关于 KRP 课程的内容梳理

赵一铮

*本文内容只代表作者的一些个人思考，并没有被纳入哪个官方文件或者哪本教材，如有不同观点，欢迎交流指正。我们希望把整门课程讲成一个逻辑严谨、通俗易懂的故事。

2022/4/27

本课程将围绕“定义”（definition）的两个不同类型展开讨论和思考：
（1）内涵式定义（intensional definition）：define a term by specifying the necessary and sufficient conditions of being it
（2）外延式定义（extensional definition）：define a term by listing everything that falls under that definition

比如 mother，使用内涵式定义：a female who has a child。female 是成为 mother 的一个必要条件，has a child 是成为 mother 的另一个必要条件，合二为一变成了 mother 的充分必要条件；而使用外延式定义会列出目前世界上所有 mother 的个体。这个个体的集合就是 mother 的外延式定义。

看待世界的方式：面向对象

学习这门课程不只是简单地对这个分类进行记忆，还要思考更深入的事情，某个事物的底层哲学。假如把当前的物理世界（physical world）看作是一个 universe，universe 内存在各种各样的 element。根据颗粒度（放大镜倍数）的不同，element 指代的事物也不同，它可以是原子、分子层面的事物，也可以是人、建筑层面的事物，还可以是比这些更为微观或者宏观的事物，只要“存在”即可。什么是“存在”？

existence: the ability to interact with the physical world

英文的解释是一个实体与物理世界的交互能力，有点哲学，不好理解。我自己给出这样一个定义：只要是能让这个物理世界或者其中 element 发生改变的任何事物都表明了该事物的“存在”，任何一点点小小的改变都可以。比如，我搬起石头，石头的位置变了，则我是存在的，石头也是存在的（位置改变了）；你的批评让另一个人委屈了，则这份批评是存在的，另一个人的委屈也是存在的。不仅仅是物理的改变，情绪、情感等都可以发生改变。

being 指的是任何存在的事物

举例：living being 生物；human being 人类；alien being 外星人；love being 爱；

becoming: 指的是 being 的变化

reality: 指的是一个充满 being 的世界

entity: 独立存在的 being

什么是“独立存在”？

比如，一块肉是独立存在的吧？这块肉就是 entity；但是这块肉在不切割的情况下，它的四分之三，就不是 entity，只是 being，因为它的四分之三并不独立存在。但如果用刀把这块肉切开，一份是四分之三，一份是四分之一，这两份肉又成了独立的 entities。

针对每一个 entity，我们首先应该为其取一个名字，这个名字是以“语言”来承载的。不管是什么样的语言，总之是某种语言。这样这个 entity 就能有了除了它自身之外的第一种表示。这样做的好处是方便交流，当 A 提到 entity X 的时候，与 A 有同样认知的 B 明白 A 指代的是哪一个 entity。entity 只有名字就够用了么？那么 entity 之间不就完全没有任何联系，都是一个个 isolated 符号名称而已么？显然是不够的，接下来我们还得对每一个 entity 进行定义，这个 entity 是什么材质？做什么的？大小如何？形状如何？是人还是非生命类物品？一个 entity 展示给这个世界的所有信息都是形成这个 entity 的必要条件。同一个 entity，看待的角度不同，形成这个 entity 的必要条件也不同。比如 Lily 这个人，她是一个 mother，也是一个 teacher，成为 mother 的必要条件和成为 teacher 的必要条件显然是不一样。但所有的一切的必要条件形成了 Lily 这个独一无二的 entity。不同的 entity，它们可能会在某些特点一致，比如 Lily 和 Lucy 都是老师，而且都是母亲，这些 entities 会由于某些相似的特点，被归为同一种群体。但是我们上课讲过，群体往往是一个概念层面的事物，不是独立存在的。比如 Lily 是一位老师，她是老师这个群体的一个 instance，是一个 entity。但是老师这个群体是看不见摸不着的，没有哪个 entity 可以独立代表老师这个群体。所以我们把这种群体叫做 concept。Teacher 是一个 concept，Mother 也是。Concept 像是一种集合。

Entity 与 entity 之间可以交互、可以存在某种关联。作为集合概念，concept 与 concept 之间也可以，比如包含关系（Dog 和 Animal），互斥关系（Female 和 Male），相交但互不包含关系（Animal 和 Male），通过某些关系建立关联的关系（Human 和 Dog 之间可以通过 hasPet 联系起来）。思考一个问题：是区分人和海豚这两个群体容易？还是区分老师与律师这两个群体容易？还是区分南京大学学生与浙江大学学生这两个群体容易？还是区分 Lily 与 Lucy 这两个人容易？显然是难度越来越大。如果用一个包含特征的集合来区分两个群体，集合里面的特征共同构成了区分两个 concepts 的充分必要条件。人和海豚可以通过包含相对较少特征的集合区分，而 Lily 与 Lucy 就要用更多的特征，因为她们都属于 human，性别都为 female，有着相同的 occupation，区分她们需要更多更细节的特征，比如她们的面部特征、声音特征等等。换句话说，相似点越多的 entities 之间，concepts 之间需要更多更细节的特征区分。这也是为什么，对于机器学习来说，有效的数据越多，算法的结果越好。

*说到这里我们从内涵式定义和外延式定义角度区分下“知识表示与推理”和“机器学习”这两个 AI 子领域的底层哲学（个人一直觉得这个部分是课程中非常重要的讨论）：

知识表示与推理研究的重点是事物的内涵式定义，以及通过内涵式定义的“语义”（目前是基于集合论和模型论）来进行智能计算行为。外延式定义集合中的每个元素可以看作是内涵式定义的一个 instance，比如 Lily 是 Mother 的内涵式定义的 instance。在物理世界里，这样的 instance 通过其它方式表示出来（data 或者一组 data）。这个世界上某个 concept 所有的 instances 合在一起反映了某个 concept 完整的内涵式定义，但遗憾的是这样的 instances 很可能是无限的。机器学习研究的重点是事物的外延式定义，以及通过外延式定义的“语义”来计算智能行为。通常是通过一组 training data 来尽可能学习到某个内涵式定义或者内涵式定义的片段（大部分情况下是片段，想学到完整的内涵式定义需要完整的数据集、完美的算法、强大的算力，理论上是不可能的）。既然外延式定义的 instances 可能是无限的，同样也受算法和算力所限，所以作为 training data 出现的就只是一部分，甚至是一小小部分的 instances，它天生无法完美地反映正确的完整的内涵式定义。所以机器学习有一个重要的课题是评估一个算法（比如分类算法）的好坏，它有多大能力去在 training data 上学习一个“内涵式定义”，再用这个定义在 test data 上展现其学习能力。

说到这里，我片面认为：只有内涵式定义才有机会称为知识。外延式定义，无论表示成什么形式，都只是知识的片面展现。最近几年，因为数据 + 知识，学习 + 推理的研究方向变得逐渐热门。一些工作尝试探索新的学习模型，并声称是将知识融入到了模型之中来提升 XX 性能。比如一种常见的方式是，引入一个知识图谱，或者其它方式表示的知识库，把知识图谱的内容映射到连续向量空间，比如 TransE，融入到模型之中。如果取得了好的实验结果则声称是知识带来的加成。这真的是知识带来的吗？首先，经典知识图谱是三元组组成的，类似于 ABox 之中的 role assertion（或者带有 nominal 的 TBox 的 concept inclusion），建立的是实体与实体之间的关系（或者 nominal 与 nominal 之间的关系）。这样的三元组不都是知识，比如“地球围绕太阳转”（$\{earth\} \sqsubseteq \exists orbits.\{sun\}$，这里我们用带有 nominal 的 TBox 来表示）是知识，可是“小明和小红是朋友”（$\{xiaoming\} \sqsubseteq \exists isFriendOf.\{xiaohong\}$）也是知识吗？俩人闹掰了怎么办？其次，知识转化为向量表示（相当于内涵式定义转化为外延式定义的 a set of instances），会失去原始的语义，转化过程涉及信息的损失和扭曲；最后，对于保留的信息，在模型训练过程中究竟用到了多少、怎么用的也不得而知。整个过程不就相当于“数据的扩充增广”吗？只不过新加入的数据的原始形态是知识图谱，给人一种加入了知识的假象。这样的工作只适合发 paper，糊弄不懂知识表示与推理的 reviewers，但对于知识 + 数据的 AI 范式的探索毫无推进作用，也没有底层哲学的启迪作用。因为人类使用知识的时候并不一定将其外延化，什么时候外延化的边界现在还不知道。

关于“知识”的定义和来源背后有一个宏大、旷日持久的争论。这是哲学里面关于认识论（epistemology）的研究范围。其中有很多不同的学派（school of thought）和代表人物。比如经验主义学派（empiricism），理性主义学派（rationalism），实用主义学派（pragmatism）、相对主义学派（relativism）等。这些学派之间的思想有的是互斥的，有的是相交但不重合，有的是包含（一种学派下面的子学派）。各个学派之间看待事物的角度不同，所以导致它们的思想并不一定都是互斥。我们简单提两个既互相对立、互相斗争，又互相影响、互相渗透的学派：经验主义学派和理性主义学派。在欧洲哲学史上，哲学家把这两种思想的冲突以及解决这两种冲突的不懈努力提到全部哲学的中心地位上来。

经验主义学派核心思想：知识仅来源于或主要来源于感官经验（knowledge comes only or primarily from sensory experience）

代表人物：弗兰西斯·培根（Francis Bacon）、托马斯·霍布斯（Thomas Hobbes）、乔治·贝克莱（George Berkeley）、约翰·洛克（John Locke）、大卫·休谟（David Hume）
还经常会听到联结主义（connectionism）这个词，它是实现经验主义的一种方法，核心思想是：智能行为通过人工神经网络实现（intellectual abilities via artificial neural networks）

理性主义学派核心思想：知识来源于理性和演绎，不依赖于感官经验（knowledge comes from reason and deduction）
代表人物：勒内·笛卡尔（René Descartes）、巴鲁赫·斯宾诺莎（Baruch Spinoza）、戈特弗里德·莱布尼茨（Gottfried Wilhelm Leibniz）

有意思是的，经验主义被广泛认为起源于欧洲大陆，但代表人物却都来自英伦；而理性主义被广泛认为起源于英伦，但却在欧洲大陆发扬光大。两个学派是认知内涵之争。一个外延的 instance 就是机器学习和知识表示与推理两个 AI 范式。这两种不同的哲学思想，体现在不同学科领域的研究方法的差异上，不仅仅是 AI 领域。以自然语言处理领域为例，带有经验主义色彩的工作包括早期的安德烈·马尔可夫（Andrey Markov）、克劳德·香农（Claude Shannon）；而理性主义的代表工作是诺姆·乔姆斯基（Noam Chomsky）的有限状态自动机对应语言模型。经验主义倾向使用概率和随机方法来研究语言，建立语言的概率模型，这种方法适合处理浅层次语言现象以及词语近距离依存关系。理性主义倾向使用符号表征方法来研究语言，适合处理深层次语言现象和词语长距离依存关系。现阶段，自然语言在经验主义方法的加持下取得了巨大的进步。不同的哲学思想甚至会外延到有神论和无神论之争。理性主义思想假设万物皆有“真理”，知识来源于真理和真理上的演绎，支持有神论，并相信神才是那个掌握真理的；而经验主义体现无神论，强调知识来源于人民群众的智慧。看待世界的不同方式（不同的哲学思想）衍生了不同的科学研究方法。英伦系科学家比如艾萨克·牛顿（Isaac Newton），他的科学思想带有明显的经验主义色彩。牛顿认为自然哲学只能从经验事实出发去解释世界事物，因而经验归纳法是最好的论证方法。他曾表达：“虽然用归纳法来从直言和观察中进行论证不能算是普遍的结论，但它是事物本性所许可的最好的论证方法，并随着归纳的愈为普遍，这种论证看起来也愈有力”。而理性主义学派代表人物莱布尼茨与经验主义学派代表人物牛顿之间关于谁先发明 calculus 的争论也成为了旷日持久的世纪争论（The Leibniz – Newton calculus controversy）。原始观点认为牛顿首先开启和完成了关于 calculus 的工作，但莱布尼茨发表得更早。现代观点认为二者各自独立开展了关于 calculus 的工作，是不是也意味着不同派别的思想其实有着更深层次的哲学来兼容所谓的理性主义学派和经验主义学派，从而衍生更先进，更接近人类本质的 AI 范式？其实我们使用两种不同哲学在 AI 领域的研究，以及在其它学科领域的研究，本质上都是在实践两种不同的哲学思想，我们都是哲学的外延式实践。

平时在学习和研究中发现，从事知识表示与推理研究的学者大多来自大陆系国家，比如德国、法国、意大利、奥地利、东欧国家；而学习领域的几位奠基人大多来自英美系国家，比如 Donald O. Hebb、Frank Rosenblatt、Geoffrey Hinton、Rich Sutton、Michael I. Jordan 等（从名字就可以看出，如果发现谁的国籍不是，也只是国籍不是，追溯一下其导师，或者导师的导师，发现最终会回归到英美系学者）。无聊的时候和身边人一起研究过各自的导师、导师的导师、导师的…的导师，发现最终都会追溯到莱布尼茨、高斯、拉格朗日。

知识表示与推理的内涵式定义（通常是一个符号表达式）的外延式语义通过 interpretation 来体现（所以通常这样的 interpretation 有无数个），但 interpretation 也是抽象层面的（比如

interpretation 定义的 domain 是不是想象的？里面的 element 是不是想象的？），不是现实中真实的数据。机器学习接触的是真实的 data。机器学习做的是一种“表”；知识表示与推理做的是一种“里”，也是一种“理”。机器学习更像是一种实验科学，一种实用的工程；知识表示与推理更像是一种理论科学，一种头脑的思维。机器学习算法的输入是数据，输出是一个与人类智能得到的结果“相似”的结果（有时候更糟、有时候更好、有时候差不多），中间的计算过程，很多算法与人类处理相同问题使用的方法并不一致。传统的机器算法基于统计和概率模型，而联结主义的神经网络目前不可解释。知识表示与推理的算法输入是一组逻辑表达式（现阶段主流用逻辑语言表示，理论上也可以不用逻辑语言，但得建立一种新的可计算的语义解释模型），输出是一个只可能比人类智能得到的结果“更好”或者至少一样的结果（因为是“理”的结果，是客观上正确的结果），中间的计算过程透明。

机器学习的缺点在于外延数据的天然不完备、数据质量难以保证、部分计算过程不透明、结果缺乏可解释性。知识表示与推理的缺点在于从来不与真实的数据打交道，不与真实世界交互，所有的研究进展都体现在对“理”、对“内涵”的探索上，使得该领域的研究与真实世界难以沟通，难以产生有实际应用价值的贡献；其次，知识的获取无法做到自动化（如果承认内涵式定义才算知识），这使得一个知识库的构建变得异常艰难（知识对不对？全不全？有没有冗余？）。还有，对于一些不确定性知识，不是非黑即白的知识难以很好地表示；最后，计算透明性的极致要求导致中间的推理过程计算复杂度极高，会受到算力的限制（所以要学习知识表示语言的可判定性和计算复杂度）。

在机器学习的科研中，当用算法 1 解决了一个问题 A，它可以平移到问题 B 去尝试效果，也可以有算法 2 来尝试解决问题 A。算法与问题之间是一个多对多的函数。面对同一个问题，人们根据现有的 SOTA 算法来开发更好的算法，可能是你有了更好的数据，可能是有了更 powerful、smart 的算法，可能是其它。所以一个问题一个算法可以有很多的 followers 去跟进研究，论文引用率往往高。知识表示与推理的科研中，问题与推理方法往往是一对一的函数关系，个别的是一对多，但这个“多”往往<3。机器学习的问题是外延式，同一类问题的 applications 往往很多，针对不同的 applications 都可以提出新的算法，哪怕只是准确率小小的提升，每一个新的算法都是一篇 paper。知识表示与推理问题是内涵式的，同一类问题只有一个内涵式问题本身，很少有外延式的 applications。问题解决就是解决了，解决的方式往往只有有限几种甚至是唯一方法。这导致客观上知识表示与推理的“理”的问题不如机器学习“表”的问题多，但其实本质上，它们研究的都是同一个“理”，底层哲学并没有区别。

从解决问题的类型上看，机器学习特别适合“学习”“不太需要可解释性”的任务，因为学习天生就是从物理世界的片段中总结“理”再去运用“理”的过程，计算重结果不重过程。知识表示与推理更适合“理性思考”“需要可解释性”的任务，比如医疗、法律等领域的部分问题，因为知识都是内涵式定义，过程的透明性才是第一位的。知识表示与推理最值得骄傲的是“theoretical soundness”，而机器学习展现了“empirical success”。

越来越多的 AI 学者开始意识到，仅仅追随某一种哲学，无论是机器学习还是知识表示与推理，都难以解决现实生活中所有的问题，或者难以模拟人类真实的智能行为。因为在面对现实问题的时候，人类的智能往往是“感知”与“理性”相结合，甚至二者不是串行发挥作用，而是有时串行、有时并行发挥作用。如何将二者有效地结合，科学地结合，需要我们对要解决的问题进行归类（哪些问题适合哪一种哲学，哪些不适合），需要我们对这两

种哲学的边界有着更深层次的思考（哪一种哲学可以解决哪些问题），或许需要我们对内涵式定义（知识）和外延式定义（数据）找到统一的知识表示方式，才有可能完成知识与数据的交互，才有可能找到学习+推理结合的契机和着陆点，才能继续推进 AI 这个领域的前进和发展。在这个过程中，我们也会更加懂得人类自己。

很多国内高校也开设了关于“知识表示与处理”的课程，但最终都是以“向量”等表示方式嵌入（one-hot encoding、bags of words、word embedding、knowledge graph），这样的知识表示方式适合深度学习模型，但计算过程依然不透明。少部分高校也有一些涉及逻辑语言作为知识表示语言的课程，但属于入门级的讲授，简单介绍一下，不涉及知识表示与推理的核心思想、理论、技术及证明。我们希望在这门课上，讲清楚这个领域的故事和它所承载的哲学思想。

话题：什么是人工智能？学习它为什么重要？

人工智能从狭义上讲，指的是让机器模拟人的智能行为（intelligent behavior）。就这样一句简单的解释，需要思考的问题却不少。第一个问题是，为什么要让“机器”，或者说得更狭义一点，要让“计算机”去模拟？为什么不是让猫、狗、或者昆虫去模拟？那一定是计算机本身有一些特别的性质，促使我们有兴趣去研究，这个特别的性质就是计算机的特性、优点、尤其是比起人类的优点。这个问题可以换一种问法：计算机比起人的优势在哪里？这个同学们应该去搜索很多资料，说法很多，但是普遍承认的有这样几个优势：
（1）计算机处理信息的速度要比人类快；
（2）计算机“记忆力”好，它们的大脑里可以装入很多信息而不遗忘，而人类的存储量就相对较少，忘性极大；
（3）计算机“体力”好，长时间工作不需休息，是全天候的（round-the-clock）。不像人类，工作一段时间就需要进食，进食完就想睡觉，电能支持的比生物能支持的先天优势；
（4）计算机“可信度”高，他们计算的结果不犯错误，除非程序有错误（但程序是人写的，计算机只负责执行），人会受到情绪、状态、感受、感情等影响，犯错几率比计算机大。比如，面对你的亲人朋友，你受情感影响很难保持客观。
（5）计算机“无人性”，适合去做一些需要智能但是物理上危险的工作。即便除了安全问题也不会引起人类的情感变化。

计算机比人类出色的地方，一个词概括，就是“计算能力”。两个词概括，加上一个“非人性”。所以，学习 AI 的原因之一，可能也是最重要的原因，是希望通过计算机强大的计算能力模拟智能行为，从而解放生产力和提高生产效率，深层目的是 advance humanity。

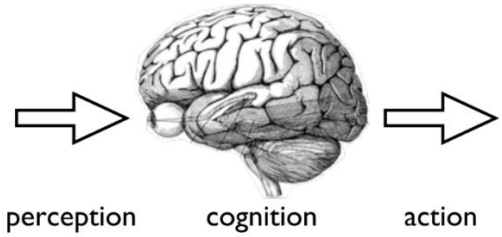
接下来一个问题：什么是人的智能行为？更广义地说，什么是智能行为？先尝试给出它的内涵式定义：

Artificial intelligence： using artifacts (often machines) to simulate intelligent behavior

然后是它的外延式定义：

Artificial intelligence： using artifacts (often machines) to simulate the abilities to see, hear, smell, feel, think, reason, calculate, communicate, read, understand, imagine…love

外延式定义的智能行为可以大致分为以下三类：



（1）perception（感知）：the abilities to see, hear, smell, feel, or become aware of something through the senses

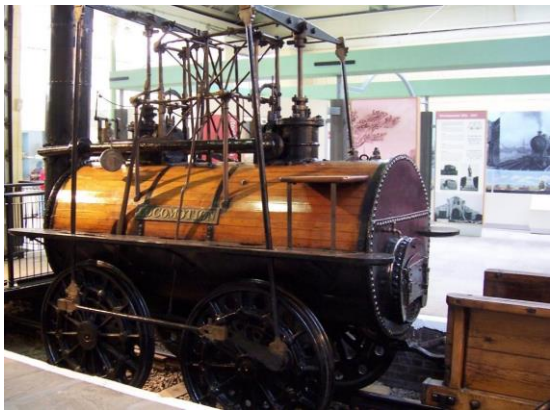
（2）cognition（认知）：the abilities to think, reason, calculate, remember, imagine, or other mental processes, and understand through thought, experience and the senses

（3）action（行为）：the abilities to move, act, speak…

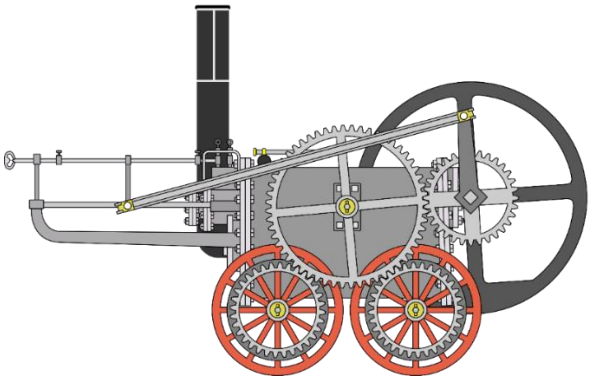
上面这个定义说智能行为是多种能力的组合，什么能力呢？从物理世界中感知，渐渐形成、补充、修正对世界的现有认知，再利用现有认知对周边环境做出行为反应的能力。人工智能旨在让机器完成上述智能行为或者其中一部分。其中一个重要的前提是这些智能行为是“可计算的”（computable）。只有可计算的“模型”，计算机才能够处理，人类处理信息的方式并不一定都是可以抽象为一个可计算的模型。

模型：对于世界上一个 entity 的抽象就是模型（model）。

比如现在要讲蒸汽机车的工作原理，有必要拿来真的蒸汽机车吗？



或许是不需要的…只需要展示给大家它的模型即可：



*注意区分“计算模型”和“可计算模型”：前者指各类计算机的模型，现存的计算模型包括：finite state machine，pushdown automata， Turing machine，Lambda calculus 等；后者指的是“被计算事物”的模型（数据模型），比如接下来要介绍的本体。

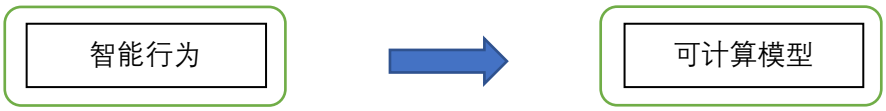
什么是“可计算的”？该怎么定义和理解“可计算的”？先理解“计算”：

compute: to execute a program (program = a set of instructions)

计算指的是执行程序的过程，程序解释为一组指令。连在一起就是按照一组指令，一步一步执行的过程就叫计算。一个计算的例子：要把大象关冰箱总共分几步？

（1）把冰箱门打开
（2）把大象装进去
（3）大冰箱们关上

按照这个步骤去执行的过程就叫做计算。可以完成计算这个行为的任何事物都叫做“计算机”。比如上述大象关冰箱步骤，人类可以完成吗？当然可以，如果大象不反对的话，所以人类本身就是一台计算机。这个就是广义的计算机的解释。随着时代的发展，计算机从机械计算机到现在的电子计算机、生物计算机。人工智能研究的是怎么把不同的智能行为变为可计算的模型，这个过程叫做“建模”：



想要一个东西可计算，一个通用方式就是转化为数学模型，这也是为什么我们要在大一大二学习那么多数学基础课程，比如高等代数、数学分析、概率论与数理统计、最优化方法、信息论、数理逻辑等。数理逻辑是知识表示与推理的数学基础。

一个数学建模的例子：m 元钱，投资 n 个项目。效益函数 $f_i(x)$，表示第 i 个项目产生 x 元的效益，i=1,2,⋯,n。求如何分配每个项目的钱数使得总效益最大？

实例：5 万元，投资给 4 个项目，效益函数 $f_i(x)$如下表：

x	$f_1(x)$	$f_2(x)$	$f_3(x)$	$f_4(x)$
0	0	0	0	0
1	11	0	2	2
2	12	5	10	21

3	13	10	30	22
4	14	15	32	23
5	15	20	40	24

建模过程：
输入：n, m, $f_i(x)$, 其中 i=1,2,⋯,n, x=1,2,⋯,m
解：n 维向量<$x_1, x_2, \cdots, x_n$>，xi 是投到第 i 个项目的钱数，使得满足下面条件：
目标函数：

$$max \sum_{i=1}^{n} f_i(x_i)$$

约束条件：

$$\sum_{i=1}^{n} x_i = m, x_i \in \mathbb{N}$$

根据这样的模型，我们可以设计不同的算法来完成，比如这里最直接的算法就是*蛮力算法*—就是计算出所有的可行方案，然后找出受益最大的那个。显然这个算法的复杂度会很高，是对于 m 和 n 是指数增长的。

上述的建模过程相对简单，但是对于人工智能的建模不会那么直观。看下面一张照片：

从这张照片我们接收到的信息是一条狗，是一条幼年金毛，它坐在开满小黄花的草坪上，张着嘴卖萌。这个过程涉及到我们用眼睛去看（感知），再把图片中的物体与先验知识结合起来（认知）。我们知道这个样子的是狗，是金毛，是幼年金毛等等信息。现在把这张图片输入给计算机，让它也像人类一样通过“感知”“认知”处理这张照片，得到与我们一样的信息。这个“感知”“认知”的过程是通过“计算”的手段实现的。

这门课的内容主要涉及如何将“知识”表示为可计算的模型（一组逻辑表达式），并在模型上进行计算（推理）获得新的知识。这个计算基础是数理逻辑（数理逻辑包括集合论、模型论、证明论、递归论），具体说来，是基于集合论和模型论的知识表示方法和语义解释方法。基于这样的理论，我们可以清楚的看到计算机是如何对知识进行表示，如何对这种表示的知识进行理解，是如何基于这种表示和理解进行推理的。

人类思考的过程中，需不需要触碰现实中真实存在的 entity，还是只针对那个物体在我们大脑中的抽象进行思考？显然是后者。我们思考的对象不是现实中的 entity，而是这个 entity 在我们大脑中的一个概念映射（conceptual mapping）。这个概念映射是用语言来承载的。理想的知识表示语言是：

（1）有足够的表达力表示（物理世界中的）知识
（2）人类可以理解它的语义
（3）机器可以理解它的语义，且该语言可计算（是一种数学语言）

为此我们提出使用 formal language，它有三个特点：

（1）有确定的字母表（意味着语言中单词的组成只能使用字母表里面的元素）
（2）有确定的语法规则（意味着必须按照这个语法规则组成复杂短语或句子）
（3）有确定的语义解释（意味着必须按照这个方法去解释句子中的每个元素）

自然语言属不属于 formal language？虽然自然语言满足前两点，但是对于第三点自然语言是不满足的。同一词汇，不同的人看，解释可以是不一样。比如“黄瓜”，大多数人的理解是 cucumber，但是有些地区 cucumber 被称为“青瓜”，而黄瓜指的是另一种蔬菜，见下图：

再比如，“饭”这个词，一些人理解为 meal，也会有人理解为 rice。“跳脚”这个词既可以

指一种动作，也可以引申为“气急败坏”。英文也是如此，一词多义比比皆是。

典型的 formal language 有 programming language 和 logical language（二者合起来可以是 logic programming language，logic programming 也是一个很有意思的研究领域）。逻辑语言中，我们把一个逻辑表达式称为 a theorem 或 axiom，把 a set of axioms 称为 a logical theory。逻辑语言可以满足知识表示语言的三个条件，未来可能有某些方面更适合作知识表示的语言；同时也不能否认，formal language 比起自然语言，也有其局限性。比如自然语言的模糊性，有的时候事物并不是非黑即白的存在，而是有一定的“fuzziness”。“六点半左右”“黄色偏灰”等等，这种 fuzziness 可以被 formal language 捕捉吗？这种 fuzziness 可以被计算吗？如果可以，这个数学模型该如何建立？

接下来开始本体（ontology）的讲解。首先，如何定义本体？因为我们学习了描述逻辑，是否可以直接定义本体为 description logic-based knowledge base；或者说，因为我们介绍了逻辑语言，是否可以直接定义本体为 logic-based knowledge base？答案是不行的。

哲学领域的本体研究的是 entities 以及 entities 是如何分类的（the classification of entities）。计算机领域本体该如何定义？Siri 之父 Tom Studer 给出的定义是：

ontology: a formal, explicit specification of a shared conceptualization

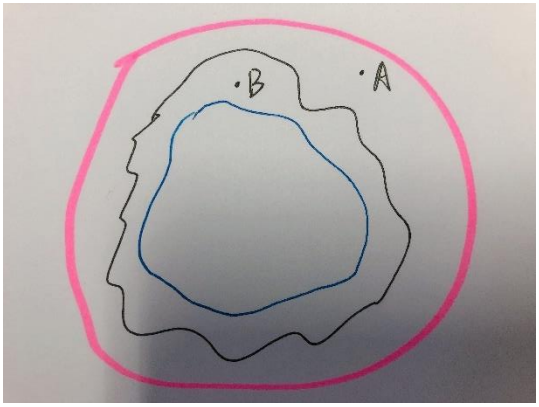
这个定义也是目前最为广泛接受的一个定义。分析一下这个定义：

conceptualization 是什么？指的是一个 physical world 的抽象。比如有这样一个 physical world，里面有实体小金毛，实体小绵羊，实体小老虎，实体小河马。那么这样一个 physical world 的抽象就是一个 conceptualization。Conceptualization 取决于个人（dependent on individuals）。因为对于同一个 physical world，不同的人形成的 conceptualization 可能不一样。有的人认为实体 a 属于概念 A，有的人认为属于概念 B。该听谁的？不可能对每一个 conceptualization 都做一个 ontology 吧？一定是最客观、最公认那个，所以定义中有 shared 这个词。

specification 是什么？词典解释为 a detailed description。detailed 好理解，description 是什么意思？表示对于一个人、事物、或事件的（口头或笔录）的描述。按照这个解释，原来的 ontology 定义就可以改写为：

ontology: a formal, explicit, detailed description of a shared conceptualization

explicit 是什么？词典解释为“described clearly and in detail, leaving no room for confusion or doubt”。就是描述得清楚、细致、不给留任何模糊的空间。看下面这张图：



假如此时我们想为图中的黑色圈出部分的概念下定义，最好就是描述符合成为这部分中元素的充分必要条件。假如我们用粉色勾出的部分来定义黑色部分，那么里面就混入了一些本来不属于黑色部分的元素，比如 A。但反过来，如果我们用蓝色勾出的部分来定义黑色部分，那么有一些实际是黑色部分的元素被排除在外了，比如说 B。所以无论把一个定义做的太 strong（蓝色部分），还是太 weak（粉色部分），都不是一个好定义。都会留出模糊的部分，留出漏洞。所以一个 explicit definition 应该是如下的解释：

explicit definition: a definition which formally sets out the meaning of a concept or expression, as by specifying necessary and sufficient conditions for being it

明确地描述成为某个概念中的元素需要满足什么条件。比如要成为“人”必须是哺乳动物，必须有头，必须有腿，可是满足这些条件的动物有很多，不仅仅是人。那么这个定义就不够 explicit，不够 precise。

formal 是什么？定义中只有这个词没有解释了，但其实这个词我们之前解释过了，formal 可以理解为严格按照字母表，语法规则和语义解释规则来组成句子。

这样一个关于本体的定义就完整地呈现在面前了。

在这样一个定义下，要求本体是一个“可计算模型”，一些基于集合论、模型论的逻辑语言就进入了视野。我们将使用描述逻辑语言描述本体知识。这里需要注意：本体只是一个知识表示模型，是一个领域知识库，它用什么语言描述知识并没有任何规定。不是只能用描述逻辑，用一阶谓词逻辑、模态逻辑、时序逻辑、命题逻辑取决于具体的任务类型，它合适什么样的语言，甚至用自然语言都可以，只要能描述知识，都可以叫做本体。这门课，我们选择用描述逻辑描述本体知识，以描述逻辑为载体介绍各种推理任务，展示各种应用，覆盖知识表示与推理这个领域研究的核心内容，一方面是因为描述逻辑是目前最热门、最前沿的知识表示语言，另一方面它就是一个用于教学的“工具人”，换了别的语言，跟随的推理类型也不会改变，只不过语言的表达力和不同推理任务的复杂度会发生改变。

描述逻辑（description logic，后面简写为 DL）是一阶谓词逻辑（first-order predicate logic，后面简写为 FOPL）的子集，并且是它的可判定子集（decidable fragments），因为 FOPL 是不可判定的（undecidable）。现阶段通常使用网络本体语言（Web Ontology Language，以下简称 OWL）开发本体，那么 OWL 与 DL 之间是什么关系呢？OWL 是面向开发端的，因为本体开发出来，需要有一个可用的、可部署的文件格式，比如.txt、.pdf、.xml 等。本体的

文件格式为.owl，大家可以用浏览器打开下面这个本体文件:

https://protege.stanford.edu/ontologies/pizza/pizza.owl

里面包含很多如下片段:

<owl:ObjectProperty rdf:about="http://www.co-ode.org/ontologies/pizza/pizza.owl#hasBase">
<rdfs:subPropertyOf rdf:resource="http://www.co-ode.org/ontologies/pizza/pizza.owl#hasIngredient"/>
<owl:inverseOf rdf:resource="http://www.co-ode.org/ontologies/pizza/pizza.owl#isBaseOf"/>
<rdf:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
<rdf:type rdf:resource="http://www.w3.org/2002/07/owl#InverseFunctionalProperty"/>
<rdfs:domain rdf:resource="http://www.co-ode.org/ontologies/pizza/pizza.owl#Pizza"/>
<rdfs:range rdf:resource="http://www.co-ode.org/ontologies/pizza/pizza.owl#PizzaBase"/>
</owl:ObjectProperty>

可以看到，里面有三个关键字：owl、rdfs、rdf。它们之间的关系：rdfs 是 rdf 的扩展，owl 是 rdfs 的扩展。这些具体是什么，可以先不关注。现阶段，需要记住.owl 文件可以被浏览器处理，内容可以被浏览器展示。DL 是一种逻辑语言，我们学习它是要学习它的字符表、语法、语义、表达力、以及在上面的推理任务。所以：DL 是 OWL 的逻辑内核，OWL 是 DL 的外部封装。他们使用不同的术语来表达同样的事物：

DL	OWL	First Order Logic
individual	individual	constant
concept	class	unary predicate
role	object property data property	binary predicate

object property 的 range 是 individual 或者 class，比如{Tom} ⊑ ∃hasChild.Lawyer；data property 的 range 是 data types（a number or string），比如{Tom} ⊑ ∃hasAge.55。这里的 55 不是一个符号，而是阿拉伯数字，interpretation 不会对它进行解释。这门课，我们不涉及 data property，因为推理不会涉及到“自然语言”，尽管在工程上，它会起到知识表示作用。

接下来我们学习几种 DL 语言。无论是哪一种 DL，都会将知识分为两个层面。第一个层面是类的层面，包括类的性质以及类与类之间的相互关系。第二个层面是实体的层面，包括实体的属性和实体与实体之间的关系。

类的性质（也就是成为类中元素的必要条件），比如：
Father 的性质是∃hasGender.Male，是∃hasChild.Top
Medalist 的性质是∃hasWon.Medal
Car 的性质是∃hasComponent.Wheel

类与类之间的相互关系（上面类的性质，也可以看做是类与复杂类之间的关系），比如：
Student 是 Human 的子类
Human 是 Mammal 的子类

类层面的知识的集合叫做 Terminological Box，简称 TBox。TBox 里面都是类与类之间的包含关系，所以我们也把 TBox 叫做 concept hierarchy（relations between concepts）。只不过这里，包含关系不仅可以在 atomic concepts 之间，还可以在 complex concepts 之间。

讲完了类的性质和类之间的关系，接下来将目光放在实体上：

实体的属性（实体属于哪个类），比如：
Jack 是学生 Student(Jack)
Lily 是一个有孩子的人 ∃hasChild.X(Lily)

实体与实体之间的关系
Lily 是 Jack 的妈妈 isMotherOf(Lily, Jack)
Jack 养了一只小狗 dodo hasPet(Jack, dodo) Puppy(dodo)

实体的属性既可以是一个原子类（atomic concept），也可以是通过逻辑连接符组成的一个复杂类（complex concept）。实体与实体的关系一定是由一个二元关系连接的两个实体。实体层面的知识的集合叫做 Assertional TBox，简称 ABox。

想一下 TBox 和 ABox 的名字是不是很贴切。看一下 terminology 的词义：

Terms: words and compound words that in specific contexts are given specific meanings

上面定义中的 words 对应 DL 的 concept names，compound words 对应 complex concepts，也叫 compound concepts。

Assert：这个词有很多自然语言语义，其中一个意思是将某个 object 归类

这里需要统一部分术语。首先一个 TBox 和 ABox 共同组成一个 Knowledge Base，简称 KB。一个本体就应该等同于一个 KB。但是在很多文献中，会发现有两种定义：一种是将本体定义为 TBox，不包括 ABox；还有一种是将本体等同于 KB。我们的规矩是在口语、写作中使用任何一种方式都可以，只需要定义清楚即可。

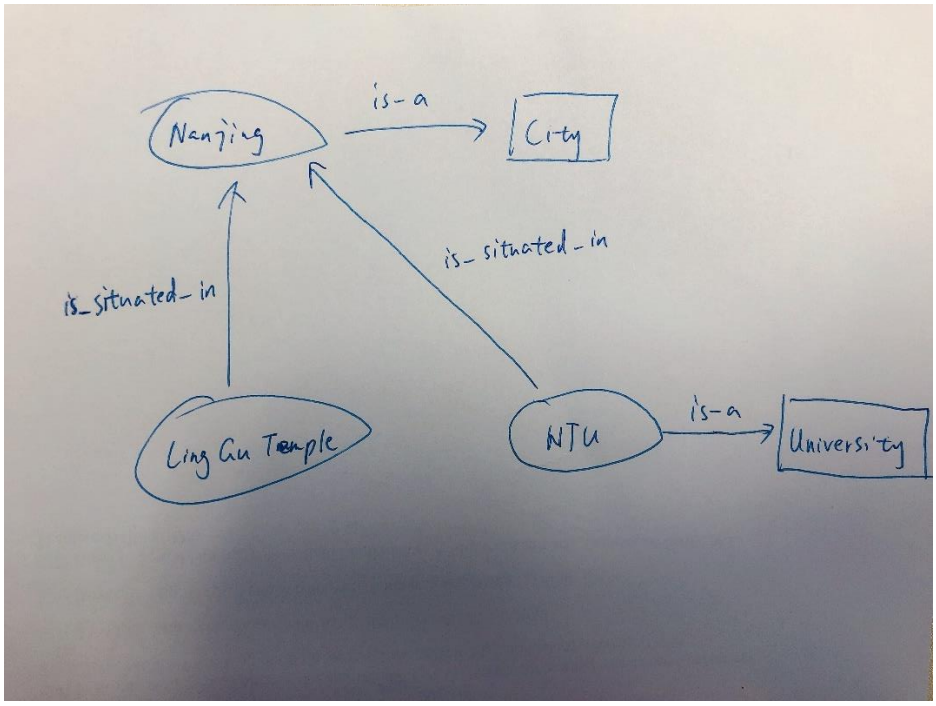
除了 TBox 和 ABox，本体有时还会包含一个 Role Box（简称 RBox）。Role Box 顾名思义，是关于 role 的性质。比如说，role 之间也可以有 inclusion 关系：hasFather 是 hasAncester 的子关系。意味着，a 如果是 b 的父亲，a 就一定是 b 的祖先。role 之间的包含关系称为 role inclusion，用字母 H 表示。EL 语言中如果加入 role inclusion，则称为 ELH；同样地，ALC 称为 ALCH。除此之外，RBox 还可以加入 role equivalence，表示两个 role 等价（同样地，a role equivalence 可以表示成两个 role inclusions）。一些 role 拥有其它的性质，比如 role 的传递性（transitivity），对称性（symmetry），自反性（reflexivity），函数性（functional）等等。这里，传递性是我们在这门课上介绍的一个代表性性质，其字母表示规则很特殊：ALC + transitivity = S，ALCHI + transitivity = SHI。一个 role r 是 transitive 的，其语义为：对于 domain 任意元素 x, y, z 来说，$\forall x \forall y \forall z((r(x,y) \wedge r(y,z)) \rightarrow r(x,z))$。比如，hasAncester 就是一个 transitive role；但 hasFriend 和 hasParent 不是。

如果有 RBox，则 KB = TBox + ABox + RBox
如果没有，则 KB = TBox + ABox

TBox 表达式称为 TBox axioms 或者 TBox inclusions（因为 TBox 里面都是 GCI），ABox 表达式称为 ABox assertions，而 RBox 表达式称为 RBox axioms。有的文献不会刻意区分 axioms 与 assertions，所以也会称 ABox assertions 为 ABox axioms。文献中也会看到 TBox statements，ABox statements 这样的表达，不要困惑。这门课把 KB 中所有表达式统称为 axioms。

一个 DL 的知识类型确定（TBox、ABox、RBox 确定），它的表达力取决于 concept 和 role 的表达力。更确切地说，取决于哪些逻辑运算符可以被使用来构建 complex concept 和 complex role。比如 EL 就只允许使用 top concept、and、exists 来构建 complex concept；而 ALC 进一步允许使用 bottom concept、negation、or、forall 来构建 complex concept；ALCI 进一步允许使用 inverse 来构建 complex role，等等。

为了说明 DL 作为当前知识表示语言的先进性，我们对比它与其它一些知识表示语言。详细介绍一个：Semantic Network。



上图为一个 semantic network。semantic network 是一个有向图（directed graph），由 nodes 和 edges 组成。其中，nodes 既可以是一个实体，比如上图中的 Nanjing、Ling Gu Temple、NJU；也可以是一个类，比如 City、University。Nodes 对于实体和类不做区分（上图中我们将实体的 node 用椭圆表示，类的 node 用矩形表示，是为了方便读者区分）。edges 用箭头标识，表示两个 nodes 之间的关系。每个 edge 上会有一个 label，代表关系的名称。

看下 semantic network 的缺点。首先，最大的缺点是没有 formal semantics。所有的 nodes 名称和 edges 名称都保持自然语言语义，没有像本体类似的基于集合论的语义。University 不会被解释为一个集合，NJU 不会被解释为集合中的一个元素，is-a 不会被解释为 domain 上的双元关系。这样一来，“计算”的基础就没有了。计算机是无法在“原始的自然语义”上进行计算的，这是它先天性的缺陷。其次，nodes 对实体和类是不做区分的，这种粗糙的处理方式对于知识表示来说不够理想，埋下了隐患。最后，像“非 University”“City 或 University”这种“复杂类”是无法表示的，说明表达力不够。其实，后两点缺点并不存在，它们只是第一个缺点衍生出来的产品。没有语义解释，意味着 semantic network 就仅仅是一

个数据模型，和传统的关系数据库没有任何区别，都是有固定的语法，没有可支撑计算的语义，作用就是存储数据用的物理框架。

上面讲了 DL 的词汇表和语法规则（怎样组成复杂的类），接下来讲 DL 的语义，也就是如何解释 DL 表达式。解释方法是基于集合论（set theory）和模型论（model theory）。两个理论都属于数理逻辑的范畴。具体做法是基于面向对象的世界观，用一个叫做 interpretation 的东西来解释 DL 表达式。Interpretation 包含两个部分：第一个是确定 domain，告诉这个故事发生在哪个世界/宇宙（world/universe）里面，以及里面会有哪些 elements（entities）；第二个是将 DL 表达式中的符号对应到这个 domain 上去：将 individual name 对应到 domain 中的一个 element；将 concept name 对应到 domain 上的一个子集，这个子集代表一个类，类中元素有共同的性质（所以才将它们分为一类）；将 role name 对应到 domain 上的一对实体之间的二元关系（可能不止一对，而是多对）。比如 hasFather，满足父子/女关系的实体对儿可能有多个。逻辑连接符的解释与 interpretation 没关系，它有逻辑上的预定义解释。

能够解释同一个或者同一组 DL 表达式（a DL axiom or a set of DL axioms）的 interpretation 不只有一个，这个合理吗？答案是合理的。比如 domain 是南京大学，里面的 element 是南京大学世界里的各种人，包括学生和老师。那么存在一个关于老师的属性是：Teacher $\sqsubseteq$ $\exists$teaches.Student。这个 axiom 只在南京大学这个 domain 才成立吗？不是，在其它大学的 domains 里依然成立。所以越是通用的知识满足它的 interpretations 越多，当然也有一些知识只在少数的 interpretations 解释下才成立，甚至在任何 interpretations 解释下都不成立。注：（无限集合之间也可以比较大小，比如整数集和正整数集）。

规定了语义的解释方法后，接下来可以根据这个方法建立起一种“可计算模型”。通过这个模型可以进一步定义几种推理运算，这些推理运算会在本体被用作各种 applications 的 KB 的时候发挥作用。比如 query answering：问某个 KB，is Lily a Teacher？换成逻辑表达式：

$$KB \models Teacher(Lily)$$

常见的推理运算包括：

（1）验证两个 concepts 之间是否存在 inclusion（subsumption）关系（equivalence 关系验证可以转化成两个 inclusions 关系验证，下面涉及 equivalence 的地方也是一样）

（2）验证一个 individual 是否属于某个 concept（membership）

（3）验证两个 individual 之间是否存在某二元关系（membership）

（4）计算哪些 concepts 之间存在 inclusion 关系

（5）计算每一个 individual 都属于哪些 concept

（6）计算哪些对儿 individuals 存在二元关系

还能想到其它的推理类型吗？

（1）（2）（3）和（4）（5）（6）的推理类型不一样，前者问的是“是否”（yes or no）的问题，称为 Boolean questions，后者 Non-Boolean questions。区别在于，Boolean questions 是事先假设一个情况，然后去验证这个情况是否成立，除了问题本身从头至尾没有任何“未知因素”。但是 Non-Boolean questions 是利用已有知识去计算未知的知识。直觉上讲，难度要比 Boolean questions 大。能不能将（4）（5）（6）问题转化为 Boolean questions？比如（4）问题，是否可以使用枚举方法，先找出逻辑表达式中所有的 concepts，再去验证每个组合是否满足 inclusion 关系？答案是不可以。因为 concepts 不仅是 concept name，还可以是

complex concepts，而 complex concepts 的数量可以是无限的：用 concept name A 和 role name r 可以造出∃r.A、∃r.∃r.A、∃r.∃r.∃r.A…等无数个 concepts。这样的方式是行不通的。Non-Boolean questions 不是都能转化为 Boolean questions 进行处理的。刚才列举了一些推理类型，是否已经将课上讲的推理类型完全覆盖？并不是，还有两种推理类型：

（7）验证一个 ontology 是否 consistent（部分文献也叫 satisfiable）

（8）验证一个 concept 是否 satisfiable

目前为止我们只关心推理类型的（1）（2）（3）（7）（8）这五个 Boolean questions，并且会在接下来展示：（1）（2）（3）（7）都可以转化为（8）问题。部分文献也会写（1）（2）（3）（8）都可以转化为（7）问题；也对，因为（7）（8）可以互相转化，看从哪个角度看待这个问题。

除了（7），其它问题又需要考虑两种情况：是否有一个背景知识？比如我说：食堂的饭菜是很难吃的。这句话正确吗？不一定正确，因为有的食堂的饭菜好吃有的不好吃。但是如果我说：XX 大学食堂的饭菜是很难吃的。一下子就变得很肯定了。这说明什么？说明增加了背景知识之后，原来不正确的可能就正确了。

我们按照（8）（7）（1）（2）（3）的顺序进行逐步讲解：

问题（8）：一个 concept C 是 satisfiable，则一定存在一个 interpretation I 使得 C^I 非空。一个 concept 是否必须是 satisfiable？从直觉角度，并不一定。物理世界里有没有一类群体里面是没有任何元素的？答案是有。比如“吃肉的素食主义者”、“没有生物学父母的哺乳动物”、“好吃不贵的南大食堂”等等。一个 ontology 里面，某些 concept 是 unsatisfiable 的，完全没有问题。看下面的例子：

$$\text{EukaryoticCell} \sqsubseteq \text{Cell}$$
$$\text{EukaryoticCell} \sqsubseteq \exists \text{hasPart.Nucleus}$$

$$\text{RedBloodCell} \sqsubseteq \text{EukaryoticCell}$$
$$\text{RedBloodCell} \sqsubseteq \forall \text{hasPart.}\neg\text{Nucleus}$$

$$\text{Blood} \sqsubseteq \exists \text{hasPart.RedBloodCell}$$

这个例子中：真核细胞一定有细胞核，血红细胞是真核细胞但是其成分中没有细胞核，很明显的 contradiction。所以血红细胞这个类里面不应该存在任何元素（任何 Interpretation I 都不可能让 RedBloodCell 非空）。如果 RedBloodCell 任何时候都为空，则 Blood 也如此（无论与常识是否符合，目前逻辑表达式展示的信息就是如此。事实上，血红细胞是真核细胞，在其成熟期的时候没有细胞核，但是成熟期之前有细胞核，它是特殊的真核细胞）。

那么我们现在如何判断一个 concept 是 satisfiable 的呢？分为两种情况：

（a）一种是没有背景知识（without TBox）

（b）一种是有背景知识（with TBox）

Concept satisfiability 的问题只和 TBox 有关，和 ABox 没关系。无论添加任何 ABox axioms 到背景知识里面，都不影响一个 concept 的 satisfiability。这句话非常重要，想一想为什么？

对于上述（a）的情况，一个 concept C 是 unsatisfiable 的，则 C is born to be unsatisfiable，它天然就是永远不可能非空的，非常类似于命题逻辑中的永假式。比如 Person ⊓ ¬Person 和 ∃hasChild.Male ⊓ ∀hasChild.¬Male。这时候想证明 C 是 satisfiable 的，最简单的方法是构建一个 interpretation I 使得 C^I 非空。但是对于 unsatisfiable 的情况，就无法这么做了，因为不可能找到世界上所有的 interpretations，并验证每一个是否都让 C 为空（这时候可以用反证法假设存在这样一个 interpretation I 使得 C 是 satisfiable 的，并试图寻找 contradiction）。于是就有了 Tableau 算法：该算法尝试构建一个 interpretation I 使得 C^I 非空，并且保证如果 C 是 satisfiable 的，就一定能够通过它构建这样一个 interpretation（这是算法的完备性保证的）。如果 C 是 unsatisfiable 的，Tableau 算法会在中间的某个步骤达到 contradiction（也叫 clash）。

contradiction：也就是⊥。在这里，我们把⊥外延解释为：一个元素属于 A 同时属于¬A，这里 A 指的是 atomic concept。

这个算法从何处开始？假设这个 concept C 是 satisfiable 的，则必定存在一个 domain element x 使得：$x \in C^I$。再根据这个假设，结合 C 的语义去构建 interpretation I 或者寻找 clash。

概念 C 有哪几种类型？换句话说，C 的根节点可以是哪几种？取决于 DL 是什么？如果是 ALC，则 C 可以是一个 atomic concept（concept name），还可以是 negation，and，or，exists，forall。是不是针对该语言中的每一个逻辑运算符，都设计一个 interpretation 的构造规则？

为了避免∃hasChild.Male ⊓ ¬∃hasChild.Male 这种客观上是 clash 的情况无法找到上面定义的 clash（因为∃hasChild.Male 并不是一个 atomic concept）我们引入 negation normal form（NNF）的概念。可能有人会问：为什么不在定义 clash 的时候直接将外延解释为：一个元素属于 C 同时属于¬C，其中 C 为任意 concept。这样 clash 的概念不更有普适性么？看这样一个例子：∃r.¬∀s.A ⊓ ¬∀r.∀s.A 确实是 clash，但是却找不到 C 和¬C 这种明显互斥的表示。这是因为原式之中¬符号出现的位置不够“对称”。NNF 要求所有的¬都只能出现在 atomic concept 之前。这样一来，¬后面只有 concept name，则是否存在 clash 就非常直观了。不满足这个语法的式子在使用 Tableau 算法之前，都需要使用如下规则转为 NNF：

$$
\begin{array}{rcll}
\neg\top & \equiv & \bot & \\
\neg\bot & \equiv & \top & \\
\neg\neg C & \equiv & C & \\
\neg(C \sqcap D) & \equiv & \neg C \sqcup \neg D & \text{(De Morgan's law)} \\
\neg(C \sqcup D) & \equiv & \neg C \sqcap \neg D & \text{(De Morgan's law)} \\
\neg\forall r.C & \equiv & \exists r.\neg C & \\
\neg\exists r.C & \equiv & \forall r.\neg C &
\end{array}
$$

图中的规则保证我们可以将 ALC 的任意 concept 转为 NNF，分别处理¬出现在 top，bottom，negation，and，or，exists，forall 这六个非 atomic concept 的情况。上述规则可能会递归使用。

每当出现一个算法，或者一组规则，“原则上”我们需要证明这个算法或者这组规则的正确性、完备性、可终止性。对于一些非常简单、直观的算法和规则，我们通常成为该算法或者规则为“standard”，意味着不需要证明也公认成立。上述这组规则就是这样一个例子。

转为 NNF 之后，我们就可以在 Tableau 算法中不再针对¬设计 interpretation I 的构造规则。另外，top 和 bottom 已经没有任何叶节点，所以就只剩下其它几种类型的根节点。如下图所示，到达哪个根节点，就使用对应的 rule 构造 interpretation I。并且 rule 的可使用性（applicability）除了根据根节点判断以外，还要检测是否满足 if 条件，满足才能使用。

$$S \rightarrow_{\sqcap} S \cup \{ x : C,\ x : D \}$$

if (a) $x : C \sqcap D$ is in S
(b) $x : C$ and $x : D$ are not both in S

$$S \rightarrow_{\sqcup} S \cup \{ x : E \}$$

if (a) $x : C \sqcup D$ is in S
(b) neither $x : C$ nor $x : D$ is in S
(c) $E = C$ or $E = D$ **(branching!)**

$$S \rightarrow_{\forall} S \cup \{ y : C \}$$

if (a) $x : \forall r.C$ is in S
(b) $(x, y) : r$ is in S
(c) $y : C$ is not in S

olicable if role successors can be found

$$S \rightarrow_{\exists} S \cup \{ (x, y) : r,\ y : C \}$$

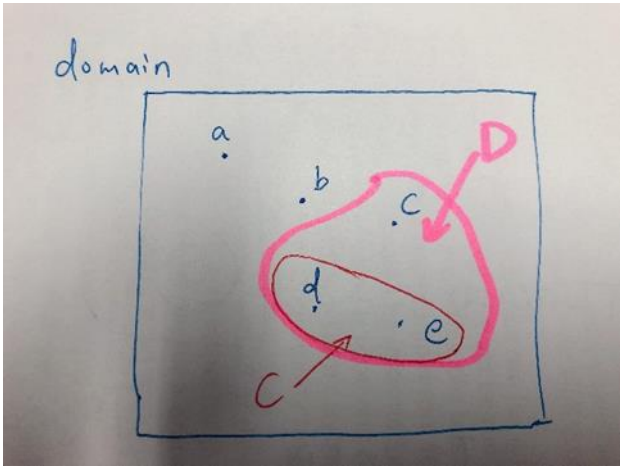
if (a) $x : \exists r.C$ is in S
(b) y is a fresh individual
(c) there is no z such that
both $(x, z) : r$ and $z : C$ are in S

Tableau 算法的本质是 decompose 或者叫 break down 整个 concept 的语义。一个不能被满足

的 concept 一定会在 decompose 的过程中遇到语义上的矛盾（这是 Tableau 算法的完备性保证的，需要证明）。按照 Tableau 算法一直 decompose 到叶节点，也就是 atomic concept 和 negated atomic concept 层面，算法才会终止。如果最后出现了某个元素同时在 A 和¬A 的情况，则到达 clash。这说明原始的 concept C 不可能存在一个 interpretation I 使得 $x \in C^I$。也就是说 C 是 unsatisfiable 的。反过来，如果直到已经无 rule 可用了，依然没找到 clash，则 C 是 satisfiable 的。这里面唯一特殊的规则是 or 这个 rule。它的每个分支都要探索到位才可以保证结果的正确性：如果在某一个 branch 上得到了已经无 rule 可用，且并没有找到 clash 的效果，则可以直接宣判 C 是 satisfiable；但是，如果一个 branch 中得到了 clash，不能着急下结论说 C 是 unsatisfiable，因为还需要探索其它的 branches 的情况。当全部 branches 都检查完了且都找到了 clash，C 才是 unsatisfiable。

对于上述（b）情况，也就是有 TBox 的情况，我们不能在全部可能的 interpretations 里面寻找让 C 不为空的那个，来证明 C 的 satisfiability。相反，我们需要在 TBox 的 models 里面寻找让 C 是 satisfiable 的。也就是说：判断一个 concept C 是否 satisfiable w.r.t. a TBox T，我们需要去看 $T \cup \{x: C\}$是否有一个 model。满足 x: C 上面已经看过了怎么做。那么怎么体现是 T 的 model?

对于 T 中每一个 axiom（GCI）$C \sqsubseteq D$，如果某个 interpretation I 是 $C \sqsubseteq D$ 的 model：$I \vDash C \sqsubseteq D$，则 $I \vDash \top \sqsubseteq \neg C \sqcup D$。意思是：C 是 D 的子集这件事如果在 I 上为真，则¬C 与 D 的并集一定是整个 domain 这件事在 I 也一定为真。看下图：



因为任何元素都一定在 domain 里，则 $x: (\neg C \sqcup D)^I$。如果 TBox 之中包含多个 GCI，则 x 都会出现在每个用同样方式表示的 domain 之中：

Check whether C is satisfiable w.r.t. a TBox = $\{A \sqsubseteq B, D \sqsubseteq E\}$。则存在一个 Interpretation I 使得 domain 中存在一个元素 x:
$x \in C^I$,
$x \in (\neg A \sqcup B)^I$,
$x \in (\neg D \sqcup E)^I$
这时候，我们需要在 Tableau 算法中添加一个新的规则：

$$S \rightarrow_U S \cup \{ x : D \}$$

if (a) $\top \sqsubseteq D$ is in S
(b) x occurs in S
(c) $x : D$ is not in S

当发现 C⊑D 出现在 T 中的时候，等价地转化为⊤⊑¬C⊔D。如果发现 x: ¬C⊔D 没有在现有的状态系统 S 里，则将此事实添加进去。所以，多了背景知识（一个 TBox，也就是一组 GCIs）其实是帮助我们添加一些 interpretation 构建过程中的限制条件。比如 C⊑D 意味着所有出现在 S 状态系统里面的 individual 都要出现在¬C⊔D 之中，这就是一个限制。

至此，我们已经学会如何解决 concept satisfiability 的问题，无论有无 TBox。接下来看如何将其它推理问题转化为 concept satisfiability 的问题。首先看问题（7）。

问题（7）：check 一个 ontology 是否是 consistent 的

对于 ontology O 的 consistency 定义为：存在一个 Interpretation I 使得 I 可以成为 O 的 model。也就是存在一个 interpretation I 使得 I 是 O 中每个 axiom 的 model。至于 I 满足一个 axiom 是从集合论和模型论角度定义：I 满足 C⊑D，则存在一个 interpretation I 满足 $C^I \subseteq D^I$。比如，如果 I 对于 C 的解释为{1, 2}，对于 D 的解释为{1, 2, 3}；如果 I 对一个 individual name a 的解释为 1，则 I 满足 C(a)和 D(a)这两个 axioms。

为什么要 check ontology consistency?

如果 O 是 inconsistent 的，意味着没有任何 interpretation 可以使得 O“自洽”。一个不自洽的本体是无法从中得到任何正确信息的。看一个例子，O ⊨ alpha（alpha 是任意一个 axiom）。如果此时 O 是不自洽的，意味着没有任何 interpretation 可以满足 O，成为 O 的 model。而 O ⊨ alpha 成立的要求是所有 O 的 model 都是 alpha 的 model，写作 for any interpretation I, if I is a model of O, then it is a model of alpha。因为 O 没有 model，所有这个式子是 vacuously true（理解为虽然为 true，但 true 得毫无意义）。我们看这样一个自然语言的例子，⊨符号左边是：萨摩耶拥有律师证；⊨符号右边是：它是律师。让左边这句话成立的 model 一定是右边这句话的 model，这没问题。但是萨摩耶是不可能拥有律师证的，这让后续的任务推理结论都毫无意义。如果这个例子还是太抽象，举一个更现实的。有一天一个人对你承诺，如果他成为了亿万富豪，一定会对你 xxx，然而…

一个 O 是 inconsistent 的例子：A(a)和¬A(a)同时出现在 O 中。另一个例子，RedBloodCell 的那个例子，我们已经得到结论 RedBloodCell 和 Blood 必然是 unsatisfiable 的。这种情况下，如果向 O 中添加一个 ABox axiom：RedBloodCell(a)或者 Blood(b)。则 RedBloodCell 和 Blood 中必然有元素，与上面事实相悖，则整个 ontology 是 inconsistent 的。这里注意，当 O 是空集时，它是自洽的。空集意味着任何 interpretation 都是它的 model。

了解了 check ontology consistency 的重要性，如何验证一个 ontology O 是否是 consistent 的呢？思路很简单，就是想办法为 O 构造一个 model，找到了它，则 O 就是 consistent 的。问

题（7）可以以如下方式转化为问题（8）：

例子：Check O = {C ⊑ D}是否是 consistent。如果是，则存在一个 interpretation I 满足 $C^I \subseteq D^I$，则$(\neg C \sqcup D)^I$一定是 I 的 domain，则 I 定义的 domain 中一定存在一个 element x 使得 domain 非空：$x \in (\neg C \sqcup D)^I$。所以 check：O = {C⊑D}是否是 consistent 的问题就成功转化为 check ¬C⊔D 是否是 satisfiable 的问题。使用 Tableau 算法将原始的表达式转化为⊤⊑¬C⊔D，并使用 U 规则进行 S 状态系统的升级。
一个更复杂的例子（O 中包含多个 axioms）：Check O = {C⊑D, D⊑E}是否是 consistent。如果是，则存在一个 interpretation I⊨C⊑D，D⊑E，则¬C⊔D 和¬D⊔E 都一定是 I 的 domain，则 domain 中一定存在一个元素 x 使得 $x \in (\neg C \sqcup D)^I$ 且 $x \in (\neg D \sqcup E)^I$。所以 check O = {C⊑D, D⊑E}是否是 consistent 的问题就成功转化为 check ¬C⊔D 和¬D⊔E 是否是 satisfiable 的问题。使用 Tableau 算法将原始的表达式转化为⊤⊑¬C⊔D 和⊤⊑¬D⊔E，并使用 U 规则进行 S 状态系统的升级。

如果 O 中包含 ABox axioms 呢：Check O = {C⊑D, D(a)}是否是 consistent？此时不需要再创造一个虚拟的元素 x 来进行判断，因为已经有了一个现成的元素：a。此时只需要将 a: D 添加到 S 状态系统，并使用 U 规则将 a: ¬C⊔D 添加到 S 状态系统里面即可继续 Tableau 算法接下来的操作。直到全部叶节点为 clash，则 O 是 inconsistent；或者某一个叶节点在饱和状态下（无 rule 可用）依然没出现 clash，则 O 是 consistent。

接下来看如何将问题（1）转化为（8）：验证两个 concepts 之间是否存在 inclusion 关系（也称为 subsumption 任务）。分为两种情况，一种是没有背景知识（without TBox），另一种是有背景知识（with TBox）。

没有 TBox 的情况下，如果两个 concepts C 和 D 之间存在 inclusion 关系，说明 C⊑D 这件事永真。说明任意一个 interpretation I，都可以满足 C⊑D。显然，我们不可能找出所有的 interpretation 来验证是否都成立。一个可行的方法，或者叫可计算的方法，假设存在一个 interpretation I 使得 C⊑D 不成立：I \not\models C⊑D。说明⊤⊑¬C⊔D 这件事在 I 中不成立，说明¬C⊔D 不再是 I 的 domain，说明 domain 中一定存在一个元素 x，x 在¬C⊔D 的补集之中，也就是在¬(¬C ⊔ D)之中，也就是在 C⊓¬D 之中。所以问题变成了 C⊓¬D 的可满足性问题。将 x: C⊓¬D 加入到 S 状态系统，使用 Tableau 算法进行验证。如果 C⊓¬D 可满足，说明确实存在这样一个 interpretation I 使得 C⊑D 不成立，则说明 C 和 D 之间不存在 inclusion 关系；反过来，如果 C⊓¬D 不可满足，说明不存在这样一个 interpretation I 使得 C⊑D 不成立，说明 C 和 D 之间存在 inclusion 关系。

有 TBox 的情况下，如果两个 concepts C 和 D 之间存在 inclusion 关系，说明 C⊑D 这件事在所有满足 TBox 的 models 中永真。问题就变为了验证 a TBox T：T⊨ C⊑D 是否成立。这个式子的语义是：所有 T 的 model 都是 C⊑D 的 model。依然是通过反证法来证明。假设存在一个 T 的 model I，它不是 C⊑D 的 model。则⊤⊑¬C⊔D 这件事在 I 中不成立，说明¬C⊔D 不再是 I 的 domain，说明 domain 中一定存在一个元素 x，x 在¬C ⊔ D 的补集之中，也就是在¬(¬C⊔D)之中，也就是在 C⊓¬D 之中。所以问题变成了 C⊓¬D 针对 TBox T 的可满足性问题。如果 C⊓¬D 针对 T 是可满足的，说明原始的 T⊨ C⊑D 不成立；如果 C⊓¬D 针对 T 是不可满足的，说明原始的 T⊨ C⊑D 成立。

比如让 T = {C ⊑ A, A ⊑ D}，check T ⊨ C ⊑ D。我们首先假设有一个 T 的 model I 不能让 C ⊑ D 成立，则 x: C ⊓¬D 就可以加入到 S 状态系统里面了。接下来，将原来 T 中的两个 GCIs 转化为⊤ ⊑ ¬C ⊔ A 和⊤ ⊑ ¬A ⊔ D。并使用 U 规则将 x: ¬C ⊔ A 和 x: ¬A ⊔ D 添加入 S 状态系统，继续执行 Tableau 算法的其它步骤。

这里注意，concept inclusion 关系的验证与 ABox 存在与否依然没有任何关系。
接下来看如何将问题（2）转化为（8）：
问题（2）：验证一个 individual 是否属于某个 concept（可以称为 membership 任务，因为是检测某个 individual，或者某两个 individuals 是否是某个 concept 或者某个 role 的 member）。依然分为两种情况，一种是没有背景知识（without ontology），另一种是有背景知识（with ontology）。这里注意，背景知识不再局限于 TBox，而是整个 ontology（TBox + ABox）。

没有 ontology 的情况下，如果一个 individual a 属于一个 concept C，说明 C(a)这件事永真。说明任意一个 interpretation I，都可以满足 C(a)。只有一种可能，对于任意 interpretation I，C^I都是 I 的 domain。也就是说对于任意 interpretation I, $(\neg C)^I$都是空集，则¬C 是 unsatisfiable 的。问题就从验证 C 是 valid 的问题（C 是 tautology）变成验证¬C 的 satisfiability 问题（要验证 C(a)是否永真，理论上不可能找出所有 interpretation，每个都验证一遍。但可以假设存在一个 interpretation I 使得 C(a)不成立，则使得¬C(a)成立）。因为原始的式子中已经出现了一个 individual a，我们直接用它来当做 domain 的 element 即可。将 a: ¬C 加入到 S 状态系统中，并继续执行 Tableau 算法其它步骤。如果结果是 clash，则说明这样的 interpretation I 不存在，说明 C(a)永真；反过来，如果是 no rule is applicable，则说明 C(a)不永真。

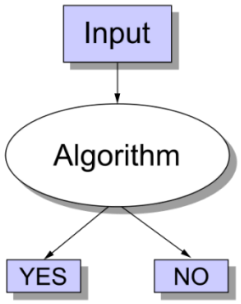
在有 ontology 的情况下，如果一个 individual a 属于一个 concept C，说明 C(a)这件事在所有 ontology 的 models 上永真。问题就变为了验证 an ontology O：O ⊨ C(a)是否成立，也就是要验证所有的 O 的 model 都是 C(a)的 model。反证法假设存在一个 O 的 model I 使得 C(a)不成立，则 I 要满足¬C(a)。所以需要将 a: ¬C 加入到当前的 S 状态系统，同时，将 O 中所有的 axioms 转化为一定形式后加入 S。比如 O 中存在 GCI C ⊑ D，则按照 U 规则添加 a: ¬C ⊔ D 进入 S；O 中存在 ABox axiom D(b)，则添加 b: D 进入 S；O 中存在 r(a,b)，则添加(a,b): r 进入 S。所以问题变成了¬C(a)针对 Ontology O 的可满足性问题。如果¬C(a)针对 Ontology O 是可满足的，说明原始的 O ⊨ C(a)不成立；¬C(a)针对 Ontology O 是不可满足的，说明原始的 O ⊨ C(a)成立。

如何将问题（3）转化为（8）是不是可以不用具体讲了？

Tableau 的基本思想是构建一个满足各方面要求的 model，并且在部分推理任务上，比如 subsumption 和 membership 两个验证任务，又体现了假设再反驳的思想（refutational）。Tableau 算法过程讲解到此为止（其计算性质和性质的证明将在后续讲解）。

至止，是不是可以看到 concept satisfiability 问题有多么重要？我们把这类问题叫做 decision problem（Boolean problem），翻译为“决定性问题”。

decision problem: a yes-or-no question on specified sets of inputs



简单地说，decision problem 就是给定一组输入，输出一定是 yes 或者 no 的问题。比如，给定一个自然数，判断其是不是偶数？比如，给定两个自然数 x 和 y，判断 x 是否可以被 y 整除？用上图表示（维基百科）：

Tableau 就是这样一个算法。当一个 concept 是 satisfiable 的时候，算法会返回 yes，否则返回 no。是不是对于所有的 decision problem 都存在一个算法，使得当“客观事实”为 yes 的时候，该算法都能返回 yes 的结果，当“客观事实”为 no 的时候，该算法都能返回 no 的结果？答案当然是否定的。一个最典型的例子就是一阶谓词逻辑的 validity checking 问题（可等价转化为 satisfiability 问题）。客观上不存在一个 algorithm 使得当一组一阶谓词逻辑表达式为 invalid 的时候，一定返回一个 no 的答案。如果从 satisfiability 的角度看待，那就是，客观上不存在一个 algorithm 使得当一组一阶谓词逻辑表达式为 satisfiable 的时候，一定返回一个 yes 的答案（随着课程进行后续展示细节）。

对于一个 decision problem，如果存在一个 algorithm 使得在客观事实是 yes 的时候返回 yes，或者在客观事实是 no 的时候返回 no，则说明该问题是可决定的（decidable）。

接下来再想一个问题，当我们确定一个 decision problem 是 decidable 的，意味着世界上存在一个 algorithm 可以解决这个 decision problem。但是不是随便开发一个关于这个问题的算法都能确保解决这个问题？答案当然不是的。一个 decision problem 是 decidable 的是这个问题本身的性质，相当于“客观条件很好”。但是能不能开发出解决这个问题的算法是主观问题。比如计算 1+1 这个问题当然是可解决的，但是你开发了一个算法永不终止，或者返回了 3，也是有可能的，所以算法也要有一些性质来保证它的“质量”。

推理算法需要满足的三个重要性质：终止性（termination）、正确性（soundness）、完备性（completeness）。

终止性：给定任何输入，该算法都会在有限步骤内终止（不存在算法无限运行下去）

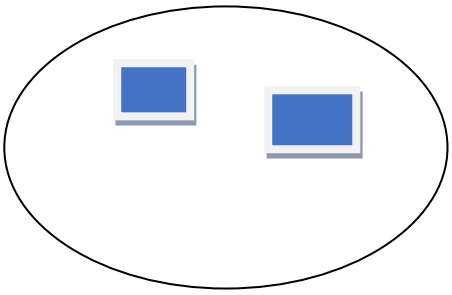
正确性：对于一个 concept C 进行 satisfiability checking，如果 Tableau 返回 yes，则客观事实上 C 一定是 consistent（不存在算法返回了 yes，但是客观事实 C 是 unsatisfiable 的；或者返回了 no，客观事实是 satisfiable 的）

完备性：对于客观事实是 satisfiable 的 concept C，Tableau 都能返回 yes（不存在一种情况，只针对某些 satisfiable 的 concept，算法才返回 yes；而对于另一些，算法计算不了）；对于客观事实是 unsatisfiable 的 concept C，Tableau 都能返回 no

当一个 decision problem 的算法满足上述三个性质的时候，我们将这个算法称为 decision procedure。这是知识表示与推理领域非常重要的一个概念。KR 学者的一个重要目标就是为各种逻辑语言寻找 decision procedure（前提是该语言本身是 decidable 的）。

Description Logic（DL）是一个语言家族，根据所允许使用的逻辑连接符和 basic building blocks 的不同，其提供的表达力，相应的计算性质也不同。我们学习了家族中的 EL, ALC, SHOIQ，其中重点学习了 EL 和 ALC。学习 EL 是因为 EL 有着特别好的计算性质，以及在现实中非常广泛的应用。学习 ALC 是因为它是目前公认的 DL 语言的核心，从 ALC 开始（包括 ALC）扩展到 SHOIQ 和 SROIQ，这中间的所有语言都被认为是 expressive DL（表达力很强的 DL），计算性质都相对较差。大量的学者在研究 DL 家族不同成员的计算性质，其中的一些结果可以在：http://www.cs.man.ac.uk/~ezolin/dl/查询到。设计和实现表达力强、计算性质好的 KR 语言（不一定是现有的 DL，可能是新的扩展，可能完全跳出 DL 框架，甚至跳出一阶谓词逻辑框架的语言）是 KR 学者孜孜不倦追求的目标。

Tableau 算法可以解决 ALC 上的标准推理问题，被认为是迄今为止 ALC 及其扩展上最有效率的推理算法。然而，对于表达力不如 ALC 的语言比如 EL，Tableau 算法就不一定是最有效率的。我们在课上讲解了 EL 的推理算法，它负责完成 subsumption 验证问题，但并不负责 concept satisfiability 和 ontology consistency 的验证问题。首先是算法本身不具备这个功能，其次是没有这个必要，因为 EL 语言下的 concept 都是 satisfiable 的，ontology 都是 consistent 的（作业题目的证明）。想一想是为什么？先从直觉上想一下再尝试进行证明：



将上面图片中的外圈想象成一个 domain。此时 EL 语法允许构建 concept name，每构建一个 concept name，都是圆圈的一个子集（比如矩形部分）。而 ALC 语法构建 concept name A 的时候，就自动被动生成了另一个 concept ¬A；EL 不会，因为不允许使用¬构建 concept。那么想让某个 concept 在 EL 表达式中为 unsatisfiable（永远为空集）的可能性还有吗？什么时候一个 concept 永空？只有 bottom concept ⊥，也就是 C⊓¬C。

==========

接下来课程进入到关于 model theory 的讲解。为什么要在一门《知识表示与处理》的课程上讲数理逻辑中“模型论”的一些内容？首先，看一下什么是模型论（model theory）？

model theory: a branch of mathematical logic which deals with the relation between a formal language and its interpretations, or models.

模型论研究的是一个 formal language 中的解释（interpretation）和模型（model）的性质。解释与模型的关系是：解释是一个表达式的“潜在模型”，有可能使这个表达式为真，也有可

能使这个表达式为假，使它为真的就是这个表达式的模型。描述逻辑是一个 formal language，它的解释的方式我们上面介绍过了；成为它的某个表达式（a formula），或者某组表达式（a set of formulas，称为本体）的模型的判断条件上面也介绍过了。二者都是基于集合论来定义的。那就要了解一下，集合论上的模型定义方式有何性质。

第一个需要了解的就是 DL 的每一个 interpretation 都可以表示为图（graph interpretation），其中节点（node）代表 domain 中的一个 element，边（edge）代表两个节点上的二元关系。如果一个 element 是某个概念 C 的解释（domain 的一个子集）里的一个元素，则可以把这个 node label 为 C。

接下来要介绍的概念，是模型论这个部分最重要的概念，叫 bisimulation。Bisimulation 是一种 relation，存在于两个 interpretations 之间。Bisimulation 不只是在模型论里面才出现，它是非常重要的概念，在很多领域都有应用。但本质上，它的作用是 capture “state equivalences and process equivalences” (behavioral equivalence)，翻译过来就是抓取“状态上的等价关系和过程中的等价关系”（也就是行为上的等价关系）。还记不记得上节课讲的 automata 的知识？一个 automata 里面有一个状态集合，一个字符表集合，对于一个确定性有限状态自动机（DFA），当机器移动到某个状态（比如 s1），在 s1 上接收到某个字符（比如 0），会移动到下一个状态（比如 s2）。这时候可以将一个 graph interpretation 对应成一个 automata：一个 node 对应状态集合里的一个 state，一个 edge 对应字符集合里的一个 symbol。区别在于 graph interpretation 对应的是一个非确定性有限自动机（NFA），因为原则上从一个 node 通过 r 关系可以达到多个 nodes。

Bisimulation 在 DL 上的定义是：给定两个 DL interpretations I1 and I2，如果 I1 中的一个元素（比如 d1）和 I2 中的一个元素（比如 d2）满足如下关系，则说 I1 中的 d1 与 I2 中的 d2 存在 bisimilar 关系，写作(I1,d1) ~ (I2,d2)：

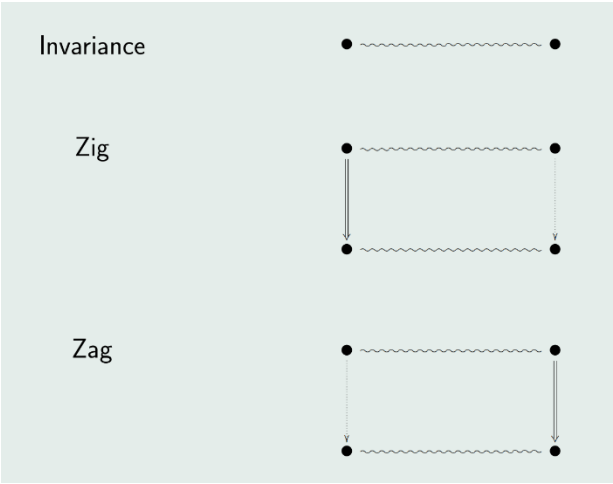
（1）如果 d1 是某个 concept name A 在 I1 解释下的集合里面的元素，则 d2 也是 A 在 I2 解释下的集合里面的元素，反之亦然；

（2）如果 I1 中存在一个 node f1 与 d1 存在 r 关系：$(d1,f1) \in r^{I1}$，则 I2 中必须也存在一个 node f2 与 d2 存在 r 关系：$(d2,f2) \in r^{I2}$，并且(I1,f1) ~ (I2,f2)；

（3）如果 I2 中存在一个 node f2 与 d2 存在 r 关系：$(d2,f2) \in r^{I2}$，则 I1 中必须也存在一个 node f1 与 d1 存在 r 关系：$(d1,f1) \in r^{I1}$，并且(I2,f2) ~ (I1,f1)。

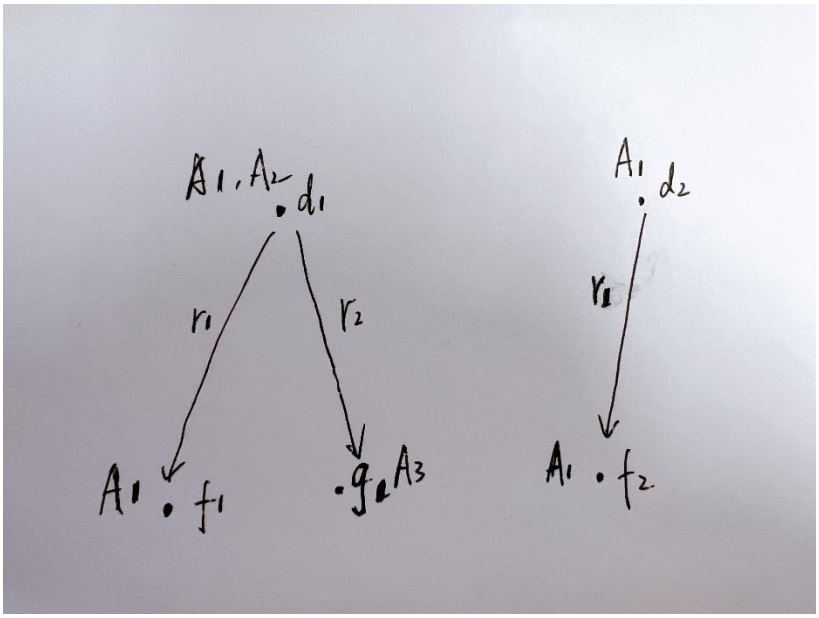
我们把（1）称为 invariance（后面详细讲），表示两个 nodes（states）所属的概念一样。我们如果用 $L_{I1}(d1)$表示在 I1 环境下，d1 所属于的概念的集合，则这里 $L_{I1}(d1) = L_{I2}(d2)$；

我们把（2）称为 forth，或者 zig，把（3）称为 back，或者 zag。



back and forth：表示来来回回，比如从篮球场的一端跑到另一端再返回就是这个意思
zigzag：表示沿着 Z 字形弯弯曲曲前进
两者都有对应的小游戏，类似于 4399 小游戏，有兴趣可以 google 一下了解下游戏中体现这些单词的语义。如果 I1 中存在一个 node d1，I2 中存在一个 node d2，两者之间有 bisimilar 关系，则说 I1 与 I2 之间存在 bisimulation 关系。这暗示着，I1 和 I2 中的元素中可以存在多对儿 bisimilar 关系的元素。如果对于 I1 中每一个元素 d1，I2 中都存在一个 d2，使得(I1,d1) ~ (I2,d2)，反过来对于 I2 中每一个元素 d2，I1 中都存在一个 d1，使得(I2,d2) ~ (I1,d1)，则称 I1 与 I2 间的 bisimulation 关系为“global bisimulation”。参考书里面定义的 bisimulation，以及上述描述的都是定义在特定的 d1 和 d2 上，因此不是 global bisimulation。我们把这样的 interpretation (I1,d1)叫做 pointed interpretation，翻译为“指定（元素 d1）的解释”。另外一点需要说明，bisimulation 并不一定定义在两个不同的 interpretations 上，定义在同一个 interpretation 上的两个元素也可以，那就是(I1,d1) ~ (I2,d2)。更有甚者，定义在同一个元素上都可以，但是没意义，一个元素与其自身

在我们的参考书里面，涉及到一个概念叫：parameterized，表示参数化。它的意思是有的时候 d1 和 d2 并不是在 I1 和 I2 上的所有的 concept name 和 role name 上都使得上述 bisimilar 的条件（1）（2）（3）成立，而只是其中一部分。比如下图 I1 和 I2 中解释了 A1，A2，A3 和 r1，r2，但 d1 与 d2 之间并无 bisimilar 关系，但在 A1 和 r1 上有 bisimilar 关系：



d1 属于 A1 和 A2，而 d2 只属于 A1；有一个 g1 与 d1 存在 r2 关系，但是在 I2 中并没有一个 node 与 d2 存在 r2 关系，所以不管怎么看 d1 和 d2 都不符合 bisimilar 的关系。但如果将参数限制在 A1 和 r1，则 d1 和 d2 有 bisimilar 关系，称为在 A1，r1 上的 bisimilar 关系。

接下来一个词叫 invariance，我们在 DL 环境下定义它：

Invariance：a concept C is called invariant for bisimulation if the following holds: if (I1,d1) ~ (I2,d2), then $d1 \in C^{I1}$ iff $d2 \in C^{I2}$

该如何理解这个定义？我们把 d1 和 d2 看作是真假美猴王，我们的目的是分辨出他俩的身份，看看哪个是孙悟空，哪个是六耳猕猴。于是我们找到第一个证人：唐僧。让他念紧箍咒，看看哪个喊疼。结果两个都喊疼，说明通过唐僧是无法分辨他俩的。这个唐僧就是我们这里的 C；后来又去了天宫，让托塔天王拿出照妖镜，也看不出区别，那么这个托塔天王就是另一个 C；后来又去了观音那，还是看不出，观音也是一个 C；直到到了谛听和如来那，才分辨出来。所以这个 C 起到的作用就是分辨两个解释中两个元素的区别。
由此，我们有了一个非常重要的 theorem（Theorem 3.2）：

If (I1,d1) ~ (I2,d2), then for all ALC concept C: $d1 \in C^{I1}$ iff $d2 \in C^{I2}$

这个 theorem 告诉我们什么？就是所有的 ALC concept 都不能区分两个 interpretations 上的一对儿 bisimilar 元素，意味着所有的 ALC concept 都不能作谛听或如来，只能作观音、唐僧、或者其他不能分辨真假美猴王的角色。

Theorem 3.2 给出了一个归纳证明，我们仔细看一下这个证明的思想：首先数学有三种基本证明方法：构建法、反证法、归纳法。不同的问题适用于不同的方法。我们在 KRistal 自己的组会上分享过面对不同问题时怎么决定使用最合适的方法，这里时间有限，不多赘述，只写一些简单的判断思路。

构造法：适用于“证明 XXX 有什么东西”，“存在一个 XXX 满足什么性质”，因为是只要找

到一个实例即可
归纳法：适用于要证明的对象是无限的，并且要证明的对象之间有联系，也就是说互相之间满足某种“偏序关系”
反证法：适用于要证明的对象是无限的，但对象之间并无偏序关系，或者并无任何联系

这里我们采用归纳法证明。因为我们无法枚举出所有的 ALC concept，一一验证它们是否都无法区分 bisimulation。但是我们能观察到 ALC concept 的构造是遵循语法规则的。

明确一点：ALC = EL + negation

表面上 ALC 比 EL 多了 negation、disjunction、forall、bottom concept 四个构造元素，实际上只多了 negation，因为 disjunction = negation + conjunction，forall = negation + exists，bottom concept = negation + top concept，所以我们只需要考虑 negation，conjunction，exists，top。

归纳法的第一步是找到偏序集里面的第 1 个元素，并证明当 case 是第 1 个元素时，结论成立；第二步！第二步！第二步！很多同学会搞错，他们会在第 1 个元素成立的基础上，接着证明第 2 个元素成立，然后第 3 个元素…第 n 个。这是不对的！这不又变成了无穷无尽的证明？本来我们使用归纳法证明的目的就是为了避免无限枚举。我们在第二步要证明的是，当偏序集里任何第 i 个元素成立，则第 i+1 个元素一定成立。这样一来，我们已经证明了第 1 个元素成立（i=1），那么根据上述结论，第 i+1 个元素（第 2 个元素）也成立，如此就可以使得整个偏序集上的元素成立。我们把偏序集的第 1 个元素叫做 base case，也叫 basic base，把偏序集里任何第 i 个元素成立，第 i+1 个元素一定成立这个假设叫做 induction hypothesis，也叫 inductive hypothesis。

这里，我们的 base case 是 C = A，其中 A 是一个 concept name（因为这是 C 最简单的情况），我们要证明 A 无法区分两个 bisimilar 的 nodes d1 和 d2。我们要证明的是：

$$d1 \in A^{I1} \text{ iff } d2 \in A^{I2}$$

那么事实是如此么？是的。因为这是(I1,d1) ~ (I2,d2)成立的第一个条件（1）明确定义的。

接下来我们要证明 induction hypothesis，首先 C 是一个 conjunction 的情况：C = D ⊓ E。这里我们需要首先假设 D 和 E 已经不能区分 d1 和 d2，再证明则 D⊓E 也不能区分 d1 和 d2。这一步一定要看懂，尤其是“假设”这个操作，这是归纳法的核心也是最难理解的一步。这时候，千万不要讨论 D 和 E 具体是什么形式，它们是任意两个 ALC concepts。任意两个 ALC concepts 不能区分 d1 和 d2，则它们的 conjunction 也一定不能区分 d1 和 d2。只有这样证明才是一个通用的证明。则我们要证明的是：

$$d1 \in (D \sqcap E)^{I1} \text{ iff } d2 \in (D \sqcap E)^{I2}$$

因为是 iff，我们要证明两个方向：

$$d2 \in (D \sqcap E)^{I2} \text{ if } d1 \in (D \sqcap E)^{I1}$$

$$d1 \in (D \sqcap E)^{I1} \text{ if } d2 \in (D \sqcap E)^{I2}$$

先证明第一个方向，假设 $d1 \in (D \sqcap E)^{I1}$，则根据 ALC 语义，$d1 \in D^{I1}$ and $d1 \in E^{I1}$。我们刚才又假设 D 和 E 已经不能区分 d1 和 d2，所以 $d2 \in D^{I2}$ and $d2 \in E^{I2}$。根据 ALC 语义，我们知道 $d2 \in (D \sqcap E)^{I2}$。第一个方向证明完毕。第二个方向的证明是镜像第一个的版本，我们不再赘述。或者像参考书里面的证明一样，直接使用 iff 关系得到证明。

接下来考虑 C 是一个 negation 的情况：C = ¬D。同样地，先假设 D 已经不能区分 d1 和 d2，再证明¬D 也不能区分。则我们要证明的是：

$$d1 \in (\neg D)^{I1} \text{ iff } d2 \in (\neg D)^{I2}$$

这里还是只证明一个方向：

$$d2 \in (\neg D)^{I2} \text{ if } d1 \in (\neg D)^{I1}$$

假设 $d1 \in (\neg D)^{I1}$，则根据 ALC 语义，$d1 \in \Delta^{I1}/D^{I1}$。我们刚才又假设 D 已经不能区分 d1 和 d2，所以 $d1 \in D^{I1}$ iff $d2 \in D^{I2}$。而 $d1 \in \Delta^{I1}/D^{I1}$ 告诉我们 d1 不在 D^{I1} 里，则 d2 也一定不在 D^{I2} 里（否则就违背了 D 已经不能区分 d1 和 d2）。则 $d2 \in \Delta^{I2}/D^{I2}$，则 $d2 \in (\neg D)^{I2}$，证毕。

接下来考虑 C 是一个 exists 的情况：C = ∃r.D。先假设 D 已经不能区分 d1 和 d2，再证明∃r.D 也不能区分。则我们要证明的是：

$$d1 \in (\exists r.D)^{I1} \text{ iff } d2 \in (\exists r.D)^{I2}$$

这里还是只证明一个方向：

$$d2 \in (\exists r.D)^{I2} \text{ if } d1 \in (\exists r.D)^{I1}$$

假设 $d1 \in (\exists r.D)^{I1}$，则根据 ALC 语义，there is a $d1' \in \Delta^{I1}$ such that $(d1,d1') \in r^{I1}$ and $d1' \in D^{I1}$。则根据(I1,d1) ~ (I2,d2)成立的第二个条件（2），there is a $d2' \in \Delta^{I2}$ such that $(d2,d2') \in r^{I2}$ and $d2' \in D^{I2}$。则根据 ALC 语义，$d2 \in (\exists r.D)^{I2}$，第一个方向证毕。另外一个方向的证明是第一个方向的镜像版本，其中使用的是(I1,d1) ~ (I2,d2)成立的第三个条件（3）。

至此，全部证明结束。我们得到一个非常重要的结论：任何 ALC concept 都不能区分两个 bisimilar 的元素，它们都不是谛听或者如来。

在这个结论的基础上，我们进一步知道：如果此时存在一个 concept 能够区分两个 bisimilar 的元素，则它一定不是 ALC concept。这就给我们证明语言之间的表达力提供了基础。

以 proposition 3.3 为例，如果想证明：ALCN is more expressive than ALC，只需要构造一个 ALCN concept C 使得 C 能够区分两个 bisimilar 的元素。首先构造两个 pointed interpretations：

(I1,d1)和(I2,e1)。再构造一个 ALCN concept C：<=1r.⊤。显然，<=1r.⊤能够区分 d1 和 e1，因为 $e1 \notin$ <=1r.⊤。则 C 一定不是 ALC concept，则存在一个 ALCN concept C 使得任意 ALC concept D such that C≠D。其它 DL 语言的表达力与 ALC 的对比也可以参考同样的思路。

上述使用 bisimulation 和 invariance 的 ideas 证明逻辑语言表达力的不同，来自于荷兰阿姆斯特丹大学教授 Johan van Benthem，中文名字范炳坤。他同时是斯坦福大学和清华大学客座教授，是国际著名逻辑学家，主要工作关注在模型论和模态逻辑上。因为描述逻辑与模态逻辑之间的镜像关系（见参考书 2.6.2），许多描述逻辑领域的概念和技术都来自模态逻辑。

上面的证明思路，我们直接被灌输到头脑中，可能不觉得有什么。但试想一下，如果你是那位开拓者，此时尚未发现某种方法来证明可计算语言之间的表达力不同，想到上述的方法是不是非常厉害。周院长在说南大 AI 学院建立的宗旨之一，就是培养那些设计者，奠基

者，不是搬砖者，希望同学们能够理解这些证明思路的底层哲学。

接下来介绍 disjoint union 的概念，它表示互不相交的集合，比如{1,2}和{3,4}和{5,6}。这里首先定义一个索引集Ω（index set），里面是 1,2,⋯,n 等正整数。然后创造一组 interpretations $(I_\nu)\ \nu \in \Omega$，比如：

$I1$：$\Delta^{I1}=\{a1,b1,c1,d1\}$
$A^{I1}=\{a1\}$
$r^{I1}=\{(a1,b1)\}$
$I2$：$\Delta^{I2}=\{a2,b2,c2,d2\}$
$B^{I2}=\{b2\}$
$s^{I2}=\{(b2,c2)\}$
$I3$：$\Delta^{I3}=\{a3,b3,c3,d3\}$
$C^{I3}=\{c3\}$
$t^{I3}=\{(c3,a3)\}$
⋯

在此基础上构造一个新的 interpretation J，定义如下：

J：$\Delta^J=\{(a1,1),(b1,1),(c1,1),(d1,1)，(a2,2),(b2,2),(c2,2),(d2,2)，(a3,3),(b3,3),(c3,3),(d3,3)\}$
$A^{I1}=\{(a1,1)\}$ $B^{I2}=\{(b2,2)\}$ $C^{I3}=\{(c3,3)\}$
$r^{I1}=\{((a1,1),(b1,1))\}$ $s^{I2}=\{((b2,2),(c2,2))\}$ $r^{I3}=\{((c3,3),(a3,3))\}$

J 的构造很简单，将 I1，I2，I3 并在一起，并给其中每个元素一个对应的索引即可。这里注意：I1，I2，I3 的 domains 并不一定是完全不相交的，可以两两相交。我们接下来的结论也不会因此而变得不成立。比如它们甚至可以是完全一模一样的 interpretation，加上索引之后并在一起，如果这样的 interpretation 是无限的，则称 J 为 countably infinite disjoint union of I with itself。

定义了 disjoint union J of $(I_\nu)\ \nu \in \Omega$之后，我们介绍一个重要的引理（Lemma 3.7）：任何 ALC concept 都无法区分 I_ν和 J 上对应的 d 和(d,v)。也就是说，我们需要证明$(I_\nu,d) \sim (J,(d,v))$。证明相对直观，不再赘述。在此基础上，我们得到 Theorem 3.8：假设 T 是一个 ALC TBox，$(I_\nu)\ \nu \in \Omega$是 T 的任意一组 models，则它们的 disjoint union J 也依然是 T 的 model。这是不是相当于 model 在 disjoint union 操作上的闭包？假设 T 有 n 个 models（n 可以是无限大），则任意两两组合，三三组合，⋯，都是 T 的 model。原则上闭包里有 2^n-1 个 T 的 models。按照这个思路，因为 n 可以是无限大，则任意针对 T 可满足的 concept C 都存在一个解释 J 使得 C 的解释 C^J 无限大。证明也很简单：因为 T 是可满足的，所以必然存在一个 I 使得 T 可满足。因为 C 针对 T 可满足，则一定存在一个 element $d \in \Delta^I$ 使得 $d \in C^I$。那么此时只需要构造一个针对 I 自身的闭包 J（countably infinite disjoint union of I）即可。根据 Theorem 3.8，J 也是 T 的 model；根据 Lemma 3.7，$(d,n) \in C^J$ with $n \in \mathbb{N}$（自然数集）。所以：任意可满足的 ALC concept 都存在无限大的模型。

下面介绍任意可满足的 ALC concept 都一定存在有限大的模型，这个性质称为 finite model property，简称 fmp。模型的有限性非常重要，它是语言“可决定”（decidable）的前提。具体来说，它让前面我们介绍的重要任务：验证针对任意 ALC TBox T 可满足的 ALC concept 是 decidable 的。这也是 Tableau 算法的理论前提。如果一个语言的模型都是无限的，或者模型有的是有限的有的是无限的（不保证一定有限），理论上 Tableau 算法无法为其构造一个 model，因为构造的过程是不终止的（not terminating）。

为了证明 ALC 语言具备 fmp，我们需要定义一些概念：ALC concept 的 size 和 subconcepts，两个定义都是递归定义，都有一个 base case。

size(C)：

（1）base case：if C = A 或者 top、bottom concept（C 是叶节点），size(C) = 1

（2）C = C1 ⊓ C2 或者 C = C1 ⊔ C2，size(C) = 1 + size(C1) + size(C2)

（3）C = ¬D 或者 C = ∃r.D 或者 C = ∀r.D，size(C) = 1 + size(D)

总结下就是：每个 concept name（包括 top 和 bottom）以及每个逻辑连接符的 size 都是 1。

sub(C)：

（1）base case：if C = A 或者 top、bottom concept（C 是叶节点），sub(C) = {A}

（2）C = C1 ⊓ C2 或者 C = C1 ⊔ C2，size(C) = {C} + sub(C1) + sub(C2)

（3）C = ¬D 或者 C = ∃r.D 或者 C = ∀r.D，sub(C) = {C} + sub(D)

总结下就是：每个 concept name（不包括 top 和 bottom）以及以每个逻辑连接符为根节点的 concept 都是 C 的 subconcept（包括 C 本身）。

上述两个定义可以扩展到 axiom 层面，或者继续扩展到 TBox 层面：

size(C ⊑ D) = size(C) + size(D)

size(T) = $\Sigma_{C \sqsubseteq D \in T}$ size(C ⊑ D)

sub(C ⊑ D) = sub(C) + sub(D)

sub(T) = $U_{C \sqsubseteq D \in T}$ size(C ⊑ D)

很容易看到：|sub(C)| 受到 size(C) 的限制，同样，|sub(T)| 受到 size(T) 的限制。于是有了 Lemma 3.11。|sub(C)| <= size(C) 是因为 top/bottom concept 不算 subconcept，另外，C 中可能存在一些 identical subconcepts，比如 C = A ⊔ ∃r.A，sub(C) 是一个集合，它只会收录 A 这个 subconcept 1 次，但在算 size 的时候，A 贡献了 2 次计数。

接下来做一个关于 subconcept 的闭包定义：a set S of ALC concepts is closed if U{sub(C) | C ∈ S} ⊆ S。显然，如果 S 是包含 ALC concept 或者 TBox 的 subconcepts 的集合，则 S 是闭合的。

再接下来定义 S-type 的概念：让 S 是 a set of ALC concepts，I 是一个解释，d ∈ Δ^I 为 domain 中一个元素。S-type 定义在 d 上，指的是 S 中所有包含 d 的概念的集合，记作 $t_S(d)$。有多少个 S-type 呢？因为 S-type 是 S 的子集，这样的 S-type 最多可以有 S 的子集的数量。又因为：If a set has “n” elements, then the number of subset of the given set is 2^n。证明非常直观：一个集合里面的元素只有两种选择，要不选择进入到 subset，要不不进入，一共 n 的元素，所以一共有 2^n 种可能性（包括全集和空集）。这个结论形成了 Lemma 3.13。

接下来我们要证明 ALC 具备 fmp，也就是证明任意可满足的 ALC concept 都一定存在有限大的模型。这里用什么方法证明？它满足构造法的特点。我们可以选定一个通用模型，在这个模型里面，所有的 ALC concept C 都只包含一个 element，非必要不创造新的 element。比如为了证明 concept C 针对 TBox T = {C ⊑ ∃r.D} 可满足，只需要找到一个 T 的 model I，其中，I 使得 C^I = {a}，一个元素就足够证明 C 的可满足性。同样地，对于∃r.D 的解释，根据 T 中 axiom 的限制，只需要$(\exists r.D)^I$ = {a}，没必要向里面再添加新的元素。但是对于 D 的解释就不一样了，需要创造一个新的 element b，使得(a,b) ∈ r^I 且 b ∈ D^I。或许有同学有疑问，

这里有必要创造新的 element b 么？直接使用原来的 a 不行么？比如(a,a)∈ r^I 且 a∈ D^I。答案是不可以，这并不是通用模型。给定一个 C 针对于 T 的随机 model，我们可以对那些有同样 S-type 的元素进行合并，合并为一个元素来构造一个通用模型。为此，我们介绍一种 S-filtration 技术。

S-filtration：让 S 是一组 ALC concepts，I 是一个 interpretation。我们定义一个等价关系，这里为了节约时间，直接用$=_S$ 符号来表示这个等价关系。两个事物之间存在等价关系，比如平面几何的全等关系、比如代数的等值关系、本质上是指这两个事物可以互相替代，并且不会影响后续的任何操作。这个等价关系定义在 I 的 domain 里面的两个元素之间：

$$d =_S e \text{ if } t_S(d) = t_S(e)$$

两个元素有$=_S$等价关系，则它们的 S-type 一致，则它们属于 S 中的同样的 concepts。那么我们可以选出一个代表，让它来代表所有 S-type 一致的元素。比如让 d 成为代表，则它所代表的元素的集合称为 d 的$=_S$ -equivalence class，用$[d]_S$ 表示。按照定义：

$$[d]_S = \{e \in \Delta^I \mid d =_S e\}$$

S-filtration 是定义在 I 上的函数：S-filtration(I) = J。结果是一个新的 interpretation J：

$$\begin{aligned}
\Delta^{\mathcal{J}} &= \{[d]_S \mid d \in \Delta^{\mathcal{I}}\};\\
A^{\mathcal{J}} &= \{[d]_S \mid \text{there is } d' \in [d]_S \text{ with } d' \in A^{\mathcal{I}}\} \text{ for all } A \in \mathbf{C};\\
r^{\mathcal{J}} &= \{([d]_S, [e]_S) \mid \text{there are } d' \in [d]_S, e' \in [e]_S \text{ with } (d', e') \in r^{\mathcal{I}}\}\\
&\quad \text{for all } r \in \mathbf{R}.
\end{aligned}$$

以前 I 的 domain 的 element 是 individual，现在 J 的 domain 的 element 是包含 individual 的集合，也就是 domain 的元素是一个一个的 S-type 集合。这些 S-type 之间是 disjoint 的。也就是说，不可能存在一个元素，它出现在某一个 S-type 中，又同时出现在另一个 S-type 中。这个很好理解，如果是这样的话，说明两个 S-type 是一样的，可以合并为一个。这是从 I 到 J 在 domain 上的改变。在解释函数上的改变更为抽象。J 把 A 解释到Δ^J 的子集上去，也就是找出一部分 S-type，要求这些集合中的元素都属于 A。J 对于 r 的解释也是遵循同样的思路，对应到 J 的 domain 的两个 S-type 之间，要求两个集合元素之间存在 r 关系。举个例子来帮助理解：

（1）先确定 S-type 中的 S = {A, B, C, ∃r.C}

（2）再确定初始的解释 I：Δ^I = {a, b, c, d, e, f}

A^I = {a, b, e} B^I = {b, c, d} C^I = {c, d, f}

r^I = {(a, c), (b, d), (e, f)}

则$(\exists r.C)^I$ = {a, b, e}

（3）找出各个元素的 S-type：

$t_S(a)$ = {A, ∃r.C}

$t_S(b)$ = {A, B, ∃r.C}

$t_S(c)$ = {B, C}

$t_S(d)$ = {B, C}

$t_S(e) = \{A, \exists r.C\}$

$t_S(f) = \{C\}$

（4）对各个元素进行归类：

$[a]_S = \{a, e\}$

$[b]_S = \{b\}$

$[c]_S = \{c, d\}$

$[d]_S = \{c, d\}$

$[e]_S = \{a, e\}$

$[f]_S = \{f\}$

（5）构建 I 的 S-filtration J：

$\Delta^J = \{\{a, e\}, \{b\}, \{c, d\}, \{f\}\}$

$A^J = \{\{a, e\}, \{b\}\}$ $B^I = \{\{b\}, \{c, d\}\}$ $C^I = \{\{c, d\}, \{f\}\}$

$r^J = \{(\{a, e\}, \{c, d\}), (\{b\}, \{c, d\}), (\{a, e\}, \{f\})\}$

在这个基础上，我们有了 Lemma 3.15：这时候的 S 指的是一组有限、闭合的 ALC concepts，意味着所有的 ALC concepts 的 subconcepts 都已经在 S 之中。让 I 是一个 interpretation，则我们可以得到：

$$d \in C^I \text{ iff } [d]_S \in C^J$$

d 是 Δ^I 中任意 element，且 C 是 S 中任意 concept。

Lemma 3.15 是想证明 S 中的任何 concept C 都无法区分 d 和 $[d]_S$。看起来很像是 d 和 $[d]_S$ 之间有 bisimilar 关系。确实，如果能证明 $(I,d) \sim (J,[d]_S)$，则上面的结论直接成立，因为任何 ALC concept 都不能区分两个 bisimilar 的元素。但很遗憾，按照 J 的定义，$(I,d) \sim (J,[d]_S)$ 不成立。但 bisimulation 关系不成立不代表 $d \in C^I$ iff $[d]_S \in C^J$ 这个结论不成立；换句话说，bisimulation 是其成立的充分条件但不是必要条件。

先看下为什么 $(I,d) \sim (J,[d]_S)$ 不成立。首先，bisimulation 并没有设置参数 S，它的要求是任何时候 d 属于某个 concept name，则 $[d]_S$ 也要属于这个 concept name，无论这个 concept name 在不在 S 中。这意味着 S 必须囊括所有的 concept name 才可以让 bisimulation 理论上成立。我们在上文介绍过 bisimulation 也可以参数化（parameterized），但即便如此，我们也依然可以证明按照 J 的定义，$(I,d) \sim (J,[d]_S)$ 无法永远成立。参考书上给了一个例子，我们这里给出另外一个：首先得承认，J 的定义可以满足成为 bisimulation 的第一个条件（1），因为这也是 J 的定义要求，它对 A 的解释方式直接就满足了 bisimulation 的条件（1）。但当有 r 关系出现的时候情况就发生了变化。比如如果 $S = \{A, B, \exists r.B\}$，按照定义，S 是闭合的：

（1）先确定 S-type 中的 $S = \{A, B, \exists r.B\}$

（2）再确定初始的解释 I：

$\Delta^I = \{a, b, c, d\}$

$A^I = \{a, b, c\}$

$B^I = \{b, d\}$

$r^I = \{(a, b), (c, d)\}$

则 $(\exists r.B)^I = \{a, c\}$

（3）找出各个元素的 S-type：

$t_S(a) = \{A, \exists r.B\}$

$t_S(b) = \{A, B\}$

$t_S(c) = \{A, \exists r.B\}$

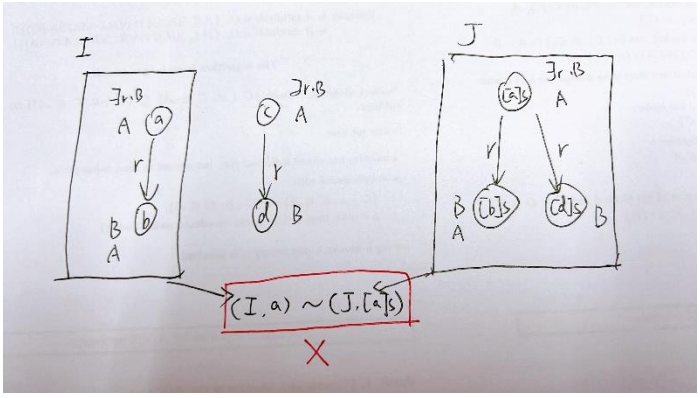
$$t_S(d) = \{B\}$$

（4）对各个元素进行归类：

$$[a]_S = \{a, c\}$$
$$[b]_S = \{b\}$$
$$[d]_S = \{d\}$$

（5）构建 I 的 S-filtration J：

$$\Delta^J = \{[a]_S, [b]_S, [d]_S\}$$
$$A^J = \{[a]_S, [b]_S\}\ B^I = \{[b]_S, [d]_S\}$$
$$r^J = \{([a]_S, [b]_S), ([a]_S, [d]_S)\}$$



看上图，(I, a) ~ (J, $[a]_S$)并不成立，因为 J 中$[a]_S$有一个 r-successor $[d]_S$，它只是 B^J 中的一个元素。但是 I 中的 a 却没有这样的 r-successor。因此，它们不能满足 bisimilar 关系。

尽管如此，我们要证明：$d \in C^I$ iff $[d]_S \in C^J$。同样的方法，我们使用归纳法进行证明：我们的 base case 是 C = A，其中 A 是一个 concept name，我们要证明 A 无法区分 d 和$[d]_S$：

$$d \in A^I \text{ iff } [d]_S \in A^J$$

因为是 iff，我们要证明两个方向：

$$[d]_S \in A^J \text{ if } d \in A^I$$
$$d \in A^I \text{ if } [d]_S \in A^J$$

先证明第一个方向：如果 $d \in A^I$，则$[d]_S \in A^J$，这是 J 对 A 的解释直接得到的。再证明第二个方向：如果$[d]_S \in A^J$，则根据 J 对 A 的解释，$[d]_S$ 中存在一个元素 d’ 使得 $d' \in A^I$。又因为 $d =_S d'$ 且 A 为 S 中一个元素，则可以由 $d' \in A^I$ 得出 $d \in A^I$。

接下来我们要证明 induction hypothesis，首先 C 是一个 conjunction 的情况：$C = D \sqcap E$。则我们要证明的是：

$$d \in (D \sqcap E)^I \text{ iff } [d]_S \in (D \sqcap E)^J$$

因为是 iff，我们要证明两个方向：

$$[d]_S \in (D \sqcap E)^J \text{ if } d \in (D \sqcap E)^I$$
$$d \in (D \sqcap E)^I \text{ if } [d]_S \in (D \sqcap E)^J$$

先证明第一个方向，假设 $d \in (D \sqcap E)^I$，根据 ALC 语义，$d \in D^I$ 且 $d \in E^I$。又因为 $d \in [d]_S$，则根据 J 的定义，$[d]_S \in D^J$ 且 $[d]_S \in D^J$。

我们刚才又假设 D 和 E 已经不能区分 d1 和 d2，所以 $d2 \in D^{I2}$ and $d2 \in E^{I2}$。根据 ALC 语义，我们知道 $d2 \in (D \sqcap E)^{I2}$。第一个方向证明完毕。第二个方向的证明是镜像第一个的版本，我们不再赘述。或者像参考书里面的证明一样，直接使用 iff 关系得到证明。

接下来，我们给出一个重要的结论，Theorem 3.16：这个结论不但告诉我们 ALC 具有 fmp，还更为直接的指出，存在一个让 C 针对 T 可满足的 model 里面元素个数的上界是 2^n，这里的 n = size(T) + size(C)。这里建议自己举几个简单的例子看一下是不是这样。看一下这个定理证明的思路。先假设存在一个 interpretation I 让 C 针对 T 可满足。让 S = sub(T) + sub(C)，也就是说 S 是闭合的，里面包含了 T 和 C 里面所有的 subconcepts。根据 Lemma 3.11，S 里面元素的个数，也就是 subconcepts 的个数一定是小于等于 n。再根据 Lemma 3.13，I 的 S-filtration J 的里面的元素一定满足：$|\Delta^J| <= 2^n$。Lemma 3.13 说了什么？它说给定一个有限的 S，比如 S = {A, B, ∃r.B}，对于一个 interpretation I 来说，Δ^I 中的某个元素 d 最多有 2^3 个 S-type，那么 $[d]_S$ 最多有 2^3 个，则 Δ^J 里面的也元素最多有 2^3 个。则现在只需要证明 J 和 I 一样，也能让 C 针对 T 可满足。我们让 d 是 C^I 中一个元素。因为 S 是闭合的，则 C 一定会在 S 中，则根据 Lemma 3.15，可以得到 $d \in C^I$ iff $[d]_S \in C^J$。所以显然 C^J 不为空。现在只需要证明 J 是 T 的 model 即可。我们泛化一个 T 中的 GCI，表示为 $D \sqsubseteq E$，以及一个 Δ^J 中元素 $[e]_S \in D^J$，现在我们需要证明 $[e]_S \in E^J$。因为 S 是闭合的，D 属于 S，根据 Lemma 3.15 可以得到 $e \in D^I$ iff $[e]_S \in D^J$，所以 $e \in D^I$。又因为我们已经假设 I 是 T 的 model，而 $D \sqsubseteq E$ 是 T 中的一个 GCI，根据语义，我们知道 $e \in E^I$。最后再根据 Lemma 3.15，得知 $[e]_S \in E^J$。可以看到，整个证明过程十分细致又十分繁杂，建议同学们用一到两个例子反复检查上述证明过程中出现的每一个过程。

那么接下来的 Corollary 3.17，也就是 ALC 具备 fmp 的结论就直接得证了。上界都找到了，一定是有限的。既然 C 针对 T 可满足的 model 里面，必然存在一个 model 的元素有限，这个 model 是 S-filtration J。则可以证明验证 C 针对 T 是否可满足这件事是 decidable 的。则一个有关计算理论的推论 Corollary 3.18 就诞生了。很多同学会问，这个引理很重要么？答案是非常非常重要！它等于直接告诉我们：ALC 是一个适合作知识表示的语言。因为 ALC 本身是 decidable 的，在其上的 reasoning（SAT 验证）是 decidable 的，任何时候都可以返回一个 yes or no 的答案。不像一阶谓词逻辑，是 undecidable 的。所以同学们看到 AI 论文中说“使用一阶谓词逻辑而不是其某个可判定子集”作知识表示并完成推理任务，一定是一个 KR 外行，作为 reviewers 直接拒稿即可。但是可惜啊可惜，实在可惜，很多 reviewers 并不是 KR 学者，允许了太多 neuro-symbolic 领域的 paper 进来，里面充斥着一阶谓词逻辑作知识表示的工作，并未对 decidability 做任何讨论（因为一阶谓词逻辑是其它领域学者接触最多的逻辑语言，和命题逻辑一样，在本科阶段学过）。这意味着，理论上这些 paper 里面的 AI 算法存在风险使得在运行到某个阶段时陷入到无限循环之中。尽管实际测试中，很少有算法遇到这种情况。这是因为那些 neuro-symbolic 的 paper 打着逻辑的旗号，实际上并未做演绎推理；另外，大家都知道最坏情况原本就很少出现。但是任何理论上不完美的东西在 KR 领域都是不被接受的，像之前说的，KR 追求的理念是 theoretical soundness，这点与 AI 其它领域追求的 empirical success 或者 practical usefulness 理念完全不同。

Theorem 3.19 阐述了 ALCIN 不具备 fmp，具体问在在于 I 与 N 的交互，详细内容需要认真过滤一遍。这个证明并不难，不多赘述。

接下来我们介绍一个重要的概念，tree model property（树状模型性质），简写为 tmp。我们要证明每一个可满足的 ALC concept 都有一个树状模型。为此，我们首先定义什么是树？简而言之，就是：（1）只有唯一根节点（无父节点了）；（2）每一个其它节点（排除根节点）只有一个父节点。

Definition 3.20 给出了 tree mode 的定义，很好理解。我们只需要记住：根节点是要验证的 concept C 里面的元素（回忆下 Tableau）。和上面的问题一样，当我想证明每一个可满足的 ALC concept 都有一个树状模型，是不是首先要找到这个树状模型？是不是要构造一个？但是构造哪个呢？因为满足 C 的模型是无限的，它的树状模型显然也是无限的（可想一想 disjoint union 的概念）。这时候头脑中要有一个思想：找一个“通用”模型，它可以使用某种算法，某项技术系统性构造。这是构造法的精髓。为此，我们介绍一种叫做 unravelling 的技术，它可以帮助我们系统性构造树状模型。这项技术同 bisimulation，invariance 一起成为 KR 语言分析中十分重要的概念。多少经典 paper 和著作中有其身影的出现，甚至不仅仅是在 KR 领域。我们以 DL 语言为背景，对其进行初步讲解。

首先进行必要概念的定义：给定一个 interpretation I 和其中一个元素 $d \in \Delta^I$。我们定义一个叫做 d-path 的概念。它指的是从 d 出发，经由任意二元关系连接（edge），一直到某个节点停止。记住，这个终止节点（end node，用 end(p)来表示）不一定是某个叶节点（无子类的节点），到任意节点终止都可以。这样一条 path 的长度就是其串联节点的数量。这样一来，原始的 I 的 domain 中的元素是 d 这样的 individual，而在新的 unravelled model 里面，domain 中的元素不再是 d，而是从 d 出发的一条路径。Unravelling 的思想很直观：Δ^I 的元素以前工作的时候，承担了很多工作，比如参考书 Figure 3.5 中的图，左边图中 d 节点既要接收来自 e 节点的问候，又要传递像 e 和 d 节点传递出问候，也就是说计算的过程中，机器会多次读到 d 节点。而 unravelling 的目的是对这样一个“automata”进行结构上的展开，让其每一个 state 都只被“途径一次”，一旦在计算路径中再次遇到这个 state，就创造一个这个 state 的镜像 copy，类似于孙悟空拔了根脖子上的毛，吹出了又一个孙悟空分身。这样一来，我们可以使用一条路径的终止节点来代表整个路径。因为在树状图中，从起始节点 d 出发到达某个其它节点的路径是唯一的，用终止节点代表整个路径，是一个一对一的函数。

有了这样一个 unravelling 的技术和由此带来的 unravelled model，我们可以定义一个新的 interpretation J，称为初始 interpretation I 的 unravelling：它的 domain 是由从 d 起始的所有路径组成；对于 A 的解释也很直观，当这个路径的之终结点属于 A 的时候，这条路径就是 A 的一个元素；对于 r 的解释采用同样的思路，当两条路径的终止节点之间满足 r 关系，则这两条路径满足 r 关系。并且大家想一下，这样的路径只可能是一条路径是另一条路径的子路径。确切地说，是路径 p1 + end(p2) = p2。

不难看出，至少直觉不难感受到，原来的 I 与现在的 J 之间存在 bisimulation 关系，且这个关系存在于任意节点 d1 与从 d 起始，以 d1 为终止节点的这条路径之间。如果用 p 表示路径，可以表示为(I, end(p)) ~ (J, p)，其中 d 默认为起始节点，d1 表示 end(p)。

为了证明(I, end(p)) ~ (J, p)，需要逐个证明 bisimulation 的三个条件成立（见 Definition 3.1）：
（1）首先对比的节点 end(p)和路径 p 满足：如果 end(p)属于 A^I，则路径 p 属于 A^J；反之亦然。这个可以直接得证，因为这是 unravelling J 对 A 的解释方式；
（2）接下来需要证明，如果存在一条路径 q 使得$(p, q) \in r^J$，则按照 bisimulation 的定义一定存在一个 element $d \in \Delta^I$使得(I, d) ~ (J, q)且$(end(p), d) \in r^I$。现在的任务就是找到这个 d。不卖关子，直接让 d = end(q)。则接下来我们需要证明(I, end(q)) ~ (J, q)且$(end(p), end(q)) \in r^I$。因为前面已经假设$(p, q) \in r^J$成立，按照其定义，$(end(p), end(q)) \in r^I$直接成立。接下来

如果觉得抽象难理解，我的建议是先理解前面的这些概念，如果对于 bisimulation，d-path 的概念理解模糊，这里很难看懂这些一环套一环的证明。如果(I, end(p)) ~ (J, p)成立，显然任何 ALC concepts 对它们都不做区分（Proposition 3.23）。

接下来，来到本章最后一个重要结论：ALC 具备 tmp。那么我们只需要证明构造的这个通用的 unravelling J（由 d 起始）能够让 C 针对 T 可满足，且 J 是一个 tree model。

假设 Interpretation I 是 T 的一个 model，且存在一个 element $d \in \Delta^I$ 使得 C^I 非空。我们需要按照上面 unravelling 的定义创造一个 I 的 unravelling J 并证明 J 是从 d 起始，C 针对 T 可满足的一个 tree model。

（1）先证明 J 是 T 的 model。假设 $D \sqsubseteq E$ 是 T 中的任意 GCI，我们要证明的是任何Δ^J 里面的元素（一条路径）p，当 $p \in D^J$ 的时候，则一定 $p \in E^J$。因为 $p \in D^J$，根据 Proposition 3.23，$end(p) \in D^I$；又因为 I 是 T 的 model，则 $end(p) \in E^I$；再根据 Proposition 3.23，$p \in E^J$。

（2）接下来证明 J 的 graph representation G_J 是一个起始于 d 的 tree。G_J 想成为 tree 的要求：只有唯一根节点且每个其它节点只有一个父节点。根据定义，d 是唯一长度等于 1 的 d-path，且任何 length > 1 的 d-path 都存在一个二元关系与前置路径相关。则 d 成为了唯一没有前置节点的节点，（1）得证。假设 p 是任意 length > 1 的 d-path 路径，则必然存在唯一 d-path q 使得 p = q + end(p)。否则如果还有另外一条 d-path o 使得 p = o + end(p)，那么意味着 q 或者 o 则 end(p)的父节点有

（3）最后证明 $d \in C^J$。因为 $d \in C^I$，根据 Proposition 3.23，可以得知 $end(d) \in C^J$；又因为 d = end(d)，所以 $d \in C^J$。

==========

接下来加入一个所谓的“interlude”，介绍一个计算机最底层的理论，也是最重要的理论 — 计算理论，的一些基本知识（prerequisites）。没有这个理论，计算机科学就不能称为是科学，只能是 computing 或者 computing technology。计算理论研究的是：

（1）计算模型（包括形式语言、自动机）
（2）哪些问题是可计算的，哪些是不可计算的（可计算性理论及算法）
（3）计算需要多少时间、消耗多少存储（计算复杂度理论）

在介绍之前，我们先树立一些思想，定义和理解一些必要的概念。首先将任何计算机想象成人。人需要懂一些语言。现在要学习的是，计算机到底可以掌握哪些语言？如果计算机 M 可以“识别”某个语言 L 的一个 string S，则称为 M accepts S，称 L 为 M 的语言。

Symbol – a number, letter, or sign used in mathematics, music, science like a, b, c, 0, 1, 2, ⊆…
Alphabet (Σ) – a collection of symbols（一种语言要使用的字符表）
String (s) – a sequence of symbols like a, ab, 101, sy78（string 可以为空，表示为ε）
Language (L) – a set of strings

Let Σ = {0, 1}.
L1 = set of all strings of length 2 = {00, 01, 10, 11}
L2 = set of all strings of length 3 = {000, 001, 010, 011, 100⋯}
L3 = set of all strings that begin with 0 = {0, 00, 01, 000, 001, 010⋯}

可以观察到，当Σ确定后，有的 language 是有限的，比如 L1、L2；有的是无限的，比如 L3。

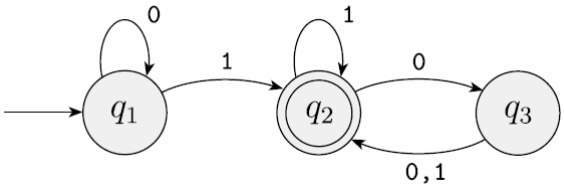
Powers of Σ:
Σ^0 = set of all strings of length 0 = {ε}，这个符号读作 eplison
Σ^1 = set of all strings of length 1 = {0, 1}
Σ^2 = set of all strings of length 2 = {00, 01, 10, 11}
⋯
Σ^n = set of all strings of length n

Cardinality of Σ: number of symbols in Σ
Cardinality of $\Sigma^n = 2^n$

Σ^* = set of all possible strings of all lengths over Σ = $\Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cdots$

定义了这些符号和概念之后我们就可以进入到计算理论的正文。首先是最简单的计算模型，叫 finite state automata，简称 FSA。其中，如果 FSA 没有输出，还可以分为 Moore machine 和 Mealy machine。如果 FSA 有输出，则可以分为 DFA、NFA、ε-NFA。我们忽略前者，只关注有输出的 FSA。

DFA – deterministic finite automata（也叫 finite automaton）
（1）it is the simplest computational model
（2）it has a very limited memory



看上面的图，里面有圆圈 q1, q2, q3，称为状态（state）；还有箭头（arrow or directed edge），箭头上有 0 和 1 这样的 symbol。一个特殊的箭头来自 nowhere，代表其所进入的状态是初始状态，一个特殊的圆圈是两个套在一起（double circled），代表其终止状态。整个计算的过程就发生在刚才介绍的这些元素上，计算必须从初始状态开始，从终止状态终止。下面对刚才的模型和计算过程进行 formalize:

DEFINITION 1.5

A ***finite automaton*** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called the ***states***,
2. Σ is a finite set called the ***alphabet***,
3. $\delta: Q \times \Sigma \longrightarrow Q$ is the ***transition function***,[1]
4. $q_0 \in Q$ is the ***start state***, and
5. $F \subseteq Q$ is the ***set of accept states***.[2]

整个的计算过程特别适合将自己想象成一台机器进行模拟（模拟人生，小时候玩的一款游戏，作者根据马斯洛需求层次理论构建了游戏中的人工智能）。假如此时此刻你站在初始状态 q1，读到了一个 symbol（信号），发现是 0；此时你打开一本操作手册，查看一下此时在 q1 点读到 0 该做什么，这个操作集就是 transition function。按照它的要求，需要从 q1 跳转到 q1，尽管你依然站在原地，但是这已经是发生了一次状态转换了。相当于在原地搭起帐篷过了一夜。用符号可以表示为：δ(q1, 0) → q1。按照这种方式，你可以踏过千山万水，只要不想回家，永远可以在外面闲逛。一个状态已经路过 1000 遍了，都可以再走第 1001 遍。但最终，你要在一个合适的时机到达终止状态，完成整个旅程。这里注意，到达终止状态不代表一定要终止，而是终止时一定要在终止状态，这个逻辑要搞清楚。你可以在终止状态上逗留一下再离开去往别的地方，但最终的最终，要再回来在这里结束整个旅程。初始状态 q0 是唯一的，但是终止状态可以是多个，所以上面用 F 这个集合来表示。

DFA 之所以叫 DFA 是指，transition function 针对一个“状态+信号”对儿，只会将你带到唯一下一个状态；整个过程是 deterministic 的。不存在你在 q1 状态读到 symbol 0，下一步有多种状态可选，不知道该往哪里走的情况出现。成为 DFA 还有一个要求，就是在其每个状态上，针对字符表中的任意 symbol，都有一个 arrow 指向下一个状态。不能字符表里面有 {0,1}，结果某个状态上只能读 1，却不能读 0 的情况出现。

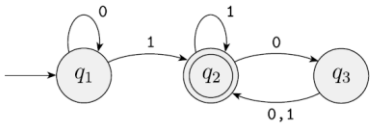


FIGURE **1.6**
The finite automaton M_1

We can describe M_1 formally by writing $M_1 = (Q, \Sigma, \delta, q_1, F)$, where

1. $Q = \{q_1, q_2, q_3\}$,
2. $\Sigma = \{0,1\}$,
3. δ is described as

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2.

4. q_1 is the start state, and
5. $F = \{q_2\}$.

如果存在 a set A of strings 能被上面 M1 这个 DFA 所接受，我们说 A 是 M1 的 language，写作 L(M) = A。这时，称为 M1 recognizes A 或者 M accepts A。因为在后面 accepts 这个词会在表达 M1 accepts strings 和 M1 accepts language 时体现的语义不同，这里我们使用 recognizes 来避免疑惑。所以，L(M) = A 等价于 M1 recognizes A。

如果一个 DFA M 的初始状态恰好同时是 M 其中的一个终止状态，那么意味着 M 可以接受 empty string ε。到这里，你可以发现每一个 DFA 都对应着一个 language，且这个 language 是可计算的 language，是 M-recognizable 的 language。

我们定义了什么是 DFA，接下来定义 DFA 上的计算。让 M = (Q, Σ, δ, q0, F)为一个 DFA 且 $w = w_1w_2\cdots w_n$ 为一个 string，且 w_i（1 《=i 《=n）属于Σ。则 M accepts w，当存在一系列的状态 $r_0,\cdots,r_n$ in Q，它满足三个条件：（1）$r_0 = q_0$，（2）$\delta(r_i, w_{i+1}) = r_{i+1}$, for i = 0,⋯,n-1，且（3）$r_n \in F$。我们说 M recognizes language L，当 L = {w | M accepts w}。

讲到这里，你会发现一个 DFA 对应一种 language。那么一定会产生一个疑问，是不是所有的 strings 都可以被 DFA accepts? 答案当然是否定的。如果某些 DFA 可以 accepts 一个语言 L，则 L 称为 regular language。regular 这个词从何处而来？只知道来自其作者 Stephen Cole Kleene，他是 Alonzo Church 的 PhD 学生，Alan Turing 的师兄。但语义上的解释并没有。

接下来你肯定会有疑问，什么样子的语言是 regular language，什么样子的不是（non-regular language）。有人说能被 DFA accepts 的就是，那不成了互相定义、循环定义。你肯定想知道满足 regular language 的内涵是什么？具体有哪些规律或者 pattern? 这里，我们首先定义几种运算，称为 regular operations，分别是 union、concatenation、star。

DEFINITION 1.23

Let A and B be languages. We define the regular operations ***union***, ***concatenation***, and ***star*** as follows:

- **Union**: $A \cup B = \{x | \, x \in A \text{ or } x \in B\}$.
- **Concatenation**: $A \circ B = \{xy | \, x \in A \text{ and } y \in B\}$.
- **Star**: $A^* = \{x_1 x_2 \ldots x_k | \, k \geq 0 \text{ and each } x_i \in A\}$.

这里需要定义一个叫做 closed 的概念。和逻辑语言里面的 closed 一样，是定义在某个运算在某个集合上。我们说某个集合针对于某个运算是 closed，意味着对这个集合里面任何元素进行这个运算，其结果都已经在这个集合里面。We say that N (the set of natural numbers) is closed under multiplication, we mean that for any x and y in N, the product x * y is also in N。很明显 N is not closed under division，因为 1/2 已经不再是自然数。基于此定义，我们可以得到 the class of regular language is closed under union and concatenation（Theorem 1.25 1.26）。

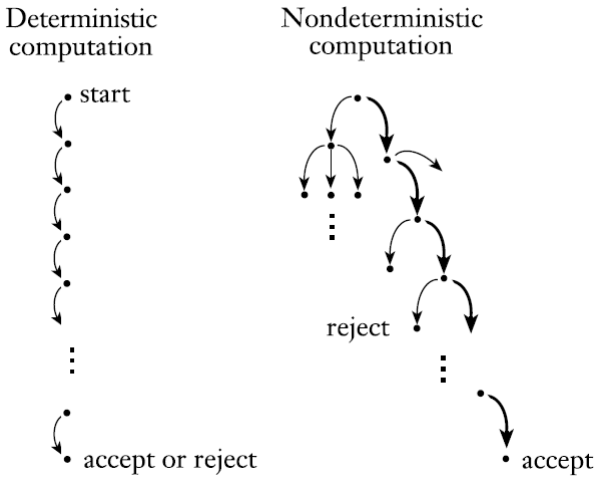
上面看到从某个状态，读到一个 symbol 之后，转移到另一个状态，这个 transition function 是 functional 的，也就是说去往的状态是确定的，唯一的。相当于来到一个分岔路口，指向标告诉你接下来要走哪一条路。那么有没有一种情况，当处在某个分叉路口的时候，不再有指向标，而是需要在多个选择中去做抉择。这种情况，我们叫做 non-deterministic finite machine，简称 NFA。

NFA 的状态数依然是有限的；字符表集合和 DFA 也没什么不同，只不过在状态转化的时候，读到的 symbol 不仅仅来自字符表集合，还可以是ε（empty string）。ε是一个很重要的符号，它不属于字符表，当某个状态上读到它的时候，意味着你可以在不读任何 symbol 的情况下去到下一个状态；当然最大的不同还是在于 transition function 不再是 functional 的，意味着在某个状态上读到某个 symbol 后选择不止一种，可以去到 a 状态，还可以是 b 状态等等。那具体的计算过程是什么样子的呢？依然把你自己想象成那个移动的主角。现在当你在路口面临多个选择时会怎么做？一种方法是先走其中一条路径，走完了之后返回到这个路口再走另外一条，类似于深度优先搜索。NFA 是这样运行吗？答案是不是的。而是，在路口处你会像孙悟空一样变身为多个分身（copy），以并行的方式继续所有的路径。到了下一个路口，又面临多个抉择，那么分身也会继续分身进行并行计算。如果到达某个状态发现 transition function 无法读取下一个 symbol b，换句话说，不存在 input 为 b 的 transition function，再换句话说，并不存在一个 label 为 b 的 arrow 从该状态出发去往别的状态，那么这个分身连同整个计算路径到这里就死亡了，因为它 rejects 了这个字符串。但是其它的分身继续工作，其它的计算路径继续活着。如果上述任意分身读到 string 最后一个 symbol 时恰好到达了 accept state，则 NFA accept this string。上面介绍了ε这个符号，如果计算过程中

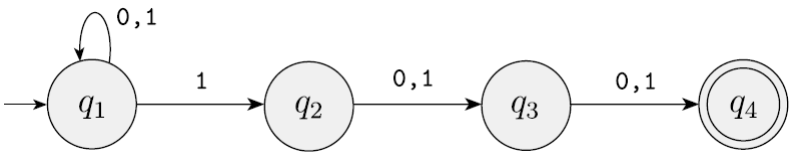
读到它，则同样地，将变为多个分身，一个分身停留在原地，其他分身去往每个当前状态上能到达的下一个状态。

到这里，有没有感觉 NFA 的思想类似于并行计算，相当于一份工作在 DFA 中一个人做，现在到了 NFA，更多的人加入其中，每个人负责其中一部分。我们用职场情景模拟下这个过程：周一，闹钟响起，关掉，第二个闹钟响起，再关掉，往复多次…，洗漱，胡乱吃了点早餐，赶地铁公交终于在 8:59 分踏入办公室。开始做昨天没完成的工作。中途想去楼下星巴克买杯美式。这时，NFA 会派出你本体的一个分身去完成买咖啡这件事，而你作为一个 DFA，只能亲自去跑一趟。这段时间，你手头的工作需要挂起，待你回来后再行继续。又过了一段时间，主管过来说这份材料有问题，需要你处理下。你不得不放下手头工作转而去处理这份材料，而 NFA 会再次派出你的分身去完成这件事。终于，厕所有了空位，抓住难得的机会冲进去，手头工作再次被挂起，而 NFA 再次派出你的分身去…（这也可以）？总之这么多工作最后是要完成的，你可以选择按照 DFA 的方式一件一件去完成，而 NFA 可以选择让这些事同时进行。事情并没有减少，但每一个分身承担的任务都减少了。这就是为什么职场上到了一定位置，需要配备 1-n 个助理，负责你的工作和工作周边。但这样也会同时增加公司的人力成本。雇主当然希望一个人完成所有的事，从而降低人员开支；而部门主管会争取一切机会增加 headcount，从而降低每个人的任务量。所以本质上可以将 NFA 理解为比 DFA 有更多的人力。但是，不代表 NFA 的工作 DFA 做不了，只是“人多而已”，并不是“技术先进”。

NFA 的计算过程无论从定义中，还是刚才的职场模拟中，都可以看作是 tree of possibilities。



下面看一个例子来加深理解，以及在计算过程中遇到“guess”这个词：



直观上，这个例子是想表示一种语言？该语言包含的所有字符串都至少长度为 3，且倒数第三个位置一定是 1。此时你站在 q1 初始位置，读到 symbol 1，有两个选择，一个是继续停留在 q1，另一个是去往 q2。如果遇到一个 string 0101111，当读到第一个 0 的时候，计算路径是确定的，依然停留在 q1，但是读到第一个 1 的时候，到底是停留在 q1 还是去往 q2，

换句话说，NFA 怎么才能知道当前这个 1 是不是倒数第三个位置的 1。答案是不知道，所以每次遇到 1，NFA 都必须 guess 这个 1 是倒数第三个，派出一个分身去往 q2，则在 q2 出读到第二个 0，前往 q3，在 q3 读到第二个 1，前往终止状态 q4，但是 string 中还有三个 1 没有被读到，但是 q4 状态上已经无法前往任何地方，所以这条路径到这里会被 kill 掉。同样地，读到第二个 1 的时候（倒数第四个 1）也会遇到同样的事情，也会被 kill 掉一次。这就是 guess 的概念，在它拿不准的时候，需要赌一下这次就是了，然后去看看到底是不是？然后被拒绝。再去，再被拒绝，精神值得赞赏~

这样，一个 NFA 的正式定义就可以得到了：

DEFINITION 1.37

A ***nondeterministic finite automaton*** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set of states,
2. Σ is a finite alphabet,
3. $\delta: Q \times \Sigma_\varepsilon \longrightarrow \mathcal{P}(Q)$ is the transition function,
4. $q_0 \in Q$ is the start state, and
5. $F \subseteq Q$ is the set of accept states.

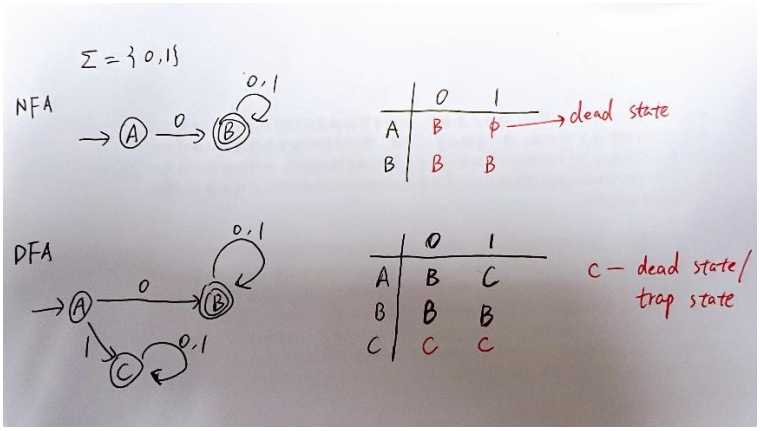
从定义可以看出，NFA 与 DFA 唯一的不同在于 3，transition function 不仅仅是 Q 中某一个状态，而是 Q 的任意一个子集里面的所有状态，所以我们用 Q 的 power set 表示 P(Q)。除此之外，还有一些定义上体现不出的不同。这些不同中，一些是硬性规定，另一些是定义和规定中推导出的“隐性不同”。硬性规定中记住两条：（1）DFA 不接受ε，而 NFA 接受；（2）DFA 在任意状态对字符表中每一个 symbol 都有对应的 transition function，而 NFA 可以只对其中一部分 symbol 有对应的 transition function。

接下来是一个很重要的结论：NFA 与 DFA 的等价性，也就是一个 NFA 与它对应的 DFA，二者 recognize 同样的语言。这一点很多人可能没有想到，有些意外。但是看到上面我们的职场情景那个例子，是不是直观多了？本质上 NFA 只是比 DFA 在面对并行任务时雇佣了更多的人，但不代表它有 DFA 达不到的能力。当然，NFA 也有好处，它让多个任务并行的能力可以节省状态数量（节省内存）。

想证明 DFA 和 NFA 的等价性，需要证明每一个 DFA 都是 NFA，反之亦然。按照定义，每一个 DFA 天然就是一个 NFA，但是反过来就不容易证明了。我们需要证明针对每一个 NFA，都存在且可以构建一个对应的 DFA（该证明我们留给学生自己去看）。

如果一台 NFA 有 k 个状态，则它最多有 2^k 个 subsets of states（一个 state 或者在集合里面，或者不在，所以是 2 选 1，一共有 k 个元素，所以是 2^k）。现在我们想为每一个 NFA 找到其对应的 DFA，有没有一个算法？答案是有。我们先将这个算法展示出来，再解释算法的正确性（soundness）。

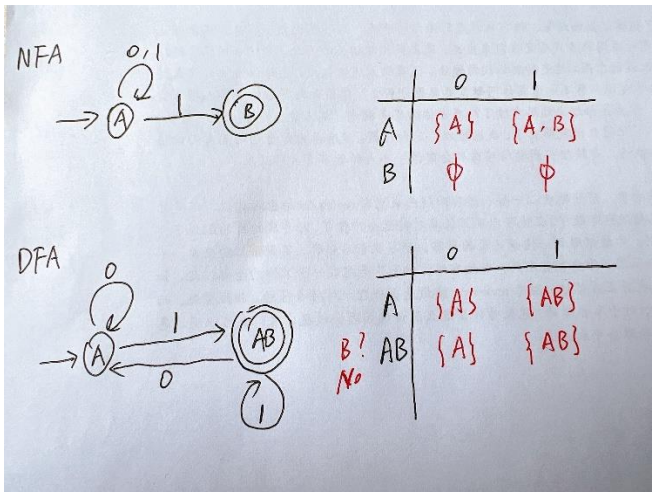
我们看一个例子来获得一些灵感。我们让 NFA 和 DFA 分别 recognize 一个 language L = {set of all strings over (0, 1) that starts with 0}



看上图，拿到一道题目，NFA 其实比 DFA 要好找、好画。画 NFA 需要一些经验和一点点规律，我们这里就不赘述了。画好了 NFA 之后我们将其转化为等价的表格（见 NFA 右边的图）。图中信息表示在状态 A，读到 symbol 0，跳转到状态 B；在状态 A，读到 symbol 1，跳转到一个叫做 dead state 的地方，用符号 phi 来表示。这是因为 NFA 中，不需要对字符表中每一个 symbol 都能反应，这里在状态 A，就只能读到 0 才有反应，针对 1 并没有一个 transition function。这时候，去到的状态叫做 dead state。接下来我们开始画对应的 DFA。我们首先根据上面你的表格来画 DFA 的表格，再转成 DFA 图。依然是从状态 A 初始，当 A 读到 1 的时候，NFA 中去往 dead state，但是在 DFA 中，任意状态都必须针对字符串的 symbol 有 transition function，所以这里我们需要创造一个新的状态，也就是 DFA 表格中的 C。当我们创造了状态 C，必然就有从 C 出去的 arrows，一个是读到 0 之后，另一个是读到 1 之后。而显然，一个 dead state 读到 symbol 时候，只能指向自己。所以这个表就做出来了。再根据这个图画出对应的 DFA 图。

这个简单的例子如果讲到这里，很多同学可能觉得已经明白了如何转化 DFA。但其实，里面有很多小细节我们都没讲到，错过的话以后画 DFA 会有很大几率发生错误。哪些细节呢？首先，当我们画 DFA 的表时，要按照顺序来生成状态。首先是从初始状态 A 进入，这个没问题，当在 A 读到 0，1 之后会跳转到 B，C。这时候，你的 DFA 里面有了三个状态：A，B，C。接下来你可以为 B 写出后面的跳转状态情况，再为 C 写。如果你发现，在 B 或者 C，读到 0 或者 1 的时候跳转到一个新的状态 D，意味着你的 DFA 里面又引入了新的状态，那么你还要为状态 D 写 transition function，直到整个系统不再引入新的状态（闭包）。这个很好理解。但是难理解的是，当“现有的表”中没有出现某个状态，是不能为其写跳转情况的，哪怕状态曾经出现在 NFA 的表格中。看下面这个例子：

让 NFA 和 DFA 分别 recognize 一个 language L = {set of all strings over (0, 1) that ends with 1}：



上图首先画了 NFA。主要关注 NFA 的表格转到 DFA 的表格：初始状态依然是 A，读到 0 的时候，跳转到 A；读到 1 的时候，跳转到 A 和 B 两个状态。而在 DFA 中，当在某个状态读到某个 symbol 的时候，只能跳转到某一个状态，那怎么办？解决方法是将 A、B 两个状态合并为一个状态，称为 AB。至此，DFA 表格中只有初始状态 A 和读到 1 跳转到的状态 AB，没有状态 B（NFA 里面才有状态 B）。接下来要为状态 AB 设计 transition function。在状态 AB 读到 symbol 0，该去往哪个状态呢？应该是 NFA 里面在 A 状态读到 0 去往的状态和在 B 状态读到 0 去往的状态的并集，也就是{A}∪∅。同样地，在 AB 读到 1 去往的状态也应该是 NFA 里面状态 A 和 B 读到 1 去往的状态的并集，也就是{A, B}∪∅。而状态 A, B 已经合并为 AB，所以我们得到该 DFA 的表格。DFA 的表格里面不再有状态 B。

NFA 转换 DFA 的思想显而易见，就是从起始状态开始，将读到同一个 symbol 而去往的状态“合并”为一个状态，并将这些被合并状态上的“发生的故事”继续合并，也就是读到某个 symbol 而发生的行为取并集，直到终止状态。从状态数量上看，是 NFA 存在优势，因为如上面所分析，最多需要 2^k 个状态才能画出对应的 DFA，k 是 NFA 的状态数量。而在计算时间上，恰恰相反，DFA 占据了明显优势，因为它的路径是确定的。

接下来，我们使用 FSA（finite state automata）来指代 DFA 和 NFA。每一个计算模型都对应着它所能处理的语言，FSA 也是一样。FSA 对应的语言称为 regular language。其字符表、语法、语义我们不做讲解，可以自己私底下去了解（非本门课程内容）。这里一定会有一个疑问：这个 regular language 有没有什么特点或者叫边界，可以帮助我们确定一个语言是不是 regular language？又或者是不是所有的语言都是 regular language？答案是否定的。不然的话，FSA 岂不是取代了图灵机成为了通用计算模型？根据上面的分析，我们首先可以得到一些重要的定理或者引理：

（1）A language is regular iff some regular expression describes it.

（2）A language is described by a regular expression iff it is regular.

上面两句话不是一回事，第一句说一个语言是 regular 的，则一定存在一个 regular expression 描述它；反过来也一样。暗示着 regular expression 的表达力比所有 regular language 的集合的表达力要大。第二句才说明了二者的等价性。

要判断一个语言是不是 regular language，我们引入一个 pumping lemma 的概念。这里不再使用模型论的 bisimulation，因为 regular language 的语义解释不是基于模型论的。

THEOREM 1.70

Pumping lemma If A is a regular language, then there is a number p (the pumping length) where if s is any string in A of length at least p, then s may be divided into three pieces, $s = xyz$, satisfying the following conditions:

1. for each $i \geq 0$, $xy^iz \in A$,
2. $|y| > 0$, and
3. $|xy| \leq p$.

Recall the notation where $|s|$ represents the length of string s, y^i means that i copies of y are concatenated together, and y^0 equals ε.

如果一个语言 A 是 regular language，则存在一个数字 p，称为 pumping length，使得 A 中任意长度不短于 p 的 string s（|s|≥p），可以分为三个部分 s=xyz，并且进一步满足以上三个条件。以上定义用数学语言描述 pumping lemma，那该如何从现实世界的角度理解呢?

（1）如果一个语言 A 是 regular language，则一定存在一个 FSA 可以 recognize A，则 A 中的任意 string 都可以被这个 FSA 所 accepts。

（2）FSA 的状态是有限的，而 A 中的 string 是无限的。如果 FSA 的状态数量是 p，则 A 中必然存在一些 strings，它们的长度，也就是符号的数量大于 p。根据 pigeon hole principle 的结论，5 个物体放入 4 个盒子一定至少存在至少一个盒子里面的物体≥2。则同样的道理，如果 FSA accepts 长度大于 p 的 strings，则必然至少存在一个 state 被访问了超过一次，比如下列的计算过程（一组状态被访问的记录）：

$$q_1, q_2, q_3, \cdots, q_k, \cdots, q_k, \cdots, q_{n-1}, q_n$$

这就意味着必然存在一个 substring 会被 $q_k, \cdots, q_k$ 读到。这个 substring 就是 lemma 里面的 y，y 之前的 substring 就是 x，y 之后的是 z。显然，y 可以被多次重复（pumped），比如 xz，xyz，xyyz，xyyyz，其产生的新的 string 依然在 A 里面。

（3）Pumping Lemma 中条件（2）要求 y 这个字符串的长度至少是 1。没有这个条件，整个 lemma 就会无意义为真，我们叫 trivially/vacuously true，因为没有 pumped 的字符串则 xy^iz 变为了 xz，也就没有了变量部分，s = xz。则 Pumping Lemma 自动成立。

（4）Pumping Lemma 中条件（3）要求 substring y 起始的地方一定要在 p 范围之内。因为第一次开始 pumped 的时候一定是 p 之内的。不然会误伤很多本来是 regular language 的语言。

Pumping Lemma 的发明者是 1976 年图灵奖得主 Michael O. Rabin 和 Dana Scott，两位都是 Alonzo Church 的博士学生，Alan Turing 的师弟。

这里注意：pumping lemma 只能证明一个语言不是 regular language，但是不能证明一个语言是 regular language（只负责找茬，不负责善后）。下面步骤证明一个语言 A 不是 regular 的：

（1）假设 A 是 regular
（2）则一定存在一个 pumping length p
（3）所有长度大于 p 的 strings 都可以压缩（pumped），|s|≥p
（4）现在找到一个 string s in A，满足|s|≥p
（5）将 s 分为三部分 s=xyz
（6）证明 xy^iz 不属于 A，for some i
（7）接下来考虑所有 S 分为 xyz 的情况（有哪些分法？）
（8）证明任何一种分法都不可能满足上述 pumping 的三个条件
（9）得出结论：s 不能被 pumped（与（3）矛盾）

接下来看一个使用 pumping lemma 证明不是 regular language 的例子：$A = \{0^k1^k \mid n \geqslant 0\}$
假设 A 是 regular 的，则存在一个 pumping length p
$s = xyz = a^pb^p = aaaaaaabbbbbbb$ $(p = 7)$
Case 1：y 在 a 部分里面，比如 aa aaaa abbbbbbb
Case 2：y 在 b 部分里面，比如 aaaaaaab bbbbb b
Case 3：y 在 a 和 b 部分里面，比如 aaaaa aabb bbbbb
$xy^iz = xy^2z$，也就是让 i = 2。你会发现，无论是上面哪种 case，都不可能。

之所以跳出这门课内容范围之外，介绍几种计算模型，包括 pumping lemma，是想说明，每一种计算模型都对应着其可计算模型（某种语言）。对于可计算语言，我们有各种方法判断其表达力强弱，侧面反应一个计算模型的“计算能力”强弱。在基于模型论解释的逻辑语言中，我们求助于 bisimulation；而在 regular language 上，我们求助于 pumping lemma。

接下来，我们要介绍的是一种新的计算模型，叫做 pushdown automata。介绍它之前，我们先看它对应的语言，叫 Context Free Language，简称 CFL。

CFL：a language generated by some context free grammar (CFG)；CFG 正式定义如下：

DEFINITION 2.2

A ***context-free grammar*** is a 4-tuple (V, Σ, R, S), where

1. V is a finite set called the ***variables***,
2. Σ is a finite set, disjoint from V, called the ***terminals***,
3. R is a finite set of ***rules***, with each rule being a variable and a string of variables and terminals, and
4. $S \in V$ is the start variable.

一个 4-tuple。其中，V 中的 variables 同时也叫 non-terminals 或者 non-terminal symbols，所以 V 需要与Σ是 disjoint 的关系。说的比较模糊的是 R，也就是 a finite set of rules，我们通常把这些 rules 称为 production rules。它的形式是：

$$A \to a$$

其中，$A \in V$ 且 $a = \{V \cup \Sigma\}^*$。举一个例子，依然是之前熟悉的例子，regular language 的处理不了那个例子 a^nb^n。用 CFG 定义结果如下：

$$G = \{V, \Sigma, R, S\} = \{(S, A), (a, b), (S \to aAb, A \to aAb \mid \epsilon), S\}$$

观察，S 是我们的初始变量，所以规则中第一个是 S 起始的：

$$\begin{aligned} S &\to aAb \\ &\to aaAbb \text{ (by } A \to aAb) \\ &\to aaaAbbb \text{ (by } A \to aAb) \\ &\to aaabbb \text{ (by } A \to \epsilon) \\ &\to a^3b^3 \end{aligned}$$

CFL 比起 regular language 表达力更强。同样的方式，我们可以判断一个语言不是 CFL，用的是专为 CFL 涉及的 Pumping Lemma，这里不再赘述，感兴趣的可以去阅读相关资料，难度不大。现在转回 pushdown automata（PDA）的介绍，它有这样几个特点：

（1）it is more powerful than FSA
（2）FSA has a very limited memory but PDA has more memory
（3）PDA = FSA + a stack（with infinite size）

stack: an abstract data type that serves as a collection of elements, with two main principal operations:

上面图里就是一个典型的 stack，一个“抽象的”数据存储模型。像一个个的小格子，每个格子里面存储一个 symbol，采用“后进先出”的方式 last in first out (LIFO)进行读写操作。这里，a 是第一个进入到 stack 的元素，接下来是 b，直到 e 最后一个进入。

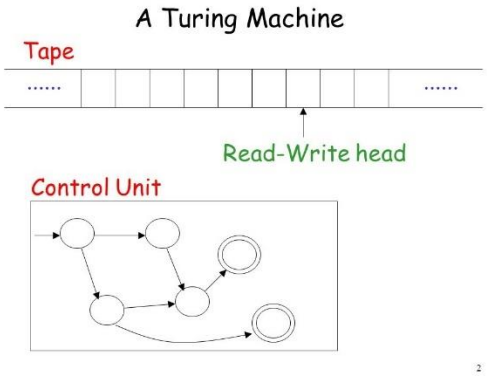
push (down): adds an element into the stack
pop (up): removes the topmost element from the stack

push 的操作是“写”的操作，将某个 symbol 写入 stack 并存储在 stack 最上面一个格子；pop 的操作是“读”的操作，读到 stack 中最上面的那个 element，并将它从 stack 中弹出去。所以，pushdown automata 是一个能在 input string 上进行“读”操作，这点与 FSA 一样；但还可以在 stack 上进行“读”和“写”双操作，这是比 FSA 厉害的地方。我们用现实生活中的其他例子来比喻下这种“只读”和“可读可写”方式带来的不同。我们在学习的时候，有的人喜欢边学边记笔记，无论是记到纸质还是电子文件，以便随时调出来查阅；有的人对自己的记忆力非常自信，觉得“读到了”和“理解了”就是“写下了”和“记住了”且“不会忘了”。对于后者，事实证明，很可能是一个陷阱（pitfall），一个记忆陷阱。大脑的 memory 是有限的，记忆的东西增多，很多以前记下的东西会按照先进先出（first in first out, FIFO）的方式被遗弃掉。所以当你可以执行 PDA 操作时，不要只做 FSA 的事情，莫名放弃掉 stack 的作用。要找到一个类似于 stack 可以永久存储的地方，比如笔记本或者各种 Pad，将知识保护好。尤其是那些需要二次加工才能产生的知识。很多知识是你随时拿起课本看一下就能理解的，而还有很多是需要很长时间深入思考进去才能在书本上的浅层知识或者基础知识上衍生出来的“深层知识”，也就是所谓的“理解”。允许在某个地方进行写的操作相当于给了一个机器记忆的能力。针对之前的例子：a^nb^n。它可以记住读到了多少个 a，然后通过 pop a 的方式找到同样数量的 b。

PDA 相当给了人们一个可以无限存储的地方，但存储的方式非常受限：只能按照顺序一个一个写和读，换句话说，PDA 的所有操作都只能在 stack 的最顶上的元素上进行，很像是自动售货机，只在每个 tray 的最前面那个物体上进行操作。而一个 stack 不能记住那么多东西，或者同时记住一些东西。于是在经典 PDA（也就是 FSA + 1 stack）的基础上，扩展为 FSA + 2 stacks，或者更多。它们的计算能力不同。我们在作业中，要求证明 PDA + 2 stacks is more powerful than PDA + 1 stack，然后证明 PDA + 3 stacks is not more powerful than PDA + 2 stacks。给一个提示，比如为什么本体里面只需要有二元关系，不需要三元关系？是不是所有的 3 stack PDA 可以转换为 2 stack PDA？而 2 stack PDA 不可以转化为 1 stack PDA？依然是上面的例子，如果字符串变为 $a^nb^nc^n$，1 stack PDA 能否处理？2 stack PDA 呢？如果变为 $a^nb^nc^nd^n$，2 stack PDA 能否处理？3 stack PDA 呢？

这样的模型依然满足不了更复杂的计算需要，或者对应表达力更强的语言。于是还有新的

计算模型，比如 Linear Bounded Automata（LBA），比如在 PDA 的基础上有 Turing Machine（TM）。我们接下来介绍下 TM，这是最重要的计算模型，也是计算的通用模型。一台计算机做的任何任务，TM 都可以做。TM 做不了的事情，说明这个事情是计算不可行的（computationally infeasible）或者叫不可计算的（not computable or uncomputable）。针对于某个语言 L，如果一个 TM 不能 recognizes it，则称为 undecidable。



看上面的图，TM 有一个 tape，长度无限（tape 左边有界，右边没有）；tape 被分为一个个离散的小格子，称为 cell，cell 上可以存储一个 symbol；有一个 tape head，可以在这个 tape 上左右移动，且每次只能移动一步；head 可以读 cell 上的 symbol，还可以将其擦掉，写上另一个 symbol；tape 上有几个位置是 accept cells，有几个位置是 reject cells。就这样一个简单的东西，就是 TM，它的计算过程是：input string 会从往右停留在 tape 上等待 TM 检阅，其它 cell 都用空白符号填充。在执行完所有操作后，head 停留在了 accept cell 的位置，则 TM accepts 这个 string；如果停留在 reject cell 位置，则 TM rejects 该 string；还有一种情况，TM 进入了一个 infinite loop，在这个 loop 里面永远走不出来，所以结果是 never halting。这是 TM 的三种终止形式。我们把上面的语言描述用数学语言定义如下：

DEFINITION 3.3

A ***Turing machine*** is a 7-tuple, $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, where Q, Σ, Γ are all finite sets and

1. Q is the set of states,
2. Σ is the input alphabet not containing the ***blank symbol*** $\sqcup$,
3. Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$,
4. $\delta: Q \times \Gamma \longrightarrow Q \times \Gamma \times \{\text{L}, \text{R}\}$ is the transition function,
5. $q_0 \in Q$ is the start state,
6. $q_{\text{accept}} \in Q$ is the accept state, and
7. $q_{\text{reject}} \in Q$ is the reject state, where $q_{\text{reject}} \neq q_{\text{accept}}$.

上图中清楚地定义了一个 standard TM 应该是什么样子的。与其他计算模型不同的地方：

2. Σ规定的是 input string 可以使用哪些 symbols（tape head 可以读到哪些 symbols），这里面不包括 blank symbol，这里我们用字母 b 来表示 blank symbol；

3. Γ规定的是 tape string 可以使用哪些 symbols，这里面包括了 blank symbol，所以能够写到 tape 上的 symbols 应该是既可以来自Σ，也可以是 b；

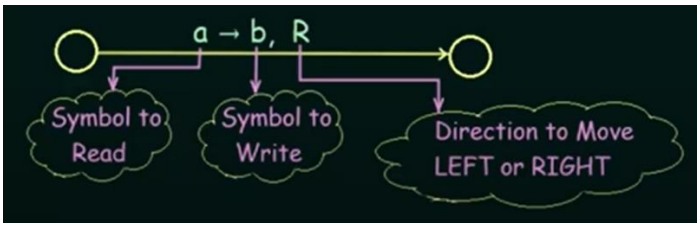
4. transition function 是最最最重要的，它的输入是一个 2-tuple，包括（1）当前所在状态，也就是当前所在的 cell，以及（2）读到的 symbol；输出是一个 3-tuple，包括（1）接下来要去到的状态，（2）当前状态上写入的 symbol（注意注意这里是写入到当前状态，不是下一个状态，{L, R}指定到了下一个状态后，head 往左移动还是往右移动。

*所有的空的 cells 都会自动填充上 b，而不是就空置着

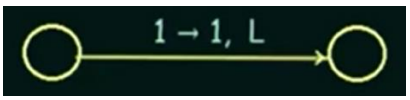
*既然Σ里面不包含 b，则第一次遇到 b 的时候意味着字符串的终结

*因为 tape 左边是有界的，当 head 已经在最左边的 cell 上，但是又接到了向左移动的指令怎么办？答案是继续停留在当前的 cell，不做移动。TM 的计算除非到了 accept 或者 reject state，不然会一直持续下去；换句话说，TM 不可以在非终止节点终止。

上面是 TM 的基本元素的介绍，下面我们看一下 TM 的计算过程：



（1）tape head 读所在 cell 的 symbol（图中左边 node）
（2）tape head 在当前 cell 写入新的 symbol（图中，a 是当前 cell 读到的 symbol，b 是写入到当前 cell 的 symbol，R 是移动方向，这个 R 是在上一个状态的 transition function 就决定好了的）
（3）如果当前 cell 的 symbol 不变，则依然做一步写入操作，只不过用当前 cell 的 symbol 代替它自己，见下图用 symbol 1 代替 1。



Standard TM 称为标准 TM 或者经典 TM，只有一条 tape。但是实际还有很多 TM 的变种，比如多个读写头的 TM，比如多个 tapes 的 TM，比如 tape 左边也无边界的 TM 等等，再比如 non-deterministic TM，但不管是什么样的变种，都可以 simulated by standard TM。TM 在 1936 年被 Alan Turing 发明，TM 这个名字是他的老师 Alonzo Church 起的，并且 Church 独立发明了 lambda-calculus，它与 TM 具有相同的计算能力。二者追随哥德尔的工作，形成了一个伟大的开创性的工作，它是计算的基础，称为 Church-Turing thesis。用直白的话解释它就是：任何计算行为，计算任务，都可以被转化为 TM 上的计算问题；反过来，凡是不能被 TM 计算的问题，就是问题本身不可计算，不存在一个更强大的计算模型，使得该问题可以被这个模型所解决，但不能被 TM 所解决。

我们看一个具体的例子 $L = \{0^n1^n2^n \mid n \geq 1\}$。这个例子能被 1 stack PDA 解决吗？不能吧？我猜是不能，但你们在作业中要证明。看看它怎么被 TM 完美解决。这个语言收集了所有 strings，满足 string s = xyz，其中 x 部分是 n 个 0，y 部分是 n 个 1，z 部分是 n 个 2。我说下 TM 的思路，首先某个字符串比如 s = 001122 会进入到 tape 上等待 tape head 读写。tape head 读到第一个 0，在这里将其擦掉，写入 X（用 X 替换 0），接着向右移动，读到第二个 0，写入 0（本状态上 symbol 保持不变），继续向右移动，读到第一个 1，写入 Y，向右移动，读到第二个 1，写入 1，继续向右移动，读到第一个 2，写入 Z，然后向左移动，每次移动一步，直到遇到 X（撞墙了），向右移动，如果再读到 0，改写为 X，然后重复上述动作。在这个过程中，你会遇到下面几种情况：
（1）最后 head 再一次读到 X，按照上面方式向右移动，发现下一步读到是 Y，意味着所有的 0 都被替换为了 X，那么后面就不该再有任何 1 或者 2，如果有，则 reject；没有，且没有任何其它非 XYZ symbol，则来到了 b 状态，accept；
（2）head 在将某一个 0 改为 X 后，向右移动，发现后面不再有 1，则 reject；
（3）同样的道理将某一个 1 改为 Y 后，向右移动，发现后面不再有 2，则 reject；

（4）如果期间遇到任何 021XYZ 之外的Σ-symbol，则 reject；

如上文所提到的，TM 判断一个 string 是否可计算，有三种结果。其中两种是 halt，一种是 not halt。而 halt 里面，又分为 accept 和 reject。如果一个语言 L，TM 对 L 有可能返回三种结果，则我们说 L 是 Turing-recognizable；TM 对 L 返回的只可能是 halt 的两种结果，则说 L 是 Turing-decidable。有没有语言是 Turing recognizable 但不是 Turing-Decidable？有，比如 first-order logic。有没有语言甚至不是 Turing recognizable？有，但例子不好举。

TM 是后面用于分析一个算法复杂度的默认使用的计算模型。比如给定一个语言，由无数个属于该语言的 strings 组成。比如 L = $\{0^n1^n2^n \mid n \geq 1\}$是 L 的内涵式定义，而 012、001122 这些是它的外延式定义。TM 接收某个 string 会有一个长度 n，则在该语言 decidable 的情况下，会判断 TM 需要多少时间和空间来确定 accept 还是 reject 该 string。请记住：某个语言要不是 decidable，要不是 undecidable。如果是 undecidable，则不再分析其复杂度，因为语言是不可决定的，计算可能没有终点，分析到可计算性这里就结束了（computational theory）；只有先确定是 decidable 的，才能进行复杂度分析（complexity theory）。

首先是时间复杂度分析，比如给定一个 string s 长度|s| = n，需要用多长时间 TM 才能判断 accept 还是 reject s？而且这个结果一定与 n 有关系，也就是说，我们会有一个函数，这个函数的 input 是 n，output 是一个与 n 有关的数字，比如 f(n)，我们需要 f(n)这么久的时间才能完成这个任务。我们可以得到一个非常精准的结果么？举个例子，判断 0011222 和判断 0101222 能否属于 L = $\{0^n1^n2^n \mid n \geq 1\}$。很明显，前者需要按照上述方法，知道最后一个 2 才可以判断不属于 L，而 0101222 到第二个 0 给出 reject 的结果。可见，同样是长度为 7 的 string，判断的时间也不一样，所以我们无法得到一个针对 input n 的精确结果，而是一个最坏结果。最差情况下，多久可以得到 accept 或者 reject 的结果？采用的是 worst-case analysis 的方式。另外，我们一直用 string 来作为 TM 的输入，用 string 的长度来作为时间复杂度分析的输入。其实很多情况下，我们的数据结构不一定是传统的 string，还可以是比如一个图，那么这时候，我们需要多少 steps 去完成一个任务，可能取决于这个图的节点数量，边的数量，或者这个图的 outdegree 的数量，又或者是上述这些参数的某些组合。

还有，当 n 是 string 的长度，我们得到一个针对 n 的时间复杂度 $f(n) = 6n^3 + 2n^2 + 20n + 45$，我们只考虑这个多项式的最高项（the highest order term），且系数（coefficient）也会被忽略掉。这是因为我们采取一种渐进式（asymptotic notation）的分析方法，称 f is asymptotically at most n^3，也叫 big-O notation，写作 $f(n) = O(n^3)$。这是因为，随着 n 的增大，f(n)的取值将最大程度，或者叫被 n^3 所 dominant。我们只抓重点，不看那些影响较小的部分。

我们用这种方法简单分析下 decides L = $\{0^n1^n \mid n \geq 1\}$的时间复杂度：
TM = “on input string w”：
（1）首先会读这个 tape，如果发现 0 出现在 1 的右边，则 reject；
这一步最坏的情况是直到最后一步，才发现 0 出现在 1 的右边，比如 0001110，所以最坏情况是对整个 string 进行遍历，需要 n steps，最后把 head 归位到最左边的 cell 又需要 n steps，一共需要 2n steps。因为采用 big-O notation 分析法，这一步需要 O(n) steps。凡是这种只影响系数的操作过程，我们都会忽略掉（omit）。
（2）再去读这个 tape，读到一个 0，换为 X，一直向右移动，读到一个 1，换为 Y，一直向左移动，遇到 X，向右移动，读到一个 0…repeat

这一步是遍历整个 tape，并且会替换掉一个 0 和一个 1，也就是一次 scan 替换 2 个元素，那么最多我们会 scan n/2 次，假设每次需要走 O(n) steps（事实是越到后面，走的 steps 越少，不需要走全部的 n steps，但是别忘了是 big-O notation 分析法），所以我们最多在这一步会走 $n/2 * O(n) = O(n) * O(n) = O(n^2)$。

（3）最后一步，查看当按照 0 与 1 一对一兑子之后，是否还剩下 0 或者是否还剩下 1，如果是，则 reject；如果 tape 上只有 X 和 Y，则 accept。

这一步又需要 scan 一次整个 tape，需要 O(n) steps。

所以完成整个 decide 任务一共需要 $O(n) + O(n^2) + O(n)$ steps，按照 big-O notation 分析法，decide 这个 L 的时间复杂度是 $O(n^2)$。

针对这个问题，有没有更好的算法使得更快地 decides L，答案是肯定的。你可以按照参考书 page 280 的算法得到一个 O(n logn)的 bound，甚至在 multi-tape TM 上得到一个 O(n)的结果。我们甚至得到更 general 的结论（Theorem 7.8）：

THEOREM 7.8

Let $t(n)$ be a function, where $t(n) \geq n$. Then every $t(n)$ time multitape Turing machine has an equivalent $O(t^2(n))$ time single-tape Turing machine.

可以看到使用 multi-tape TM 比 single-tape TM 要在时间上，有优势，优势是从多项式时间到线性时间，对于人来说，够大么？够大。对于计算机呢？不算大。所以使用 multi-tape 还是 single-tape TM 并不影响大局。那么下面这个呢？

THEOREM 7.11

Let $t(n)$ be a function, where $t(n) \geq n$. Then every $t(n)$ time nondeterministic single-tape Turing machine has an equivalent $2^{O(t(n))}$ time deterministic single-tape Turing machine.

可见，一个 non-deterministic TM 有着至少一个指数量级的时间优势。计算机在意么？答案是在意。说明使用 non-deterministic TM 还是 deterministic TM 区别非常大，所以在做时间复杂度的时候，一定要说明使用的 TM 是哪一种，如果不说，默认为经典 TM。对于计算机来说，一个重要的界限就在于这里：多项式时间还是指数时间。如果证明了决定某个 L 最坏情况下需要 $O(n^3)$，还是 $O(n^6)$，其实本质上并无量级的差别，当你找到了一个多项式时间的 bound，就不需要去寻找一个 linear 的 bound，尽管 linear 的 bound 当然比 polynomial 的更紧，时间上更漂亮，但是对于真实计算机的计算速度来说区别不大。但是一旦突破了指数界限，那就不能任性了，如果你可以找到两倍指数的 bound，就千万别停留在三倍指数，就像你要是一个亿万富豪，可以肆无忌惮的消费；但是如果你是一个朝九晚五的打工仔，过日子就得精打细算，一个道理。如果你能为一个目前指数时间复杂度算法才能解决的问题找到一个多项式的算法，那绝对值得发表一篇漂亮的 paper。我们做这门课的时间复杂度分析，停留在这条界线上足够，不需要找到更精确的 bound。但是到了 exponential 的界限，就需要精打细算，到底是 $O(2^n)$ 还是 $O(2^{2^{n}})$ 区别很大。

对于一个计算问题来说目前已知的最好算法时间复杂度是 $O(n^3)$，则它是一个在 polynomial 时间内可以解决的问题，我们把这一类问题称为 P 问题，是计算问题里面最为简单的问题。

DEFINITION 7.12

P is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine. In other words,

$$\mathrm{P} = \bigcup_k \mathrm{TIME}(n^k).$$

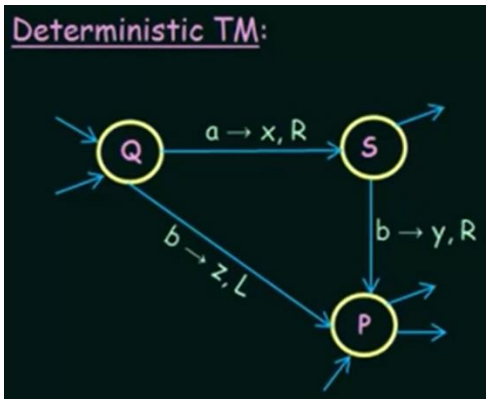
这里，polynomial time 指的是 the number of steps the algorithm has to perform to solve the problem is in the order of a polynomial in the input size。见上图，无论 k 赋值于几，本质上都是 P 问题，不会因为指数位置上数字变大，就会有什么不同。当然，也不能太过分，比如 n^{100} 结果大不大，当然大，但是实际生活中，什么时候才能遇到 for 循环 100 次的情况呢？我们把 P 问题称为 realistically solvable 的问题。

现实生活中典型的 P 问题有很多，包括 PATH 问题，RELPRIME 问题，另外，decides CFL 语言也属于 P 问题。

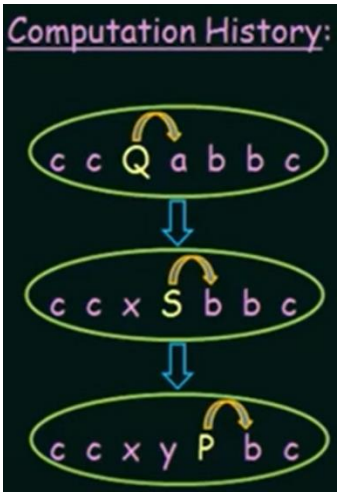
除了 P 问题，还有一种问题类型称为 NP 问题。这一类问题也比较好理解，它指的是能在 polynomial time 内由 non-deterministic TM 决定的问题。在我们的作业中，涉及到了关于使用 non-deterministic TM 的问题，这里我们详细讲一下它，与 deterministic TM 区别在于：

$$\delta: Q \times \Gamma \longrightarrow \mathcal{P}(Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\}).$$

Deterministic TM 的 transition function 可以去到(Q, Γ, {L, R})，而在 non-deterministic TM 里，变为了它的 powerset，P(Q, Γ, {L, R})。依然难以理解，看一个例子：

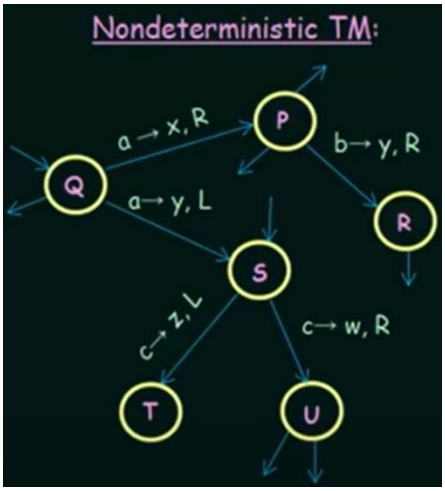


可以看到这是一个 deterministic TM 的片段，我们依然用传统的 state 呈现形式来代替 TM 的 tape，这样显得更加直观。将上述的 states Q S P 想象成 tape 的 cell:

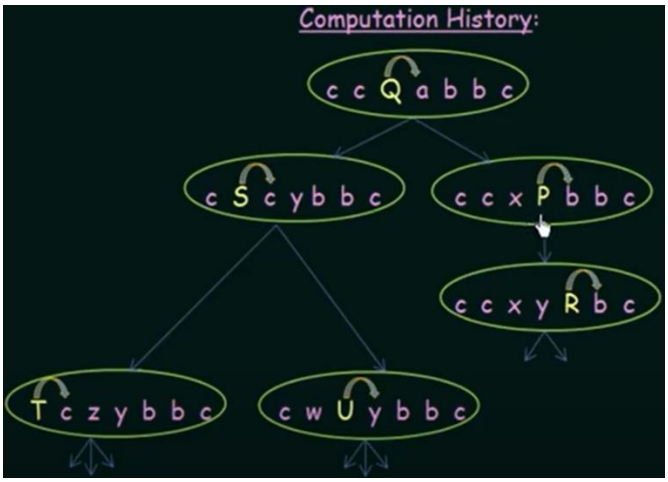


这是其中的计算过程，每个椭圆成为一个 configuration，每个 configuration 记录了当前状态，当前 tape 的内容，以及当前的 tape head 在哪里。具体表示为 substring1 Q substring2，其中 Q 表示当前所处的状态，substring1 和 substring2 表示 Q 状态前的 string 和 Q 状态后 string，而 Q 状态后的 substring2 的第一个 symbol 就是 head 现在读到的 symbol，比如 abcQedf 表示

当前状态为 Q，之前已经读过了 abc，目前 head 读到了 e。按照这样的解释，上图表示的计算过程就是目前来到了 Q 状态，读到了 symbol a，并将 a 这个 symbol 改写为 x，转移到下一个状态 S，在 S 状态读到了 symbol b，并将 b 改写为 y，转移到状态 P，在 P 状态读到了 symbol b。在这样一段过程中，deterministic TM 无论来到哪一个状态，在这个状态读到某一个 symbol 之后都只能有唯一的下一个“configuration”：（1）将当前状态改写为什么 symbol；（2）走向哪一个状态；（3）接下来去往哪个方向，左还是右？而在一个 non-deterministic 中情况不一样，当来到某个状态，在这个状态上读到某个 symbol，可能面临多种选择：at each moment in the computation there can be more than one successor configuration。



比如上图，在 Q 状态读到 symbol a，可以将当前 symbol a 改写为 x，去往 P 状态，并在 P 状态上向右移动；还可以将当前 symbol a 改写为 y，去往 S 状态，并在 S 状态上向左移动。相当于在每一步给了我们更多选择的自由，计算的效率增高了（计算过程中走的 steps 整体上变少了）。如果将这个计算过程用和刚才一样的 configuration 的形式表示出来：



可以看出，形状变为了树状结构。按照其结构的特点，如果任何一个 branch 达到了 accept 状态，则 non-deterministic TM accepts the input string；否则当所有的 branches 都 reject，则该 TM rejects the input string。很明显，如果要“验证”（verify）某个 branch 是否被该 TM accepts 是不是就是一个 deterministic 的路径？所以可以在 polynomial time 内被一个 deterministic TM 所解决，但只是“验证”，不是“解决”。一个典型的问题是 traveling salesman problem，it is possible for a postman to visit all cities while traveling at most k distance。可否在 polynomial time 找出所有这样的路径？不知道。但是很明显，如果给定一条路径，去验证这条路径是否满足 distance 的要求可以在 polynomial time 解决。这就是注明的 P＝NP？问题：一个 polynomial

time 内可验证的问题是否也可以在 polynomial time 内可解决。

Non-deterministic TM 并没有比 deterministic TM 增加了计算能力：

THEOREM 3.16

Every nondeterministic Turing machine has an equivalent deterministic Turing machine.

由此可以得出推论，一个语言 L 是 Turing-recognizable iff 有一些 non-deterministic TM 可以 recognizes L；一个语言 L 是 Turing-decidable iff 有一些 non-deterministic TM 可以 decides L。

根据上述 Theorem 7.11，我们知道一个到目前为止确定的结论：一台 non-deterministic TM N 可以 simulated by 另一台 deterministic TM T。并且存在一个 constant c，使得，对于任意 input string s，如果 N 可以在 t steps 内 accepts s，则 T 最多可以在 c^t steps 内 accepts s。下面是分析的过程，最坏情况下，N 可能会不得不在任何一个状态上进行 non-deterministic 的选择，这就会产生一个 computation tree，像是一个完整的 binary tree of depth t(n)，其中 n 的 s 的长度。这个 tree 会产生 $2^{t(n)}$个不同的 branches，每一个 branch 都是一个 valid computation of N。那么用 T 去 simulate 整个计算过程需要 depth * |branches|：

$$t(n) * 2^{t(n)} = 2^{t(n) + \log 2 t(n)} \leqslant c^{t(n)}$$

for some sufficiently large constant c > 2。没有人知道是否有更好的 T simulates N 的办法，让这个计算过程更加简单，steps 更少！现在关于复杂度类之间最好的结论是：

$$\mathrm{NP} \subseteq \mathrm{EXPTIME} = \bigcup_k \mathrm{TIME}(2^{n^k}),$$

NP 问题是可以被一个 deterministic TM 在 EXPTIME 内解决的。NP 与 EXPTIME 之间有没有其它的类可以插入进去？不知道！如果知道，可以申请 100 万美元奖金。

接下来我们引入 NP-complete 和 NP-hard 这两个概念，来帮助我们更好的划分复杂度的类。除了 P 问题和 NP 问题，我们还有更多的类。这些类依然是围绕 NP 提问展开的。上世界 70 年代，一位美国科学家 Stephen Cook（1982 年图灵奖得主）和一位俄裔美国科学家 Leonid Levin（Donald E. Knuth Prize 得主），他们发现 NP 问题里面有一类子问题，它们的复杂度与整个 NP 问题类的复杂度有非常大的关系。换句话说，NP 问题里面的各个问题之间可能会存在某种联系。什么联系呢？这一类子问题非常有“代表性”，就是如果这些子问题能被一个 deterministic TM 在 polynomial time 内解决，则 NP 的其它问题也一定能被 deterministic TM 在 polynomial time 内解决。换句话说，这些子问题是 NP 问题里面最难的问题，称为 NP-complete 问题。他们为什么要去找到这样一类子问题呢？试想一下，如果此时我想证明 NP 问题都可以被 deterministic TM 在 polynomial time 内所解决，是不是需要证明每一个 NP 问题都能被 deterministic TM 在 polynomial time 所解决。问题是 NP 问题是无限的，这意味着这个证明永远都做不出来。因为理论上我们无法总结出所有的 NP 问题，并去一一证明。但是如果又这么一个 NP 问题，解决了它，则其它所有的 NP 问题都可以解决了，是不是就说明这个问题是“有代表性的”？第一个被证明是 NP-complete 的问题，也是被上述两位学者独立证明，是 Boolean satisfiability problem (简称 propositional logic 上的 SAT 问题)：

SAT: determine if a propositional formula can be made true by an appropriate assignment of truth values to its variables.

显然，如果验证一个 assignment 是否让某个命题逻辑表达式为真，相对简单；但是要直接

判断该表达式是否可满足（是否存在某个 assignment 使得其为真），用的最好的方法就是对所有可能的情况进行穷举（最坏情况下），查看每一种可能的取值。假如这个表达式包含 n 的 proposition symbol，则最多有 2^n 种 truth value 的可能性。这意味着只要为 SAT 问题找到一个deterministic TM 上的 polynomial time 的算法，那么就等于证明了 P = NP。可以总结为：

SAT belongs to P iff P = NP

Donald E. Knuth 教授大家都认识吧？TEX 打字系统的发明者，算法分析方面的引领者，著有《The Art of Computer Programming》一书，可以说在我们上学那个年代，无人不知，无人不晓。不知道现在 Python 流行的年代还是否这样。当时编程的两本启蒙教材，不是学校里面发的教科书（学校里面的教科书差点给我学抑郁了），而是《Thinking in Java》和《The Art of Computer Programming》。以他名字命名的 Donald E. Knuth Prize 主要是奖励在理论计算机方面的杰出个人，与哥德尔奖奖励的是杰出的工作（某一篇 paper）不同。1996 年设立该奖项后的第一位得主是姚期智教授，而最近的 2021 年得主是美国莱斯大学的 Moshe Vardi 教授，他是逻辑和知识表示与推理方面的顶级学者，著有《Reasoning About Knowledge》和《Finite Model Theory and Its Applications》，其中很多内容都是我们上课学到的，也是我博士导师在逻辑方面的领路人之一。看一下图灵奖得主 Stephen Cook 教授（照片来自维基百科，公开授权）：

年纪轻轻就做出了数学上非常漂亮的，计算理论底层的开拓性贡献。

接下来讲一个概念叫 reducibility，它的思想基础是：如果一个问题A可以reduced to 问题B，则问题 B 的解也可以拿来解决问题 A。比如我想考上南京大学，需要高考考 660+，则考上南京大学的问题（问题 A）就被 reduced to 高考考 660+的问题（问题 B），那么只需要去寻找高考考 660+的方法即可。解决了问题B，A 也就解决了。但是这个 reduced 的过程不都是计算上非常简单，我们定义一种转换，叫做 polynomial time reduction：

DEFINITION 7.28

A function $f: \Sigma^* \longrightarrow \Sigma^*$ is a ***polynomial time computable function*** if some polynomial time Turing machine M exists that halts with just $f(w)$ on its tape, when started on any input w.

首先定义一个转化函数，称为 polynomial time 可计算的函数，然后：

DEFINITION 7.29

Language A is ***polynomial time mapping reducible***,[1] or simply ***polynomial time reducible***, to language B, written $A \leq_P B$, if a polynomial time computable function $f: \Sigma^* \longrightarrow \Sigma^*$ exists, where for every w,

$$w \in A \iff f(w) \in B.$$

The function f is called the ***polynomial time reduction*** of A to B.

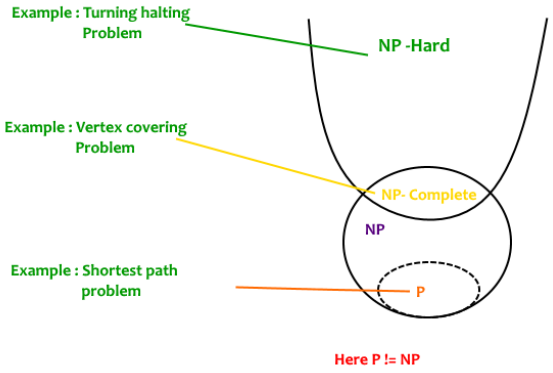
一个语言 A 到语言 B 之间存在一个 polynomial time 的转换函数 f，意味着对 A 中任意 string，都存在一个 f(w)属于 B。这个 f 就是 A 到 B 的 polynomial time reduction。至此，我们得到一个新的结论：

THEOREM 7.31

If $A \leq_P B$ and $B \in P$, then $A \in P$.

如果 AB 之间存在一个这样的 f，且 B 属于 P 问题类，则 A 也属于 P 问题类。

还有一类问题是 NP-hard 问题，它指的是任何 NP 问题都可以在 polynomial time 内转化为该类问题，则这类问题被称为 NP-hard 问题；如果刚刚好，这类问题也在 NP 类里面，则称为 NP-complete 问题。所以 NP-complete 问题是 NP-hard 问题的子集，同时也是 NP 问题的子集，用图表示的话如下：



想要证明一个问题 A 是 NP-hard，只需要找到一个已知的 NP-hard 问题 B，再证明存在一个 polynomial time reduction 使得 B 可以在 polynomial time 内 reduced 为 A：$A \leqslant_P B$。
想要证明一个问题 A 是 NP-complete，只需要找到一个已知的 NP-complete 问题 B，再证明存在一个 polynomial time reduction 使得 B 可以在 polynomial time 内 reduced 为 A：$A \leqslant_P B$。

上面是最简单的证明方法；否则按照定义，上述二者都需要我们去证明任何 NP 问题都可以在 polynomial time 内 reduced 为该问题，这一步我们很难做到。所以我们只能借助现有的东西，去做这个证明。正所谓站在爸爸的肩膀上，嘲笑爸爸矮。而如果想证明某个问题 A 属于 P 问题或者 NP 问题，只需要找到对应的 TM recognizes it 即可。

除了关注在时间复杂度，还有一个评价计算复杂度的方法就是用了多少的空间。这里面，很显然对于同一个问题，空间复杂度不会超过时间复杂度，比如完成一个任务需要 n steps，那么不可能完成这个任务需要大于 n 个 states（cells）；相反如果完成一个任务需要 n states，倒是有可能需要超过 n states 的数量，比如需要 n^3 steps 可能吗？可能。但是也不是无限的，没有边界，我们得到的结论是：

$$P \subseteq NP \subseteq PSPACE = NPSPACE \subseteq EXPTIME.$$

如果解决一个问题需要 TM 占用 tape cells 的长度是 f(n)，确实 TM 可以走超过 f(n)的步骤来完成整个计算步骤，但是它最多可以有 $f(n) * 2^{O(f(n))}$ 个不同的 configurations，证明请看计算理论参考书 Lemma 5.8 或者任意其它资料（建议阅读，因为重要）。

另外，在这里还要介绍一个重要的定义，关于上述 inclusions 排列中的 PSPACE = NPSPACE 这一结论。我们知道 PTIME = NPTIME 是百年悬而未决的问题，但是空间上早已被证明是等价的，证明者为 Stephen Cook 教授的学生，Walter John Savitch。他已经于 2021 年去世，

去世之前是美国加州大学，圣地亚哥分校的教授。这个结论叫做 Savitch's theorem。我们先用自然语言描述下这个定理：deterministic TM 可以使用非常小的增加的空间代价去 simualte non-deterministic TM。如果你还记得上面的定理，目前我们知道，deterministic TM 需要指数时间才可以 simulate non-deterministic TM 的计算行为，但是在空间上的结果却非常令人惊讶！具体说，就是 Savitch's theorem 证明了任意使用 f(n) space 的 non-deterministic TM 都可以等价转化为使用 $f^2(n)$ space 的 deterministic TM，数学表达式为：

THEOREM 8.5

Savitch's theorem For any[1] function $f: \mathcal{N} \longrightarrow \mathcal{R}^+$, where $f(n) \geq n$,

$$\text{NSPACE}(f(n)) \subseteq \text{SPACE}(f^2(n)).$$

又因为：the square of any polynomial is still a polynomial，我们完全可以安全地认为 PSPACE 类和 NPSPACE 是同一个类。所以在证明中，如果你使用了 non-deterministic TM 证明了某个问题是 f(n) space 可解的，换句话说是 NPSPACE 的，那么就等于证明了它是 PSPACE 可解的。

在空间上所有的 completeness 和 hardness 概念，以及其证明与时间上的一模一样。

接下来我们进入到参考书第五章的讲解，关于描述逻辑 ALC 及其 extensions 上的 complexity 的结果。回忆一下以前讲的知识，ALC 上的推理的核心问题在于判断一个 ALC concept C 针对一个 ALC TBox T 是否是可满足的：是否存在一个 T 的 model 使得 $C^{\mathcal{I}}$ 不为空。其他定义的推理问题可以直接转化为 concept satisfiability 的问题。寻找 complexity 结果的时候，TBox 被发现很重要，首先取决于有没有 TBox；如果有，其次取决于 TBox 是 cyclic 还是 Acyclic，我们在这里只讲解 with acyclic TBox 的情况（ayclic TBox 只包含 concept definitions，也就是说 equivalence 或者 inclusion 符号左边都是 concept names），并证明 concept satisfiability with acyclic TBox 是 PSpace-complete，也就是说这个问题是 PSpace 可解的（upper bound），且任何问题已知的 PSpace-hard 问题都可以在 polynomial time 内 reduced 为该问题（lower bound）。Upper bound 证明的是 concept satisfiability with acyclic TBox 是一个 PSpace 问题，lower bound 证明的是这个问题同时也是一个 PSpace-hard 问题，则二者共同使得该问题是 PSpace-complete 问题，是 PSpace 类里面最难的那一类问题，其它任何 PSpace 问题都可以在 polynomial time 内 reduced 为 concept satisfiability with acyclic TBox 问题。可能会有同学问为什么在 SPACE 环境下依然要求是 polynomial time reduction，而不是 polynomial space reduction？直观地解释，因为从问题 A 到问题 B 的 reduction 本身一定是计算实际可行的，PTIME 可以保证 reduction 可以顺利进行，而 PSpace 就不好说了，很可能需要 EXPTIME 才能完成。

首先证明 upper bound：

已知 ALC 具有 tree model property，这意味着，为了判断一个 ALC concept C 针对一个 ALC TBox T 是否是可满足的，可以尝试为其构建一个 tree model，并计算构建过程所需要付出的资源上的代价，比如 TM 需要多少 steps，比如需要多少 states（cells）。Tableau 恰好是帮助构建 tree model 的（在 Tableau 算法中，被构建的 tree model 就叫做 Tableau），而且回忆一下，之所以构建出的 model 是“典型”tree-shaped 的，肯定要归功于其中的$\sqcup$-rule，它带来了 non-determinism，所以这里可以使用 non-deterministic TM 来计算构建的成本（使用这种类型的 TM 意味着当你在任何 non-determinism 的地方需要选择时，都默认选择的是对的那条路径，这里也就是 clash-free 的那条路径）。且如果结果是 NPSPACE，则根据 Savitch's theorem，其结果是 PSPACE。另外还要明确一点，Tableau 是以整个 concept 为根节点进行

构建，只有遇到 exists 的时候才会发展叶节点。这样一来，一个 tree 的 outdegrees 的数量就 bounded by exists 的数量，而间接 bounded by C 和 T 的 size。当通过 exists 进行叶节点的构建的时候，最多需要一个 element，也就是一个 node 完成构建即可，所以其实对 element 的数量要求也是有限的。直观上看，这个数量肯定与 C，T 的 size 以及树的深度有关系。比如：

Consider the concept C_n defined inductively as follows;

$$\begin{aligned} C_1 &= \exists P.A \sqcap \exists P.\neg A \\ C_{i+1} &= \exists P.A \sqcap \exists P.\neg A \sqcap \forall P.C_i, \qquad \text{for } i \in \{1,\ldots,n\} \end{aligned}$$

Check the form of the interpretation induced by the completion graph obtained by starting from $C_n(x_0)$.

看这个例子，整个 TBox 是递归定义的。这个树构建出来应该是一个二叉树（binary tree），因为每个 concept C_i 都是由两个 exists 定义的，所以每个 node 下面需要创造两个叶节点，那么如果 C_n 这棵树构建出来，应该是深度为 n 层，那么可以知道构建这样一个 model 需要的 domain element 数量为 2^n-1（-1 是因为根节点只有一个）。

如果树的深度与 input 的 size 之间存在，深度是 polynomial size w.r.t. that of the input，且树的各个 branch 之间是相互独立的（每个 node 上的 label，也就是 concept，只取决于该 node 的 father node 和 child nodes，而不取决于其它 branches 上的 node），且 TM 采用深度优先搜索，则 non-deterministic TM 在某一时刻可以只存储一个 branch，而不需要考虑其它 branches。一旦树上面发生了回溯行为（在某个 branch 上遇到了 clash），则回溯到距离当前最近的发生 non-determinism 的 node，清空这条 branch 上的信息，存储另一个 branch 的信息。所以，任何时刻，non-deterministic TM 就只需要存储一个 branch 即可完成整个 Tableau 的构建工作。所以虽然上面例子显示，有的时候需要 exponential 个 nodes 才可以完成构建，但是同一时间只需要 polynomial 的 nodes。

如上所述，使用 Tableau 算法进行构建可以完成这个证明。参考书上，使用了一种从模态逻辑（modal logic）领域引入的新的算法来完成构建，叫做 K-worlds 的算法（K 是 modal logic 语言家族中的一个成员，对应的是 description logic 中的 ALC，所以称为 ALC-worlds 算法）。原因如它自己所说，更接近上述思想的本质。该算法有一些形式上，或者叫语法上的要求，需要满足（问题原本是判断任意 ALC concept C 针对任意 acyclic ALC TBox T，也就是说正常情况下 ALC-worlds 算法的输入是 C 和 T）：

（1）ALC-worlds 算法要求 C 必须是一个 concept name，而不能是 general concept。这个可以进行简单的转化：a complex concept C is satisfiable w.r.t. an acyclic TBox T iff a concept name A is satisfiable w.r.t. $T \cup \{A \equiv C\}$

（2）ALC-worlds 算法要求 T 中只能有 concept definitions，不能包含 primitive ones，也就是说只能是 equivalences，且每个 equivalence 的左边只能是 concept name（这一点 acyclic TBox 已经满足）。那么需要对 inclusions 进行转化，方法是对于每一个 inclusion $A \sqsubseteq C$ 转化为 $A \equiv C \sqcap A'$。该转化的正确性见参考书 Lemma 2.8。

（3）ALC-worlds 算法要求 the TBox T 一定处于 NNF，并且重新定义了 NNF，与我们之前 Tableau 算法中的 NNF 不完全一样。它要求 negation 只出现在 primitive concept names 前面。所谓 primitive concept names 就是在 T 中，只出现在 equivalence 右边的 concept names。如果把一个 equivalence 看成是一个 definition，则左边是被定义的 concept name（defined name），右边是用来定义 defined name 的 names，成为 defining name 或者 definer。如果一个 concept

name 只作 definer，从未被定义过，那么就是 primitive concept names，否则就是 defined one。关于 NNF 的转化方法在 Proposition 5.1 中也很清楚：对于 T 中任意的 defined name A，都为其准备一个“备胎” $\underline{A}$，一旦 A 出现在了 equivalence 右边，且 negation 符号出现在了 A 前面，则用这个备胎 $\underline{A}$ 去替换所有 equivalence 右边的¬A。在这期间，equivalence 右边的内容按照正常的 NNF 转化规则（Tableau 那里学的）进行转化（正确性需要证明：A is satisfiable w.r.t. T iff A is satisfiable w.r.t. T’）。

（4）在此基础上，ALC-worlds 算法要求 T 还需要是 simple 的，也就是说 equivalence 的右边只能是 role restriction 的单层嵌套，不可以多个嵌套。很明显，simple TBox 本身就是 in NNF。因为 negation 只出现在了 P 前面，而 P 是一个 primitive concept name。Lemma 5.2 进一步给出转化的规则，因为考虑的语言是 ALC，equivalence 右边可能的根节点有 and，or，exists，forall（没有 negation，因为是 NNF，negation 只可能出现在 primitive concept names 前面，这种情况就已经是 simple 了），针对每一个根节点类型，都有一个转化规则与之对应，思路就是用 concept name 进行替换。观察 simple TBox 不允许 defined name 单独出现在 equivalence 右边，则同样设计了第二个 bullet 的规则消除这种情况（正确性需要证明）。