (P \vee \neg Q) \wedge (P \vee Q) \wedge (\neg P \vee Q) = M_1 \wedge M_2 \wedge M_3 = \wedge_{1;2;3}$$

Eg.根据真值表来列写 A、B、C 的主析取范式、主合取范式



P	Q	A	B	$\neg A$	C
F	F	T	T	F	T
F	T	T	T	F	F
T	F	F	F	T	T
T	T	T	F	F	任意

图 2.3.1

利用 T 来列写 A 的主析取范式

A 的成真条件：P=F Q=F, P=F Q=T, P=T Q=T

$A = (\neg P \wedge \neg Q) \vee (\neg P \wedge Q) \vee (P \wedge Q)$，成真条件满足任意一个，即为真

利用 F 来列写 A 的主合取范式

A 的成假条件：P=T Q=F，也就是 $(\neg P \vee Q)$ 为假

$A = (\neg P \vee Q)$，成假条件均不满足，即为真

利用 T 来列写 B 的主析取范式

B 的成真条件：P=F Q=F, P=F Q=T

$B = (\neg P \wedge \neg Q) \vee (\neg P \wedge Q)$，成真条件满足任意一个，即为真

利用 F 来列写 B 的主合取范式

B 的成假条件：P=T Q=F，P=T Q=T 也就是 $(\neg P \vee Q)$ 为假和 $(\neg P \vee \neg Q)$ 为假

$B = (\neg P \vee Q) \wedge (\neg P \vee \neg Q)$，成假条件均不满足，即为真

C 的列写类似，取值为任意时可以自行取定 T 和 F，得到最终表达式简单。

主析取范式和主合取范式的转换：P28

已知主析取求主合取：
$$A=\vee_{0,1,4,5,7}=\wedge_{(\{0,1,\cdots,7\}-\{0,1,4,5,7\})_{补}}$$
$$=\wedge_{\{2,3,6\}_{补}}=\wedge_{1,4,5}$$

已知主合取求主析取：
$$A=\wedge_{1,4,5}=\vee_{(\{0,1,\cdots,7\}-\{1,4,5\}_{补})}$$
$$=\vee_{(\{0,1,\cdots,7\}-\{2,3,6\})}=\vee_{0,1,4,5,7}$$

DPLL 算法：SAT Solver Notes

首先，为了求解一个表达式的可满足性，我们需要转化为 CNF（合取式）。因为 CNF 转到 DNF 会有指数爆炸问题，而 DNF 转到 CNF 只需要添加额外的开关变量即可。

Eg:$(A_1\wedge A_2\wedge A_3)\vee(B_1\wedge B_2\wedge B_3)$ 可以添加开关变量转化为

$(Z\to A_1\wedge A_2\wedge A_3)\wedge(\neg Z\to B_1\wedge B_2\wedge B_3)$，转换前后的式子虽然不等价，但是可满足性相同(equi-satisfiable)。我们对转换后的式子展开：

$$(\neg Z\vee A_1\wedge A_2\wedge A_3)\wedge(Z\vee B_1\wedge B_2\wedge B_3)$$
$$=(\neg Z\vee A_1)\wedge(\neg Z\vee A_2)\wedge(\neg Z\vee A_3)\wedge(Z\vee B_1)\wedge(Z\vee B_2)\wedge(Z\vee B_3)$$

可以看到 3 个命题变量转化为了 6 个式子(3+3)，并没有指数爆炸。

对于 DPLL 算法，其本质是找到一种解释，使得表达式为 T。

DPLL 的规则：

1. Decide Rule（决定规则）：如果我们没有认定过某一个文字的真值，那么这个文字叫做 undefined literal。Decide Rule 就是选取一个未定义的文字，猜测它的真值，并且标记为 decision literal（决策文字），猜可以瞎猜，可以不用到 CNF 的信息。
2. Unitpropagate Rule（单元传播规则）：当一个 literal（文字）有了真值以后，可以通过 CNF 的信息，推导出其他文字的真值。如果推导后使得整个 CNF 为 T，那么输出 sat 以及各个文字的真值。
3. Backtrack Rule（回溯规则）：如果发现某个析取子句的真值为 F，说明前面有文字的赋值赋错了，需要回溯找到最近的一个 decision literal，将其赋为另一个值，并标记为 non-decision literal，并且通过 decision literal 推导出的值需要推翻再次推导。
4. Fail Rule（失败规则）：如果在 backtrack rule 的回溯过程中，找不到任何 decision literal，说明这个公式是不可满足的，输出 unsat。
5. Pureliteral Rule（纯文字规则）：如果 CNF 中只含 A 而不含 $\neg A$，那么 A 的值为 T；

   如果 CNF 中只含 $\neg A$ 而不含 A，那么 A 的值为 F。

注意：在书写中把 CNF 的 ∧ 换成”,” 来表示

Eg1:(PPT3, P123)

目标：理解两个 decision literal 的情况

$$
\begin{array}{rll}
\emptyset \parallel & \bar{1}\vee 3,\ \ \bar{2}\vee\bar{4},\ \ 2\vee 4,\ \ 3\vee\bar{4} & \Longrightarrow (PureLiteral) \\
\bar{1} \parallel & \bar{1}\vee 3,\ \ \bar{2}\vee\bar{4},\ \ 2\vee 4,\ \ 3\vee\bar{4} & \Longrightarrow (Decide) \\
\bar{1}\ \ 3^d \parallel & \bar{1}\vee 3,\ \ \bar{2}\vee\bar{4},\ \ 2\vee 4,\ \ 3\vee\bar{4} & \Longrightarrow (Decide) \\
\bar{1}\ \ 3^d\ \ 2^d \parallel & \bar{1}\vee 3,\ \ \bar{2}\vee\bar{4},\ \ 2\vee 4,\ \ 3\vee\bar{4} & \Longrightarrow (UnitProp) \\
\bar{1}\ \ 3^d\ \ 2^d\ \ \bar{4} \parallel & \bar{1}\vee 3,\ \ \bar{2}\vee\bar{4},\ \ 2\vee 4,\ \ 3\vee\bar{4} &
\end{array}
$$

Eg2:(PPT3, P104)
目标：理解 pureliteral 和 fail

$$
\begin{array}{rlll}
\emptyset \parallel & 1\vee\bar{2},\ \ \bar{1}\vee\bar{2},\ \ 2\vee 3,\ \ \bar{3}\vee 2,\ \ 1\vee 4 & \Longrightarrow & \text{(PureLiteral)} \\
4 \parallel & 1\vee\bar{2},\ \ \bar{1}\vee\bar{2},\ \ 2\vee 3,\ \ \bar{3}\vee 2,\ \ 1\vee 4 & \Longrightarrow & \text{(Decide)} \\
4\ 1^d \parallel & 1\vee\bar{2},\ \ \bar{1}\vee\bar{2},\ \ 2\vee 3,\ \ \bar{3}\vee 2,\ \ 1\vee 4 & \Longrightarrow & \text{(UnitProp)} \\
4\ 1^d\ \bar{2} \parallel & 1\vee\bar{2},\ \ \bar{1}\vee\bar{2},\ \ 2\vee 3,\ \ \bar{3}\vee 2,\ \ 1\vee 4 & \Longrightarrow & \text{(UnitProp)} \\
4\ 1^d\ \bar{2}\ 3 \parallel & 1\vee\bar{2},\ \ \bar{1}\vee\bar{2},\ \ 2\vee 3,\ \ \bar{3}\vee 2,\ \ 1\vee 4 & \Longrightarrow & \text{(Backtrack)} \\
4\ \bar{1} \parallel & 1\vee\bar{2},\ \ \bar{1}\vee\bar{2},\ \ 2\vee 3,\ \ \bar{3}\vee 2,\ \ 1\vee 4 & \Longrightarrow & \text{(UnitProp)} \\
4\ \bar{1}\ \bar{2}\ \bar{3} \parallel & 1\vee\bar{2},\ \ \bar{1}\vee\bar{2},\ \ 2\vee 3,\ \ \bar{3}\vee 2,\ \ 1\vee 4 & \Longrightarrow & \text{(Fail)} \\
fail & & &
\end{array}
$$

书本 2.7~2.10：待复习
第四章：

约定有小写字母表示命题，用大写字母表示谓词。

可以引入 x 来表示主词（个体词），把 P(x)称作谓词。有一元谓词和多元谓词之分。P(x)中的 x 称作个体变项（个体变元）。P 表示任一谓词是，称为谓词变项。将个体变项的变化范围称为个体域或者论域，用 D 表示，谓词逻辑的个体域除明确指明外，都认为是包括一切事物的一个最广的集合。谓词变项的变化范围，不特别声明时，指一切关系或一切性质的集合。谓词是给定的个体域到集合{T,F}上的一个映射。可以认为一个命题是没有个体变元的零元谓词。

函数是某一个体域到另一个体域的映射。

	arguments	results
connectives	propositions	a proposition
predicates	individuals	a proposition
functions	individuals	an individual

图 1 联结词、谓词、函数的辨析

全称量词：命题 $(\forall x)P(x)$ 当且仅当论域中的所有 x 均满足 $P(x)=T$ 时，$(\forall x)P(x)$ 才为真。

存在量词：命题 $(\exists x)Q(x)$ 当且仅当论域中的至少有一个 x_0 满足 $Q(x_0)=T$ 时，$(\exists x)Q(x)$ 才为真。

自由变元：不受量词约束的变元。
约束变元：受到量词约束的变元。
量词的辖域：量词所约束的范围。

变元易名规则：$(\forall x)P(x)=(\forall y)P(y)$

谓词逻辑下的合式公式的定义：P59

1. 命题常项、命题变项和原子谓词公式（不含联结词的谓词）都是合式公式。
2. 如果 A 是合式公式，那么 $\neg A$ 也是合式公式。
3. 如果 A,B 是合式公式，而无变元 x 在 A，B 的一个中是约束的而在另一个中是自由的，则 $(A\wedge B)$，$(A\vee B)$，$(A\rightarrow B)$，$(A\leftrightarrow B)$ 也是合式公式。
4. 如果 A 是合式公式，而 x 在 A 中是自由变元，则 $(\forall x)A$，$(\exists x)A$ 也是合式公式。

Eg:

$(\forall x)F(x)\wedge G(x)$ 不是，因为 x 在 $(\forall x)F(x)$ 中是约束的，而在 $G(x)$ 是自由的。

$(\exists x)((\forall x)F(x))$ 不是，因为 x 在 $(\forall x)F(x)$ 中是约束的，不满足 4。

$(\forall x)F(y)$ 不是，因为 x 在 $F(y)$ 中不存在（不是自由变元），不满足 4。

第五章：谓词逻辑的等值和推理演算（P69）
量词的性质：predicate logic P35

Quantifier（量词）

PROPERTY

1) $(\forall x)(\forall y)P(x,y)=(\forall x)((\forall y)P(x,y))$
2) $(\forall x)(\exists y)P(x,y)=(\forall x)((\exists y)P(x,y))$
3) $(\exists x)(\forall y)P(x,y)=(\exists x)((\forall y)P(x,y))$
4) $(\exists x)(\exists y)P(x,y)=(\exists x)((\exists y)P(x,y))$
5) $(\forall x)(\forall y)P(x,y)=(\forall y)(\forall x)P(x,y)$
6) $(\exists x)(\exists y)P(x,y)=(\exists y)(\exists x)P(x,y)$

否定型等值式：

$$\neg(\forall x)P(x)=(\exists x)\neg P(x)$$
$$\neg(\exists x)P(x)=(\forall x)\neg P(x)$$
，否定移到辖域内时，需要改变量词的种类。

量词对∨、∧的分配律：

$$(\forall x)(P(x)\vee q)=(\forall x)P(x)\vee q$$
$$(\forall x)(P(x)\wedge q)=(\forall x)P(x)\wedge q$$
$$(\exists x)(P(x)\vee q)=(\exists x)P(x)\vee q$$
$$(\exists x)(P(x)\wedge q)=(\exists x)P(x)\wedge q$$

对于与 x 无关的命题变项 q，可以直接移出。

量词对 → 的分配律：

$$(\forall x)(P(x)\to q)=(\exists x)P(x)\to q$$
$$(\forall x)(q\to P(x))=q\to(\forall x)P(x)$$
$$(\exists x)(P(x)\to q)=(\forall x)P(x)\to q$$
$$(\exists x)(q\to P(x))=q\to(\exists x)P(x)$$

q 在前，直接提；q 在后，变量词。

含个体变元情况下，量词 $\forall$ 对 $\wedge$、量词 $\exists$ 对 $\vee$ 的分配律：

$$(\forall x)(P(x)\wedge Q(x))=(\forall x)P(x)\wedge(\forall x)Q(x)$$
$$(\exists x)(P(x)\vee Q(x))=(\exists x)P(x)\vee(\exists x)Q(x)$$

通过变元易名，实现量词 $\forall$ 对 $\vee$、量词 $\exists$ 对 $\wedge$ 的分配律（一般情况下不成立）：

$$(\forall x)(\forall y)(P(x)\vee Q(y))=(\forall x)P(x)\vee(\forall x)Q(x)$$
$$(\exists x)(\exists y)(P(x)\wedge Q(y))=(\exists x)P(x)\wedge(\exists x)Q(x)$$

前束范式：A 中的一切量词都位于该公式的最左边（不含否定词）且这些量词的辖域都延伸到公式的末端。

前束范式 A 的一般形式为 $(Q_1x_1)\cdots(Q_nx_n)M(x_1,\cdots,x_n)$，其中 Q_i 为全称量词或存在量词，M 称为 A 的母式（基式），M 中无量词。

定理 5.3.1：谓词逻辑的任一公式都可化为与之**等值**的前束范式，但其**前束范式并不唯一**。

求取前束范式的步骤：P75

1. 消除联结词 →，↔
2. ¬ 内移（反复使用摩根律）
3. 量词左移（使用分配等值式）
4. 变元易名（使用变元易名分配等值式）

Eg：

$$\neg((\forall x)(\exists y)P(a,x,y)\to(\exists x)(\neg(\forall y)Q(y,b)\to R(x)))$$

Step1: eliminate arrows.

$$=\neg(\neg(\forall x)(\exists y)P(a,x,y)\vee(\exists x)(\neg\neg(\forall y)Q(y,b)\vee R(x)))$$

Step2: move "not" inside.

$$=(\forall x)(\exists y)P(a,x,y)\wedge(\forall x)((\exists y)\neg Q(y,b)\wedge\neg R(x))$$

Step3: move quantifiers to the left.

$$=(\forall x)((\exists y)P(a,x,y)\wedge(\exists y)\neg Q(y,b)\wedge\neg R(x))$$

Step4: change names of bound variables.

$$=(\forall x)((\exists y)P(a,x,y)\wedge(\exists z)\neg Q(z,b)\wedge\neg R(x))$$
$$=(\forall x)(\exists y)(\exists z)(P(a,x,y)\wedge\neg Q(z,b)\wedge\neg R(x))$$

Move quantifiers to the left.

$$=(\forall x)(\exists y)(\exists z)S(a,b,x,y,z)$$

Skolem 标准形：P76

Skolem 标准形与原公式不等值，只能保持在某种意义下的等值关系。

定义：只保留全称量词的前束形。计算方式：从量词最左边开始逐个消掉存在量词，消完用此存在量词左边的所有全称量词的约束变元的函数代替。

Eg：求 $(\exists x)(\forall y)(\forall z)(\exists u)(\forall v)(\exists w)P(x,y,z,u,v,w)$ 的 Skolem 标准型。

易得，答案为：$(\forall y)(\forall z)(\forall v)P(a,y,z,f(y,z),v,g(y,z,v))$

推理及推理规则：待复习。

SMT solver：

SMT Solver 是自动判定一个谓词逻辑表达式是否可满足的工具

Eager SMT Technique：复杂度较高，直接把谓词逻辑表达式转化为命题表达式后通过 SAT Solver 求解。针对存在函数的情况，为了保证函数的关系信息不丢失，需要额外补充函数关系的表达式。

Lazy SMT Technique：

步骤：

1. 把式子中每个原子命题替换成命题变项。
2. SAT Solver 求解命题逻辑表达式。如果 SAT Solver 返回 unsat，则直接退出返回 unsat。
3. 如果 SAT Solver 返回了一组解，那么调用 T-Solver(theory solver)去验证这组解在谓词层面能否同时成立。如果可以，则返回 sat 和这组解。
4. 如果不能成立，则 T-Solver 通知 SAT Solver 这组解不能成立。把这组解成立的非作为一个约束加到表达式中。
5. 重复 2-4，直到 T-Solver 成功找到一组解或者 SAT Solver 返回 unsat。

优化：

1. DPLL 算法执行中间就调用 T-solver，检查已有的对部分命题变项的赋值有没有问题，这样可以尽早发现问题，提前终止 DPLL 算法，减少 SAT solver 的无用功。
2. T-solver 不仅可以检查 SAT solver 的赋值有没有问题，而且还可以在 DPLL 算法运行中间推测出某些 literal 的真值，并通知 SAT solver，这样可以帮助减少 SAT solver 的工作量。

EUF：

EUF 是 T-Solver 的一种，用来处理包括**等于和函数**的公式。处理函数时需要用到 congruence rule。

congruence rule：$x1 = y1, ..., xn = yn \Rightarrow f(x1, ..., xn) = f(y1, ..., yn)$

步骤：

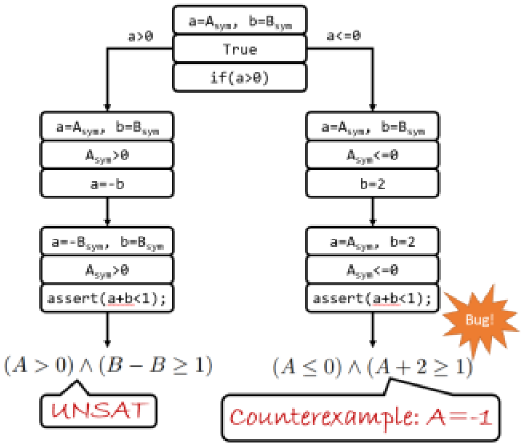
1. 把公式中的函数替换为个体词
2. 每个个体词画一个圆圈。
3. 利用等式把圆圈合并。
4. 利用 congruence rule 进一步合并，合并操作结束。
5. 查看不等式，如果不相等的两个个体词在同一个圆圈中，则报错。

符号执行（Symbolic Execution）：

符号执行技术为程序的每一条可能的执行路径生成一个公式，分别求解。和上面的方法一样，符号执行也是用符号而不是具体的数值表示当前程序状态。符号执行过程中维护 3 块数据。

1. Path constraints：路径的条件
2. Next code：下一条代码指令
3. Symbolic store：每个变量的值或符号表达式

```
void func(int a, int b) {
  if(a > 0)
    a = -b;
  else
    b = 2;
  assert(a + b < 1);
}
```



霍尔逻辑：

引入霍尔逻辑的目的：在形式化验证程序正确性的过程中，需要引入霍尔逻辑来解决 unbounded loop 的问题。

霍尔逻辑使用霍尔三元组来定义程序的正确性：$\{P\}C\{Q\}$，其中 P 是前置条件，C 是代码，Q 是后置条件。前置条件表示执行程序前的程序状态满足什么条件，后置条件表示程序执行后满足什么条件。注意霍尔逻辑只能保证 partial correctness，

对于程序不能结束的情况：$\{P\}while(1);\{Q\}$，任意 P，Q 均成立。

公理：

$\{P\}skip\{P\}$

赋值公理 AS-FW（forward assignment axiom）：$\{P\}a := e\{(\exists v)(a = e[v/a] \wedge P[v/a])\}$ 其中

后置条件的意思就是：令 $v = a_{old}$，执行完赋值语句后有 $a_{new} = e(a_{old})$

Eg: $\{a=1\}a := a+1\{(\exists v)(a = a+1[v/a] \wedge (a=1)[v/a])\}$ $\{a=1\}a := a+1\{(\exists v)(a = v+1 \wedge v=1)\}$ $\{a=1\}a := a+1\{a=2\}$

赋值公理的另一个形式 **AS**：$\{P[e/a]\}a := e\{P\}$，其中 P[e/a] 是把 P 中的 a 全部换成 e。意思是如果执行代码前 P[e/a] 成立，那么执行 a:=e，现在 a 的值变成了 e，所以 P 肯定成立。

Eg:

$\{b+1=42\}a:=b+1\{a=42\}$
$\{42=42\}a:=42\{a=42\}$
$\{a-b>3\}a:=a-b\{a>3\}$

推理规则：

弱化后继(Weakening consequent, **WS**)规则：如果有 $P \to Q$，显然 P 的成立条件比 Q 强。

那么我们有 $\frac{\{P\}C\{Q\}, Q \to R}{\{P\}C\{R\}}$,其中横线意为 inference（推理）

Eg:

$$\frac{\{a=1\}a:=a+1\{(\exists v)(a=(a+1)[v/a] \wedge (a=1)[v/a])\} \quad (\exists v)(a=(a+1)[v/a] \wedge (a=1)[v/a]) \to a=2}{\{a=1\}a:=a+1\{a=2\}}$$

强化前驱(Strengthen precedent, **SP**)规则：对于 $P \to Q$，P 的成立条件比 Q 强。

那么我们有 $\frac{P \to Q, \{Q\}C\{R\}}{\{P\}C\{R\}}$

顺序组合规则(Sequential composition rule, **SC**):

$$\frac{\{P\}C1\{R\}, \{R\}C2\{Q\}}{\{P\}C1; C2\{Q\}}$$

Eg:试证明 $\{b>3\}a:=2*b, a:=a-b\{a\geq 4\}$

(1) $\{a-b\geq 4\}a:=a-b\{a\geq 4\}$ AS,利用 $\{P[e/a]\}a:=e\{P\}$，p= $P=a\geq 4, P[e/a]=a-b\geq 4$

(2) $\{2b-b\geq 4\}a:=2*b\{a-b\geq 4\}$ AS

(3) $b>3 \to b\geq 4$ predicate logic

(4) $\{b>3\}a:=2*b\{a-b\geq 4\}$ SP(2)(3)

(5) $\{b>3\}a:=2*b, a:=a-b\{a\geq 4\}$ SC(1)(4)

条件规则(Conditional Rule, CD)：

$$\frac{\{P \wedge istrue(b1)\}C1\{Q\},\{P \wedge isfalse(b1)\}C2\{Q\}}{\{P\}if(b1)then: C1; else: C2\{Q\}}$$

Eg：证明$\{T\}if(a<b)then: c:=b; else: c:=a\{c=\max(a,b)\}$

观察可得，尝试构造出$\{a<b\}c=b\{c=\max(a,b)\}$和$\{a\geq b\}c=a\{c=\max(a,b)\}$

Proof:

(1) $\{b=\max(a,b)\}c=b\{c=\max(a,b)\}$ AS

(2) $b=\max(a,b) \leftrightarrow a\leq b, a<b \rightarrow a\leq b$ predicate logic

(3) $\{a<b\}c=b\{c=\max(a,b)\}$ SP(1)(2)

(4) $\{a=\max(a,b)\}c=a\{c=\max(a,b)\}$ AS

(5) $a=\max(a,b) \leftrightarrow a\geq b$ predicate logic

(6) $\{a\geq b\}c=a\{c=\max(a,b)\}$ SP(4)(5)

(7) $\{T\}if(a<b)then: c:=b; else: c:=a\{c=\max(a,b)\}$CD(3)(6)

循环规则(While Rule, **WHP**)：

$\frac{\{I \wedge istrue(b1)\}C\{I\}}{\{I\}while(b1)C\{I \wedge isfalse(b1)\}}$，其中 I 为循环不变量。无论循环执行几次，I 始终成立。

需要手动指定循环不变量。（loop invariant）

Eg:

$\{a\leq 10\}\ while(a!=10)\ a:=a+1\ \{a=10\}$

$Proof.$

$(1)\{a+1\leq 10\}a:=a+1\{a\leq 10\}$ AS

$(2)a\leq 10 \wedge a\neq 10 \rightarrow a+1\leq 10$ $predicate\ logic$

$(3)\{a\leq 10\wedge a\neq 10\}a:=a+1\{a\leq 10\}$ $SP(1)(2)$

$(4)\{a\leq 10\}\ while(a!=10)\ a:=a+1\ \{a\leq 10\wedge a=10\}$ $WHP(3)$

$(5)a\leq 10\wedge a=10 \rightarrow a=10$ $predicate\ logic$

$(6)\{a\leq 10\}\ while(a!=10)\ a:=a+1\ \{a=10\}$ $WC(4)(5)$

$$\{\textbf{true}\}\ \textbf{while}\ x \neq 10\ \textbf{do skip}\ \{x = 10\}$$

Proof:

1. $\{\textbf{true} \wedge x \neq 10\}\ \textbf{skip}\ \{\textbf{true} \wedge x \neq 10\}$ — SK
2. $\textbf{true} \wedge x \neq 10 \Rightarrow \textbf{true}$
3. $\{\textbf{true} \wedge x \neq 10\}\ \textbf{skip}\ \{\textbf{true}\}$ — WC, 1, 2
4. $\{\textbf{true}\}\ \textbf{while}\ x \neq 10\ \textbf{do skip}\ \{\textbf{true} \wedge \neg(x \neq 10)\}$ — WHP, 3
5. $\textbf{true} \wedge \neg(x \neq 10) \Rightarrow x = 10$
6. $\{\textbf{true}\}\ \textbf{while}\ x \neq 10\ \textbf{do skip}\ \{x = 10\}$ — WC, 4, 5

最弱前置条件（Weakest Precondition，**wp**）：
给定一个程序 C 和后置条件 Q，P 是最弱前置条件当前仅当对于任意 P'，有：$\{P'\}C\{Q\}$，$P' \rightarrow P$。用 **wp(C,Q)**来表示

$$\{a = 2\}a := a + 1\{a = 3\}$$
$$\{a = 2 \wedge b = 4\}a := a + 1\{a = 3\}$$
$$\{a = 2 \wedge a > 1\}a := a + 1\{a = 3\}$$

weakest precondition

$$a = 2 \wedge b = 4 \rightarrow a = 2$$
$$a = 2 \wedge a > 1 \rightarrow a = 2$$

证明霍尔三元组 {P}C{Q} 成立就等价于证明 P → wp(C, Q)或者 sp(C, Q) → P 成立，也就是证明 $P \wedge \neg wp(C,Q)$ 或者 $sp(C,P) \wedge \neg Q$ 是不可满足的。

这样就把霍尔逻辑的公式转成了谓词逻辑公式，接下来就交给强大的 SMT solver 来处理。由于 AS-FW 给出了最强后置条件，而 AS 给出了最弱前置条件，且 AS 没有存在量词，所以我们研究 wp 的问题。在实际应用中，我们使用 $wlp(C,Q)$ 代替 $wp(C,Q)$，也就是不考虑终止的情况。

$$wlp(skip, P) = P$$
$$wlp(a := e, P) = P[e/a]$$
$$wlp(C1; C2, P) = wlp(C1, wp(C2, P))$$
$$wlp(if(b)\ then\ C1\ else\ C2, P) = (istrue(b) \rightarrow wlp(C1, P)) \wedge (isfalse(b) \rightarrow wlp(C2, P))$$

如果程序中有循环，那么需要用户手动定义循环不变式 I，然后证明 $\{P\}while(b)C\{Q\}$ 就变成证明下面 3 个公式都成立。

$$P \rightarrow I$$
$$\{I \wedge iftrue(b)\}C\{I\}$$
$$(I \wedge isfalse(b)) \rightarrow Q$$

这样循环就去掉了，接下来就可以按照前面的做法生成谓词逻辑公式，交给 SMT solver 验证。

Eg:

$$wlp(a := b + 1, a = 42) = b + 1 = 42$$

$$wlp(a := a - b, a > 3) = a - b > 3$$

$$wlp(a := a + 1; b := a, a = 3 \wedge b > 0)$$
$$= wlp(a := a + 1, wlp(b := a, a = 3 \wedge b > 0))$$
$$= wlp(a := a + 1, a = 3 \wedge a > 0)$$
$$= a + 1 = 3 \wedge a + 1 > 0$$

$$wlp(if(a == 0)\ then\ \{b := 1\}\ else\ \{b := 0\}, b = 1 \vee b = 0)$$
$$= (a = 0 \rightarrow wlp(b := 1, b = 1 \vee b = 0)) \wedge (a \neq 0 \rightarrow wlp(b := 0, b = 1 \vee b = 0))$$
$$= (a = 0 \rightarrow 1 = 1 \vee 1 = 0) \wedge (a \neq 0 \rightarrow 0 = 1 \vee 0 = 0)$$

第 9 章 集合

空集：不含任何元素的集合，记做 $\varnothing$

空集的定义也可以写成 $\varnothing = \{x \mid x \neq x\}$

全集：在给定的问题中，所考虑的所有事物的集合，记做 E。

全集的定义也可以写成 $E=\{x\mid x=x\}$

图灵机部分

DFA(Deterministic Finite Automaton，确定有限自动机)
DFA 包括：

Q: A finite set of states（确定且有限的状态集合）

Σ: A finite set of input symbols（确定且有限的输入的符号集合）

δ: A transition function（状态转移函数）$Q\times\Sigma\to Q$：也就是 Q 和 Σ 的笛卡尔积，到状态的函数，也就是任意一对 Q 和 Σ，需要能够映射到下一个 Q

q_0: The start state（开始的状态）

F: A set of final or accepting states.（接受的状态集合），且满足 $F\subseteq Q$

我们可以用一个五元组来表示一个 DFA：$A=(Q,\Sigma,\delta,q_0,F)$

DFA 把一个有限长度的字符串作为输入（字符均来自于有限的输入符号集合），根据状态转移方程 δ 进行转移。DFA 一定会停下来，如果停下来的时候在 accepting states(F)中，那么这个 DFA 接受这个字符串，否则就拒绝这个字符串。

我们可以定义一个扩展转移函数(extended transition function) $\hat{\delta}:Q\times\Sigma^*\to Q$，也就是把定义域扩展到了空字符的情况。如果输入了一个空字符，那么状态不变。$\hat{\delta}(q,\varepsilon)=q$

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

给定一个 DFA：$A=(Q,\Sigma,\delta,q_0,F)$，所有 A 接受的字符串的集合称作 A 的语言，记做 $L(A)$。

$L(A)=\{w\mid\hat{\delta}(q_0,w)\in F\}$，并且我们称作 L 是**正则语言(regular language)**

我们通常使用**正则表达式（regular expression）**来表示正则语言。
对于一个字符表（alphabet）Σ

$\varnothing,\varepsilon,a$ 是对应语言 $\varnothing,\{\varepsilon\},\{a\}$ 的正则表达式，其中 $a\in\Sigma$

如果 r 和 s 是对应于正则语言 L_r 和 L_s 的正则表达式

那么有以下符号：$r+s(r\mid s):L_r\cup L_s$ 出现 r 或出现 s

$rs:\{wu\mid w\in L_r\wedge u\in L_s\}$ 字符串拼接

$r^k:\{w_1w_2\cdots w_{k-1}w_k \mid w_1\in L_r \wedge w_2\in L_r \wedge\cdots\wedge w_k\in L_r\}$允许有 k 次 Lr 重复出现

$r^+: L_{r^1}\cup L_{r^2}\cup L_{r^3}\cup\cdots\cup L_{r^n}\cdots$允许有穷次不定个数（至少一次）重复出现

$r^*: L_{r^+}\cup\{\varepsilon\}$允许有穷次不定个数（可以 0 次）重复出现

但是 DFA 不能解决$\{0^n1^n \mid n\in N^+\}$的问题，这时候就需要图灵机。

磁带：磁带被切分成一个个单元，每个单元中有一个符号。磁带是无限长的。
磁带头（type head）一开始指向输入符号的开始。

转移有如下形式：$(s1\to s2,d)$，其中$d\in\{R,L\}$代表当我们磁带头遇到 s1 的时候，将磁带头指向的符号 s1 修改为 s2，然后按照 d 来决定向左还是向右移动磁带头。

一个图灵机由七元组组成 $A=(Q,\Sigma,\Gamma,\delta,q_0,B,F)$

Q: A finite set of states（确定且有限的状态集合）

Σ: A finite set of input symbols（确定且有限的输入的符号集合）
Γ: The complete set of tape symbols. $\Sigma\subset\Gamma$

δ: A transition function（状态转移函数） $Q\times\Sigma\to Q\times\Gamma\times\{L,R\}$：也就是 Q和Σ的笛卡尔积，到状态的函数，也就是任意一对 Q 和Σ，需要能够映射到下一个 Q、修改的符号和左右移动的方向

q_0: The start state（开始的状态） $q_0\in Q$

B: The blank symbol. （磁带上的空白符号） $B\in\Gamma\wedge B\notin\Sigma$

F: A set of final or accepting states. （结束或接受的状态集合），且满足 $F\subseteq Q$

The Language of a TM

- Similar to DFA, the language of a TM $\boldsymbol{M}=(\boldsymbol{Q},\boldsymbol{\Sigma},\boldsymbol{\Gamma},\boldsymbol{\delta},\boldsymbol{q_0},\boldsymbol{B},\boldsymbol{F})$ is set of strings that it accepts, or:

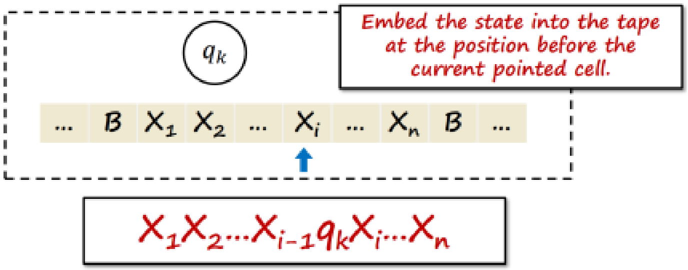
$$L(\boldsymbol{M})=\{w|w\in\Sigma^*\wedge p\in F\wedge q_0w\vdash^*\alpha p\beta\}$$

- For a certain language L, if there exists a TM $\boldsymbol{M}$ such that $L=L(\boldsymbol{M})$, then we call L is a **recursively enumerable language(递归可枚举语言)**, or ***RE***.

We'll talk more about RE in the next courses.

Instantaneous Descriptions for TM（图灵机的即时描述）

也就是把当前状态 q_k 插入到当前指向的磁带头的位置



用 $\vdash_M$ 来描述图灵机的移动

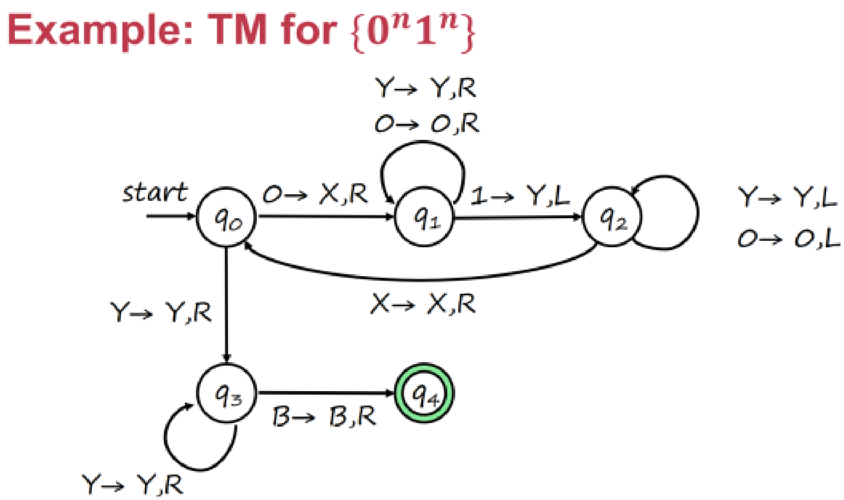
比如 $\delta(q,X_i)=(p,Y,R)$ 代表的是在 q 状态下遇到符号 X_i 后，会转移到 q 状态并且修改 X_i 为 Y，并向右移动一个单位。故有 $X_1X_2\cdots X_{i-1}qX_iX_{i+1}\cdots X_n \vdash_M X_1X_2\cdots X_{i-1}YpX_{i+1}\cdots X_n$

比如 $\delta(q,X_i)=(p,Y,L)$ 代表的是在 q 状态下遇到符号 X_i 后，会转移到 q 状态并且修改 X_i 为 Y，并向左移动一个单位。故有 $X_1X_2\cdots X_{i-1}qX_iX_{i+1}\cdots X_n \vdash_M X_1X_2\cdots pX_{i-1}YX_{i+1}\cdots X_n$

$\vdash_M^*$ 代表 0 次、1 次或多次移动。

Example: TM for $\{0^n1^n\}$

- Our solution:
  - When we see a $\mathbf{0}$ at the beginning, we change this $\mathbf{0}$ to $\boldsymbol{X}$ to indicate we have counted it, and try to find an $\mathbf{1}$ by moving tape head to right.
  - When we find an $\mathbf{1}$, we change this $\mathbf{1}$ to $\boldsymbol{Y}$, and turn back to find another $\mathbf{0}$ at the beginning (right after an $\boldsymbol{X}$).
  - Repeat the above process when there are only $\boldsymbol{X}$ and $\boldsymbol{Y}$ left on the tape, then we accept the input. Otherwise we reject the input.



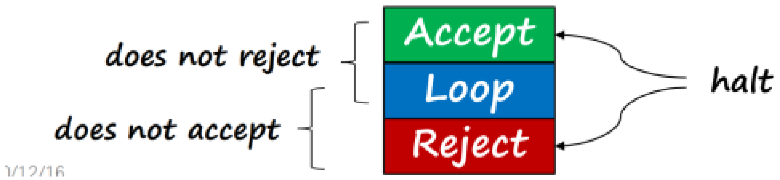
对于一个输入的字符串，图灵机的输出可能是以下三种情况：

1. 图灵机接受这个字符串；2.图灵机拒绝这个字符串；3.图灵机无限循环

一些概念

Let M be a Turing machine and let s be a string.

– M **accepts** s if it enters an accepting state when running on s
– M **rejects** s if it enters a rejecting state when running on s.
– M **loops infinitely** on s when running on s if it enters neither an accepting nor a rejecting state.
– M **does not accept** s if it either rejects s or loops on s.
– M **does not reject** s if it either accepts s or loops on s.
– M **halts** on s if it accepts w or rejects s.

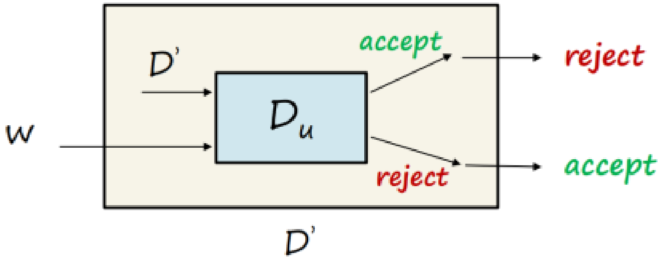


拥有以上行为的图灵机，我们称 TM 是一个 recognizer，能够 recognize（识别）语言 L，语言 L 是图灵可识别的（Turing-recognizable）。

对于一部分图灵机，不会进入死循环，一定能在有限步后停机。我们称这种是 decider。这种图灵机识别的语言，是图灵可判定的（Turing-decidable），称作 recursive language（递归语言），递归语言（R）是递归可枚举语言（RE）的子集。

那么如何证明 R 是 RE 的子集，$R \neq RE$ 呢？

首先需要知道通用图灵机（universal Turing machine）的概念。通用图灵机把一个图灵机和一个字符串作为输入，输出那个图灵机输入字符串的结果，记为U_{TM}。如果 M 接受 w，那么U_{TM}接受$\langle M,w\rangle$；如果 M 拒绝 w，那么U_{TM}拒绝$\langle M,w\rangle$；如果 M 在 w 上死循环，那么U_{TM}在$\langle M,w\rangle$上死循环。



D' accepts $w \Leftrightarrow D'$ rejects w

D_u can not exist!

构造出如上图所示的通用图灵机的反例，及即可证明这个图灵机D_u对应的语言L_u并不是图灵可判定的。

如何得到 Non-RE 的语言？

首先，我们需要知道所有图灵机可以通过 01 字符串编码，从而得到一个能够和正整数一一对应的有序图灵机序列，字符串同理。

那么我们可以通过构造，得到一个任意图灵机都无法识别的字符串的集合，也就是得到一个 language，称为对角语言，记做L_d。满足$L_d=\{w_i \mid w_i \notin L(M_i)\}$，其中 L 是递归可枚举语言 RE。或$L_d=\{M \mid M \notin L(M)\}$。

	1	2	3	4	5	6	...
M_1	0	0	0	1	1	0	...
M_2	0	1	1	0	1	1	...
M_3	0	0	1	1	0	1	...
M_4	1	1	0	1	0	1	...
M_5	1	0	0	0	0	0	...
M_6	0	1	1	0	1	0	...
...	...	...	...	...	...	...	...
	1	0	0	0	1	1	

Flip this language. Swap what's included and what's excluded

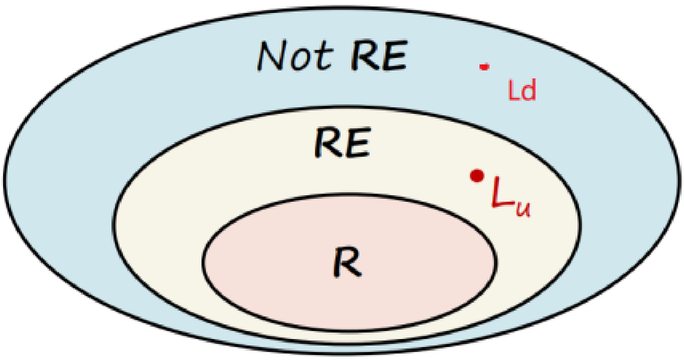
- The language L_d, the diagonalization language, is the set of string w_i such that w_i is not in $L(M_i)$

$$L_d = \{w_i | w_i \notin L(M_i)\}$$

- Notice that the code for M_i is just w_i, so we can write L_d in a more interesting way:

$$L_d = \{M | M \notin L(M)\}$$

- The diagonalization language is all the codes of Turing machine which does not accept its own code.



图灵机附录：

图灵机的编码方法（Encoding TM）

Encoding TM

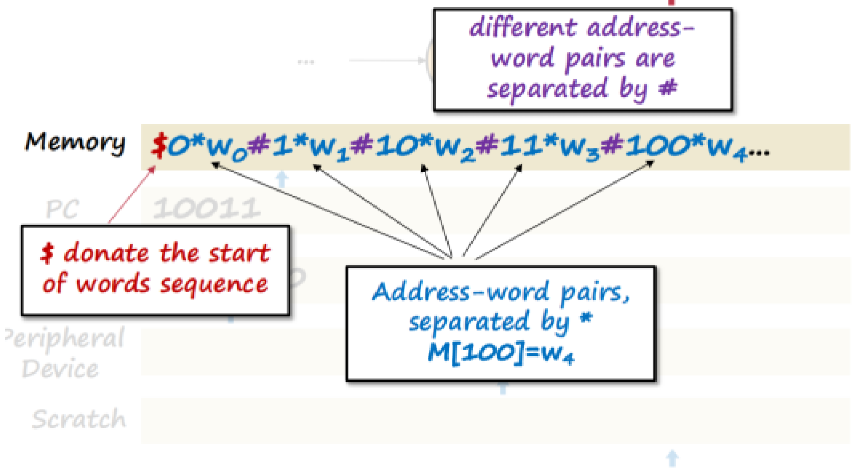
- In order to take a Turing Machine as an input, we need to encoding the TM. Similarly, we shall encode TM with binary.
- We first assign integers to the states, tape symbols and directions
  - We assume the states are $q_1, q_2, \ldots, q_k$ for some k. $\boldsymbol{q_1}$ is the **input state** and $\boldsymbol{q_2}$ is the **only accept state**.(Is it right?)
  - The tape symbols are $X_1, X_2, \ldots, X_m$ for some m. $\boldsymbol{X_1, X_2, X_3}$ **are** $\boldsymbol{0, 1, B}$ respectively.
  - Refer to direction $\boldsymbol{L}$ **as** $\boldsymbol{D_1}$, $\boldsymbol{R}$ **as** $\boldsymbol{D_2}$

Encoding TM

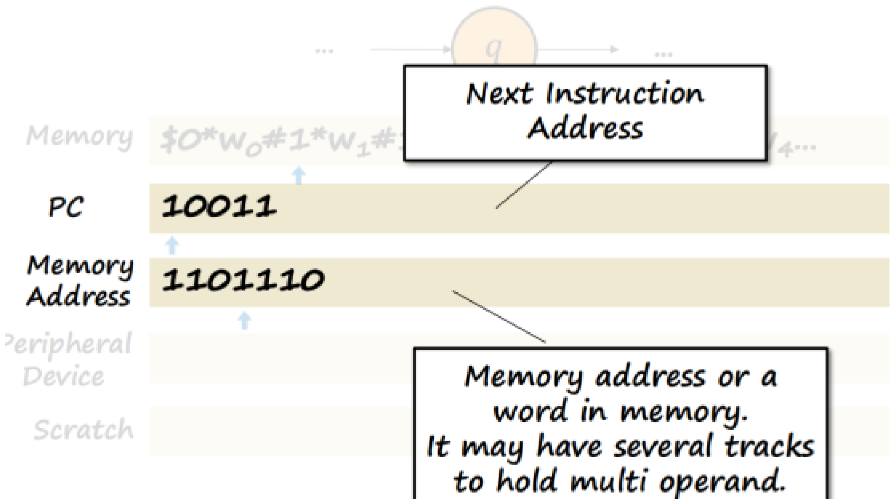
- After assigning each state, symbol and direction an integer, we can encode the transition function $\boldsymbol{\delta}$.
- Suppose one transition rule is $\boldsymbol{\delta(q_i, X_j) = (q_k, X_l, D_m)}$, we shall code this rule by the string $\mathbf{0^i10^j10^k10^l10^m}$.
  - Notice $\boldsymbol{i, j, k, l, m}$ are at least one, so there're no occurrences of two or more consecutive 1's with in the code for a transition
- So let $\boldsymbol{C_i}$ donate the code for the ith transition rule, we can encode the whole TM as:
  - $C_1 11 C_2 11 C_3 11 \ldots 11 C_n$

图灵机模拟现代计算机：

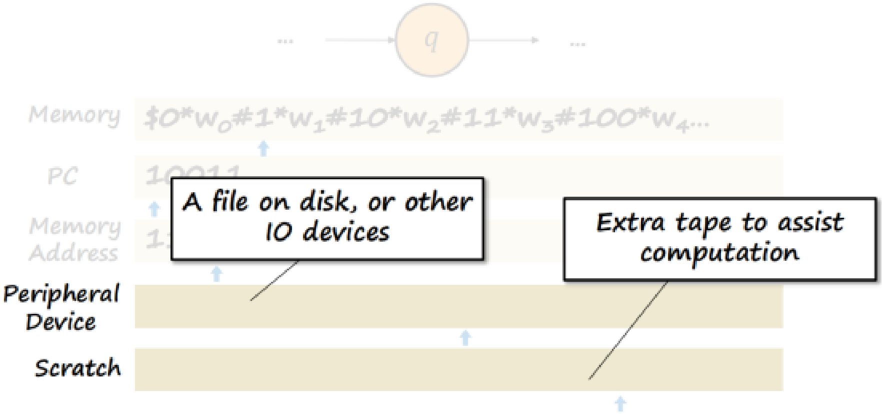
A TM Simulates Idealized Computers.



A TM Simulates Idealized Computers.



A TM Simulates Idealized Computers.



TMs and Computers

- We've informally proved that a TM can simulate computers by simulating its instruction cycle.
  - It may require a leap of faith to believe since we do not prove it rigorously.
- So we know TM and computer can simulate each other, which means that TM has same computation capability with computer.

$$TM \approx Computer$$

Effective Computation

- An ***effective method of computation*** is a form of computation with the following properties:
  - The computation consists of a set of steps.
  - There are fixed rules governing how one step leads to the next.
  - Any computation that yields an answer does so in finitely many steps.
  - Any computation that yields an answer always yields the correct answer.
- This is not a formal definition. Rather, it's a set of properties we expect out of a computational system.

Church-Turing Thesis

Every effective method of computation is either equivalent to or weaker than a Turing machine.

- This is not a **theorem** but a falsifiable scientific hypo**thesis**. But it has been thoroughly tested! So we have strong faith in its correctness.
- This means Turing Machine can model any “computation”.