软件测试

介绍

钮鑫涛

大纲

1. 测试 2. 输入构造 3. 预言 4. 反馈

为什么是这个主题?


同学们
我想做科研

为什么是这个主题?

- 对软件进行测试，是保障软件正确性的关键手段，甚至是现实世界中唯一的可行手段
  - 毕竟找到真正的bug比什么都有说服力！
- 软件测试是典型的入门简单（堪称无门槛），但深入却不容易
- 想要真正测好一个待测对象，其实没有一劳永逸的方法，你必须对待测对象有深刻的了解
  - 实际上能真正在SIGMOD、PLDI、OSDI、S&P等等top tier的会议期刊发表的测试文章，无一不是对待测对象（入数据库、编译器、内核...）有着非常深入的了解，从而得到一些不凡的洞见！
- 当然，这不是说测试本身不重要，只是现在测试本身的方法学很难得到认可，这次讨论班会给出测试的简单介绍，然后会在一个待测方向（即代码优化）深入了解


软件测试概述

软件测试

- 软件测试是通过执行软件来分析软件某种性质的方法
  - 是的，软件测试就是一种软件分析方法，不过其和我们熟知的静态分析不同，其是一种动态分析技术
  - 但，说实话，静态分析本质也需要对程序的实体进行扫描并加以分析，从某种角度，也是“执行”，只不过是基于不同的语义罢了：测试是具体的操作语义，静态分析是抽象的语义 （还有其他的领域，比如定理证明，用的是公理语义）
- 对于大部分测试，需要分析的性质往往是改软件的运行行为有没有符合规约
  - 比如是否功能正确，性能是否达标等等

软件测试

- 从定义可知，测试的关键有两点：
  - 如何执行?
    - 待测软件的输入形式是什么？用什么语法可以描述？怎么构造合适的输入?
    - 对于非法的输入会导致未定义行为(Undefined Behavior)吗?
  - 如何判断是否符合规约?
    - 有形式化的规约吗?
    - 没有的话，如何自动化判定?

软件测试

- 除了这两点之外，由于测试天然的动态特性，其无法枚举所有可能的程序输入空间
  - 静态分析某种程度上可以做到（但我们已经学习过莱斯定理，本质上做不到对非平凡性质的完备推理），利用抽象解释的结束，可以推断出在所有输入下的某种性质
- 因此，如何选择最有“价值”的输入是软件测试的一个重点
  - 静态：最大代码覆盖（最简单的白盒），所有可达路径（符号执行），输入空间的刻画（组合测试、等价类），最能体现软件错误特征（基于历史）
  - 动态：基于插桩等信息来动态获取当前测试信息来指导下面测试的生成（自适应）

输入构造

输入

- 一个程序的输入就是能够被这个程序所能正确处理的语言（language）
- 语言本质上就是字符串的集合
  - 呃，那图像处理程序呢？图像也是语言吗？
    - 实际上，在形式语言理论中，图像也有形式语言来描述：图像语言（picture language）是指一组图像，其中图像是某个字母表上一组字符组成的二维数组
  - 那，什么是字符串呢？以及相关的概念呢？

D. Giammarresi, A. Restivo, Two-dimensional languages, in: A. Salomaa, G. Rozenberg (Eds.), Handbook of Formal Languages, Vol. 3, Beyond Words, Springer, Berlin, 1997, pp. 215–267

一些定义

- 字母表Σ是一个有限的符号的集合，e.g., 中文字符集
- 字母表Σ上的字符串s是由Σ中符号构成的一个有穷序列：特殊的串，空串ϵ, $|\epsilon| = 0$
  - 串可以进行"拼接"操作，一个串可以和自己拼接（幂运算，$s^0 \triangleq \epsilon, s^n \triangleq s^{n-1}s$）
- 语言是给定字母表Σ上一个任意的可数的串集合
  - 比如，变量**id**: $\{a, b, c, a1, a2, ...\}$，空白：{**blank**, **tab**, **newline**}
  - 由于其是集合，因此可以通过集合的操作来构造新的语言
    - 其中比较特殊的操作是Kleene闭包，$L^* = \bigcup_{i=0}^{\infty} L^i$，其中 $LM = \{st \mid s \in L \,\&\, t \in M\}$

形式文法（Formal Grammar）

- 语言是字符串的集合，但字符串从何来?
- 我们可以通过形式文法 $G = (N, \Sigma, P, S)$来刻画字符串
  - 有限的非终结符号集N， 与G生成的字符串没有交集
  - 有限的终结符号集Σ, 与N没有交集
  - 有限的产生式规则集P，每个规则都为如下形式
    - $(\Sigma \cup N)^* \ N \ (\Sigma \cup N)^* \rightarrow (\Sigma \cup N)^*$
  - 开始符号$S \in N$
- 这种形式语法也被称为重写系统（re-writing system）或短语结构文法(phrase structure grammar)

Noam Chomsky

形式文法（Formal Grammar）

- 产生式样式$(\Sigma \cup N)^* \ N \ (\Sigma \cup N)^* \rightarrow (\Sigma \cup N)^*$
  - 这里的 $*$是克莱尼星号，∪表示并集。也就是说，每个产生式规则从一个符号串映射到另一个符号串，并且产生式左侧的字符串中必须至少包括一个非终结符号
- 推导（Derivation）：
  - 推导即是将某个产生式的左边替换成它的右边
  - 每一步推导需要选择替换哪个非终结符号, 以及使用哪个产生式（可能有多个产生式）

例子

- 给定文法$G = (N, \Sigma, S, P)$:
  - $N = \{S\}$
  - $\Sigma = \{1, (,), +\}$
  - $P = \{S \to (S + S), S \to 1\}$
- 那么我们可以从开始符号S推导出如下：
  - $S \implies 1$
  - $S \implies (S + S) \implies (S + 1) \implies (1 + 1)$
  - $S \implies (S + S) \implies ((S + S) + S) \overset{*}{\implies} ((1 + 1) + 1)$

多步推导

形式文法和语言

- 句型（Sentential Form）：
  - 如果$S \overset{*}{\Longrightarrow} \alpha, \alpha \in (\Sigma \cup N)^*$, 则$\alpha$是文法$G$的一个句型
- 句子（Sentence）
  - 如果$S \overset{*}{\Longrightarrow} \omega, \omega \in \Sigma^*$, 则$\omega$是文法$G$的一个句子
- 文法G生成的语言$L(G)$:
  - 文法G生成的语言$L(G)$是它能够推导出的所有句子构成的集合
    - $L(G) = \{\omega | S \overset{*}{\Longrightarrow} \omega\}$

形式文法

- 在编译原理里，给定文法G和某个句子s，我们会关心是否$s \in L(G)$，即, 检查 s 是否符合文法G
  - 为此，你们要学习很多语法解析器的算法（词法分析、语法分析、构建语法树等等），从而给出判定
- 在软件测试领域，我们可以“反“过来：
  - 给定文法G,我们可以生成符合该文法的句子集合
    - 这是自然的，因为文法G含有句子的产生式，我们只要不断的选择产生式直到全是终止符即可！

例子

- 给定文法$G = (N, \Sigma, S, P)$:
  - $N = \{S, E, T, F, I, D\}$
  - $\Sigma = \{ +, -, *, /, ., (,), 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
  - P :
    - $S \to E$
    - $E \to T + E \mid T - E \mid T$
    - $T \to T * F \mid T / F \mid F$
    - $F \to + F \mid - F \mid (E) \mid I \mid I . I$
    - $I \to DI \mid D$
    - $D \to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

例子

- 给定文法$G = (N, \Sigma, S, P)$:
  - $N = \{S, E, T, F, I, D\}$
  - $\Sigma = \{+, -, *, /, ., (,), 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
  - P :
    - $S \rightarrow E$
    - $E \rightarrow T + E | T - E | T$
    - $T \rightarrow T * F | T / F | F$
    - $F \rightarrow + F | - F | (E) | I | I . I$
    - $I \rightarrow DI | D$
    - $D \rightarrow 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9$

S

例子

- 给定文法$G = (N, \Sigma, S, P)$:
  - $N = \{S, E, T, F, I, D\}$
  - $\Sigma = \{+, -, *, /, ., (,), 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
  - P :
    - $S \to E$
    - $E \to T + E \mid T - E \mid T$
    - $T \to T * F \mid T / F \mid F$
    - $F \to + F \mid - F \mid (E) \mid I \mid I . I$
    - $I \to DI \mid D$
    - $D \to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

E

例子

- 给定文法$G = (N, \Sigma, S, P)$:
  - $N = \{S, E, T, F, I, D\}$
  - $\Sigma = \{+, -, *, /, ., (,), 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
  - P :
    - $S \rightarrow E$
    - $E \rightarrow T + E \mid T - E \mid T$
    - $T \rightarrow T * F \mid T / F \mid F$
    - $F \rightarrow + F \mid - F \mid (E) \mid I \mid I . I$
    - $I \rightarrow DI \mid D$
    - $D \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

The simplest strategy: by random

?

例子

- 给定文法$G = (N, \Sigma, S, P)$:
  - $N = \{S, E, T, F, I, D\}$
  - $\Sigma = \{+, -, *, /, ., (,), 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
  - P :
    - $S \to E$
    - $E \to T + E \mid T - E \mid T$
    - $T \to T * F \mid T / F \mid F$
    - $F \to + F \mid - F \mid (E) \mid I \mid I . I$
    - $I \to DI \mid D$
    - $D \to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

$$T - E$$

例子

- 给定文法$G = (N, \Sigma, S, P)$:
  - $N = \{S, E, T, F, I, D\}$
  - $\Sigma = \{+, -, *, /, ., (,), 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
  - P :
    - $S \rightarrow E$
    - $E \rightarrow T + E | T - E | T$
    - $T \rightarrow T * F | T / F | F$
    - $F \rightarrow + F | - F | (E) | I | I . I$
    - $I \rightarrow DI | D$
    - $D \rightarrow 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9$

Select which non-terminal to derive? We can also use random strategy

$T - E$

例子

- 给定文法$G = (N, \Sigma, S, P)$:
  - $N = \{S, E, T, F, I, D\}$
  - $\Sigma = \{+, -, *, /, ., (,), 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
  - P :
    - $S \to E$
    - $E \to T + E \mid T - E \mid T$
    - $T \to T * F \mid T / F \mid F$
    - $F \to + F \mid - F \mid (E) \mid I \mid I . I$
    - $I \to DI \mid D$
    - $D \to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

$$F - E$$

例子

- 给定文法$G = (N, \Sigma, S, P)$:
  - $N = \{S, E, T, F, I, D\}$
  - $\Sigma = \{+, -, *, /, ., (,), 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
  - P :
    - $S \to E$
    - $E \to T + E \mid T - E \mid T$
    - $T \to T * F \mid T / F \mid F$
    - $F \to + F \mid - F \mid (E) \mid I \mid I . I$
    - $I \to DI \mid D$
    - $D \to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

$$I . I - E$$

例子

- 给定文法$G = (N, \Sigma, S, P)$:
  - $N = \{S, E, T, F, I, D\}$
  - $\Sigma = \{+, -, *, /, ., (,), 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
  - P :
    - $S \to E$
    - $E \to T + E \mid T - E \mid T$
    - $T \to T * F \mid T / F \mid F$
    - $F \to + F \mid - F \mid (E) \mid I \mid I . I$
    - $I \to DI \mid D$
    - $D \to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

$$D . I - E$$

例子

- 给定文法$G = (N, \Sigma, S, P)$:
  - $N = \{S, E, T, F, I, D\}$
  - $\Sigma = \{+, -, *, /, ., (,), 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
  - P :
    - $S \to E$
    - $E \to T + E \mid T - E \mid T$
    - $T \to T * F \mid T / F \mid F$
    - $F \to + F \mid - F \mid (E) \mid I \mid I . I$
    - $I \to DI \mid D$
    - $D \to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

$$9.I - E$$

例子

- 给定文法$G = (N, \Sigma, S, P)$:
  - $N = \{S, E, T, F, I, D\}$
  - $\Sigma = \{+, -, *, /, ., (,), 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
  - P :
    - $S \to E$
    - $E \to T + E \mid T - E \mid T$
    - $T \to T * F \mid T / F \mid F$
    - $F \to + F \mid - F \mid (E) \mid I \mid I . I$
    - $I \to DI \mid D$
    - $D \to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

$$9.D - E$$

例子

- 给定文法$G = (N, \Sigma, S, P)$:
  - $N = \{S, E, T, F, I, D\}$
  - $\Sigma = \{+, -, *, /, ., (,), 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
  - P :
    - $S \to E$
    - $E \to T + E \mid T - E \mid T$
    - $T \to T * F \mid T / F \mid F$
    - $F \to + F \mid - F \mid (E) \mid I \mid I . I$
    - $I \to DI \mid D$
    - $D \to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

$$9.3 - E$$

例子

- 给定文法$G = (N, \Sigma, S, P)$:
  - $N = \{S, E, T, F, I, D\}$
  - $\Sigma = \{+, -, *, /, ., (,), 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
  - P :
    - $S \to E$
    - $E \to T + E \mid T - E \mid T$
    - $T \to T * F \mid T / F \mid F$
    - $F \to + F \mid - F \mid (E) \mid I \mid I . I$
    - $I \to DI \mid D$
    - $D \to 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

可以任意复杂!

$$\overset{*}{\Longrightarrow} 9.3 - (5 + 4)$$

从开始符生成的局限

- 随机不可控
- 很多生成的字符串的价值都很低
- 一个启发式的想法：
  - 从现实中的串出发，而不是开始的非终结符出发，然后通过“变异”手段来演化出新的串
    - 这些串和现实中的串有很高的相关度，并且又更进一步变的多样和复杂，可谓“青出于蓝而胜于蓝”
  - 但问题是，你怎么能够保证变异的串是符合文法的了？
    - 一个微小的扰动完全让一个句子从一个合法的转化为非法的：E.g., 如果我们要fuzz url, 从一个合法的串“https://www.nju.edu.cn” 变异为 “https//www.nju.edu.cn”就是非法的了！

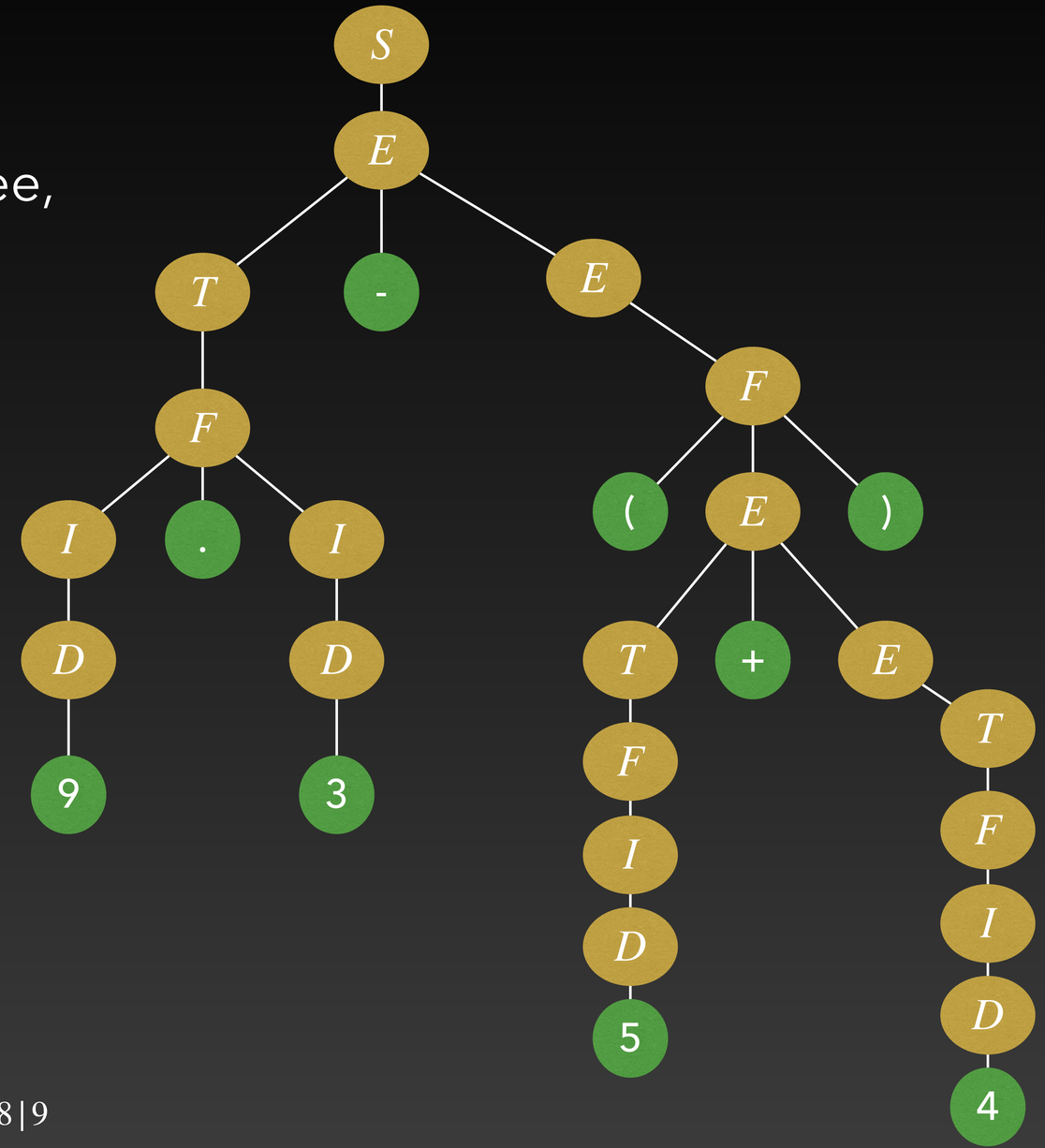
基于变异的句子生成

- 基本思想：
  - 每一个valid sentence都一定是从开始符S演变过来，其每一步都是"合法"的，最终才能得到最终的句子。
  - 但是根据我们刚才的语法制导生成的过程中，每一步的选择并不唯一，生成式和非终结符的挑选都会影响最终的句子生成。
  - 如果我们能根据这个valid sentence的生成过程逆推，回退到某个含有非终结符的时候，此时选择一个不一样的产生式/非终结符进行推导，那最终就是一个不一样的但是合法的句子，而且和原来的valid sentence在结构上具有共性，因此是一个合法的"变异"。
- 那么如何能得到这个valid sentence的原推导过程呢?
  - 简单！编译原理中的parse过程会把句子的演化过程以树状的形式展现出来。

分析树（Parse Tree）

- 分析树（也称具体语法树，Concrete Syntax Tree, CST）是输入字符串在解析过程中构建的树结构
  - 叶子节点为终止符
  - 非叶子节点为非终止符

$$9.3 - (5 + 4)$$

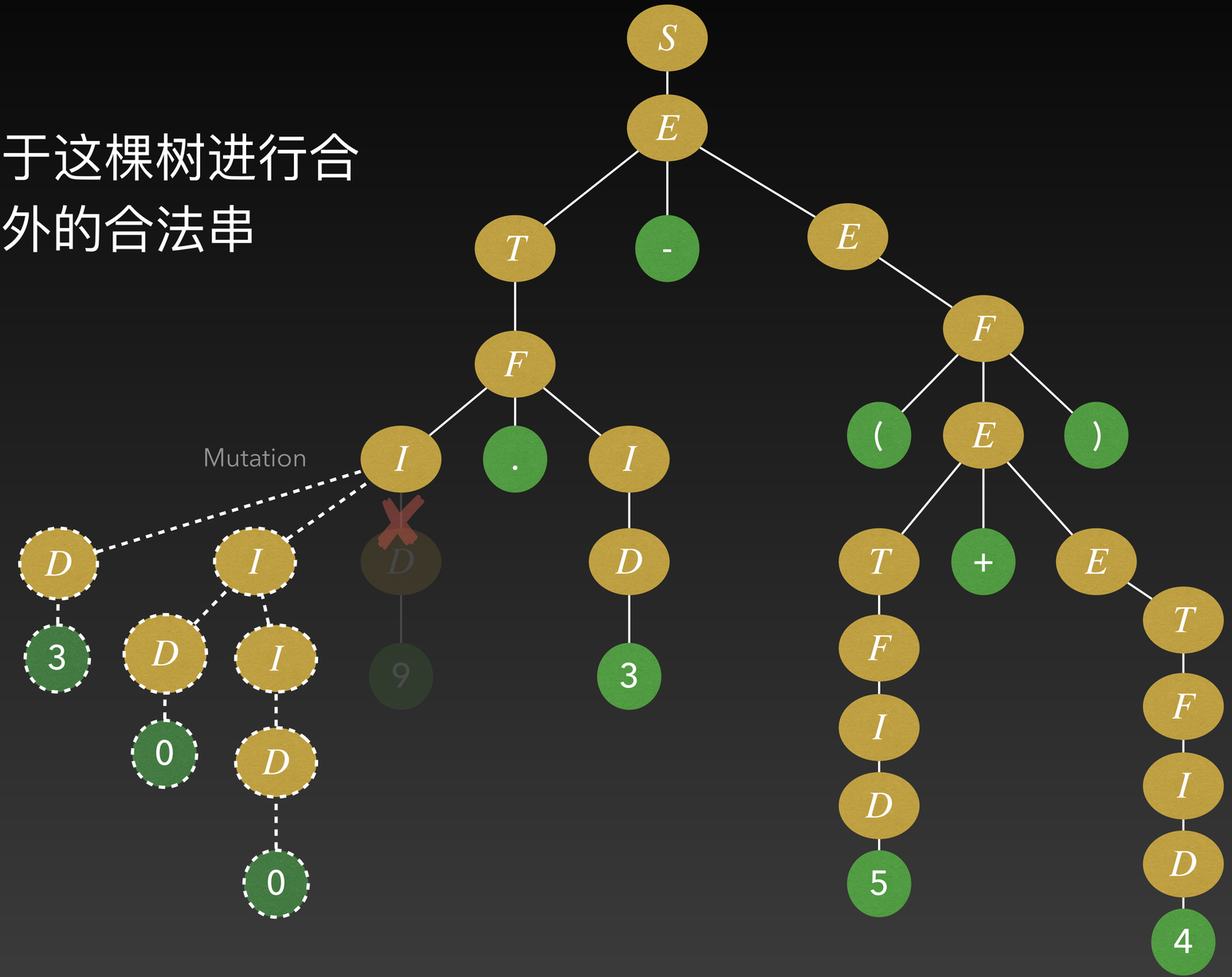


- $S \rightarrow E$
- $E \rightarrow T+E|T-E|T$
- $T \rightarrow T*F|T/F|F$
- $F \rightarrow +F|-F|(E)|I|I.I$
- $I \rightarrow DI|D$
- $D \rightarrow 0|1|2|3|4|5|6|7|8|9$

分析树（Parse Tree）

- 有了分析树，我们就可以基于这棵树进行合法的编辑，使得其变异成另外的合法串

  - $S \rightarrow E$
  - $E \rightarrow T+E|T-E|T$
  - $T \rightarrow T*F|T/F|F$
  - $F \rightarrow +F|-F|(E)|I|I.I$
  - $I \rightarrow DI|D$
  - $D \rightarrow 0|1|2|3|4|5|6|7|8|9$



$9.3-(5+4)$ → $300.3-(5+4)$

问题是怎么得到这样的语法树?

自动机

- 得到一个分析树的过程，本质上就是能够从一个字符串逆向回开始符S的过程，换句话说，本质上就是判定一个字符串是否属于某个语言（由文法描述）的过程
- 自动机(Automata)就是做这类工作的理论机器！
  - 其是一个抽象的自推进计算设备：即按照预定的操作顺序自动运行

自动机的形式描述

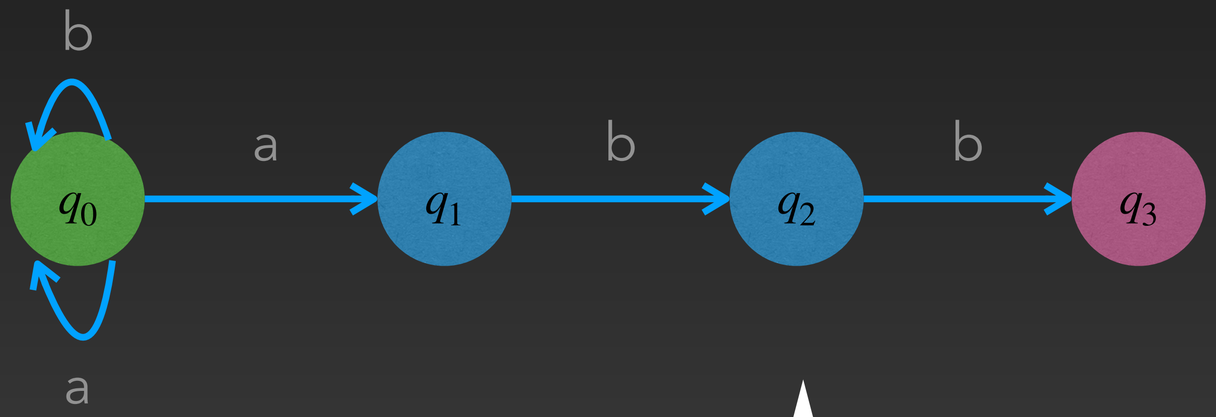
- 自动机可以通过五元组$M = < \Sigma, \Gamma, Q, \delta, \lambda >$来表示
  - Σ：是有限的符号集合，为自动机的输入字母表
  - Γ：也是有限的符号集合，为自动机的输出字母表
  - Q：为自动机的状态集合
  - δ：是状态转换函数(transition function), $\delta : Q \times \Sigma \rightarrow Q$, 即将当前状态和输入符号映射到下一个状态
  - λ：是下个输出函数(next-output function), $\lambda : Q \times \Sigma \rightarrow \Gamma$, 即根据当前状态和输入符号映射到下一个输出

自动机的形式描述

- 输入字符串
  - 有限的字符串$a_1a_2\ldots a_n$，其中的每个符号$a_i \in \Sigma$
- 运行
  - 给定初始状态q_0，输出串$a_1a_2\ldots a_n$，那么自动机会不断运行：对于当前状态q_{i-1}和读取第$i-1$个字符a_{i-1}，那么根据状态转移函数$\delta(q_{i-1}, a_{i-1}) = q_i$，转移到状态$q_i$，此外，也会根据下个输出函数$\lambda(q_{i-1}, a_{i-1}) = b_i$输出$b_i$
- 终止/接收
  - 如果最终的状态q_n是该状态机额外定义的终止状态状态之一，我们称输入为$a_1a_2\ldots a_n$的计算终止，如果该计算任务为识别字符串，那么我们说$a_1a_2\ldots a_n$属于该状态机所能识别(recognize)的语言

有穷状态机

- 对于状态集合为有穷自动机(Finite Automaton)，我们称为有穷自动机，其可以识别正则表达式(regular expression, or regex)所表达的语言
- 比如，对于正则表达式 (a | b) *abb 所代表的语言，其对应的(非确定性)有穷自动机（NFA）为：
  - 注：非确定性自动机的状态转移函数 $\delta : \Sigma \times Q \to 2^Q$



Thompson 构造法可以从正则语法直接构造NFA

(a | b) *abb 是正则表达式的表现形式，也可以用之前的形式文法表达：

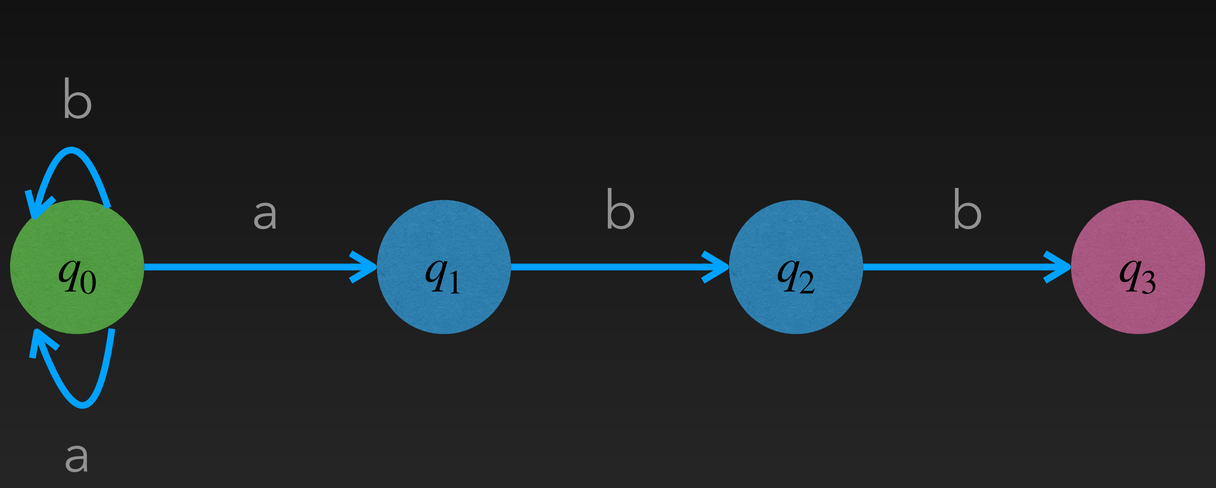
$S \to Ab$

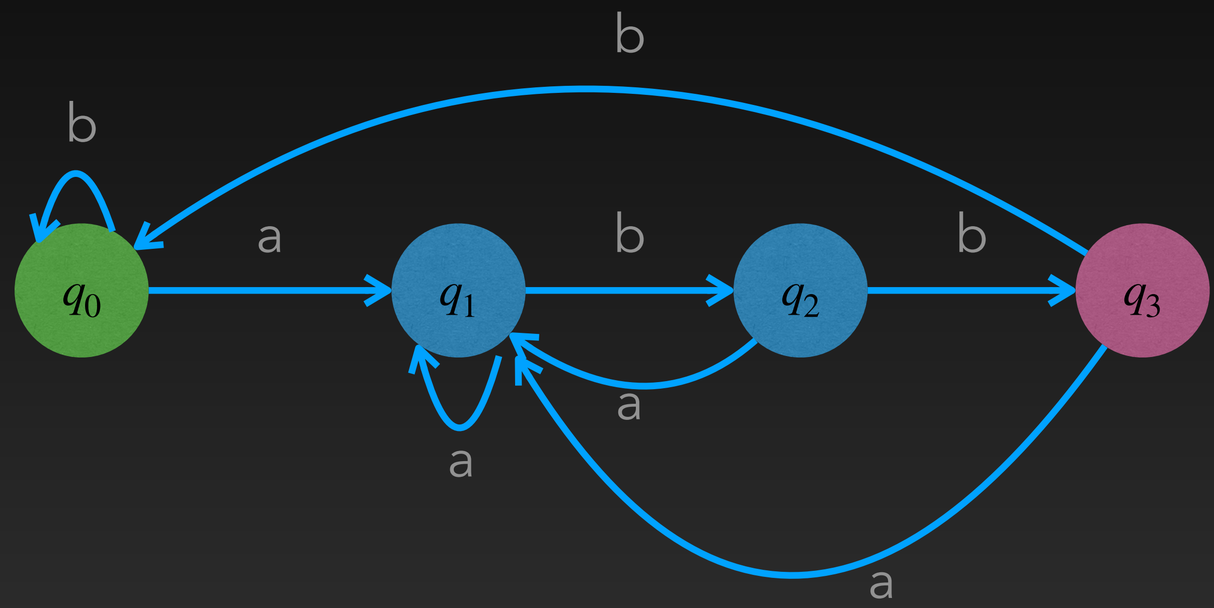
$A \to Bb$

$B \to Ca$

$C \to Ca \mid Cb \mid \epsilon$

确定性，非确定性

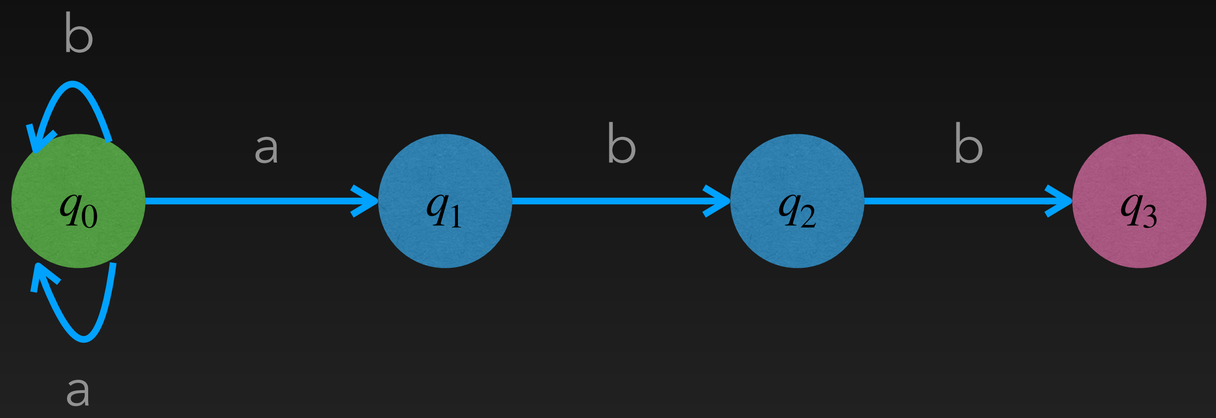




- Michael O. Rabin和Dana Scott首次引入非确定性自动机的概念，并与1976年获得图灵奖，非常简洁，易于理解，能够清晰地描述语言，
  - 从NFA可以通过子集构造法的方式得到等价的确定性自动机，DFA要复杂一点，但是很适合构造识别算法

有穷状态机

和确定性有穷自动机等价的非确定性自动机（转移函数$\delta : \Sigma \times Q \to 2^Q$）



(a | b) *abb 是正则表达式的表现形式，也可以用之前的形式文法表达：

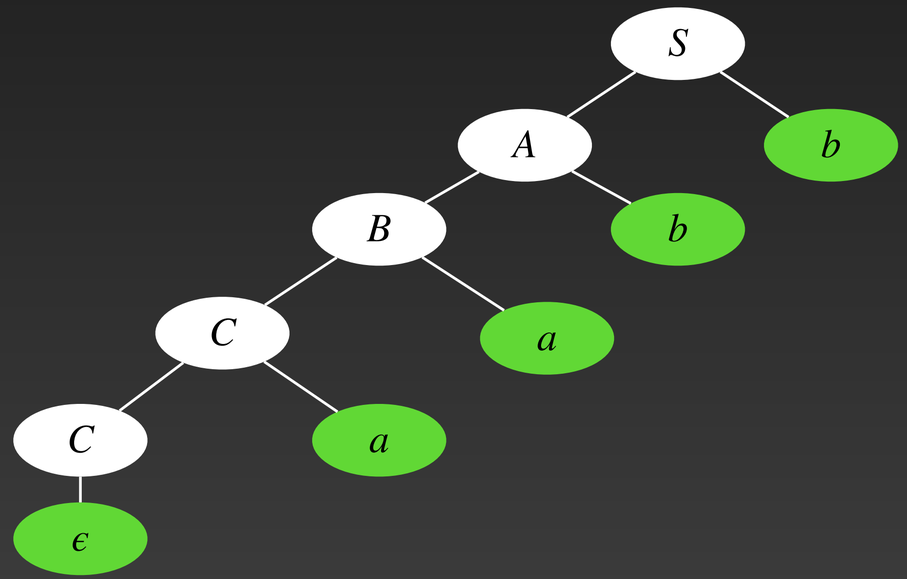
$S \to Ab$

$A \to Bb$

$B \to Ca$

$C \to Ca \mid Cb \mid \epsilon$

- 加入给定字符串aabb，那么经过有穷状态机的的parse，就会发现其经过了:$q_0q_0q_1q_2q_3$的状态序列，很容易逆推出相应的分析树

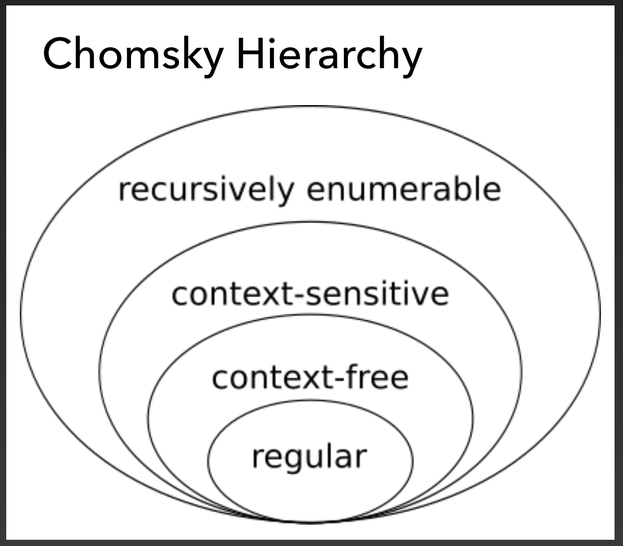


有限状态机的局限

- 正则语言并不能描述所有语言，比如语言$L = \{a^n b^n \mid n \geq 0\}$，（可通过Pumping Lemma证得），相应的，有限状态机也无法识别出该类语言，因此需要更强的自动机

Grammar	Languages	Recognizing automation	Production rules
Type-3	正则	有限自动机	$A \to x \mid xB$
Type-2	上下文无关	非确定性下推自动机	$A \to \alpha$
Type-1	上下文相关	线性有界非确定性图灵机	$\alpha A \beta \to \alpha \gamma \beta$
Type-0	递归可枚举	图灵机	$\gamma \to \alpha$

$x \in \Sigma \quad A, B \in N \quad \alpha, \beta, \gamma \in (\Sigma \cup N)*$

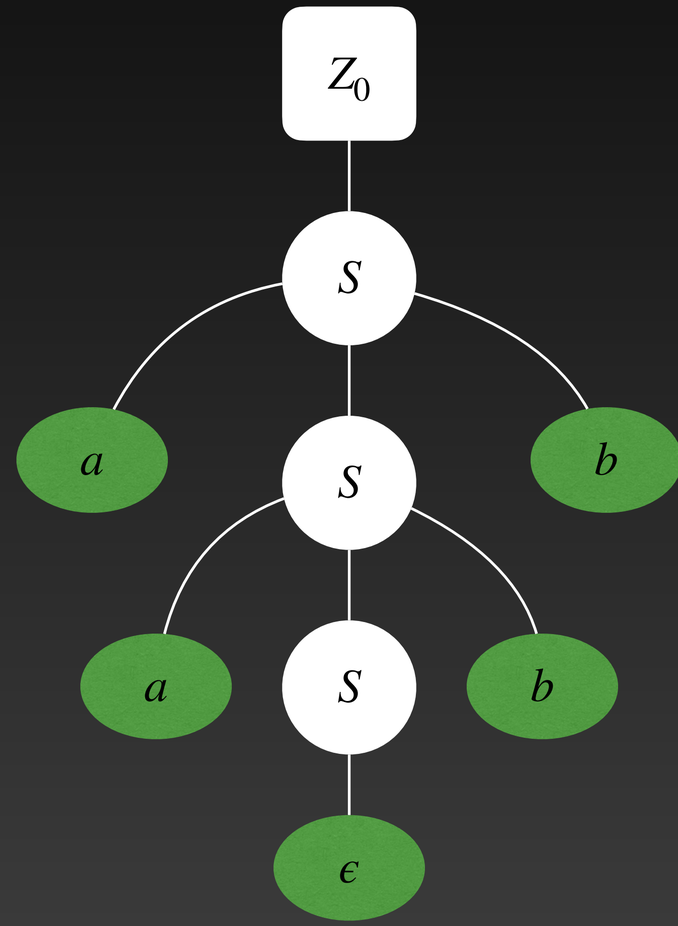


更复杂的机器

- 下推自动机就是比普通的有限自动机多了一个栈，转换函数需要根据当前状态、栈的符号、和当前输入来确定下一个状态，因此可以表达嵌套的语义，对于上述文法，可以给出类似如下的下推自动机
  - 初始状态q_0 栈顶符号Z_0
  - 栈记录着生成式中的非终结符，并且每次输入字符时，自动机会根据状态转移规则来压栈或弹栈
- 图灵机则更加灵活
  - 其可以任意操纵内存上的数据，而无需像下推自动机那样只能按照“先入后出”的栈存储模式操纵内存，因此可以识别更强的语言（递归可枚举语言）

一个更加复杂的例子

- 考虑上下文无关文法$S \rightarrow aSb \mid \epsilon$, 以及输入串$aabb$，我们可以通过如下的下推自动机进行识别
- **输入**： aabb
- 处理：
  - **初始状态**：栈中只有Z_0 ，表示开始处理
  - 读入第一个 a：当前栈顶是Z_0，根据生成式 S→aSb， 压入一个S，后续如果遇到b可以出栈, 读入第二个a时，栈顶为S，再次压栈，然后遇到第一个b，此时栈顶是第二个S，说明已经符合S→aSb的生成式，出栈S，然后遇到第二个b，并且栈顶是第一个S，说明再次符合S→aSb的生成式，出栈S，栈顶为Z_0，并且没有更多的输入，解析完成
  - 根据这个解析过程，很容易构造语法树



实用的工具

- 实际上，你已经不必去写这些自动机以及如何根据自动机的结果来构造解析树，语法的解析已经是非常成熟的领域（当然前沿还有研究）
- 有很多工具可以辅助你去构造分析树
  - ANTLR (Another Tool for Language Recognition)就是其中之一，可以非常方便的生成分析树，其背后的算法是 LL(*) 算法 （OOPLSA 2014， PLDI 2011）
  - YACC/Bison（比较过时）

Quiz

- 1.如果给定一个串并不是该语言所属的串，那么如何修改使其符合该语法呢?
  - 很多时候生成器并不能完整和准确的对应语言，那么生成的串可能就不符合程序所应该接受的语言，那么这个串就会引发这个程序所谓的**Undefined Behavior**
    - 如何变异该串使的其“重返大道”?
- 2. 如果一个串有多个语法树，即可以从开始符经多个不通路径到达相同的串，我们就称该语法具有歧义，有没有办法消除歧义?
- 3. 很多时候并没有给出形式语法，能否通过某种方式“合成”该程序的形式语法?

形式语言之外

- 很多时候，程序的输入难以通过形式文法完全表达，比如图像、声音、视频处理程序（尽管我们之前说过也可以通过类似字符串的形式给出，但毕竟不是主流表示）
  - 没有形式文法意味着没有类似的生成式可以“完美”的生成符合程序要求的输入
- 但我们可以“近似”地给出输入
- 一个很容易想到的做法是：
  - 假设程序的合法输入符合某种分布（多个随机变量的联合分布）
  - 你只需要在这个分布中“采样”即是合法的输入

例子

- 经典的对抗神经网络(Generative Adversarial Network, GAN)就是可以生成符合样本的分布的数据的方法之一
- GAN 由两个网络组成：生成器（Generator） 和 判别器（Discriminator）
  - 生成器接收一个随机噪声向量，并生成图像
  - 判别器判断生成的图像是真实数据还是生成的数据
  - 生成器通过欺骗判别器逐渐学会生成越来越逼真的图像

例子

- 其目标函数：$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_{\text{data}}(x)}\left[\log D(x)\right] + \mathbb{E}_{z\sim p_z(z)}\left[\log(1-D(G(z)))\right]$
  - $\min_G$：表示对生成器 G 进行最小化
  - $\max_D$：表示对判别器D 进行最大化
  - $x \sim p_{\text{data}}(x)$：表示 x 是从真实数据分布中抽取的样本
  - $z \sim p_z(z)$：表示z是从生成器的输入噪声分布(通常为正态分布)中抽取的样本, $G(z)$表示生成器基于这个输入的输出
  - $\log D(x)$，表示对判别器样本x的判断：如果来自真实样本，判定器希望接近1，否则接近0

例子

- GAN的训练过程即为交替优化：
  - Step 1. 固定生成器 G，更新判别器 D。判别器通过最大化目标函数来学习区分真实和伪造样本
  - Step 2. 固定判别器 D，更新生成器 G。生成器通过最小化目标函数来欺骗判别器，使得生成的样本更逼真。
- 通过反复迭代（最终会到达一个平衡点，即纳什均衡），生成器生成的样本会越来越逼真，而判别器则会不断改进以区分真实样本和伪造样本。

生成模型

- 生成模型（Generative Model）可以为我们解决没有形式文法的情况下生成符合程序要求的输入，一些经典的生成模型（希望能够给出一次报告）
  - Gaussian mixture model
  - Probabilistic context-free grammar
  - Diffusion model
  - Boltzmann machine
  - Hidden Markov model
  - Autoregressive model


测试预言

程序预言的不可判定性

- Oracle本质上在探索这个问题：待测软件是否正确实现了规约?
- 我们可以换一种视角来看待这个问题：
  - 规约本质上是某种形式语言写成的具备某种语义的“程序”
  - 而你实现的程序本质上是用编程语言写成的某种“程序”
  - 因此，当你问出你的程序是否实现正确时，你本质上在问这样一个问题：这两个“程序”是否在语义上等价
- 我们在软工1的课程中已经完全能证明这个问题：
  - 这是一个不可判定问题！

现实更加糟糕

- 如果有形式化的规约，虽然预言问题不可判定，我们还是可以给出现实的近似算法
  - 比如对于软件是否能够终止，虽然是不可判定，但并不是对任意的程序我们都无法判定是否会终止，只是存在这样的程序我们无法判定而已。我们可以对很多程序都能给出是否终止的判定，对于那些给不出的，回答“不知道”即可，或者给出一个保守的（近似）答案
  - 程序预言也是如此，给定一个形式化规约，很多性质我们都可以给出判定结果
- 但如果没有这样的规约呢？– 非常普遍的情况！
  - 两种情况：通过非形式化的语言（比如自然语言）描述，或者根本就没有（或难以得到，这种情况下一般来说软件的规约有一个符合人类常识认知的约定）

解决方案

- 用人吧！—— 太废人了：正确的做法，如何最少量的使用人！
- 用“假人” —— 人工智能
  - 目前确实有一些，特别是大语言模型在做
    - 比如利用自然语言处理技术对所写的规约进行学习
    - 从给定程序源码、注释、或论坛评价等信息，来学习
  - 但没有soundness的保证
  - 当然，“真人”也没有soundness的保证

Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey.

To the best of our knowledge, we are the first…

只要我paper读的少，我就有无数novel ideas

典型的specification挖掘工作

- 不变式（invariants）的动态挖掘 – Daikon
  - 提供程序及其一组输入（测试用例），程序将被执行多次，每次使用尽可能差异大的输入数据，收集运行的信息
  - 定义一组潜在的不变式（这些不变式和程序的内部变量相关联），比如量之间的简单关系，如 $x + y = z$
  - 根据运行结果筛除不符合的不变式

M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M.S. Tschantz, and C. Xiao, “The Daikon system for dynamic detection of likely invariants,”
M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, “Dynamically discovering likely program invariants to support program evolution,”
Kexin Pei, David Bieber, Kensen Shi, Charles Sutton, and Pengcheng Yin. 2023. Can large language models reason about program invariants?

典型的specification挖掘工作

- Weyuker 早在1983年就将将推理和测试视为逆过程。
  - 测试过程从一个程序开始，寻找能够表征预期和实际行为每个方面的输入/输出对
  - 而推理过程则从一组输入/输出对开始，推导出一个符合给定行为的程序
- 这激发了很多后来的机器学习算法用来根据部分测试用例和输出来“学习”合成一个程序（往往是一个FSM），试图挖掘出待测程序的specification

Elaine J. Weyuker. 1983. Assessing Test Data Adequacy through Program Inference

那，有没有别的?

- 规约不过是定义了软件的行为，没人规定非得用形式化语言
- 比如，如果我们用另外的程序写的规约呢?
- Differential Testing!
- Regression Testing!

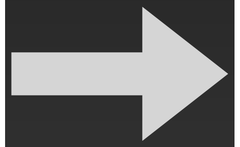
蜕变测试

- 完整的规约非常困难，有没有简单一点的?
- 蜕变测试（Metamorphic Testing）
  - Sin(x) function —> Sin(x+360) = Sin(X).
  - Hence, when design test inputs, we can have 30, 30+360, 30+360+360. They must equal to each other.

Intramorphic

- Differential本质是多个不同版本，target相同的语义，如果没有多个版本?
- Intramorphic：自己构造!
  - 修改系统在测（SUT）的一个或多个组件
  - 使得修改后系统和原始系统在一组输入上的输出关系是已知的

```
def bubble_sort(arr):
  length = len(arr)
  for i in range(length):
    for j in range(0, length - i - 1):
      if arr[j] > arr[j+1]:
        arr[j], arr[j+1] = arr[j+1], arr[i]
  return arr
```



```
def  bubble_sort_reverse():
  length = len(arr)
  for i in range(length):
    for j in range(0, length - i - 1):
      if arr[j] < arr[j+1]:
        arr[j], arr[j+1] = arr[j+1], arr[i]
  return arr
```

那么给定任何输入，这两者的输出都应该是严格相反的


反馈

测试的好坏标准

- 有了输入构造和正确性判定，我们已经可以开始测试了！
  - 是的，任何程序你都可以尝试去测
- 如果你想测的足够多
  - 只要在构造是用随机性就可以了
    - 随机选择非终结符
    - 随机选择生成式
  - 主打一个大力出奇迹！

但随机性不是任何时候都高效

- 随机很好，但是如果程序的行为特性与你的输入的文法的分布并不是特别相关，那随机的效率就很低
  - 比如程序中如果含有 if (a = ‘c’ & b = ‘(’ & c = ‘#’ & d = ‘$’){…} 在这个程序中，只有一种类型的pattern可以进入这个分支，但你很难用随机方法进入
- 解决这类问题有两种方式：
  - 获得白盒信息，利用它类似符号执行、静态分析等方法获得这种分支的语义，然后推演出合适的输入
  - 从“之前“的输入中获得信息，然后吸取教训，进行更好的尝试 – 这就是反馈（or 自适应）

关于自适应的两种方法学

- TY Chen的Adaptive Random Testing
- Kaiyuan Cai的Adaptive Testing