黑盒环境下复杂编程语言的上下文无关文法推断

Feifei Li[1,6], Xiao Chen[2], Xiaoyu Sun[3], Xi Xiao[1]，[6*]，Shaohua Wang[4], Yong Ding[5*]，Sheng Wen[5], Qing Li[6]

[1]中国清华大学深圳国际研究生院[2]澳大利亚纽卡斯尔大学[3]澳大利亚澳大利亚国立大学[4]中国中央财经大学[5]中国东莞广东科技学院计算机科学学院，523668[6]中国鹏城实验室

arXiv:2601.12385v1 [cs.PL] 18 Jan 2026

摘要

复杂编程语言的语法推断仍是一项重大挑战，因为现有方法无法在实际可行的时间限制内扩展到真实世界的数据集。在我们的实验中，包括 Arvada、Treevada 和 Kedavra 在内的最先进工具，均无法在 48 小时内推断出 C、C++ 和 Java 等复杂语言的语法。Arvada 和 Treevada 直接对完整长度的输入示例进行语法推断，这对于此类语言中常见的大型文件而言效率低下。Kedavra 通过引入数据分解来创建更短的语法推断示例，但其词法分析仍依赖于原始输入。此外，其严格禁止过度泛化的约束限制了复杂语法的构建。

为克服这些局限，我们提出了 Crucio。它构建分解森林，并通过分布矩阵提取短示例，用于词法和语法推断。实验结果表明，Crucio 是唯一能够在合理时间内成功推断复杂编程语言语法的方法（其非终结符数量最多比以往基准测试高出 23× ）。在以往的简单基准测试中，Crucio 的平均召回率分别比 Treevada 和 Kedavra 高 1.37× 和 1.19× ，F1 分数分别提高了 1.21× 和 1.13×。

CCS概念

• **请勿使用此代码 → 为您的论文生成正确的术语**；为您的论文生成正确的术语；生成为您的论文生成正确的术语；为您的论文生成正确的术语。

关键词

语法推断、复杂编程语言、黑盒

ACM 引用格式：

Feifei Li[1,6], Xiao Chen[2], Xiaoyu Sun[3], Xi Xiao[1,6*], Shaohua Wang[4],Yong Ding[5*], Sheng Wen[5], Qing Li[6]。2026 年。黑盒环境下复杂编程语言的上下文无关文法推断。载于第*48*届 *IEEE/ACM* 国际软件工程大会论文集（*ICSE '26*）。ACM，纽约，纽约州，美国，12页。https://doi.org/10.1145/3744916.3773147

1 引言

从示例字符串推断上下文无关文法（CFG） [13] 在软件工程中有广泛的应用，包括代码理解 [27], 逆向工程 [24],协议规范恢复 [11, 25], 以及模糊测试的输入生成 [2, 6, 14, 15, 26, 30, 31]。

大多数现有的语法推断技术都在灰盒或白盒环境中运行，在这些环境中，可以访问解析器的内部结构或源代码 [10, 17, 22]。相比之下，黑盒文法推断旨在无法访问解析器内部实现的情况下恢复其底层语法，仅依赖输入样本以及解析器提供的二元反馈（接受或拒绝）。其目标是构造一种能够近似预言机行为的语法，准确预测任意输入是否会被接受。

近期的黑盒文法推断研究包括 Arvada [19], Treevada [5], 和 Kedavra [21]等方法。虽然这些方法在小型合成文法上展现出潜力，但在处理真实世界编程语言时却表现不佳。用于评估 Arvada、Treevada 和 Kedavra 的数据集通常对应少于20个非终结符的解析器文法。为评估可扩展性，我们引入了源自 C、C++、Java、Lua 和 Rust 的高难度基准测试，其中包括含有20至200个非终结符的文法，以及平均长度超过2,000个字符的输入样本。现有工具无法在这些数据集上生成有效文法。

对真实编程语言进行语法推断面临诸多挑战。首先，输入程序规模庞大——通常包含数千个词元——使得直接进行语法推断的计算成本高昂。Arvada 和 Treevada 都通过根据上下文相似度选择要合并的气泡，来泛化文法规则；这种相似度基于原始输入程序

*通讯作者：Xi Xiao 和 Yong Ding。电子邮箱：Feifei Li (lff23@mails.tsinghua.edu.cn)、Xiao Chen (xiao.chen@newcastle.edu.au)、Xiaoyu Sun (xiaoyu.sun1@anu.edu.au)、XiXiao (xiaox@sz.tsinghua.edu.cn)、Shaohua Wang (davidshwang@ieee.org)、Yong Ding (dingyong@gdust.edu.cn)、Sheng Wen (wensheng@gdust.edu.cn)、Qing Li (liq@pcl.ac.cn)。

允许免费为个人或课堂用途制作本作品的全部或部分数字副本或纸质副本，但前提是不得为营利或商业利益而制作或分发副本，且副本须附有本声明及首页上的完整引文。对于本作品中由作者以外的其他方持有版权的组成部分，必须尊重其版权。注明出处后可以摘录。以其他方式复制、重新出版、发布到服务器或重新分发至列表，均须事先获得特定许可和/或支付费用。请联系 permissions@acm.org 申请许可。*ICSE '26，2026*年*4*月*12*日至*18*日，巴西里约热内卢© 2026 版权所有归所有者/作者所有。出版权已授权给 ACM。ACM ISBN 978-1-4503-XXXX-X/18/06 https://doi.org/10.1145/3744916.3773147

进行计算，结果生成的语法过于复杂，推断效率低下。尽管 Kedavra 引入了数据分解以提升可扩展性，但其词法推断仍在未经分解的原始样本上进行，造成了显著的性能瓶颈。此外，其分解策略无法保证分解后的片段保留原始输入的语法覆盖率，从而可能导致泛化损失。Kedavra 还通过施加严格约束来防止过度泛化，但由于依赖基于采样的验证，错误可能无法被发现。这些未被发现的错误会随时间累积，并可能使后续的语法泛化步骤失效或变得不稳定。

为解决这些局限性，我们提出了 Crucio，这是一种新颖的黑盒语法推断框架，旨在扩展至复杂编程语言，同时保持精确率和可解释性。Crucio 引入了多项关键创新。首先，它提出了分解森林这一概念，系统地探索输入样本的结构化分解。借此可以发现简短且可泛化的样本，同时保留语法覆盖率，从而克服 Kedavra 中观察到的泛化损失。其次，Crucio 通过处理更短的分解样本来改进词法推断。它使用决策树对词元进行分类，并应用 L* 算法 [4] 高效、稳健地泛化词元级词法规则。第三，在语法推断方面，Crucio 构建分布矩阵，并在解析树上进行语法泛化，同时强制满足一阶交换正确性。这可确保正确性保证有界，且无需在初始阶段之后进行额外的预言机查询。

据我们所知，Crucio 是**首个能够在黑盒环境中于切实可行的时间内，为 C、C 和 Java 等复杂编程语言推断出可用语法的方法**。

本文的贡献如下：

• 我们提出了 Crucio，这是一种新颖的语法推断框架，可提升黑盒语法推断的泛化能力和可扩展性。该工具已开源，可通过 https://github.com/Sinpersrect/crucio 获取。

• Crucio 是首个能够在切实可行的时间限制内为真实世界编程语言推断出准确语法的工具。在复杂基准测试（包括 C、C++、Java 等）中，它是唯一能在超时阈值内得出结果的方法。在简单基准测试中（用于评估最先进（SOTA）工具，涉及 Lisp、JSON、海龟等语言），Crucio 的表现可与最先进的语法推断工具相媲美。

• 我们提出了交换精确率，这是一种全新的语法评估指标，旨在解决基于样本的精确率的关键局限。交换精确率避免了基于样本的精确率所带来的偏差；这种偏差会偏袒仅关注少数样本的无效文法，以及生成短样本的语法，从而使评估更加可靠。

2 背景与相关工作

上下文无关文法 (CFG) [13] 是一种成熟的数学形式体系，用于定义形式语言的句法结构，尤其常用于规定编程语言语法。形式上，上下文无关文法定义为四元组 $G = (V, \Sigma, R, S)$，其中 V表示有限的非终结符集合，Σ 表示终结符集合（与 V不相交），R是有限的产生式规则集合，其形式为 $A \to \alpha$，其中 $A \in V$和 $\alpha \in (V \cup \Sigma)^*$，并且 $S \in V$是指定的开始符号。

上下文无关文法可以通过有限的一组规则递归地定义潜在的无限语言。例如，考虑如下语法：

$$V = S \quad \Sigma = a\ b, \qquad S \to \{\} \mid aSb \mid ab. \qquad a^n b^n \mid n \geq 1$$

{,}，其产生式规则为.....该语法生成的语言为 { }，其中包含“ab”、“aabb”、“aaabbb”等字符串。推导过程从开始符号 S 开始，并递归应用产生式规则，直到得到一个仅由终结符组成的字符串。

上下文无关文法在表达能力与可处理性之间实现了平衡。它们足以描述大多数编程语言的语法，同时又受到足够的约束，能够支持 LL、LR 和 CYK 解析器等高效解析算法。因此，上下文无关文法构成了 C++ [20]、Java [16], 和 JavaScript [35] 等编程语言的编译器和解释器的语法基础。此外，上下文无关文法还广泛用于验证 JSON [29] 和 XML [23], 等结构化数据格式的语法，以及自然语言处理。

黑盒文法推断。黑盒文法推断由 Glade[7], 开创，为后续研究奠定了基础，但在处理高度递归的语法时表现出局限性。为解决这些问题，Arvada[19] 提出了一种通过合并气泡来泛化语法的方法，从而更有效地处理递归结构。在此基础上，Treevada[5]进一步优化了推断语法中括号的处理方式，使学习到的规则能够递归应用，并进一步支持生成确定性文法。Arvada 和 Treevada 都直接从原始样本中推断语法；随着输入复杂度增加，它们的性能会下降，因此不适用于可能长达数千字符的大型复杂输入。此外，它们的气泡合并策略依赖于上下文相似性，却未考虑语法的泛化能力，因此泛化能力有限。Kedavra[21] 引入了一种数据分解策略，将输入字符串切分为更小的单元并进行增量推断，从而显著提高了语法的精确率、召回率和可解释性。然而，Kedavra 的词法推断仍然基于原始样本进行。此外，其泛化过程要求误差为零；随着语法复杂度增加，这一条件会越来越难以满足。因此，它在处理更复杂的语法时效果不佳。另一方面，REINAM[32]专注于概率上下文无关文法（PCFG）：它首先使用 Glade 提取上下文无关文法，然后通过强化学习对其进行优化，以调整产生式的概率。最近，Jia 等人[18] 提出了 V-Star，该方法结合主动学习与可见下推自动机，能够准确推断具有嵌套层次结构的输入语言，例如 XML 和 JSON。然而，V-Star 往往会生成过于复杂的语法，且效率较低。在相对简单的数据集上，其运行速度明显慢于 Arvada 和 Treevada。

灰盒语法推断。灰盒语法推断适用于无法完全访问预言机源代码，但可以利用覆盖率等部分内部信息的场景。GRIMOIRE [10] 采用了这种方法，通过提取

类似语法的结构，这些结构基于模糊测试期间代码覆盖率的反馈。虽然它不会构建显式的上下文无关文法，但会利用行为信号，更有效地指导输入生成。

白盒文法推断。白盒文法推断是指在能够访问解析器源代码的情况下推断输入文法。代表性方法包括采用动态分析的 Lin [22],，以及使用动态污点分析跟踪输入流并重建上下文无关文法的 AUTOGRAM[17], 。近期，Leon[9] 提出了一种专为递归下降解析器设计的符号分析方法，通过直接分析控制流和源代码结构来提取文法规则。这些方法都需要访问解析器的源代码；如果无法获取源代码，它们便无法奏效。

深度学习。尽管循环神经网络等深度学习模型已被用于语法推断，但研究表明，它们难以捕捉上下文无关文法的结构特性 [8, 28, 34]。Yellin [33] 尝试通过从循环神经网络中合成上下文无关文法来克服这一局限。然而，Arvada、Treevada 和 Crucio 缺少主动学习 [3], 这一核心机制，导致其结果明显逊于这些方法。

3 方法

图 1 展示了我们提出的系统 Crucio 的工作流程，该系统专为复杂编程语言的语法推断而设计。整个流程包含三个主要阶段：分解森林构建、词法推断，以及基于分布矩阵的语法推断。

分解森林是 Crucio 的核心结构。它可以将输入序列构建为一棵树，并通过在树上搜索满足给定条件的样本，获取满足词法推断或语法推断条件的短样本，从而显著降低整体推断任务的复杂度。它可以解决 Kedavra 中的泛化损失问题。

词法推断 从分解森林中获取短样本用于推断，并使用 L* 算法构建自动机作为词法规则。

基于分布矩阵的语法推断 首先根据从分解森林中获取的示例构建分布矩阵，以捕获示例的上下文分布。然后，通过反复合并气泡极大团，对扁平解析树的结构进行泛化，最终推导出满足一阶交换正确性的上下文无关语法。构建语法后，Crucio 会继续在分解森林中搜索能够泛化当前语法的样本，然后重构该语法。

在以下各节中，我们将详细介绍 Crucio 的核心阶段及其背后的技术细节。

3.1 分解森林

我们提出了一种名为分解森林的新型数据结构，它由多棵分解树组成。分解树是一种层次化表示，其中每个节点的子节点对应的样本复杂度都低于其父节点。分解森林用于搜索满足给定条件的短样本，这些样本随后可用于词法或语法推断。越简单的输入样本，推断效率越高。图1(b)展示了一个分解森林的示例。

每棵分解树都会动态扩展。没有必要也不切实际预先计算整棵树，因为这样会带来巨大的计算开销。相反，子节点会根据特定的分解策略按需生成。目前，我们设计了三种用于计算每个节点子节点的分解策略：二**分最大子序列删除**：将序列分成两部分，然后交替对第一部分或第二部分应用最大子序列删除。**最大子序列删除**：该策略优先删除较大的子序列，从较大的片段逐步处理到较小的片段，重点消除循环结构。**子序列替换**：用自身的一个子序列替换某个子序列，主要用于简化或删除递归结构。

搜索算法遍历每棵树，以确定其根节点是否满足给定条件。如果满足条件，则尝试遍历其子节点（此时分解树会动态扩展）。如果没有子节点满足条件，则返回当前节点的值。如果某个子节点满足条件，则在该子节点上递归搜索。

3.2 词法推断

词法推断的目的是将原始输入字符串转换为词元，从而能够在词元层面进行语法推断。这大幅降低了后续推断过程的计算成本。在许多编程语言中，输入可能非常复杂，包含数千个词元。在这种情况下，Kedavra 中默认使用的词法推断效率低下，因为它直接作用于原始输入样本。为应对这一可扩展性挑战，我们引入了一个专门的词法推断框架，旨在处理大型且结构丰富的输入。O

ur lexical inf推断过程 d结构消 ists of f我们的主in steps:

- **分词：**根据一组预定义的词法规则，将输入示例拆分为单个词元。
- **分类：** 根据上下文分布将分段后的词元归入词法类别。具有相同分布的词元归入同一类别。
- **字符级泛化：** 针对每个词元类，使用 L 算法推断有限自动机，以实现字符级泛化。推断出的自动机用作词元识别的词法规则。
- **词法验证：** 在原始输入示例上验证推断出的词法规则，并在检测到差异或错误时迭代改进规则。该框架利用分解森林 搜索能够实现词法推断的最短示例。

3.2.1 分词。**分词将字符级分析转换为词元级分析，从而缩短序列长度并提高效率**。这一过程首先应用一组预定义的词法规则，将每个输入字符串切分为候选词法单元。为进一步细化词法单元边界，我们会找出语法无关的字符，也就是说，这些字符可以自由地出现在词法单元边界处，且重复出现不会影响句法有效性。此类字符会从语法中排除。

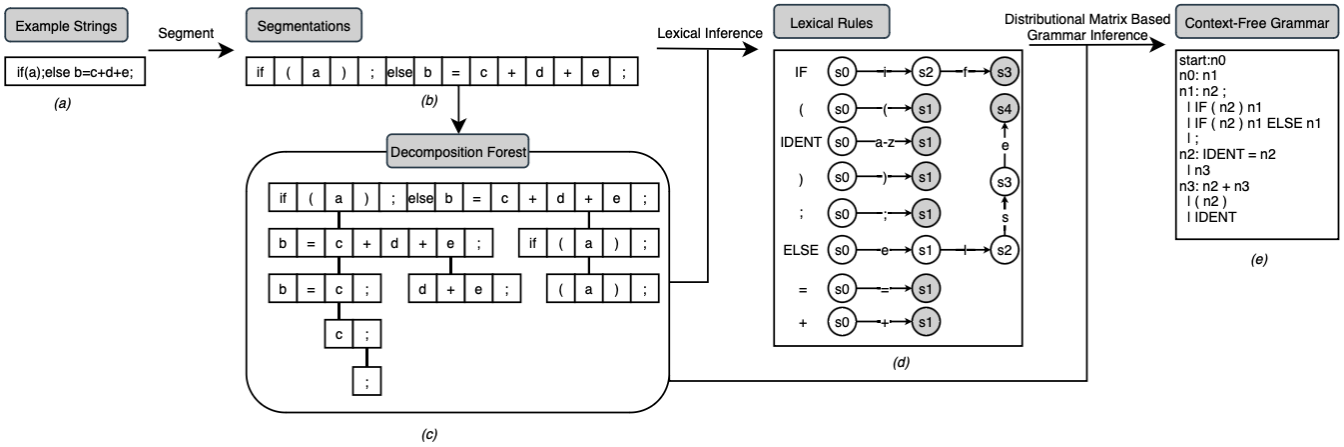


图1：Crucio 的工作流程

我们特别检查空格字符（ASCII 32）是否与语法无关。如果是，我们便利用空格字符在编程语言中通常充当分隔符这一特性来细化词法单元边界。具体而言，我们会在每个候选边界处迭代插入一个空格字符，并查询预言机以检查修改后的输入是否仍然有效。如果插入操作被接受，则确认该边界；否则，将相邻片段合并。

示例：请看图1(a)所示的 tinyc 输入。

```
if(a);else␣b=c+d+e;
```

基于预定义规则的初始分割会得到以下候选词法单元序列：

```
如果(a);否则␣b=C+d+e;
```

为了检查空格字符是否与语法无关，我们测试在同一位置插入额外空格是否会影响语法有效性。例如，以下变体——在 else 和 b 之间使用两个连续空格——是有效的：

```
if(a);否则␣␣b=c+d+e;√
```

这表明空格字符与语法无关，可用于确认或细化词元边界。接下来，我们测试在每个候选边界处插入空格。可以插入空格的边界被视为有效边界；如果无法插入空格，则需要合并相邻的两个片段。

3.2.2 词法单元分类。 **词法单元分类旨在识别哪些字符串属于同一词元类。** 例如，在 TinyC 中，a 和 b 都是标识符，因此属于同一类别。这种分类的目的是减少不同词法单元类型的数量，从而减少需要处理的词元种类，简化语法推断。将输入分割为原始词法单元后，下一步是把属于同一类别的词元分组。这通过决策树分类器实现，该分类器利用词元在不同上下文中的分布行为。

假设以下示例已经完成分解，并用于构建初始决策树。

```
;
```

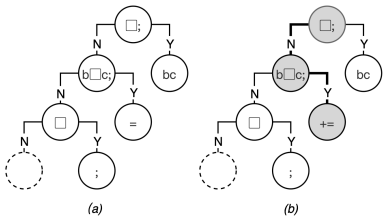


图2：决策树

```
c;b
=c;
```

接下来，我们分解一个新示例：b +c;

我们使用新示例进行词法推断。具体而言，我们遍历每个词元，将其插入决策树的根节点，并调整树以满足以下性质：对于每个叶节点，在给定上下文集 C 下，所有字符串都必须表现出一致的行为。对于每个内部节点，左子节点中的字符串不得在当前节点的上下文中被接受，而右子节点中的字符串必须被接受。这里，上下文集 C 指的是从当前选定的、用于词法推断的短示例中得到的所有上下文（而不是原始示例中的上下文）。词元的上下文定义为由其前面的所有词元和后面的所有词元组成的一对。例如，词元 c 的上下文是

□ b = □ = □ b =；以及 □ ;，我们将其表示为：Con(c) { ; ; }。

我们在算法1中给出了此步骤的伪代码。关键假设是，同一词元类中的词元具有相同的上下文分布，这意味着它们可以在相同的语法上下文中互换使用。基于这一原则，我们构建一棵决策树，其中内部节点表示基于上下文的区分，叶节点对应词元类别。

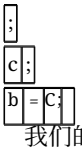
3.2.3 字符级泛化。 **字符级泛化增强了推断语法的通用性。** 在对词元进行聚类后，我们继续对每个词元类的字符级表示进行泛化。由于词元通常由

算法1 决策树更新过程

```
输入： 决策树 T，示例 E，词元值 V
输出： 更新后的决策树
函数ClassifyToken(T,E,V)
    T.v ← T.v ∪ {V}                          // 将值添加到根节点
    T_1 ←UpdateTree(T,E)                     // 调整决策树
    返回 T_1
结束函数
函数更新节点(T,E)
    如果 T 是叶节点 则
        C ←Con(E)
        c ←检查上下文(T.v,C)                 // 检查所有上下文的行为是否一致
        如果 C=无则
            返回
        结束如果
    else
        c ← T.c
    结束如果
    T.r.v ← {i ∈ T.v | O(i ⊙ c)}             // 使用上下文拆分节点。
    T.l.v ← V \ T.r.v
    更新节点(T.l)
    更新节点(T.r)
结束函数
```

正则语言描述，我们采用 L* 算法 [4] 为每个词元类推断一个确定性有限自动机（DFA）。

请看以下示例：



我们的决策树如图 2(a) 所示，其中 b 和 c 被归入同一类别，而 = 和 ; 则分别单独分类。决策树只能执行分类，无法泛化到未见过的数据，因此不能完整表示词法规则，也不能充当等价查询的预言机。我们需要将每个叶节点转换为自动机，以识别属于该词元的字符串。每个叶节点都包含一个字符串集合 S，其字符串空间由从根节点到该叶节点的所有上下文 C 确定：如果一个字符串在 C 中的行为与 S 中的字符串相匹配，则该字符串属于该词元。

为了泛化 b 和 c，我们使用 L* 算法。原始预言机 F(x) 只检查单个字符串的正确性。我们将其封装为词元预言机 G(x, C)，用于确定 x 在上下文 C 中的行为是否与 S 一致，从而检查 x 是否属于该词元。在此基础上，我们构造一个近似等价查询预言机 H(X)：它从假设 X 中进行采样，返回被 G 拒绝的字符串（估计 X-T），并检查 S 中的所有字符串是否都包含在 X 中（估计 T-X）。虽然无法保证完全等价，但经验表明，这足以推断出完整的词法规则。

使用 L* 进行泛化时，必须提供字符集：如果词元包含字母，则包含所有字母；如果包含数字，则包含所有数字。这样便可将 b,c 泛化到整个标识符空间。

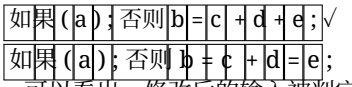
最后，利用为每个词元推断出的 DFA，我们可以构建一套完整的词法规则。这些词法规则与推断出的文法规则相结合，便构成了一个能够分析输入字符串的完整解析器。

算法 2基于分布矩阵的语法推断

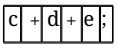
```
输入： 示例 E
输出： 上下文无关文法
函数InferGrammar(E)
    M ←BuildDistributionalMatrix(E)
    T ←构建平面解析树(E)
    while true 执行
        G ← 构建交换图(M, T)
        C ←获取极大团(G)
        C_merge ←选择最佳团(C, C_e)
        if C_merge = None 则
            跳出                             // 如果未找到可泛化的极大团，则终止推断。
        结束条件判断
        MergeBubbles(T, C_merge)             // 重复合并气泡以泛化语法
    结束 while
    返回 ToGrammar(T)
结束函数
```

3.2.4 词法验证。**由于词法规则是根据短示例推断得出的，因此在复杂输入上可能失效，因而需要进行验证。**为验证这些规则是否适用，我们首先使用当前的词法规则解析原始示例。随后，根据词法推断确定的对应等价项，替换原始示例中的每个词元。再使用预言机验证得到的修改后示例。如果检测到错误，我们便利用分解森林找出最短的失败示例，并将其作为反例纳入其中，以改进词法规则。

在图2所示的示例中，我们将 + 和 =视为属于同一个词元类。为验证此规则，我们在原始示例上进行测试，分别用 = 替换 + ，反之亦然。



可以看出，修改后的输入被判定器拒绝，这表明假定的等价性无效。随后，我们遍历分解森林，以定位一个更短的反例来展示这一失败，并将其纳入词法推断中，如以下示例所示。



3.3 基于分布矩阵的语法推断

本节将详细介绍基于分布矩阵的语法推断方法。该过程从一个空语法开始，通过从分解森林中识别并泛化短示例，逐步构建语法。这一迭代过程持续进行，不断纳入新的短示例，以泛化和扩展语法，直到覆盖所有训练示例。

算法 2 概述了我们方法的步骤。首先，我们根据给定的示例构造一个分布矩阵。该分布矩阵充当观察表 [1, 4, 12]。然后，我们为每个样本构建一棵扁平解析树，即根节点为开始符号、子节点为该样本所有词元的解析树。基于这些树，我们构造一个交换图，其中每个节点代表一个气泡，边表示两个气泡 可以互换。最后，我们在

图中找出极大团，选择一个包含广义示例的团，并合并相应的气泡。然后重复这一过程。

该方法可保证一阶交换正确性。值得注意的是，判定器仅在初始构建分布矩阵时查询一次。之后的所有步骤均无需判定器参与。

为了正式描述所提出的基于分布矩阵的语法推断，我们引入以下记号。子串指从输入字符串中提取的任意连续片段 s_i。例如，abc 的子串为 a、b、c、ab、bc 和 abc。上下文记作 $p \square q$，其中 p 和 q 是表示缺失片段前缀和后缀的字符串。例如，abc 的上下文为 $\square$、$\square c$、$a\square$、$\square bc$、$a\square c$ 和 $ab\square$。我们用记号 $a \odot (b\square c)$ 表示字符串 a、b c $a \odot b\square c = abc$ C 和的拼接，即也就是说</style，。对于上下文集和子串集合，我们定义{, $u \in S$}。

一阶交换正确性：设 $T = \{t_1, t_2, \ldots\ldots, t_n\}$ 为一组解析树。对于每棵解析树 t_i，令 $NT(t_i)$ 表示其中包含的非终结符集合。所有解析树中的非终结符全集记为 $N = \bigcup_{t_i \in T} NT(t_i)$。对于一个非终结符 x，令 Con(x) 表示它在这些解析树中出现的所有上下文构成的集合，令 Sub(x) 表示由 x 推导出的所有子序列构成的集合。将这些上下文与子序列组合后，可得到一组示例

$S =$ $x \odot$ x |{ $x \in N$ $\odot$ Sub( ) Con( ) }，其中 表示串接操作。若集合 S 中的所有示例均有效，则称这组解析树满足一阶交换正确性 。

3.3.1 分布矩阵的构建。**分布矩阵对预言机中的分布信息进行编码，从而无需额外查询即可进行语法推断**。分布矩阵的每一行 M 对应一个子串 s_i，每一列对应一个上下文 c_i。

对于每个子串 s_i 和上下文 c_j，我们定义 $M[i, j] = 1$ 当且仅当将 s_i 插入空位后得到一个有效字符串：g

$$M[i,j] = \begin{cases} 1 & \text{if oracle}(s_i \odot c_j) = \text{accept} \\ 0 & \text{otherwise} \end{cases}$$

为了构造 M，我们枚举输入示例中的所有子串 s_i，并通过从每个示例字符串中删除一个连续片段，枚举所有可能的上下文 $p \square q$。

示例： 假设我们已经对以下示例进行了分解，这些示例将用于语法推断。

;

a ;

(a) ;

对应这些子串和上下文的分布矩阵 M 如下所示。所有子串，以及通过从输入示例中删除连续子串得到的上下文，构成了分布矩阵的行和列。

子\Con	□	□;	(□);	a□	(a)□
;	1	0	0	1	1
a	0	1	1	0	0
a;	1	0	0	0	0
(a)	0	1	1	0	0
(a);	1	0	0	0	0

3.3.2 扁平解析树构建。**对于每个输入样本，我们都会构建一棵扁平解析树，作为语法推断的初始结构**。扁平解析树不包含层级结构，样本中的每个词元都被视为一个原子单元，并直接置于根节点之下。

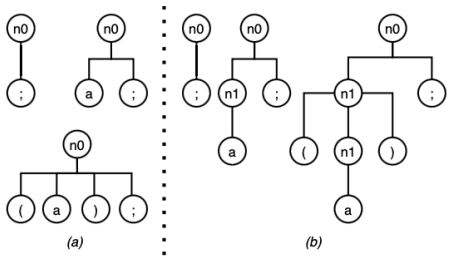


图 3：(a) 扁平解析树 (b) 广义解析树

扁平解析树的结构如图 3(a) 所示。这些树为后续的泛化和抽象奠定了结构基础。随后，我们将通过合并这些树中的气泡来推导新的语法结构。

3.3.3 构建交换图。**交换图用于识别可交换以便合并的气泡，从而实现语法泛化**。交换图是一个无向图，其中每个节点代表解析树中的一个气泡，每条边表示两个气泡是否可以交换。每个气泡对应一个连续的子树，该子树可能被抽象为一个非终结符。借助交换图，我们可以识别哪些气泡可以合并而不违反一阶交换正确性。我们仅依据分布矩阵构建交换图，不需要向预言机进行任何额外查询。

每个气泡都可以映射到一组上下文和一组子串。具体来说：

- 如果气泡是非终结符，则其子串定义为与该非终结符关联的所有气泡所含字符串的并集。否则，其子串就是样本中对应位置上的字符串。
- 如果气泡的父节点是非终结符，则其上下文定义为与该非终结符父节点关联的所有气泡的上下文的并集。否则，其上下文就是在原始样本中该位置观察到的上下文。

图4中每个气泡对应的上下文和子串列于下方。由于此处使用了扁平解析树，每个气泡恰好对应一个子串和一个上下文。

气泡	Sub	Con	气泡	Sub	Con
1	;	□	5	a	□;
2	;	a□	6	a;	□
3	a	(□);	7	(a)	□;
4	;	(a)□	8	(a);	□

通过查询分布矩阵，我们检查两个气泡的所有子串是否都可以置于其各自关联的所有上下文中。如果可以，则在它们之间添加一条无向边，构成交换图。构建出的交换图如图5所示。

当一组节点构成一个团时，即其中每对节点都相连，因此可以将它们安全地合并为一个共享的非终结符，而不会违反一阶交换正确性。

3.3.4 气泡合并。**为了推广语法，我们旨在将可交换的气泡 合并到共享的非终结符中**。该过程始于识别 极大团 交换图中的极大团。每个

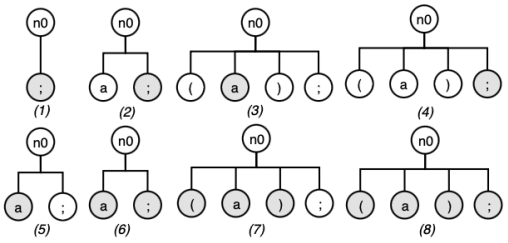


图4：气泡。每个气泡对应树中若干个阴影节点。

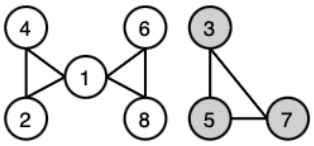


图5：交换图。阴影节点表示我们选出的极大团。

极大团表示一组气泡，其中任意一对气泡都相互连接，表明它们满足一阶交换正确性。

我们按大小降序枚举所有极大团。一旦找到一个极大团，其上下文和子串的组合能够生成一个可泛化当前语法的样本（即，一个尚未被该语法覆盖的样本），如图5所示，我们就合并该团。合并结果如图3(b)所示。每次合并后，都会重建交换图，并继续选择后续气泡。如果找不到有效的极大团，泛化过程就会终止。不过，这一过程只能确保一阶交换正确性；生成的语法可能无法达到完美的精确率。

4 评估

我们所提出方法的最终目标，是从给定程序集合中自动推断形式文法，尤其针对具有大量复杂文法规则的编程语言。为评估是否实现了这一目标，我们将 Crucio 应用于八个微基准测试、三个宏基准测试，以及五个由复杂编程语言编写的程序组成的新基准测试，并将其性能与最先进的方法进行了比较。随后，我们对结果进行了深入分析，以回答以下三个研究问题。

研究问题1 文法质量：就接受有效输入并拒绝无效输入的能力而言，我们的方法推断出的文法有多准确?

研究问题2 效率：与以往工具相比，在大型复杂数据集上的运行时间方面，我们的方法效率如何?

研究问题3 可读性：推断出的文法是否简洁?

4.1 实验设置

为了评估 Crucio 在复杂语法上的有效性和效率，我们将 Crucio 应用于一个构建的基准数据集，其中包含来自高级编程语言的16种语法，既有简单语法，也有复杂语法

表 1：数据集的文法大小：NT/T = 唯一的（非）终结符；A =文法规则（备选项）；l(A) = 平均规则长度；S =规则长度之和。

Name	NT	T	A	l(A)	S
算术	4	8	9	1.56	14
math	6	14	16	2.06	33
fol	8	16	18	2.89	52
json	7	12	17	1.76	30
lisp	5	7	11	2.27	25
海龟	4	10	8	3.12	25
while	4	10	8	3.12	25
xml	7	10	19	2.58	49
curl	-	-	-	-	-
tinyc	9	13	18	2.28	41
MiniC	17	21	35	2.23	78
c	91	113	320	1.88	601
cpp	228	126	624	1.75	1,095
java	210	124	534	1.72	917
lua	45	64	135	1.90	257
rust	253	115	706	1.71	1,205

。具体而言，我们纳入了 Kedavra 的所有原始数据集[1]，，它代表了语法推断领域的最先进水平。由于 Kedavra 基准中的语法相对简单，我们为数据集增加了五种额外语法，分别对应 C、Lua、Java、Rust 和 C++，这些语法的选择旨在体现真实世界编程语言的复杂性。这五种语法对应的训练程序和测试程序取自一个真实世界数据集 算法[2]。这些语法的预言机是使用 *ANTLR4 grammars-v4*[3] 仓库中的语法构建的。由于编程语言中的注释通常易于处理且格式标准化，并且注释一般不会影响代码在语法上的正确性，因此我们从数据集中移除了所有注释。

因此，用于实验的总体数据集包含涵盖16种语法的31个训练数据集。表1总结了语法大小：复杂语法平均包含165个非终结符号，而Kedavra的基准测试仅包含7个非终结符号。数据集统计信息进一步汇总于表2。对于复杂语法（即 C、Lua、Java、Rust 和 C++），数据集平均包含98个示例，每个示例的平均长度为2008个字符。对于简单语法，数据集平均包含29个示例，每个示例的平均长度为63个字符。这些差异反映了语法复杂度和数据集规模的变化，为评估 Crucio 在不同语法复杂度下的表现提供了基础。该基准数据集已在 GitHub[4] 上公开，供未来使用。

所有实验均在一台配备 Intel(R) Core(TM) i9-14900F CPU 和 64GB RAM 的 Linux 服务器上进行，每项实验的超时阈值为 48 小时。对于每种方法，所有语言实验（11 种简单语言和 + 5 种复杂语言）均在同一台服务器上同时执行，并为每种方法采用相同的设置，以确保公平性。

[1] https://github.com/Sinpersrect/kedavra [2] https://github.com/thealgorithms [3] https://github.com/antlr/grammars-v4 [4] https://github.com/Sinpersrect/gi-benchmarks

表2：示例字符串 S 及其平均字符大小；# = 个程序。

	R0		R1		R2		R5		Test	
	#	avg	#	avg	#	avg	#	avg	#	avg
算术	17	2.3			n/a				1,000	65.9
fol	36	14.5			n/a				1,000	106.5
math	62	5.5			n/a				1,000	72.1
json	71	3.9	30	11.7	30	8.6	n/a		1,000	10.8
lisp	26	2.5	30	79.2	30	24.8	n/a		1,000	73.2
海龟	33	7.7	35	41.1	35	25.6	n/a		1,000	34.6
while	10	15.5	30	171.4	30	217	n/a		1,000	215.2
xml	40	11.6	20	27.8	20	27.5	n/a		1,000	25
curl	25	20.7	25	22.1	25	22.0	n/a		1,000	22.2
tinyc	25	80.5	25	96.2	25	86.4	10	514	1,000	94.2
MiniC	10	107	n/a		n/a		n/a		1,000	267.3
c	81	1,175			n/a				326	1,488
cpp	70	2,493			n/a				283	2,610
java	254	1,730			n/a				1,019	1,738
lua	30	1,822			n/a				120	1,276
rust	57	2,821			n/a				232	2,776

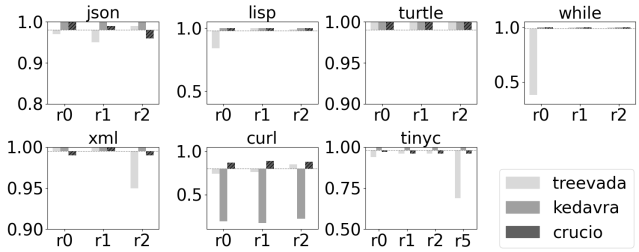


图6：Treevada、Kedavra 和 Crucio 在每个数据集（R0、R1、R2、R5）上 10 次运行的平均 F1 分数。

表 3：在相同的示例字符串上重新运行 Treevada（左）、Kedavra（中）和 Crucio（右）；Kedavra/Crucio 的数值为 10 次运行的平均值；f1 =F1 分数；± = 标准差；粗体表示 =Crucio 优于或等于 Treevada 和 Kedavra。

	Treevada			Kedavra			Crucio		
	p	r	f1	p	r	f1	p	r	f1
算术	1	1	1	1.0±.0	1.0±.0	1.0±.0	**1.0±.0**	**1.0±.0**	**1.0±.0**
math	1	.47	.64	.97±.04	.76±.2	.84±.15	.79±.02	**.9±.02**	**.84±.02**
fol	1	.53	.69	1.0±.01	.71±.25	.81±.16	**1.0±.0**	**1.0±.0**	**1.0±.0**
json	1	.94	.97	1.0±.0	.96±.02	.98±.01	**1.0±.0**	**1.0±.0**	**1.0±.0**
lisp	.99	.73	.84	1.0±.0	1.0±.0	1.0±.0	**1.0±.0**	**1.0±.0**	**1.0±.0**
海龟	1	1	1	1.0±.0	1.0±.0	1.0±.0	**1.0±.0**	**1.0±.0**	**1.0±.0**
while	1	.24	.38	1.0±.0	1.0±.0	1.0±.0	**1.0±.0**	**1.0±.0**	**1.0±.0**
xml	1	1	1	1.0±.0	1.0±.0	1.0±.0	**1.0±.0**	.99±.0	.99±.0
curl	.95	.6	.74	.91±.12	.12±.14	.19±.2	.85±.01	**.88±.01**	**.87±.01**
tinyc	.98	.9	.94	1.0±.0	1.0±.0	1.0±.0	.93±.02	**1.0±.0**	.97±.01
MiniC	.96	.48	.64	.89±.03	.51±.0	.65±.01	**.98±.0**	**1.0±.0**	**.99±.0**
c	-	-	-	-	-	-	**.65±.04**	**.85±.03**	**.73±.04**
cpp	-	-	-	-	-	-	**.62±.02**	**.82±.02**	**.7±.02**
java	-	-	-	-	-	-	**.75±.04**	**.93±.01**	**.83±.03**
lua	-	-	-	-	-	-	**.59±.05**	**.83±.05**	**.69±.04**
rust	-	-	-	-	-	-	**.42±.04**	**.78±.03**	**.54±.03**

4.2 研究问题1：文法质量

我们的第一个研究问题着重于评估 Crucio 生成的语法质量，以及它与现有工具相比表现如何。与以往研究一致，我们采用精确率、召回率和 F1 分数来衡量语法质量。同时，我们引入一种更可靠的精确率测量方法（即 交换精确率），以克服现有方法中基于采样的精确率测量（即 采样精度）的关键局限，具体而言，

这些方法无法考虑结构偏差，也对样本长度分布较为敏感。

现有的所有方法 Arvada、Treevada 和 Kedavra 都使用基于采样的估计来评估精确率。然而，这种方法存在两个主要局限：

• 结构偏差：采样精度 衡量的是使用采样算法从语法生成的有效示例所占的比例。然而，这一指标可以通过结构操纵。例如，可以刻意构造一种语法，其中大多数产生式规则都旨在生成一小组真实示例。如表 4 所示，在这种情况下，采样过程几乎总是生成这几个示例，因而采样精度为 1。若再加入少量过于宽泛的规则（例如匹配任意字符串），召回率也能达到 1。实际上，尽管这类语法对训练数据严重过拟合，并纳入了许多无效泛化，但按照 Arvada、Treevada 和 Kedavra 所使用的评估方案，它们仍可能看似取得 F1 分数为 1。因此，这些语法并不能代表底层语言有效或有意义的抽象。

• 样本长度偏差：在采样过程中，生成示例的长度通常无法控制，并且会受到语法结构本身的影响。倾向于生成较短示例的语法，更容易产生与真实输入相匹配的字符串，从而导致采样精确率虚高。这表明，采样精确率不仅受到语法表达能力的干扰，也受到生成样本长度分布的影响，因此无法作为评估语法质量的可靠客观指标。对于复杂语法，采样过程尤其容易生成极长且不正确的样本，进一步降低这一指标的可靠性。

为了解决这一问题，我们提出了一种新的精确率评估方法：交换精确率。

交换精确率：给定一些示例 $E = \{e_1, e_2, \ldots\ldots, e_n\}$，我们可以为每个示例构建一棵解析树，从而形成集合 $T = \{t_1, t_2, \ldots\ldots, t_n\}$。解析树 t_i 中所有非终结符的集合记为 $NT(t_i)$。所有非终结符构成的全集记为 $N = \bigcup_{t_i \in T} NT(t_i)$。

我们将 $\mathrm{Con}(x)$ 表示为所有解析树中出现非终结符号 x 的全部上下文的集合，将 $\mathrm{Sub}(x)$ 表示为由 x 派生的所有子序列的集合。通过组合这些上下文和子序列，我们得到示例集合 $S = \bigodot_{x \in N} x \odot x \mid \{\mathrm{Sub}(\,)\ \mathrm{Con}(\,)\}$，其中 $\odot$ 表示一种组合运算。

交换精确率定义如下：

$$p_{\text{swap}} = \frac{|\{s \in S \mid s \in L(G)\}|}{|S|} \tag{1}$$

验证 S 中的每个示例都非常耗时。因此，如果 $|S| > 1000$，我们会从 S 中随机选择 1,000 个示例进行评估。为评估交换精确率，我们使用测试集中能够由当前语法解析的子集。

这通过评估在测试集分布中识别等价子串的正确性，为语法精确率提供了更可靠的衡量方式。因此，在接下来的实验中，我们使用交换精确率 而非采样精度，以更好地反映语法质量。我们将所有指标定义如下：

表 4：我们构造的无效文法上的采样精确率和交换精确率表现。

Name	p_{sample}	p_{swap}	r	$f1_{sample}$	$f1_{swap}$
算术	1	0.089	1	1	0.16
math	1	0.063	1	1	0.12
fol	1	0.024	1	1	0.05
json	1	0.035	1	1	0.07
lisp	1	0.018	1	1	0.04
海龟	1	0.028	1	1	0.05
while	1	0.009	0.987	0.99	0.02
xml	1	0.013	1	1	0.03
curl	1	0.894	1	1	0.94
tinyc	1	0.024	1	1	0.05
minic	1	0.02	0.972	0.99	0.04
c	1	0.035	0.89	0.94	0.07
cpp	1	0.013	0.93	0.96	0.03
java	1	0.05	0.98	0.99	0.1
lua	1	0	0.96	0.98	0
rust	1	0.035	0.8	0.89	0.07

交换精确率：此处，交换精确率 定义为：在成功解析的测试集中，将解析树中对应于相同非终结符的子串互换后，所得字符串中符合语法的比例。召回率：我们将推断语法编译为解析器，并统计该解析器接受给定测试程序的数量。解析时间超过 60 秒的程序视为解析失败。*F1* 分数：F1 分数是精确率与召回率的调和平均数。

结果。我们在 R0 数据集上的实验结果如表 3 所示，该表展示了 Crucio 与另外两种最先进的工具之间的比较。在此，我们选择 Treevada[5] 和 Kedavra[21]作为基线方法，因为它们被认为是该领域中最先进且公开可用的语法推断工具。

总体而言，如表 3 所示，在 14 个语法数据集（即 arith、math、一阶逻辑、JSON、Lisp、海龟、while、curl、minic、C、C++、Java、Lua 和 Rust）上，Crucio 取得了等于或高于 Treevada 和 Kedavra 的 F1 分数，这表明我们的工具在大多数数据集上的表现更好。例如，Crucio 在 arith、一阶逻辑、JSON、Lisp、海龟和 while 上取得了完美分数（精确率、召回率和 F1 分数均为 1.0 ± 0.0）。即使对于 C、C++、Java、Lua 和 Rust 等复杂的程序输入数据集，Crucio 仍保持了出色的性能，平均 F1 分数为 0.698，而 Treevada 和 Kedavra 未能在这些数据集上推断出语法。

具体而言，Treevada 在 C、C++、Java、Lua 和 Rust 上分别运行 103 小时、206 小时、56 小时、126 小时和 337 小时后，因内存不足而终止。相比之下，Kedavra 在 Java 和 Rust 上分别运行 70 小时和 66 小时后终止，这是因为峰值内存使用量耗尽后，Ubuntu 系统会终止内存消耗最大的进程；而在 C、C++ 和 Lua 上运行超过 360 小时后，Kedavra 仍未完成。这些结果凸显了 Crucio 在生成高质量语法方面的有效性。

例外包括 XML 和 tinyc 数据集：在 XML 上，Crucio 的 F1 分数为 0.99，略低于 Treevada 和 Kedavra 的 1.0；在 tinyc 上，其表现也逊于 Kedavra 的满分 1.0。XML 上较低的召回率（0.99）是由于词法推断使用了 L* 算法，该算法在上下文约束下推断自动机，因此得到的模型更严格，但泛化能力较弱。对于 tinyc，语法的构建旨在确保一阶交换正确性，而非通过采样进行验证，因此其精确率低于 Treevada 和 Kedavra。关于精确率差异的详细讨论见第5节。尽管有这些少数例外，Crucio 仍持续优于最先进工具，证明了其生成高质量语法的有效性。

除了评估语法质量，还需要考察 Crucio 能否在不同数据集上保持稳定的性能。为此，我们进一步分析了同一语法在多个数据集（R0、R1、R2 和 R5）上的 F1 分数，如图6所示。该图展示了三种工具在七个语法数据集上的 F1 分数。柱形与水平参考线之间的偏差反映了 F1 分数的波动。请注意，每个子图中都手动添加了一条水平线，以便直观呈现各推理算法在不同数据集上的 F1 分数变化情况。

Crucio 的性能波动较小（即柱形到水平轴的距离较短），而 Treevada 对不同数据集特征更为敏感。对于 Lisp、海龟和 while 等语法，Crucio 在所有数据集上的 F1 分数始终达到 1.0。相比之下，Treevada 在 json、lisp、while、xml 和 tinyc 等数据集上的 F1 分数波动较大。这表明，对于由相同底层语法衍生的不同数据集，Crucio 的性能稳定性与 Kedavra 不相上下。仔细比较 Crucio 与另外两种方法的设计后，我们认为，Crucio 的 F1 分数稳定性更高，主要归功于其设计：与 Kedavra 一样，它对分解示例进行推断，但它使用的分解森林使其泛化能力远强于以往方法。

研究问题1的回答： Crucio 在简单数据集上展现出明显优势：与 Treevada 和 Kedavra 相比，其召回率分别提升了 1.37× 和 1.19×，F1 分数分别提升了 1.21× 和 1.13×，而精确率仅略有下降（分别为 0.97× 和 0.98×）。更值得注意的是，Crucio 是唯一能够处理 C、C++、Java 和 Rust 等复杂真实世界语言的方法，并成功推断出语法；其他最先进方法要么无法终止，要么根本无法运行。

4.3 研究问题2：效率

虽然我们已证明推断出的语法有效，但效率仍是一个关键因素，体现了推断过程的速度和资源需求。在本研究问题中，我们通过测量 Crucio 的计算资源消耗和处理时间来评估其效率。为确保公平比较，我们报告的指标与最先进研究中使用的指标一致：**运行时间**：执行工具所需的时间。**预言机时间**：我们测量预言机检查过程中的耗时。**预言机调用次数**：预言机调用的次数。**峰值内存**：我们使用 Linux 的“time”命令测量语法推断过程中的峰值内存使用量。

表5展示了 Treevada、Kedavra 和 Crucio 在 R0 数据集上的资源使用情况。总体而言，Crucio 对每个程序进行语法推断平均需要 0.12 千秒，比 Treevada（每个程序 0.17 千秒）更快

表5：Treevada 与 Kedavra 在 R0 上运行 10 次的平均结果； t **= 运行时间；** t_O **= 预言机时间；** q **= 发送至预言机的查询；** m **=峰值内存使用量；** ± **= 标准差；粗体 = 表示 Crucio 优于 Treevada 与 Kedavra。**

	Treevada				Kedavra				Crucio			
	$t[ks]$	$t_O[ks]$	$q[k]$	$m[GB]$	$t[ks]$	$t_O[ks]$	$q[k]$	$m[GB]$	$t[ks]$	$t_O[ks]$	$q[k]$	$m[GB]$
算术	.00±.0	.00±.0	0.37	0.02	.00±.0	.00±.00	.58±.02	.08±.01	**.00±.0**	**.00±.0**	1.43±1.3	.07±.0
math	.01±.0	.01±.0	6.33	0.02	.84±2.1	.06±.03	17.62±8.91	.08±.04	.10±.0	.09±.0	35.66±10.2	.09±.0
fol	.04±.0	.03±.0	16.25	0.03	.28±.2	.20±.01	57.00±4.20	.11±.10	.49±.0	.47±.0	181.67±5.3	.15±.0
json	.01±.0	.00±.0	3.26	0.02	.06±.1	.01±.00	4.00±1.13	.11±.05	.05±.0	.05±.0	20.39±11.1	.08±.0
lisp	.00±.0	.00±.0	1.84	0.02	.01±.0	.01±.00	2.16±.03	.12±.03	.01±.0	.01±.0	2.13±.2	.08±.0
海龟	.01±.0	.01±.0	5.11	0.02	.02±.0	.02±.00	4.53±.19	.03±.00	**.01±.0**	**.01±.0**	**4.31±.1**	.08±.0
while	.01±.0	.01±.0	3.23	0.02	.18±.0	.14±.03	42.96±8.53	.10±.05	.12±.0	.12±.0	46.74±.0	.09±.0
xml	.05±.0	.02±.0	9.48	0.03	.23±.0	.20±.03	62.27±8.28	.09±.00	.14±.0	.13±.0	56.06±.7	.09±.0
curl	.09±.0	.07±.0	23.39	0.04	.11±.0	.09±.02	22.27±3.83	.03±.00	.22±.0	.20±.0	45.79±3.4	.09±.0
tinyc	.65±.0	.05±.0	78.02	0.17	.06±.0	.03±.00	19.83±.46	.03±.00	**.02±.0**	**.01±.0**	**8.30±.3**	.08±.0
MiniC	.95±.0	.42±.0	77.52	0.12	.26±.1	.13±.02	32.91±3.92	.10±.11	**.15±.0**	**.13±.0**	44.91±.6	**.10±.0**
c	-	-	-	-	-	-	-	-	**10.04±1.4**	**7.42±1.5**	**988.45±227.5**	**.90±.2**
cpp	-	-	-	-	-	-	-	-	**84.94±11.6**	**49.14±9.6**	**1349.21±505.5**	**4.46±.4**
java	-	-	-	-	-	-	-	-	**65.61±15.9**	**16±.7**	**1778.05±132.5**	**2.34±.1**
lua	-	-	-	-	-	-	-	-	**5.14±.8**	**1.86±.7**	**402.23±206.1**	**3.47±.5**
rust	-	-	-	-	-	-	-	-	**78.86±5.9**	**40.57±1.4**	**1432.73±108.5**	**3.30±.7**

，也比 Kedavra（每个程序 0.19 千秒）更快。正如 Arefin [5] 所示，提高语法质量往往会降低效率。这解释了为什么 Crucio 在数学、一阶逻辑、JSON、Lisp、while、XML 和 curl 数据集上耗时更长：Crucio 具有更强的泛化能力，会为语法推断分解出更多示例，计算分布矩阵，并采用 L* 算法进行词法推断；与以往方法相比，该算法更精确，但需要更多预言机查询。不过，我们认为，效果与效率之间的这种权衡不可避免。此外，值得注意的是，在数学、一阶逻辑、JSON、Lisp、while、XML 和 curl 数据集上，Crucio 与 Treevada/Kedavra 之间的时间差异相对较小（仅几百秒），完全可以接受。

在处理 tinyc、minic、C、C++、Java、Lua 和 Rust 等更复杂的数据集时，我们的方法仍保持着具有竞争力的速度。这主要得益于分解森林的使用，它避免了对已经覆盖完整语法的示例进行冗余分解，同时还在分解示例上应用了词法推断。此外，Crucio 不依赖采样来确保语法正确性；相反，它在泛化与过度泛化之间取得了良好平衡。这表明，即使对于某些语法而言运行时间略长，Crucio 凭借其出色的语法质量和相对较低的总体时间成本，仍然具有实用性和可用性。

关于内存使用情况，我们观察到 Crucio 在所有数据集上消耗的内存都更多。然而，即使对于最复杂的数据集——C、C++、Java、Lua 和 Rust，内存使用量仍低于 5GB，这是可以接受的。内存消耗增加主要源于 Crucio 更强的泛化能力，以及其在词法推断中使用 L* 算法；后者增加了预言机查询的数量，并需要缓存这些查询的结果。

研究问题2的答案： Crucio 在复杂的真实语言（即 C、C++、Java、Lua 和 Rust）上展现出很强的可扩展性，成功推断出语法，平均运行时间为 13.6 小时，而最先进方法在 48 小时内仍未能完成。在较小的数据集上，Crucio 相较于 Treevada 和Kedavra，分别将平均运行时间缩短了 1.56× 和 1.39×，但由于其更精确的词法推断，以及分解森林所支持的更强泛化能力。

4.4 研究问题3：语法可读性

在程序理解等上下文中，可读性至关重要，因为语法必须便于人类理解。另一个重要用例涉及语义探索：过于复杂的语法通常包含大量歧义，这可能导致下游分析困难。在本研究问题中，我们使用以下指标评估推断语法的可读性：**文法大小**：我们计算唯一非终结符和终结符的数量，以及产生式的数量和每个产生式的长度（即，终结符和非终结符右部序列的长度）。文法大小是所有产生式长度之和。**解析时间和内存**：我们评估使用由 Kedavra 或 Crucio 推断的语法生成的解析器解析测试程序时，每个程序的平均耗时和所需的峰值内存使用量。

还需要注意的是，Crucio 不使用上下文无关文法来描述词法规则，而是单独处理词法分析。相比之下，Treevada 使用上下文无关文法统一表示词法规则和文法规则。因此，在整体文法大小或结构方面，无法直接将 Crucio 与 Treevada 进行比较。因此，表6中的比较仅关注语法，因为 Crucio 和 Kedavra 都基于相同的上下文无关文法假设运行。

表6报告了 Kedavra 和 Crucio 推断出的语法的大小及解析性能。对于小型和中型数据集（即 arith、math、fol、json、lisp、turtle、while、xml、curl、tinyc、minic），这两种最先进的工具生成的语法在非终结符数量（NT）、规则数量（A）、平均规则长度和规则总大小方面相近。在 mathexpr 和 tinyc 上，Crucio 生成的语法包含更多非终结符；而在 arith 和 turtle 上，其生成的语法包含更少的非终结符。这些差异反映了语法构造策略的不同；值得注意的是，Crucio 不会进行语法简化。

表6：R0 上的语法大小和解析性能：10 次运行中推断出的语法的平均值；NT/T = 唯一（非）终结符；A = 规则（备选项）；l(A) = 平均规则长度；S =规则长度总和； t_P = 解析一个测试程序所需的时间（ks）；m_P = 解析时的峰值内存（GB）；粗体 = 表示 Crucio 优于 Kedavra。

	Kedavra							Crucio						
	NT	A	l(A)	S	T	t_P	m_P	NT	A	l(A)	S	T	t_P	m_P
算术	1	3	2.3	7	4	.06	.22	2	4	2	8	4	**.02**	**.19**
math	7	18	2.1	39	11	.09	.12	13	36	1.7	61	15	**.04**	.16
fol	4	16	3.8	60	18	.02	.11	5	18	3.9	70	19	.04	.27
json	5	17	2.1	36	13	0	.04	6	18	2	**35**	13	.01	.07
lisp	2	5	3	15	5	.03	.14	4	9	2.3	21	6	**.01**	.17
海龟	3	7	3.3	23	9	0	.06	**2**	**7**	4	28	9	.01	.08
while	5	14	3.8	52	17	.01	.18	**5**	**14**	3.7	**51**	17	.05	.34
xml	5	17	3.5	59	12	.01	.05	**5**	**17**	2.8	**48**	12	**.01**	.08
curl	15	24	1.6	39	8	.03	.03	**10**	26	1.6	42	10	**0**	.1
tinyc	3	15	3.1	47	14	.02	.12	6	19	2.8	52	14	**.02**	**.11**
MiniC	12	30	2.7	80	17	.01	.04	**11**	34	2.7	91	17	.06	.58
c	-	-	-	-	-	-	-	**46**	**147**	1.9	**286**35		**4.31**	**1.39**
cpp	-	-	-	-	-	-	-	**77**	**248**	1.9	**461**48		**14.86**	**2.44**
java	-	-	-	-	-	-	-	**89**	**280**	2.2	**610**53		**8.47**	**3.06**
lua	-	-	-	-	-	-	-	**34**	**122**	1.7	**212**31		**10.18**	**1.94**
rust	-	-	-	-	-	-	-	**75**	**260**	1.9	**490**56		**14.59**	**2.19**

我们还观察到，解析效率在不同数据集上表现不一。具体而言，Crucio 在 arith、JSON 和 Lisp 上的解析时间和内存用量较低，而在一阶逻辑、while 和 MiniC 上，解析时间和峰值内存占用较高。对于大型语言，解析的计算开销仍然很高，但切实可行；例如，解析一个 Java 程序需要 8.47 秒和 3.06 GB 峰值内存。

Crucio 的主要优势在于能够扩展到大型的真实世界编程语言和 Rust，而 Kedavra 无法为这些语言生成任何语法。这种可扩展性大幅拓展了语法推断的实际适用范围。

对研究问题3的回答： 与 Kedavra 相比，Crucio 生成的语法中，非终结符数量为其 1.11× 倍，规则数量为其 1.21× 倍，规则总长度为其 1.11× 倍，这表明二者的整体复杂度和可读性相近。值得注意的是，这些结果表明，尽管 Crucio 具有更强的泛化能力和可扩展性，它仍能保持相当的文法简洁性。

5 讨论与未来工作

各模块的效果。从理论角度来看，基于分布矩阵的推断是影响最大的一项。它引入了确定性有限正确性验证，而不是像现有工作 Kedavra 那样依赖不确定且不完整的采样，从而确保语法绝对正确，并在利用分布信息构建简洁且泛化能力强的语法的同时，提供有界的正确性保证。与 Kedavra 相比，它只需枚举极大团（而非所有子集），使遍历在实践中具备计算可行性。另一方面，分解森林缩小了问题规模，使得能够计算更多用于构建语法的信息，从而确保计算可行性。它支持对一个样本进行重复分解，直到语法完整纳入该样本中存在的所有规则，从而避免 Kedavra 中出现的规则丢失问题；它还可用于词法推断，大幅提升效率。尽管这两个组成部分都不可或缺，但基于分布矩阵的推断提供了核心进展，推动 Crucio 展现出优越性。

提升解析效率。Crucio 推断出的复杂语言语法解析效率较低：解析单个输入可能需要几秒，甚至十几秒。相比之下，解析较简单语言的输入通常只需几十毫秒。效率低下主要源于两个因素：（1）输入本身固有的复杂性；（2）推断语法中的歧义。尽管输入复杂性无法避免，未来可以改进解析树上泛化过程中气泡的合并方式，以减少文法歧义并提升解析效率。

更准确的语法。Crucio 确保推断过程中构建的解析树满足一阶交换正确性。然而，这并不意味着推断出的语法完全正确。一方面，分布矩阵仅捕捉一阶交换信息，无法反映高阶结构，而由于计算复杂度高，计算这些结构并不可行。另一方面，尽管泛化过程中构建的解析树能够确保观测样本满足一阶交换正确性，但文法歧义可能导致实际解析时生成不同的解析树，从而难以保证交换操作的正确性。此外，由于语法是对潜在无限字符串集合的压缩表示，因此很难进行完整验证。总的来说，语法推断中的泛化与精确率之间存在权衡：尽管 Crucio 的泛化能力强于现有方法，但这可能以精确率略有降低为代价。

6 结论

我们提出了 Crucio，这是一种新颖的语法推断方法，能够有效应对为 C、C++ 和 Java 等复杂编程语言推断语法时面临的可扩展性挑战。借助分解森林和分布矩阵，Crucio 的表现显著优于现有最先进的方法，包括 Arvada、Treevada 和 Kedavra；这些方法在处理上述语言时，无法在实际可接受的时间内得出结果。尽管复杂编程语言的解析效率仍是一项尚待解决的挑战，Crucio 通过生成既有效又易于人类阅读的语法，取得了重大进展。

致谢

本研究获得广东省自然科学基金（2025A1515011946）和鹏城实验室重大重点项目（PCL2023A06-4）的资助。

参考文献

[1] P Adriaans. 1999年。根据简单分布学习浅层上下文无关语言。未发表手稿 (1999 年)。 [2] Ziyad Alsaeed 和 Michal Young. 2023 年。通过基于语法的搜索更快地找到短而慢的输入。载于*ACM SIGSOFT* 软件测试与分析国际研讨会第*32*届会议论文集。1068–1079。

[3] Dana Angluin. 1981年。关于识别正则语言所需查询数量的说明。《信息与控制》51, 1 (1981年), 76–87。 [4] Dana Angluin. 1987年。通过查询和反例学习正则集合。《信息与计算》75, 2 (1987年), 87–106。 [5] Mohammad Rifat Arefin, Suraj Shetiya, Zili Wang, 和 Christoph Csallner. 2024年。快速确定性黑盒上下文无关文法推断。载于 IEEE/ACM 第46届软件工程国际会议论文集。1–12。[6] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, 和 Daniel Teuchert. 2019年。NAUTILUS：利用文法搜寻深层漏洞。。载于 NDSS。 [7] Osbert Bastani, Rahul Sharma, Alex Aiken, 和 Percy Liang. 2017年。合成程序输入文法。载于 ACM SIGPLAN 编程语言设计与实现会议第38届会议论文集。ACM, 95–110。 [8] Jean-Philippe Bernardy. 2018年。循环神经网络能否学习嵌套递归?《语言技术中的语言学问题》16 (2018年)。 [9] Lukas Bettscheider 和 Andreas Zeller. 2025年。从代码中通过符号解析推断输入文法。arXiv 预印本 arXiv:2503.08486 (2025 年)。 [10] Timo Blazytko, Matthew Bishop, Cornelius Aschermann 等. 2019年。GRI-MOIRE：在模糊测试的同时合成结构。载于第28届 USENIX 安全研讨会（USENIX Security 19）。1985–2002。 [11] Juan Caballero, Heng Yin, Zhenkai Liang, 和 Dawn Song. 2007年。Polyglot：通过动态二进制分析自动提取协议消息格式。载于第14届 ACM 计算机与通信安全会议论文集。317–329。 [12] Alexander Clark 和 Ryo Yoshinaka. 2016年。上下文无关文法和多重上下文无关文法的分布式学习。《文法推断专题》(2016年), 143–172。 [13] A. Cremers 和 S. Ginsburg. 1975年。上下文无关文法形式。《计算机系统科学杂志》11, 1 (1975年), 86–117。 [14] Patrice Godefroid, Adam Kiezun, 和 Michael Y Levin. 2008年。基于文法的白盒模糊测试。载于第29届 ACM SIGPLAN 编程语言设计与实现会议论文集。206–215。 [15] Rahul Gopinath 和 Andreas Zeller. 2019年。构建快速模糊测试器。arXiv 预印本 arXiv:1911.07707 (2019年)。 [16] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, 和 Gavin Bierman. 2022年。《Java 语言规范》：Java SE 19 版。Oracle。 [17] Matthias Höschele 和 Andreas Zeller. 2016年。从动态污点中挖掘输入文法。载于第31届 IEEE/ACM 自动化软件工程国际会议论文集（ASE）。ACM, 720–725。 [18] Xiaojin Jia 和 Gang Tan. 2024年。V-Star：从程序输入中学习可见下推文法。《ACM 编程语言论文集》8, PLDI (2024年), 2003–2026。 [19] Neil Kulkarni, Caroline Lemieux, 和 Koushik Sen. 2021年。学习高度递归的输入文法。载于 2021年第36届 IEEE/ACM 自动化软件工程国际会议（ASE）。IEEE, 456–467。 [20] Thomas Köppe 等. 2022年。工作草案：C 编程语言标准 ++。技术报告 N4917。ISO/IEC。 [21] F. Li, X. Chen, X. Xiao 等. 2024年。黑盒环境中的增量上下文无关文法推断。载于第39届 IEEE/ACM 自动化软件工程国际会议（ASE）论文集。1171–1182。 [22] Zhiqiang Lin 和 Xiangyu Zhang. 2008年。从执行中推导输入语法结构。载于第16届 ACM SIGSOFT 软件工程基础国际研讨会论文集。83–93。 [23] Benoît Marchal. 2002年。《XML 实例教程》。Que 出版社。 [24] Leon Moonen. 2001年。使用岛屿文法生成健壮解析器。载于第八届逆向工程工作会议论文集。IEEE, 13–22。 [25] John Narayan, Sandeep Shukla, 和 Charles Clancy. 2015年。自动协议逆向工程工具综述。《计算机调查》48, 3 (2015年), 1–36。https: //doi.org/ 10 1145 2732254 。/ Hoang Lam Nguyen, Nebras Nassar, Ti [26] mo Kehrer, 和 Lars Grunske. 2020年。Mo-fuzz：用于测试模型驱动软件工程工具的模糊测试器套件。载于第35届 IEEE/ACM 自动化软件工程国际会议论文集。1103–1115。 [27] Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, 和 Satoshi Nakamura. 2015年。使用统计机器翻译学习从源代码生成伪代码。载于第30届 IEEE/ACM 自动化软件工程国际会议论文集（ASE）。IEEE, 574–584。 [28] Luzi Sennhauser 和 Robert C Berwick. 2018年。评估长短期记忆网络学习上下文无关文法的能力。arXiv 预印本 arXiv:1811.02611 (2018年)。 [29] Ben Smith. 2015年。《入门 JSON》。Apress。 [30] Prashast Srivastava 和 Mathias Payer. 2021年。Gramatron：有效的语法感知模糊测试。载于 ACM SIGSOFT 软件测试与分析国际研讨会第30届会议论文集。244–256。 [31] Junjie Wang, Bihuan Chen, Lei Wei, 和 Yang Liu. 2019年。Superion：语法感知灰盒模糊测试。载于 2019年 IEEE/ACM 第41届软件工程国际会议（ICSE）。IEEE, 724–735。 [32] Z. Wu、E. Johnson、W. Yang等人。2019年。REINAM：用于输入语法推断的强化学习。载于 *2019*年第*27*届*ACM*欧洲软件工程会议与软件工程基础研讨会联合会议论文集。488–498。[33] Daniel M Yellin和Gail Weiss。2021年。从循环神经网络综合上下文无关文法。载于 系统构建与分析工具和算法国际会议。施普林格, 351–369。[34] Xiang Yu、Ngoc Thang Vu和Jonas Kuhn。2019年。使用基于注意力的序列到序列模型学习Dyck语言。载于 *2019*年*ACL*研讨会*BlackboxNLP*：分析和解释用于自然语言处理的神经网络论文集。138–146。[35] Shu yu Guo、MichaelFicarra和KevinGibbons。2022年。*ECMAScript 2022*语言规范（第*13*版）。技术报告ECMA-262。Ecma International。