基于示例和树自动机的语法修复扩展版（含附录）

Yunjeong Lee，新加坡国立大学，新加坡Gokul Rajiv，新加坡国立大学，新加坡Ilya Sergey，新加坡国立大学，新加坡

arXiv:2602.18166v1 [cs.PL] 20 Feb 2026

上下文无关文法（CFG）是用于声明式描述编程语言具体语法并生成解析器的事实标准形式体系。定义所需语法的一大挑战，是排除上下文无关文法产生式中的所有可能歧义；这些产生式决定了作用域规则以及运算符优先级和结合性。实际的解析器生成工具通常采用临时拼凑的方法来解决此类歧义，这可能导致解析器的行为与语言设计者的意图相悖。在这项工作中，我们提出了一种用户友好的方法，能够可靠地修复存在歧义的文法；该方法受到自动程序合成中示例编程研究方向的启发。我们方法的核心是将初始上下文无关文法和定义优先级与结合性所需限制的补充示例，都表示为树自动机（TA）。我们方法的技术创新包括：（1）一种新的树自动机学习算法，它根据原始文法和示例构造一个自动机，在单个树自动机中编码用户偏好的歧义消解方式；（2）一种高效的树自动机交集算法，它利用可达性分析和优化大幅缩减所得自动机的规模，从而生成适用于解析器生成器的惯用上下文无关文法。我们证明了这些算法的可靠性，并将该方法实现为名为 Greta 的工具，通过一系列案例研究展示了其有效性。

CCS 概念：• **软件及其工程** → **上下文无关文法**；语法；解析 *rs.*

其他关键词　词语和短语：上下文无关文法、解析、树自动机、示例编程

ACM 引用格式：

Yunjeong Lee、Gokul Rajiv 和 Ilya Sergey。2026年。基于示例和树自动机的语法修复[2版]扩展版本（含附录）。*ACM* 程序设计语言会议论文集10，OOPSLA1，第134号文章（2026年4月），34 页。

https://doi.org/10.1145/3798242

1 引言

近年来，在为上下文无关语言自动合成文法和解析器方面已取得重大进展 [6, 8, 14, 21]。然而，精确描述一种编程语言的具体语法，仍然是一项具有挑战性且难以自动化的任务。主要挑战在于，定义语言的上下文无关文法（CFG）时，存在多种可能的方式，会导致将语言字符串解释为语法树时产生歧义。例如，根据以下算术表达式语言的上下文无关文法

$$\mathrm{S} \longrightarrow \mathrm{S} + \mathrm{S} \mid \mathrm{S} * \mathrm{S} \mid (\mathrm{S}) \mid x \mid y \mid z$$

字符串 $x + y * z$既可以解析为 $(x + y) * z$，也可以解析为 $x + (y * z)$，尽管语言设计者通常只希望采用其中一种解释。

作者联系信息：Yunjeong Lee，新加坡国立大学，新加坡，yunjeong.lee@u.nus.edu；Gokul Rajiv，新加坡国立大学，新加坡，grajiv@u.nus.edu；Ilya Sergey，新加坡国立大学，新加坡，ilya@nus.edu.sg。



本作品依据知识共享署名 4.0 国际许可协议授权。©2026年版权归所有者/作者所有。ACM 2475-1421/2026/4-ART134https://doi.org/10.1145/3798242

用于 LR(*k*) 解析器生成的现代框架，例如Yacc [16], Beaver [12], ScalaBison [9],和 Menhir [29]，能够检测运算符优先级、结合性和嵌套方面此类歧义的过度近似形式，并将其表示为所谓的移进/归约冲突和归约/归约冲突，向用户报告这些冲突，甚至自动解决它们。尽管这简化了语言语法的开发，但自动消解歧义可能导致对语法的解释不同于语言设计者心中隐含的设想。

为了更好地控制运算符优先级、结合性和嵌套，现有的解析器生成工具提供了显式指定这些属性的机制 [23, 31]，而标准编译器教材则提供了描述上下文无关文法、从一开始就避免歧义的一般策略 [17, 35]。尽管如此，采用特定工具的约定并遵循组织上下文无关文法的“良好实践”，往往会导致文法难以理解和维护。更糟的是，如果上下文无关文法中引入了歧义，就需要具备对象语言结构和解析器生成器工具工作原理方面的专业知识，才能正确消除歧义。

Adams 和 Might 提出了一种更具声明式的方法来消除上下文无关文法中的歧义 [2]：将施加于二义文法之上的限制表示为树正则表达式（TRE），从而明确禁止不需要的解析树类别。Adams 和 Might 的方法利用了上下文无关文法、树正则表达式和树自动机（TA）之间的基本联系，并采用后者这种在交集运算下封闭的语言表示方式，生成文法的修复后（即无歧义）版本。遗憾的是，对于非专家而言，编写一个能够消除歧义的正确树正则表达式，并不比直接修正文法中的歧义容易，因为这需要充分理解树正则表达式的语义，才能设计出恰当的限制。本研究旨在改进基于树自动机的语法修复方法，并借鉴自动程序合成研究中的一种流行的示例编程范式，为用户提供一种友好且可靠的方式来修复存在歧义的文法 [15, 18, 19, 26, 33, 36]。

关键思路。我们提出了一种新方法，称为基于示例的文法修复。该方法通过向用户推荐成对的示例，展示以互斥方式解决解析冲突的不同途径，从而消除文法中的歧义；随后请用户选择其中一种，并利用选定的示例生成“修复后”的文法。具体来说，我们的方法会：(a) 将检测到的、表现为 LR(1) 解析器生成冲突的歧义，转换为一小组具体示例，并以解析树备选项的形式呈现给用户。在这些示例中，(b) 用户选择符合目标语言所需语法的示例。接着，(c) 系统会自动将选定的示例转换为树自动机形式的文法约束；然后，(d) 将该约束与对应于原始文法的自动机求交，从而消除歧义。最后，(e) 将求交的结果转换回上下文无关文法形式，再次检查其中是否存在歧义；如果检测到歧义，则重复步骤 (a)–(e)。

挑战。 要将这一方法付诸实践，需要解决的技术问题与步骤 (c) 和 (d) 所需算法的设计有关。具体而言，步骤 (c) 看似是教科书式的自动机学习任务，但标准的 Angluin 风格算法 [4, 5, 7]并不适用于我们的场景，因为它们要求为字母表中的所有 符号构造树示例。换言之，这些示例合在一起必须表示整个文法，而非直接涉及歧义的子集 ——而我们旨在向本方法的用户提供的正是这些子集。此外，按照现有方法中树自动机交的标准定义来更新文法并加入学到的“修复” [2, 10] 原样应用，会产生不符合惯例且难以理解的文法。

我们通过实现两种新颖的算法应对了这些挑战：一种用于从输入“基础”文法的示例子树中进行树自动机学习（用于步骤 (c)），另一种用于树自动机交（用于步骤 (d)）。最后，我们在一个工具中实现了所述的端到端语法修复流水线。

贡献。 总之，我们作出了以下贡献：

- 我们的主要概念性贡献是 基于示例的文法修复——一种新颖的方法，只需用户简单输入哪些解析（子）树是可接受的，即可自动消解上下文无关文法中的歧义。我们的方法依赖于上下文无关文法与树自动机之间的一种基本关系（第 2 节）。
- 我们的第一项技术贡献是一种新颖的算法，可根据正负解析子树示例被动学习树自动机，并给出了其可靠性证明，表明合成的自动机确实能正确接受所有正例并拒绝所有负例（第 3.1 节）。
- 我们的第二项技术贡献是一种新算法，用于计算专为生成正则树文法而设计的树自动机交集，所得结果可转换为惯用的上下文无关文法；我们还证明了该算法的正确性（第 3.2 节）。
- 我们将基于示例的文法修复方法实现为一个名为 Greta 的工具，该工具构建于Menhir之上——这是一个用于 OCaml 解析器生成器的框架 [29]。我们在本科编译器课程中的示例、StackOverflow 上提出的问题以及真实世界语言的文法上进行评估，结果表明，我们的方法实用且高效：在大多数案例研究中，它确实能在用户只需提供少量帮助的情况下消除歧义，而且速度很快（第 4 节）。

2 概述

本节通过演示 Greta 的一次运行，说明如何通过与用户进行轻量级交互来消除上下文无关文法中的歧义：从用户提供的示例中学习树自动机（TA）形式的“修复方案”，随后通过树自动机交集更新文法。我们从一个具有歧义的典型输入上下文无关文法示例开始（第2.1节），说明如何将其转换为树自动机（第 2.2 节），介绍如何根据用户指定的树示例和输入上下文无关文法生成树自动机（第 2.3 节），并展示如何通过对两个树自动机取交集，再将结果转换回上下文无关文法来消除歧义（2.4）。

2.1 上下文无关文法与歧义

上下文无关文法（CFG）形式化定义为一个元组（V、 Σ、 S、 P），其中 V是非终结符号集合， Σ 是终结符号集合， $S \in V$是起始非终结符，而 $P \subseteq V \times (V \cup \Sigma)^*$是产生式集合。图 1 展示了一个上下文无关文法 G 的示例；其非终结符 V 包括 stmt、decl、expr 和 ident，起始非终结符为 stmt；其终结符 Σ 包括用于声明语句的 SEMI、TINT、IDENT 和 EQ，用于仅含 then分支或同时含 then分支和 else 分支的条件语句的 IF 、 THEN 和 ELSE，用于含二元运算的表达式的 PLUS 、 STAR 和 INT，以及表示左、右括号的 LPAREN 和 RPAREN；图 1 所示的产生式规则 P 将非终结符号替换为由非终结符号和/或终结符号组成的序列。

如果通过产生式规则可以用多种方式匹配终结符字符串，那么该上下文无关文法就称为有歧义 。图 1 中的 G 就是这样的文法，其中包含四种不同的歧义，也称为冲突。具体而言，这些冲突源于缺少有关加号和星号 运算符结合性的信息，以及缺少加号 与星号 之间的优先级顺序信息，还有如果带那么 分支与如果 带那么 和否则 分支之间的优先级顺序信息。图 2 展示了会产生这些冲突的表达式。第一个表达式(1) 可以通过应用文法的产生式规则表达式 → 表达式 加号 表达式 推导得出：先处理左侧的 表达式 加号 表达式 ，再处理右侧的，也可以反过来，因为该文法没有明确加号 是左结合还是右结合。同样，第二个表达式(2) 也存在星号 是左结合还是右结合的歧义。

V = { 语句 , 声明 , 表达式 , 标识符 }Σ ={ 分号 , 如果, 那么 , 否则 , 加号 , 星号 , 整数 , 左括号 , 右括号 , 整数类型 , 等号 } S = 语句 P =

{ 语句 → 声明 分号
stmt → 如果 表达式 那么 语句
stmt → 如果 表达式 那么 语句 否则 语句
decl → 整数类型 标识符 等号 表达式
标识符 → 标识符

expr → 表达式 加号 表达式
expr → 表达式 STAR 表达式
expr → INT
expr → LPAREN 表达式 RPAREN
expr → 标识符 }

图 1。一个存在歧义的上下文无关文法 G 示例。

第三个表达式 (3) 受到 PLUS 和 STAR之间未指定的优先级顺序影响。第四个表达式(4) 展示了嵌套 IF表达式的解析歧义：这些表达式只有 then 分支，或同时具有 then 和 else 分支。这一问题广为人知，名为 悬空 *else* 问题 [1]。

同一表达式的这些不同解析方式可以通过树示例来表示，如图 3所示。我们依赖此类树的层次属性：树示例中较深的符号比深度较浅处的符号具有更高的优先级顺序。这些示例是表示上下文无关文法中歧义的子解析树，因此只使用部分转移。当树示例中只有一个符号时，我们认为它表示结合性歧义。我们的想法是

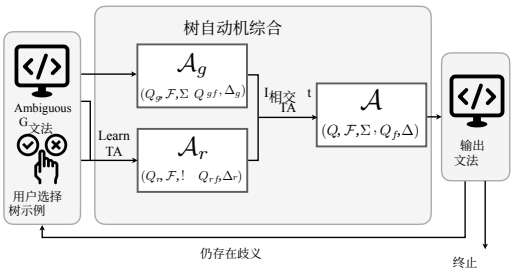


图 4。Greta 文法消歧工作流程。

允许用户为每个冲突选择一棵此类树。由此得到的、未被用户选中的树示例 T^- 集合随后可用于消除文法歧义，并相应地更新其规则。简而言之，Greta 通过以下方式实现这一点：(a) 根据用户指定的树示例和初始（输入）上下文无关文法生成一个树自动机；(b) 将学习得到的树自动机与从原始文法获得的树自动机求交，以消除歧义（图 4）。本节余下部分将以存在冲突的文法Greta 为例，介绍其工作流程。文法$\mathcal{G}$ 是一个示例。

2.2 从上下文无关文法到树自动机

我们方法的第一步是将输入文法转换为树自动机。众所周知，任何上下文无关文法都可以表示为一种树自动机，它能够识别其解析树所构成的语言——即所谓的正则树语言 [10, 13]。下面简要介绍树自动机的基本概念，并说明如何将图 1中的文法转换为相应的树自动机。

一个树自动机（TA）是一个元组（Q，F Σ，Q_f，Δ），其中 Q表示状态集合，F 是构造器标签的集合（又称带秩字母表），Σ 是终结符集合，$Q_f \subseteq Q$ 是终态集合，Δ 是转移规则集合。F 中的每个有秩符号都有一个对应的元数（秩）

(1) 表达式 加号 表达式 加号 表达式 (2) 表达式 星号 表达式 星号 表达式 (3) 表达式 加号 表达式 星号 表达式 (4) 如果 表达式 那么 如果 表达式 那么 语句 否则 语句

图 2. 展示图 1中文法中歧义的表达式。

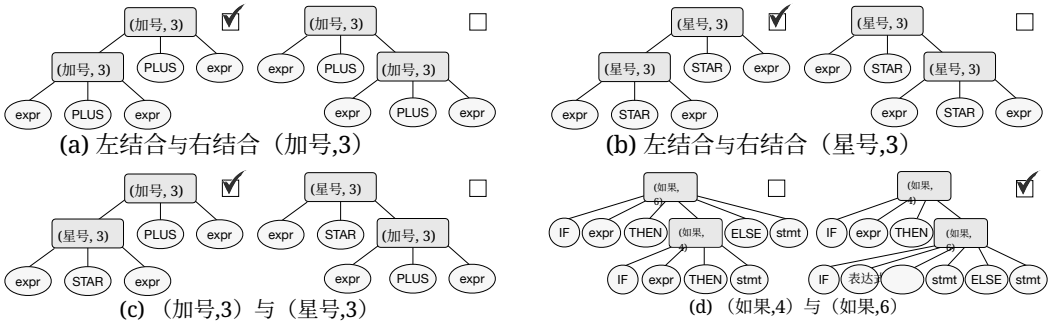


图 3. 表示 G 中冲突的树示例。

对应于原始上下文无关文法中某个产生式右部所出现的终结符和非终结符的数量。它还包含一个符号（符号），每个产生式都有一个独一无二的符号。如果存在一条从叶节点到根节点终态的运行，并且遵循 Δ 中的转移规则（在补充材料中有形式化描述），则树自动机接受该树。 图5 展示了$\mathcal{G}$的一组易读的有秩符号，这些符号选用于本文。

由文法 G 转换而来的树自动机 A_g 如 图6 所示，并结合了来自 图5 的带秩字母表。它按如下方式由该文法生成。首先，将 G 的产生式转换为具有相应元数的有序字母表符号，从而得到 图5。形如 表达式 → ident 的非终结符到非终结符产生式被映射为使用 $(\delta,1)$-标签的转移 表达式 $\leftarrow_{(\delta,1)}$ident，并将该标签加入带秩字母表。G 的非终结符集合 V 与开始符号 S 分别映射为状态集合 Q_g 以及作为最终接受状态 Q_{gf}的单元素集合 S，如图6所示。G 的终结符集合成为该树自动机的终结符集合 Σ。最后，用每条规则的带秩字母表符号对 G 的产生式规则进行标注，从而得到 A_g 的转移规则。

2.3 由树示例和输入文法生成树自动机 r

正如我们从第2.1节中了解到的，文法中的歧义是由缺乏关于终结符之间结合性和/或优先级顺序的信息造成的。我们发现，半具体解析树，如图 3所示，有助于说明结合性和优先级顺序方面的各种选项。我们的方法会向语言设计者展示这类树，并请他们选择符合其对相关字母表符号之间层次关系偏好的选项。根据用户的选择以及原始上下文无关文法，我们生成一个拒绝不需要的解析树的树自动机。该树自动机还必须接受与这些选择一致的树，以及涉及$\mathcal{G}$ 中与这些歧义无关的符号的树。

2.3.1 冲突示例及其解决。来自图 2中表达式的每一种歧义，都会以具有不同语法分析顺序的解析树呈现给用户，如图 3所示。我们的方法依靠标准的 LR(1) 语法分析器生成器来识别冲突，并根据包含这些歧义的产生式集合生成树示例。例如，(1) expr 加号 expr 加号 expr 在图 2中可以通过应用产生式规则expr → expr

F ={ (分号 ,2) , (如果 ,4), (如果 ,6) , (整数类型 ,4) , (加号 ,3) , (星号 ,3) , (整数 ,1) (() ,3) , (δ, 1), (标识符 ,1) }

图5. 运行示例的带秩字母表 F。

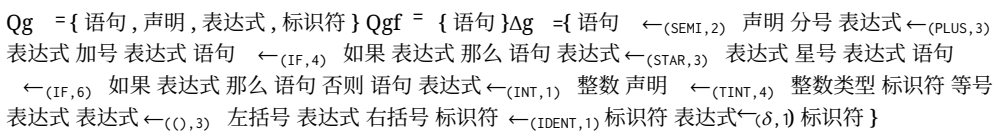


图6。根据 G. 重新诠释的树自动机 $A_{\mathcal{g}}$

加号 表达式 然后对左侧或右侧表达式 产生式右部的内容应用相同规则，实质上体现了(PLUS,3)运算符的左结合性或右结合性。一旦为规则标注了相应的树构造器标签，(PLUS,3) 此例中的解析树就会以不同方式构建，如图 3a所示。同样，(STAR,3) 在图 3b 中表示其左结合和右结合的解析树，代表了对(2) expr STAR expr STAR expr（见图 2）进行解析的两种不同方式。正如这些产生式集合所示，如果冲突仅涉及一个符号，我们可以推断这是因为该符号的结合性不明确。

除了结合性之外，冲突也可能由不同符号之间有歧义的优先级顺序引起。请看(3) expr PLUS expr STAR expr （见图 2）。可以先应用产生式表达式 → expr PLUS expr ，再将表达式 → expr STAR expr 应用于右侧的表达式，从而生成图 3c左侧的解析树。也可以先应用表达式 → expr STAR expr，再将表达式 → expr PLUS expr 应用于左侧的表达式，从而生成图 3c右侧的树。当这些符号之间的优先级顺序（例如，(PLUS,3) 和(STAR,3)）存在歧义时，这两种方式会让符号处于树中的不同深度，从而生成两种不同的解析树。同样，对(4) IF expr THEN IFexpr THEN stmt ELSE stmt （见图 2）进行解析时产生的歧义，即悬空 *else* 问题，可用图 3d中的树表示：左侧的树表示(IF,4) 的优先级高于(IF,6) ，而右侧的树则表示相反的优先级关系。

回到我们的运行示例，假设用户通过选择图 3中打勾的树，表明了他们的解析偏好。这使 Greta 除了学习(PLUS,3) < (STAR,3) 和(IF,4) < (IF,6) 之间的关系外，还学会了(PLUS,3) 和 (STAR,3) 是左结合的。这些用户偏好可以视为对输入文法的限制。Greta 通过遍历用户未选择的每棵树来收集这些限制，并将它们与原始文法的所谓基础优先级顺序结合起来。

基础优先级顺序 O_{bp} 表示从代表输入上下文无关文法$\mathcal{A}_{\mathcal{g}}$ G 的树自动机中获得的所有带秩字母表符号的集合及其对应的次序；这些次序可以看作是到最短距离终态$\mathcal{A}_{\mathcal{g}}$ 的图 6中所示的最终接受状态的。对于我们的示例，其带秩字母表记为F ，如图5 O_{bp} 所示，具体如下：

{ ((如果 ,4) ,0) , ((如果 ,6) ,0) , ((分号 ,2) ,0) , ((整数类型 ,4) ,1) , ((加号 ,3) ,1) , ((星号 ,3) ,1) , ((整数 ,1) ,1) , ((() ,3) ,1) , ((δ,1) ,1) }

这个 1元标识符 被识别为平凡符号 (详见第 3.1.1 节)，并单独处理，因为我们知道它不会与其他符号发生冲突，而且其对应的非终结符在 Greta 的整个运行过程中保持不变。其关联状态 标识符 以及转移 标识符 ←(IDENT,1)标识符 在生成的树自动机中保持不变。我们给出其形式化定义 O_{bp} ，并在 第 3.1 节中描述构造它的过程。

2.3.2 将解析偏好表示为树自动机。 接下来这一步——构造一个树自动机，为输入文法添加额外限制——是本工作的关键创新思想。

Q_r ={ e_0, e_1, e_2, e_3, e_4, 标识符 }
Q_{rf} = { e_0 }
Δ_r =

{ $e_0 \leftarrow_{(IF,4)}$ IF e_0 THEN e_0
$e_0 \leftarrow_{(SEMI,2)}$ e_0 SEMI
$e_0 \leftarrow_{(\epsilon,1)}$ e_1
$e_1 \leftarrow_{(IF,6)}$ IF e_1 THEN e_1 ELSE e_1
$e_1 \leftarrow_{(SEMI,2)}$ e_1 SEMI
$e_1 \leftarrow_{(\epsilon,1)}$ e_2
$e_2 \leftarrow_{(PLUS,3)}$ e_2 PLUS e_3
$e_2 \leftarrow_{(TINT,4)}$ TINT e_2 EQ e_2
$e_2 \leftarrow_{(INT,1)}$ INT
$e_2 \leftarrow_{((),3)}$ 左括号 e_2 右括号
$e_2 \leftarrow_{(\delta,1)}$ 标识符
$e_2 \leftarrow_{(\epsilon,1)}$ e_3
$e_3 \leftarrow_{(STAR,3)}$ e_3 STAR e_4
$e_3 \leftarrow_{(TINT,4)}$ TINT 标识符 EQ e_3
$e_3 \leftarrow_{(INT,1)}$ INT
$e_3 \leftarrow_{((),3)}$ 左括号 e_3 右括号
$e_3 \leftarrow_{(\delta,1)}$ 标识符
$e_3 \leftarrow_{(\epsilon,1)}$ e_4
$e_4 \leftarrow_{(TINT,4)}$ TINT 标识符 EQ e_4
$e_4 \leftarrow_{(INT,1)}$ INT
$e_4 \leftarrow_{((),3)}$ 左括号 e_4 右括号
$e_4 \leftarrow_{(\delta,1)}$ 标识符
标识符 $\leftarrow_{(IDENT,1)}$ 标识符 }

图 7. 由用户指定的解析偏好生成的树自动机 A_r。

请注意，图 3 中的树并非输入文法 $\mathcal{G}$的解析树；它们其实是该文法允许的某些解析树的子树。具体而言，它们包含符号 (PLUS,3)、(STAR,3)、(IF,4)和(IF,6)，但缺少字母表 $\mathcal{F}$ 中其他符号的优先级顺序信息，如 或，相对于示例中符号的优先级顺序信息；这些信息可以通过 O_{bp}恢复。鉴于用户在所提供示例中指明的偏好，我们的目标是将其纳入一个新的树自动机 $\mathcal{A}_r$，该自动机也要具有足够的“宽容度”，能够接受原始文法对应的树自动机 $\mathcal{A}_{\mathcal{G}}$ 所允许的目标“完整”解析树。本节余下部分将对这一过程作非正式说明，详述其细节并给出正确性论证，见第 3.1 节。

我们首先把从图 3 中的示例所学到的限制（例如，(加号,3) <(星号,3)，(如果,4) < (如果,6)，等）与基础优先级顺序 O_{bp}结合起来，得到以下适用于所有符号的优先级顺序集合——我们称之为 O_p:

{ ((如果 ,4) ,0) , ((分号 ,2) ,0) , ((如果 ,6) ,1) , ((分号 ,2) ,1) , ((加号 ,3) ,2) , ((整数类型 ,4) ,2) , ((整数 ,1) ,2) , ((() , 3) ,2) , ((δ,1) ,2) , ((星号 ,3) ,3) , ((整数类型 ,4) ,3) , ((整数 ,1) ,3) , ((() ,3) ,3) , ((δ,1) ,3) , ((整数类型 ,4) ,4) , ((整数 ,1) ,4) , ((() ,3) ,4) , ((δ,1) ,4) }

在这个更新后的集合中，比较所选示例里的符号次序 O_p，其结果与优先级关系(PLUS,3) < (STAR,3) 以及(IF,4) < (IF,6)一致。此外，对于基础优先级顺序中的所有其他符号对，都存在一个 O_p 中的符号对，其相对次序相同。O_p 也支持之后整合结合性限制 O_a，相关完整细节见第 3.1 节。

随后，给定 O_p 中由有秩符号及其次序构成的对 (f, o)，Greta 会根据 Prod(f) 右部中的状态和终结符，生成一条标记为 f、指向 e_o 的转移。例如，来自 O_p 的 ((如果,4),0) 会产生转移 $e_0 \leftarrow_{(IF,4)}$ 如果 e_0那么 $e_{0\circ}$ 与此同时，每个有序状态 e_o 通过一条 ϵ 转移与层级中的下一个有序状态 e_{o+1} 相连：例如，$e_0 \leftarrow_{(\epsilon,1)} e_{1\circ}$ 与符号 (如果,4) 对应的状态 e_0 表示语法分析顺序层级中的一个层级。此外，如果某个符号曾在示例中被用于指定结合性，Greta 会生成一条将其考虑在内的转移：例如，根据来自 O_a 的左结合 (加号,3) 以及来自 O_p 的 ((加号,3),2)，生成 $e_2 \leftarrow_{(PLUS,3)} e_2$ 加号 e_3。这些步骤从而构造出图 7 中的 A_r。该算法的细节在第 3.1 节中加以说明。

请注意，这样得到的自动机并不会立刻成为我们所期望的、对应于修复后的文法的结果。这是因为所构造的 A_r 不仅识别 A_g 所允许的所有那些与示例无关的解析树，甚至还识别那些不被 A_g 允许的项。例如，考虑表达式 如果 1 那么 2 星号 3。与该表达式对应的树会被 A_r 接受，因为出现在 那么 之后的 e_1 可被重写为 e_3 星号 e_4，，而这在原始文法中是不可能的，因此不 被 A_g 接受。这正是我们需要对这些树自动机取交集的原因，以便最终产生一种文法，它既不 过于宽松相对于 原始文法，又具限制性相对于用户指定的解析偏好。

2.4 通过求交树自动机修复文法

现在我们有两个树自动机：从 G 得到的 $A_g \triangleq (Q_g, \mathrm{F}\ \Sigma, Q_{gf}, \Delta_g)$，以及从树示例和 G 中学习得到的 $A_r \triangleq (Q_r, \mathrm{F}\ \Sigma, Q_{rf}, \Delta_r)$。基于示例的歧义修复结果由树自动机 $A_{res} \triangleq (Q, \mathrm{F}\ \Sigma, Q_f, \Delta)$ 表示，它是通过对 A_g 和 A_r 求交得到的。它定义为一个由各个组成部分的笛卡尔积构成的元组：即 $Q = Q_g \times Q_r$、$Q_f = Q_{gf} \times Q_{rf}$ 和 $\Delta = \Delta_g \times \Delta_r$。下面我们将逐一讨论本例中的各个组成部分。

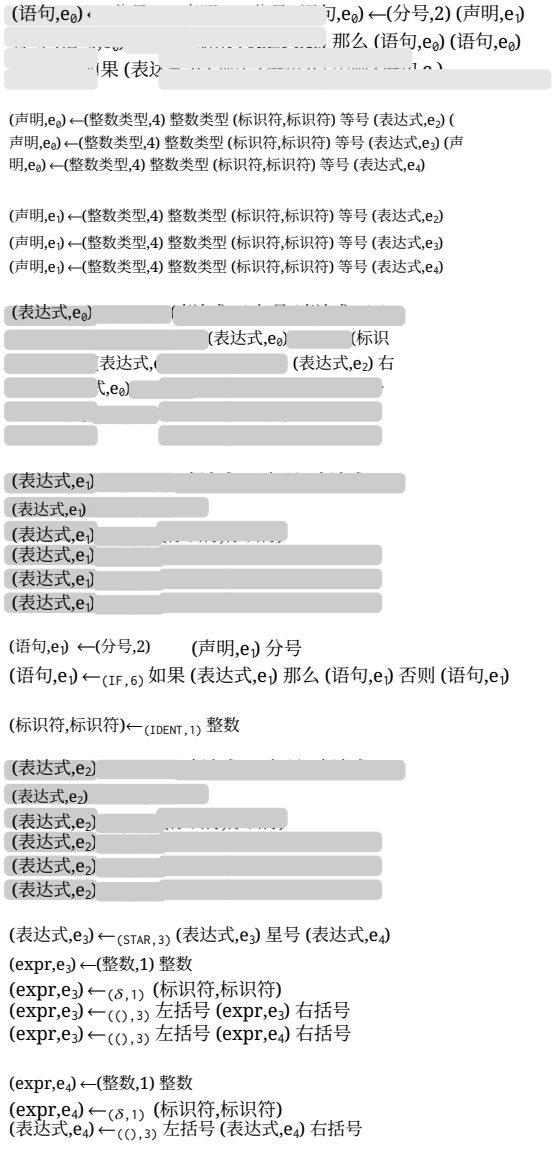
首先，对 A_g 和 A_r 的接受状态取笛卡尔积，会得到新状态 $(\texttt{stmt}, e_\emptyset)$，它是交集树自动机的初始状态。接下来，从状态 $(\texttt{stmt}, e_\emptyset)$ 出发，将其作为转移的左部，遍历 Δ_g 中生成 stmt 的转移，以及 Δ_r 中生成 $e_\emptyset$ 的转移，可以发现有三个构造子标签——$(\texttt{SEMI},2)$、$(\texttt{IF},4)$ 和 $(\texttt{IF},6)$——它们都具有匹配的右部。请注意，生成x 的转移，实质上指的是那些转移到树自动机中状态 x 的规则。给定状态 (α_g, α_r)，当转移 $\alpha'_g \leftarrow \beta_g \in \Delta_g$ 和 $\alpha'_r \leftarrow \beta_r \in \Delta_r$（其中 α'_g 和 α'_r 分别是 α_g 和 α_r，或可通过 ε 转移从它们到达）中的 β_g 和 β_r 长度相同（终结符和状态的数量相同），并且在每个位置上，两者要么都是状态，要么是相同的终结符时，这些转移就被视为匹配，且。

例如，考虑符号 $(\texttt{IF},4)$ 的转移。在 Δ_g 中，转移 $\texttt{stmt} \leftarrow_{(\texttt{IF},4)} \texttt{IF expr THEN stmt}$ 与转移$e_1 \leftarrow_{(\texttt{IF},4)} \texttt{IF } e_1 \texttt{THEN } e_1$ 在 Δ_r 中匹配，因为它们的右部元素数量相同，并且每个位置上的元素要么都是状态，要么是相同的终结符。另一方面，当我们查看符号$(\texttt{PLUS},3)$时，找不到 Δ_g 中任何以 stmt 为结果、并以 $(\texttt{PLUS},3)$ 作为构造子标签的对应转移。因此，在树自动机交集中，符号 $(\texttt{PLUS},3)$ 不会产生$(\texttt{stmt}, e_1)$转移。

按照这一思路，对产生 $(\text{stmt}, e_\emptyset)$ 的转移取笛卡尔积，会得到图 8a 中标签为 (SEMI,2)、[illegible]转移。此外，这也表明，从产生 $(\text{stmt}, e_\emptyset)$ 的转移可以到达以下状态：$(\text{decl}, e_\emptyset)$、$(\text{decl}, e_1)$、$(\text{expr}, e_\emptyset)$、$(\text{expr}, e_1)$、$(\text{stmt}, e_\emptyset)$ 和 (stmt, e_1)。我们考察这些所谓的可达状态，找出产生每个状态的转移；考虑这些转移可能会将新状态加入可达状态集合。由此得到图 8a 中其余的转移规则。

获得所有生成可达状态的转移后，我们会检查这些规则，以消除重复状态，以及相应地消除转移到重复状态的规则。在运行示例中，状态 $(\text{expr}, e_\emptyset)$、$(\text{expr}, e_1)$ 和 (expr, e_2)，以及 $(\text{decl}, e_\emptyset)$ 和 (decl, e_1) 被判定为重复状态。我们用深灰色突出显示前一组转移，它们的右部完全相同。因此，我们移除其中两个——例如 (expr, e_1) 和 (expr, e_2)——及其转移规则，同时将所有出现的这些状态替换为 $(\text{expr}, e_\emptyset)$。对于 $(\text{decl}, e_\emptyset)$ 和 (decl, e_1)，我们也进行同样的处理。移除这些重复状态并重命名状态后，我们得到图 8b 中更小但等价的转移集合。

最后，我们引入 $(\epsilon, 1)$-转移，以进一步简化所得转移。例如，观察 表达式$_2$-生成转移，表达式$_\emptyset$- 和 表达式$_1$-生成转移会重复这些转移，如图 8b中的虚线框所示。也就是说，转移规则采用



(a) Δ_g 和 Δ_r的笛卡尔积结果。 (b) 去除重复项并重命名状态之后。

图 8. 转移 Δ_g 与 Δ_r的笛卡尔积的最小化。

具有相同的树构造器标签集合，并分别转移到相同的右部集合。因此，我们可以通过分别添加以下规则来简化 表达式$_\emptyset$- 和 表达式$_r$产生转移表达式$_\emptyset \leftarrow_{(\epsilon,1)}$ 表达式$_2$ 和 表达式$_1 \leftarrow_{(\epsilon,1)}$ 表达式$_2$, , 同时移除重复的转移。我们对所有状态及其转移重复这一过程，得到最终的树自动机 $\mathcal{A}_{\text{res}}$ ，如 图 9所示。得到的树自动机 $\mathcal{A}_{\text{res}}$ (图 9) 可以通过去掉转移上的标签，直接转换回对应的上下文无关文法。[1] 然后将该文法传回 Greta。如果仍存在歧义

[1]由于 ϵ-转移不会消耗树节点（由符号表示），因此不能 直接将其取消标记，因为它们会导致树中出现原始文法里没有的产生式。相反，取消标记后得到的单元产生式需要

Q = { 语句0, 语句1 , 表达式0, 表达式1 , 表达式2 , 标识符 } Q_f = { 语句0 }Δ ={ 语句$_0$ $\leftarrow_{(\texttt{IF},4)}$ 如果 表达式0 那么 语句0 表达式$_0$ $\leftarrow_{(\texttt{PLUS},3)}$ 表达式0 加号 表达式1 语句$_0$ $\leftarrow_{(\epsilon,1)}$ 语句1 表达式$_0$ $\leftarrow_{(\epsilon,1)}$ 表达式1 语句$_1$ $\leftarrow_{(\texttt{SEMI},2)}$ 声明 分号 表达式$_1$ $\leftarrow_{(\texttt{STAR},3)}$ 表达式1 星号 表达式2 语句$_1$ $\leftarrow_{(\texttt{IF},6)}$ 如果 表达式0 那么 语句1 否则 语句 1 表达式^1 $\leftarrow_{(\epsilon,1)}$ 表达式 2 声明 $\leftarrow_{(\texttt{TINT},4)}$ 整数类型 标识符 等号 表达式 0 表达式 $_2$ $\leftarrow_{(\texttt{INT},1)}$ 整数 标识符 $\leftarrow_{(\texttt{IDENT},1)}$标识符 表达式^2 $\leftarrow_{(\epsilon,1)}$ 标识符 表达式$_2$ $\leftarrow_{(\texttt{(},3)}$ 左括号 表达式0 右括号 }

图 9。由 A_g 与 A_r 的交集得到的自动机 A_{res} 。

在其中，那么 Greta 会再次执行之前的所有步骤。当文法中不再存在歧义时，它便停止运行，或者 剩余的歧义不属于 Greta 或 Menhir的处理范围；相关细节将在第 4.4.2 节中讨论。Greta 最终会终止，因为原始上下文无关文法、由其构造的树自动机及其歧义都是有限的，而歧义数量（因此上下文无关文法所允许的解析树数量）会随着每次Greta (第 4.4.2 节)运行而减少。我们将在第 3.2 节介绍我们的交集算法。

2.5 融会贯通

在 Greta框架中，文法消歧的整体工作流程如下，正如图 4所示。如果输入的是二义性上下文无关文法，就会将其解释为树自动机，并向用户展示其中的歧义；对于每个歧义，系统会请用户从两个备选树示例中选择一个，它们共同表示同一表达式在每种冲突下的不同解析方式（第 2.3.1 节）。用户通过选择一组树示例指定偏好后，Greta 随后会利用选定的示例以及初始文法中所有字母表符号的优先级顺序，学习一个树自动机，以按照该文法编码用户的解析偏好（第 2.3.2 节）。接着，对这两个树自动机求交集，得到一个新的树自动机（2.4），然后将其转换回对应的文法。如果结果文法仍存在任何歧义，Greta 就会再次运行，直到文法完全消歧，或仅剩下无法处理的歧义。在此过程中，用户只需通过树示例来澄清解析偏好，无须计算树自动机操作，甚至不必知道这些操作确实在执行。

3 通过示例修复语法：形式化描述

本节描述Greta 框架的技术细节。首先，我们给出一种从树示例和基础文法学习树自动机的算法，并在第 3.1 节中给出其可靠性保证。接下来，我们解释用于树自动机求交的算法，并展示其正确性，见第 3.2 节。

3.1 从树示例到树自动机

学习过程大致可分为两个部分：(a) 通过树示例，利用 LearnOaOp (算法 3.1) 学习有关结合性和优先级顺序的限制；(b) 随后通过 GenTA 过程（算法 3.2）构造一个树自动机。在介绍各个算法之前，我们先对树示例和树自动机作形式化描述。

被移除，以生成新的上下文无关文法。在 Menhir 中，我们通过花括号中的语义动作来规避这一问题，（例如 expr0 -> expr1 { $1 }），从而避免此类产生式出现在最终的抽象语法树中。

与完整解析树不同，树示例可以从任意非终结符开始，并且叶节点可以是非终结符而不是终结符，例如， 加号3(星号3(表达式 ∗，表达式) +，表达式)。一个树示例表示一组有效的完整解析树，其中：(1) 树示例中的所有子非终结符都替换为以该非终结符为根的有效子树；(2) 替换后得到的树是完整上下文无关文法解析树的一棵子树。给定一个树示例 t，我们用解析树(t)表示它所代表的解析树集合。假设我们有树语言 $\mathrm{L_g}$，表示以起始非终结符为根的文法 G 的解析树集合；$\mathrm{T_g^{nt}}$，表示以 G 中非终结符 nt 为根的解析树；以及函数 $\mathrm{T_g^{nt}} \to \mathrm{L_g}$，表示 G 中某个非终结符 nt 对应的解析树上下文。于是，以 nt 为根、包含 k 个不完整非终结符 nt_1，…，nt_k 的树示例是函数 $\mathrm{T}^{nt_1}\mathrm{g} \times \cdots \times \mathrm{T}^{nt_k}\mathrm{g} \to \mathrm{T}^{nt}\mathrm{g}$，并且解析树($t$)定义如下：

$$ParseTrees(t) \triangleq \{t' \mid \exists\, nt, nt_1, \ldots, nt_k, \exists\, C \in \mathcal{T}_{\mathrm{g}}^{nt} \to \mathcal{L}_{\mathrm{g}}, t \in \mathcal{T}_{\mathrm{g}}^{nt_1} \times \cdots \times \mathcal{T}_{\mathrm{g}}^{nt_k} \to \mathcal{T}_{\mathrm{g}}^{nt},$$
$$\exists\, t_1 \in \mathcal{T}_{\mathrm{g}}^{nt_1}, \ldots, t_k \in \mathcal{T}_{\mathrm{g}}^{nt_k}.\ t' = C[t(t_1, \ldots, t_k)]\}$$

Greta 目前支持恰好使用两个产生式的树示例，这涵盖了实践中的绝大多数歧义。我们定义一些便于处理这些示例的记号：给定一个树示例 t，t_T 表示 t 的根（顶层）符号，t_B 表示嵌套（底层）符号，t_{idx} 表示 i（从左侧的 0 开始）子节点的索引，在该节点处出现 t_B，其中 $0 \le i < \mathrm{Rank}(t_T)$。给定符号 α、β 和索引 i，Eg(α、β、i) 表示满足以下条件的树示例：$\mathrm{Eg}(\alpha, \beta, i)_T = \alpha$，$\mathrm{Eg}(\alpha, \beta, i)_B = \beta$，$\mathrm{Eg}(\alpha, \beta, i)_{idx} = i$，所有其他子节点均为通配符。例如，Eg(PLUS3, STAR3, 0) 表示树示例 PLUS3(STAR3(expr ∗、expr) +、expr)。树示例的另一种形式会使用相同的两个符号，但采用不同的配置。Greta 支持的树示例就是那些可以用 Eg 表达的示例。

如果满足 $t_T = t_B$，则示例 t 是与结合性相关的示例；如果满足 $t_T \neq t_B$，则是与优先级顺序相关的示例。当用户看到两个树示例时，未被选中的那个对应于我们希望从修复后的文法中排除的树。$P^-(t)$ 表示要移除的解析树集合，对应于用户未选择的树示例 t：

$$P^-(t) \triangleq \begin{cases} ParseTrees(t) & \text{if } t_T = t_B \\ \bigcup\limits_{0 \le i < Rank(t_T)} ParseTrees(Eg(t_T, t_B, i)) & \text{otherwise} \end{cases}$$

在解决优先级顺序相关冲突时，我们会强制规定符号层级：优先级较高的符号在树中的位置必须严格深于优先级较低的符号，因此需要对所有可能的子节点位置上的排除解析树取并集。对于涉及优先级或结合性相关冲突的符号集合 $\mathrm{S_C}$，Greta 构造出 $\mathrm{S_C}$ 的以下划分：

$$\mathcal{S}_{\mathrm{E}} = \{S \subseteq \mathcal{S}_{\mathrm{C}} \mid \forall s_i, s_j \in S, s_i \neq s_j, \exists\, n, m \in \mathbb{N} \text{ such that}$$
$$ParseTrees(Eg(s_i, s_j, n)) \neq \emptyset \vee ParseTrees(Eg(s_j, s_i, m)) \neq \emptyset,\ S \text{ is maximal}\}$$

$\mathrm{S_E}$ 中的每个符号集合都是 $\mathrm{S_C}$ 的极大子集，并且对于集合中的所有符号对 s_i 和 s_j，文法（描述语言 $\mathrm{L_g}$）都允许在某些子节点位置 n 和 m 上出现树 $Eg(s_i, s_j, n)$ 或 $Eg(s_j, s_i, m)$。

对于这些集合中的每一个，如果成对的符号 $Eg(s_i, s_j, n)$ 和 $Eg(s_j, s_i, m)$ 能以两种次序解析，Greta 就会为它们生成优先级相关树示例，以确定集合中符号的全序。对于存在冲突的符号，它还会生成结合性相关树示例。随后，它通过与用户交互来了解用户期望的限制。

我们现在已经介绍了树示例、结合性、优先级和排除的解析树等概念。我们关于树自动机以及从上下文无关文法转换为树自动机的形式体系相对标准，可在补充材料中找到。现在我们转而介绍 Greta 的树自动机学习算法。

3.1.1 基础优先级顺序。 算法 3.1 首先为 $\mathcal{F}$ 中 $\mathcal{G}$ 中的所有符号计算一组现有的优先级顺序，称为基础优先级顺序 O_{bp}，具体如下：

- 首先，为每个非终结符 $e \in V$ 定义一个层级（或称其与起始非终结符之间的距离）：如果 e 是 G 的起始非终结符，则 d（e）为– $= 0$ e。–否则， d（e）是满足以下条件的最小 n：存在一个非终结符序列 $nt_0, \ldots, nt_n$，其中 $nt_n = e$、 nt_0 是起始非终结符，并且对于每个 nt_i， nt_{i+1}，都存在一个产生式$nt_i \to \beta$，属于 P，且 β 包含 nt_{i+1}。
- 接下来，使用顺序函数 $\widehat{o}$确定 F 中每个符号 s的次序：F $\mapsto \mathbb{Z}$，定义为$\widehat{o}$（s）（左部（$Prod$（s））），其中$Prod$（s）是有秩符号唯一对应的非终结符，而Lhs表示产生式的左侧非终结符。
- 最后， Obp 被构造为集合 $\{(s, \widehat{o}(s)) \mid s \in \text{F} \setminus \text{Ftr}\}$，其中 Ftr 是下文定义的平凡符号集合。

可以将符号 s 的次序理解为从起始非终结符到达它所需的最短距离 s。在没有环的情况下，次序较高的符号在解析树中的位置总是比次序较低的符号更深。根据用户偏好正确地划分符号次序，可以消除不同符号之间的优先级歧义，以及同一符号自身的结合性歧义。为了处理次序中的环，我们稍后需要在第 3.1.3 节所述的 TA 学习过程中重新引入这些环。现在来看图 1 中的运行示例 G，其起始非终结符为 stmt，各非终结符的层级如下：stmt 为 0，decl 和 expr 均为 1，ident 为 2。基于此，我们可以确定所有符号的次序。以符号 (IF,6) 及其产生式 stmt →IF expr THEN stmt ELSE stmt 为例，(IF,6) 的次序计算为 d(stmt) = 0。同样，我们可以计算所有符号的次序，从而得到 O_{bp} 第 2.3 节中的结果。

平凡符号集 F_{tr} 是一个一元符号集合，定义如下：

$$\mathcal{F}_{\text{tr}} \triangleq \left\{ s \;\middle|\; \begin{array}{l} s \in \mathcal{F} \wedge \mathit{Rank}(s) = 1 \wedge \delta_s \triangleq \alpha \leftarrow_s \beta \in \Delta \text{ where } \beta \in \Sigma \wedge \\ \forall\, \delta_{s'} \in \Delta \; s.t. \; \delta_{s'} = \alpha \leftarrow_{s'} \beta', \beta' \in \Sigma \end{array} \right\}$$

$\mathcal{F}_{\text{tr}}$ 是指其左侧非终结符仅导向单个终结符的符号。由于我们知道，$\mathcal{F}_{\text{tr}}$ 中的符号不会涉及任何歧义，关于结合性或优先级顺序，Obp 不包含$\mathcal{F}_{\text{tr}}$ 中的符号，并且生成的树自动机中涉及平凡符号的产生式保持不变。不过请注意，对平凡符号的这种单独处理仅是一种优化，不会影响学习算法的正确性。这也是算法 3.1 和算法 3.2中唯一的此类优化。

基本顺序的构造 O_{bp} 对于获取未涉及树示例的符号之间的相对优先级顺序至关重要。如果仅基于树示例合成树自动机，那么学习得到的树自动机将只包含与歧义相关的符号，这通常只是原始符号字母表中相对较小的一个子集。同时，我们认为，生成能够共同包含输入文法所有符号的示例，会降低我们工具的可用性。因此，Greta 提供的示例仅包含文法符号的一个子集，原始文法中未涉及歧义的符号则不予考虑。保留原始文法的优先级顺序 O_{bp} 使得Greta 能够高效地学习涉及所有优先级和结合性冲突的树自动机，同时不损害其正确性陈述方面的可靠性，关于第 3.1.4 节中的正确性陈述。

3.1.2 计算结合性和优先级顺序。如算法 3.1所示，LearnOaOp过程接收由用户 T^-未选择的树示例、输入上下文无关文法 G，以及一个映射 M_{to}，该映射将次序映射到 S 中在该次序下的一组有序符号集 $_{\text{E}}$，次序从低到高

算法 3.1： LearnOaOp：学习结合性和优先级顺序。

输入： 树示例 T^-、有序符号映射 M_{to}、输入上下文无关文法 G
输出： 结合性 O_{a}、优先级顺序 O_{p}
$O_{\text{bp}} \leftarrow$ G 的基础优先级顺序
O_{a}, $O_{\text{p}} \leftarrow \{\}$; $O_{\text{tmp}} \leftarrow O_{\text{bp}}$ // 使用 O_{b} 中的所有元素初始化临时集合 p
对 $t \in T^-$ **执行**
  如果 $t_T = t_B$ **那么**
    $O_{\text{a}} \leftarrow O_{\text{a}} \cup \{(t_T, t_{idx})\}$
对 降序 (o, G) 在 M_{to} **执行**
  大小 $\leftarrow$ 最大大小(G)
  $S =$ 符号(ofOrder(O_{tmp}, o)) \ $\bigcup_{g \in G} g$
  $O_{\text{tmp}} \leftarrow$ 压入N($O_{\text{tmp}}, o+1, size-1$) \ 次序($O_{\text{tmp}}, o$)
  对 i 在 [0,大小) **执行**
    第i个符号 $\leftarrow$ getAtIndex(G, i)
    $O_{\text{tmp}} \leftarrow O_{\text{tmp}} \cup$ 带次序（$S \cup$ 带符号 $o+i$）
    如果 $i = size - 1 \wedge \exists\, s \in ithSymbols, (s,_) \in O_a$ **那么**
      $O_{\text{tmp}} \leftarrow$ pushN(O_{tmp}<style 标识符='3'>, $o+i+1$<style 标识符='6'>, 1)
      $O_{\text{tmp}} \leftarrow O_{\text{tmp}} \cup$ with次序$(S, o+i+1)$
$O_{\text{p}} \leftarrow O_{\text{tmp}}$
返回 $(O_{\text{a}}, O_{\text{p}})$

根据 T^-推断出的优先级。[2] 直观地说，我们特别关注负树示例，因为学习算法的目标是排除解析树中任意位置出现的不良模式，而非保留期望的模式。这一点在算法正确性的描述中得到了明确说明，参见第3.1.4节。然后，算法返回对结合性的限制O_{a} 和优先级顺序的限制 O_{p}。

算法 3.1 的目标是推导出一组 O_{a} 结合性限制（非法子节点位置）以及一组 O_{p}，以便在冲突符号之间强制建立优先级顺序的层级关系，同时保留 O_{bp}中的其他两两顺序。在第 3.1.3节中，O_{p} 中的每个顺序都将对应于学习得到的树自动机中的状态，其中，某个与顺序 i 对应的状态 e_i 可以通过 ε 转移从状态 e_j 到达，适用于 $j > i$。借助 O_{bp}，LearnOaOp 分别在 O_{a} 和 O_{p}中收集与结合性和优先级顺序相关的限制。

首先，检查每棵树，判断它是否为与结合性或优先级顺序相关的示例。如果它是与结合性相关的示例，我们就将要禁止的符号及其嵌套符号的位置这一对信息存入 O_{a}。例如，在图 3a 中，树未选中包含符号(加号,3) 作为右子节点（位置 1）。随后，将一对 ((加号,3), 1) 加入集合 O_{a}。结合图 3b中选中的树，用户交互会生成集合 $O_{\text{a}} = \{((\text{加号},3), 1), ((\text{星号},3), 1)\}$。

接下来，算法按符号顺序降序遍历 M_{to}。对于每个次序o 及其对应的有序符号集 G，它首先计算 G中最大集合的大小。然后，将次序为 o 且未涉及任何冲突的符号集存入 S。接着，它从 O_{tmp} 中移除次序为 o 的符号，并调用pushN，将所有次序为 $\geq o+1$ 的符号的次序增加 $size - 1$，以便为新符号腾出空间。

然后，对于从 o 到 $o + size - 1$ 的每个次序 $o + i$，它会将符号集中的第 i 个集合插入 G，并插入不涉及冲突的辅助符号集合 S。getAtIndex 会在 G中的每个集合里返回索引为i 的符号（如果存在），而 withOrder 会为给定的符号指定次序。

As 例如，映射 $M_{\text{to}} = \{0 \rightarrow ((\text{星号},3)、(\text{加号},3))\}$ 将更新 O_{tmp} 如下 s:

[2]由于 S_E 中的所有符号对都可能发生优先级冲突（即 可以按任一顺序解析），因此它们必须具有相同的次序（这样我们就能用它来索引 M_{to}），或者处于相邻次序。如果它们处于相邻次序，将这两个次序上的符号合并为一个是合理的。

算法 3.2： GenTA：根据结合性、优先级和输入文法生成树自动机。

输入： O_{a}结合性集合，O_{p}优先级顺序集合，$\mathsf{G} = (V, \Sigma, S, P)$ 输入上下文无关文法

输出： $\mathsf{A} = (Q, \mathsf{F}\ \Sigma, Q_f, \Delta)$ 一个树自动机

$m \leftarrow$最大次序 O_{p}；$Q \leftarrow \{e_0, \ldots, e_m\}$；$Q_f \leftarrow \{e_0\}$

G的 $\leftarrow$ 秩符号；G的$_{\mathrm{tr}} \leftarrow$ 平凡符号

$\delta_{\mathcal{F}} \leftarrow \delta$-生成器 *w.r.t.*产生式 P

/* 非平凡符号的转移 */

对于 $(s, i) \in O_p$ **执行循环**

  if $\exists p,\ (s,\ p) \in O_a$ **then**

    // $\delta_s \triangleq e_i \leftarrow_s \ldots e_{i+1} \ldots$ （e_{i+1} 位于位置 p 在右部中，带有 n 前导和 m 尾随非终结符）

    $\Delta \leftarrow \Delta \cup \{\delta_{\mathcal{F}}(e_i,\ [e_i;\ \ldots e_{i+1};\ \ldots e_i],\ s)\}$ // n前导和 m 尾随的 e_i

  else

    $\Delta \leftarrow \Delta \cup \{\delta_{\mathcal{F}}(e_i,\ \overline{e_i},\ s)\}$

/* 平凡符号的转移 */

对 $s' \in \mathcal{F}_{tr}$ **执行**

  $Q \leftarrow Q \cup \{e_{s'}\};\ \Delta \leftarrow \Delta \cup \{\delta_{\mathcal{F}}(e_{s'}, [\,], s')\}$ // *i.e.*, $\delta_{s'} \triangleq e_{s'} \leftarrow_{s'} \alpha$

/* 连接有序状态的转移 */

对于 $i \in [0..m-1]$ **执行**

  $\Delta \leftarrow \Delta \cup \{\delta_{\mathcal{F}}(e_i,\ [e_{i+1}],\ (\epsilon,\ 1))\}$ // *i.e.*$\delta_{(\epsilon,1)} \triangleq e_i \leftarrow_{(\epsilon,1)} e_{i+1}$

/* 按次序处理环 */

对于 $(sl, ol), (sh, oh) \in HighToLow(G, O_p)$ **执行**

  $\Delta \leftarrow \Delta \cup \{\delta_{\mathcal{F}}(e_{oh},\ \overline{e_{ol},\ sh})\}$

返回 $(Q,\ \mathcal{F}\ \Sigma,\ Q_f,\ \Delta)$

```
{((SEMI,2),0), ((PLUS,3),0), ((STAR,3),0)}
                    ↓
{((SEMI,2),0), ((STAR,3),0), ((SEMI,2),1), ((PLUS,3),1)}
```

Here，集合 $S = \{$(分号,2)$\}$，getAtIndex$(G,\ 0)$ 为 (星号,3) 且 getAtIndex$(G,\ 1)$ 为 (加号,3).

该过程保留了未涉及树示例的符号对之间的优先级顺序关系。最后，对于 O_{a}中的所有与结合性相关的符号，我们希望：如果某个符号出现在层级 o，那么该层级上的非冲突符号也出现在层级$o+1$，以便后续构造树自动机。算法中，次序从 o 到 $o + size - 2$ 时会自动满足这一要求，但次序为 $o + size - 1$时可能需要额外执行一次推入和插入操作。此外，LearnOaOp 在重新插入冲突符号时会反复移位并重新分配次序，因此需要 $O(|\mathsf{F}\ |^2)$的时间和 $O(|\mathsf{F}\ |)$的额外空间，其中 $|\mathsf{F}\ |$表示 F 的大小。在上面的例子中，(PLUS,3) 和(STAR,3) 涉及结合性冲突，最终集合 O_p 为：

```
{((SEMI,2),0), ((STAR,3),0), ((SEMI,2),1), ((PLUS,3),1), ((SEMI,2),2)}
```

3.1.3 推断偏好的树自动机。基于结合性集合 O_{a}、优先级顺序集合 O_{p}以及输入上下文无关文法 G 所指定的偏好，算法 3.2 构造一个树自动机 $\mathsf{A} = (Q, \mathsf{F}\ \Sigma, Q_f, \Delta)$，用于对 F 上给定的解析偏好进行编码。其状态集合 Q最初由一系列与不同层级相关联的状态构成，层级范围从 0 到最大阶 m，对应于阶层级中的每个层级：即 $Q \triangleq \{e_0, \ldots, e_m\}$。这些有序状态用于生成转移，以编码$O_{\mathrm{p}}$中指定的符号之间的优先级关系，其中 $Q_f \triangleq \{e_0\}$。

接下来，我们有一个 δ-生成器 $\delta_{\mathcal{F}}$：$Q \times Q^* \times \mathsf{F} \mapsto \Delta$。函数 $\delta_{\mathcal{F}}$接受左侧状态、右侧状态列表以及 F 中的一个符号，以生成在正确位置带有这些符号的转移。换言之，给定由产生式 P在 G中得到的、针对 F 中某一符号的转移结构，$\delta_{\mathcal{F}}$会用左侧状态和右侧状态替换所有非终结符

所提供的状态，同时保留所有终结符。我们用 e_i 表示一个长度适当的列表（其长度由产生式中非终结符的数量决定），其中填充状态 $e_{i\circ}$。例如，给定符号 (如果,4)，当我们提供长度为 2 的 e_1 作为输入状态列表时，$\delta_{\mathcal{F}}$(e1, e1, (如果,4)) 返回 $\mathsf{e}_1 \leftarrow_{(\mathtt{IF},4)}$ 如果 e那么 e_1，这是将原始产生式语句 →如果 表达式 那么 语句中的所有非终结符替换为状态 e_1、同时保持所有终结符如果和那么不变所得的结果。[3] 从映射中检索模板（预定义的转移规则），并用所提供的内容（新的非终结符）替换某些（非终结符）占位符的这一过程，类似于基于模板的代码生成 [30]。

对于每个 $(s,i) \in O_{\mathrm{p}}$，如果存在一对 $(s,p) \in O_{\mathrm{a}}$（即 s涉及结合性冲突），那么就向 Δ添加一条转移规则 $\delta_{\mathcal{F}}(e_i, [e_i; \ldots e_{i+1}; \ldots e_i], s)$，其中右部状态列表包含 n 个前导 e_i，接着是 e_{i+1}，后跟 m 个尾随 $e_{i\circ}$。这对应于 n 个前导非终结符的产生式右部中位置 p 上出现的 $e_{i+1\circ}$。这样可以确保子符号不会出现在第 p 个位置。例如，O_{a} 中的 ((PLUS,3), 1) 与 O_{p} 中的 ((PLUS,3), 2) 一起，会向 Δ添加规则 $\delta_{\mathcal{F}}(e_2, [e_2; e_3], s) \triangleq \mathsf{e}_2 \leftarrow_{(\mathtt{PLUS},3)} \mathsf{e}_2$ 加号 e_3。

当符号 s（次序为 i）不包含在 O_{a} 中时，我们只需添加 $\delta_{\mathcal{F}}(e_i, e_i, s)$。例如，在 $\overline{O}_{\mathrm{p}}$ 中，((如果,4), 1) 会导致将 $\delta_{\mathcal{F}}(e_1, e_1, (\text{如果},4)) \triangleq \mathsf{e}_1 \leftarrow_{(\mathtt{IF},4)}$ 如果 e那么 e_1 添加到 Δ中。

接下来，对于 F_{tr} 中的每个符号 s' 及其关联状态 e'，我们将 $\delta_{\mathcal{F}}(e', [], s')$ 添加到 Δ，并将 e' 添加到Q，从而允许将类似 标识符 $\leftarrow_{(\mathtt{IDENT},1)}$标识符 的转移包含在 Δ中。在为平凡符号添加规则后，我们通过依次生成$(\epsilon, 1)$转移来连接不同层级的状态，从 e_0 到 e_1、从 e_1 到 e_2，依此类推，直到从 e_{m-1} 到 $e_{m\circ}$

为了正确处理符号次序中的环，我们还需要在学习得到的自动机中添加允许这些环的转移。考虑符号 s_l和 s_h，其中 s_h 在 O_{bp} 中的次序高于 s_l，且 G 中存在产生式 $e_l \rightarrow_{s_l} \beta_l$ 和 $e_h \rightarrow_{s_h} \beta_h$，使得$e_l \in \beta_h$ 也就是说，s_l 在解析树中的位置可能比s_h更深。直观而言，需要在自动机中重新引入符号次序中的这种循环关系。函数$HighToLow(G, O_{\mathrm{p}})$可识别此类符号对，并返回 $(s_h, o_h), (s_l, o_l)$，其中 o_h是 s_h的最高次序，且 o_l是 O_{p}中 s_l的最低次序。对于每个此类符号对，我们添加转移 $\delta_{\mathcal{F}}(e_{o_h}, e_{o_l}, s_h)$将它们连接起来。

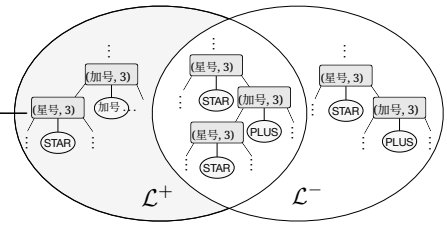


图 10。L^+ 和 L^-中的树。

最后，Δ 中的转移规则以及 Q、$\mathcal{F}$和 Q_f共同构成一个树自动机，如图 7所示。鉴于 $|\mathcal{F}|$是符号的总数，GenTA 的时间和空间复杂度为 $O(|\mathcal{F}|)$。复杂度分析的详细信息见补充材料。

3.1.4 树自动机学习的可靠性。 算法 3.2 在以下意义上是可靠的：它构造的树自动机忠实地编码了结合性和优先级顺序，并且不会丢失输入文法 $\mathcal{G}$

给定 T^+ 用户选中的树，以及 T^- 用户未选中的树，我们定义以下树语言：$\mathcal{L}^+ \triangleq \bigcup_{t \in T^+}$ 解析树(t)和$\mathcal{L}^- \triangleq \bigcup_{t' \in T^-} P^-(t')$。$L^+$是由选定的树示例所描述的所有解析树的集合，而 L^- 是被Greta排除的树的集合；Greta 会严格遵循符号层级，如第 3.1 节所述。这些集合可能存在重叠，如

[3] 在这种情况下，我们用e_1 替换所有非终结符——stmt、expr 和 stmt，因为它被列为唯一用于替换旧状态的新状态，而且 F 中的符号没有关联的非终结符$_{\mathrm{tr}}$。请注意，如果输入列表的长度为 > 1，我们就从输入列表的头部开始，依次用其中的新状态替换每个旧状态（与平凡符号关联的状态除外）。例如，给定结构为 (PLUS, 3)-转移 expr $\leftarrow_{(\mathtt{PLUS},3)}$ expr PLUS expr，应用 δ-生成器 $\delta_{\mathcal{F}}(e_2, [e_2; e_3]$,(PLUS, 3)) 会产生$\mathsf{e}_2 \leftarrow_{(\mathtt{PLUS},3)} \mathsf{e}_2$ PLUS e_3。

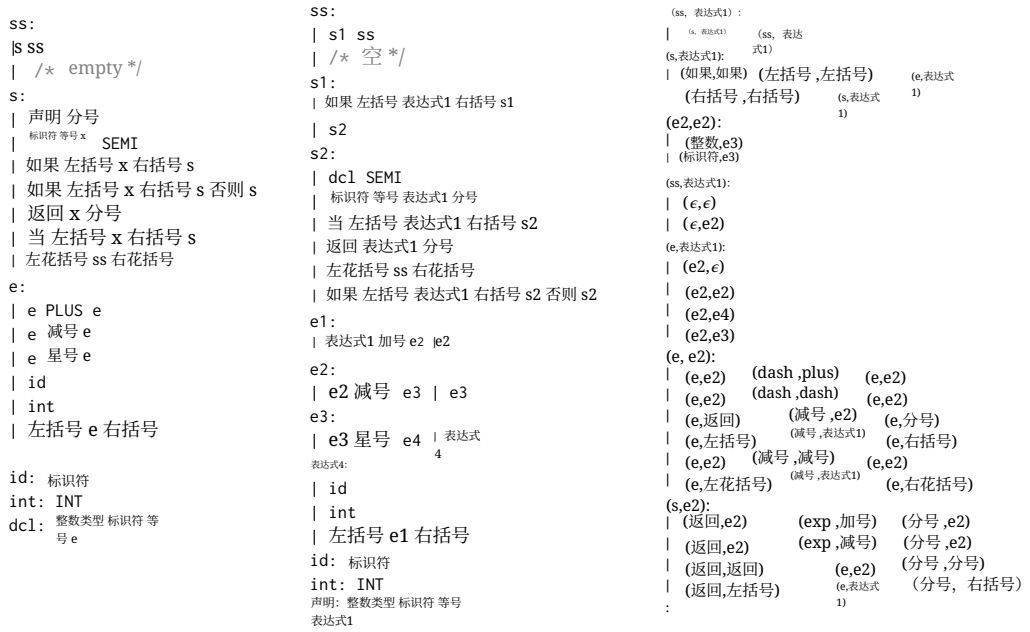


图 11. Greta 返回的消歧文法与简单笛卡尔积的比较。

图 10。我们希望学习得到的自动机拒绝所有由 L^-所描述的树，这也会排除 L^+的这一相交区域。

有了这些定义，我们可以陈述主要的可靠性结果。

定理 *3.1*（*GenTA* 的可靠性）。设 A_r 是算法 *3.2* 返回的有限树自动机，A_g 是由上下文无关文法 G 推导出的树自动机，T^- 是负树示例。此外，$L_r = L(A_r)$、$L_g = L(A_g)$ 和 $L^- = \bigcup_{t \in T^-} P^-(t)$。那么，$L_r \supseteq L_g \setminus L^-$，且 $L_r \cap L^- = \emptyset$。

证明。详见补充材料。 □

第一个陈述断言，A_r 不会丢失 L_g 中任何不必要的解析树，只要这些解析树未被 T^- 中的负例排除（其中包括用户选定的解析树）。第二个陈述断言，A_r 不会接受任何属于 L^- 且已被用户舍弃的树。

最后需要指出，由所得自动机转换而来的上下文无关文法并非可直接用作原始文法的修复方案。这是因为生成该树自动机时，为了纳入原始文法中所有与冲突无关的解析树，它被设计得足够宽松；但这也使树自动机接受了本不应允许的树。例如，A_r 接受`IF 1 THEN 2 PLUS 3` 的解析树，而 A_g 不接受。这引出了我们的下一个贡献：A_r 和 A_g 的交集算法。

3.2 用于上下文无关文法的相交树自动机

我们的交集算法最终计算状态集、接受状态集和转移规则的逐分量笛卡尔积 [10]。在有限树自动机（FTA）交集的标准定义中，这种叉积构造会一致地应用于所有状态和转移，仅根据带秩符号的元数匹配来组合转移，并枚举所有

算法 3.3： IntersectTA

输入： $A_g = (Q_g, F, \Sigma, Q_{gf}, \Delta_g)$ 和 $A_r = (Q_r, F, \Sigma, Q_{rf}, \Delta_r)$ 树自动机
输出： $A = (Q, F, \Sigma, Q_f, \Delta)$ 一个树自动机
$Q_f \leftarrow Q_{gf} \times Q_{rf}$
/* 关于可达状态学习 Δ */
$Q, Q_{tmp} \leftarrow Q_f$
当 $Q_{tmp} \neq \emptyset$ **做**
  对于 $(e_g, e_r) \in Q_{tmp}$ **do**
    $\mathcal{F}_g \leftarrow$ 符号 of e_g in Δ_g; $\mathcal{F}_r \leftarrow$ 符号 of e_r in Δ_r /* 包括从以下对象可达的符号 $(\epsilon, 1)$-转移 */
    $F' \leftarrow F_g \cap F_r$
    对于 $s \in \mathcal{F}'$ **执行**
      $\delta_s^g \leftarrow \delta_s$ 在 Δ_g 中生成 e_g ; $\delta_s^r \leftarrow \delta_s$ 在 Δ_r 中生成 e_r
      $\Delta \leftarrow \Delta \cup \{\delta_s^g \times \delta_s^r\}$
      $Q' \leftarrow \delta_s^g \times \delta_s^r$ 中 (e_g, e_r) 的可达状态
      $Q_{tmp} \leftarrow Q_{tmp} \cup \{q \mid q \in Q' \text{ 和 } q \notin Q\} \setminus \{(e_g, e_r)\}$; $Q \leftarrow Q \cup Q'$
/* 移除重复状态 */
$Q_{dup} \triangleq$ FindDupStates(Q, Δ)
对于 $((x_g, x_r), (y_g, y_r)) \in Q_{dup}$ **执行**
  $Q \leftarrow Q \setminus \{(y_g, y_r)\}$
  $\Delta \leftarrow \Delta \setminus \{\delta \mid \delta$ 生成 $(y_g, y_r)\}$
  将 y_g, y_r 替换为 x_g, x_r，位于 Δ 中
/* 引入 ϵ-转移以简化 Δ */
$L_q \leftarrow Q$ 的有序列表，其 $|\delta|$ 从小到大排列
对于 $i \in [0 .. |L_q|)$ **执行**
  $\Delta_i \leftarrow \{\delta \mid \delta \in \Delta$ 生成 i^{th} Q，其中 $L_q\}$
  对于 $j \in [i+1 .. |L_q|)$ **执行**
    $\Delta_j \leftarrow \{\delta \mid \delta \in \Delta$ 生成 j^{th} Q，其中 $L_q\}$
    如果 $\Delta_i \subset$ 的右部 Δ_j 的右部**那么**
      $\Delta'_j \leftarrow (\Delta_j \setminus \Delta_i) \cup \{e_j \leftarrow_{(\epsilon,1)} e_i\}$
      $\Delta \leftarrow (\Delta \setminus \Delta_j) \cup \Delta'_j$
返回 $(Q, \mathcal{F}, \Sigma, Q_f, \Delta)$

由此产生的状态元组，而不考虑它们是否能够参与接受运行。算法 3.3在两个关键方面偏离了教科书中的构造，而这两个方面对于我们基于文法的场景至关重要：(1) 它会对涉及终结符的转移进行特殊处理（即，生成转移的乘积时，不仅依据元数匹配，还依据右部中终结符和非终结符的精确序列），以及 (2) 它将基于可达性的优化直接纳入构造过程。这是因为：(1) 标准定义没有考虑涉及终结符的转移，导致生成的文法更加复杂，因为需要为所有现有终结符映射状态。这样可以生成更符合惯用表达的文法，如图 11b 中的文法所示，相较于根据图 11c中教科书式的笛卡尔积定义得到的结果而言。我们引入 (2) 是因为，简单地对每个分量取笛卡尔积会产生许多无法到达终态的状态和转移；借助可达性分析，我们可以完全避免计算这些状态和转移。

算法 3.3 概述了针对可达性以及涉及终结符的转移进行优化的算法，以高效计算树自动机的交集。下面我们详细说明其中的一些组成部分。首先，算法通过对输入树自动机的接受状态集合取笛卡尔积，计算接受状态集合 Q_f： Q_{gf} 和 Q_{rf}。由于我们正在求两个树自动机的交集，Q_f 只包含一个状态（即一个状态对，分别由 Q_{gf} 和Q_{rf}中的状态组成）。接下来，算法更新状态集合 Q 和临时状态列表 Q_{tmp}，并将 Q_f加入其中。注意，如果我们只是对 Q_g 和 Q_r 取笛卡尔积来填充 Q，所得集合 Q

算法 3.4： FindDupStates
输入： Q 一组状态，Δ 一组转移
输出： Q_{dup} 一组状态对
对于 $e_i \in Q$ **执行**
  $\Delta_i \leftarrow$ 转移到 e_i 于 Δ
  $\Delta_i \leftarrow$ 将 e_i 替换为e_{tmp} 于 Δ_i
  对于 $e_j \in Q \backslash \{e_i\}$ **执行**
    $\Delta_j \leftarrow$ 转移到e_j 于 Δ
    $\Delta_j \leftarrow$ 将 e_j 替换为e_{tmp} 于 Δ_j
    如果 $\Delta_i = \Delta_j$ **那么**
      $Q_{\text{dup}} \leftarrow Q_{\text{dup}} \cup \{ (e_i, e_j) \}$
返回 Q_{dup}

可能包含不可达状态，因此这些状态无需纳入 Q。因此，我们只将能够到达 Q_{f} 中的终态的状态加入 Q，从而得到一个树自动机。我们通过将首次遇到的状态加入Q_{tmp} 并在生成产生这些状态的转移时取出这些状态，以工作列表的方式维护Q_{tmp}。具体而言，对于 Q_{tmp}中的所有（e_{g}， e_{r}），我们取得所有与（e_{g}， e_{r}）对应的转移，其标签是e_{g} 和 e_{r} 在 Δ_{g} 和 Δ_{r}中分别可能取值的有秩符号的交集，并且可能包含 ϵ-转移。然后，我们将尚未由 Q 收集且从（e_{g}， e_{r}）可到达——即，出现在从（e_{g}， e_{r}）出发的转移右部中的——状态加入 Q和 Q_{tmp}，同时从 Q_{tmp}中移除（e_{g}， e_{r}）。重复这些步骤，将未见过的状态加入 Q，并将新的转移集合加入 Δ，直到 Q_{tmp} 为空。

收集完所有原始转移的笛卡尔积后，我们使用RemoveDupStates （见算法 3.4）识别并移除重复状态及其对应的转移，从而按照树自动机最小化的思路简化树自动机 [10]。最后，我们再次检查 Δ，以确定哪些转移子集在状态间重复，并引入$(\epsilon, 1)$-转移，进一步简化树自动机。此外，设 $|Q_{\text{g}}|$， $|Q_{\text{r}}|$和 $|\Delta_{\text{g}}|$， $|\Delta_{\text{r}}|$分别为两个输入自动机的状态数和转移数。那么，最坏情况下，IntersectTA 的时间复杂度为 $O((|Q_{\text{g}}| \cdot |Q_{\text{r}}|)^2 \cdot |\Delta_g| \cdot |\Delta_r|)$，空间复杂度为 $O((|Q_{\text{g}}| \cdot |Q_{\text{r}}|)^2 + |\Delta_g| \cdot |\Delta_r|)$。复杂度分析的相关细节见补充材料。

定理 3.2（Greta 的正确性）。Greta）。自动机 $\mathcal{A}_r$ （*GenTA* 的结果）与$\mathcal{A}_g$ （从上下文无关文法 $\mathcal{G}$ 导出的自动机）的交集会生成一个识别语言 $\mathcal{L}_g \backslash \mathcal{L}^-$ 的树自动机。

证明。由定理 3.1和集合交集可得。 □

4 实现与评估

我们将所提出的方法实现为一个名为 Greta 的工具。Greta 使用 OCaml 编写，并依赖 OCaml 的标准 LR(1) 语法分析器生成器 Menhir [29]，用于识别给定文法中的歧义。我们使用 Menhir，因为它能够处理比 ocamlyacc（一种 LALR(1) 语法分析器生成器）更复杂的文法，并且能生成更详细、更易理解的错误消息，便于调试有问题的文法，因此它成为 OCaml [24] 中首选的解析器生成器。因此，初始上下文无关文法采用 Menhir 的输入格式表示。随后，根据 Menhir 识别出的每个冲突所涉及的产生式集合，创建树示例。

4.1 交互设计

我们设计 Greta 的用户界面时，尽量减少用户需要做出的选择，因此 Greta 可以在与用户进行至多 N次交互后解决 N歧义。为此，我们会根据 Menhir 的报告，在发生冲突的每个状态下，针对相关词元（即符号）这一组，展示一组选定的语法树备选项。例如，如果某个文法有两个优先级次序

冲突，一个发生在(如果,4) 与(加号,3) 之间，另一个发生在(如果,4) 与(星号,3)之间，那么 Menhir 会报告一个冲突状态。该状态是在处理(如果,4)产生式后到达的，其中涉及的词元是加号和星号。在这种情况下，Greta 只展示一组树，每棵树中符号的深度交替变化，例如，分别是(如果,4) 和(加号,3)。

我们的目标是通过不 要求用户为文法中的每个歧义都选择一个树示例，来减轻用户的交互负担。考虑以下情形，其中有三个歧义：一个涉及(IF,6) 和(STAR,3)，另一个涉及(IF,4) 和(STAR,3)，第三个涉及(IF,4) 和(IF,6)。根据我们的交互设计，我们向用户展示两组树：一组展示符号(IF,6) 和(STAR,3)交替出现的树，另一组展示符号(IF,4) 和(STAR,3)交替出现的树。假设用户选择了优先级关系(IF,6) > (STAR,3)和(IF,4) < (STAR,3)。据此，我们可以推断(IF,4) < (IF,6)。因此，Greta 仅通过两次树选择就解决了三个歧义。请注意，如果用户选择了(IF,6) > (STAR,3) 和(IF,4) > (STAR,3)，Greta 就需要再进行一次用户交互。因此，向用户发出的提示数量不仅取决于输入上下文无关文法中的冲突，也取决于用户在每次交互中选择的树集合。

4.2 实验设置

一次 Greta 运行包含一系列用户提示，用户在每次提示中指定自己对两个树示例的偏好。Greta 随后应用 第 3 节 所述方法生成新的文法，并将其作为新的输入提供给Menhir。在此过程中，用户无需执行或了解 Greta会对树自动机进行任何操作。如果新生成的上下文无关文法仍存在歧义，便会继续运行Greta并进行用户交互，直到 Greta 成功解决所有歧义，或 留下了Greta无法处理的歧义。我们将在 第 4.4 节详细讨论这些无法处理的歧义。因此，Greta的端到端工作流包含一系列提示。由于每个提示都会提供两个树示例，所有场景的数量会随着提示数量呈指数增长，很快就会变得难以手动测试。因此，我们的测试框架使用 Expect [22], 这一终端文本界面自动化工具，自动完成此交互。此外，由于我们不涉及真实用户，因此会展示每种文法中的所有歧义，并测试所有可能的场景。所有实验均在一台运行 Ubuntu 22.04 的普通计算机上进行，该计算机配备 16GB 内存和 16 个逻辑 CPU 核心。

基准测试。 为了评估我们的方法，我们收集了各种文法，其规模在终结符、非终结符和产生式的数量方面各不相同，如表 1前三列所示。 $G0$ 与图 1中的运行示例完全相同。 $G1$ 与 $G0$ 相似，但更有意思，因为它允许二元运算的符号（加号,3） 和（星号,3） 与符号（如果,4） 和（如果,6）发生冲突，因此该文法可能存在更多歧义。接下来的三个文法 $G2$ 到 $G4$ 取自一门热门编译器设计课程的作业。 $G2$ 描述了一种简单的布尔语言， $G3$ 和 $G4$ 则定义了更复杂的语言，除了布尔表达式外，还包含 while 循环和各种二元运算。 $G5$ 和 $G6$ 收集自 StackOverflow 上发布的问题。 $G5$ 是一个为逻辑表达式编写的简单文法，而 $G6$描述了一种用于表达含不等式和二元运算的约束的语言。[5] $G1$ 到 $G6$有 2 或 3 个变体，即 GNa 到 GNb（或 GNc）； 这些变体包含数量或类型各异的歧义，用于观察随着歧义数量增加，Greta 在准确性和速度方面的表现。

我们编译了真实世界语言的实用文法： $G7$、 $G8$和 $G9$。 $G7$ 是用于指定 Tezos 区块链上智能合约的 Michelson 文法，而 $G8$ 是

[4] https://stackoverflow.com/questions/910445/issue-resolving-a-shift-reduce-conflict-in-my-grammar。
[5] https://stackoverflow.com/questions/4588397/fixing-lemon-parsing-confilcts?rq=3。

表1。Greta 运行的汇总结果。该表列出了终结符数量（|Σ|）、非终结符数量（|V|）、产生式数量（|P|）、Menhir 报告的各文法歧义总数（$A_{p,\ a}$），其中 p 指优先级顺序方面的歧义数量（关于优先级顺序），a 指结合性方面的歧义数量（关于结合性；平均提示数量（Δ）；将文法转换为树自动机所花费的时间（毫秒，Conv）；树自动机学习所花费的时间（毫秒，Learn）；树自动机交集时间（毫秒），并以下标形式标出成功消歧的场景百分比（$I^{\text{def}}_{\%}$）；未采用基于可达性的优化（$I^{1}_{\%}$）、未采用重复项删除优化（$I^{2}_{\%}$）、未采用 ε 引入优化（$I^{3}_{\%}$），以及未采用上述全部三项优化（$I^{123}_{\%}$）时的树自动机交集时间（毫秒），每项均以相应百分比作为下标；手动消歧总耗时（秒，T_{man}）；以及手动消歧所需的产生式编辑数量（Δ_{man}）。

	|Σ|	|V|	|P|	$A_{p,a}$	Δ	Conv	学习	$I^{\text{def}}_{\%}$	$I^{1}_{\%}$	$I^{2}_{\%}$	$I^{3}_{\%}$	$I^{123}_{\%}$	T_{man}	Δ_{man}
$G0$	13	5	10	$5_{2,2}$	4	0.06	1.68	0.62_{50}	3.06_{50}	0.91_{0}	0.50_{20}	2.21_{0}	265	9.5
$G1a$	11	4	10	$4_{3,1}$	4	0.06	0.75	0.84_{46}	3.53_{46}	1.36_{0}	0.67_{15}	3.02_{0}	163	6
$G1b$	11	4	10	$7_{6,1}$	7	0.06	0.93	0.83_{47}	2.42_{47}	0.97_{0}	0.58_{23}	1.85_{0}	202	7
$G1c$	11	3	9	$9_{6,2}$	8	0.05	1.07	0.80_{44}	1.94_{44}	0.89_{0}	0.62_{6}	1.36_{0}	169	9.5
平均值				$7_{5,1}$	6.3	0.06	0.92	0.82_{45}	2.63_{46}	1.08_{0}	0.62_{15}	2.08_{0}	178	7.5
$G2a$	10	3	10	$4_{2,2}$	4	0.07	0.72	0.94_{100}	3.15_{100}	1.76_{100}	0.86_{100}	2.97_{100}	294	12
$G2b$	10	3	10	$6_{3,2}$	5	0.06	0.76	0.68_{100}	1.56_{100}	0.83_{100}	0.53_{100}	1.26_{100}	123	8
$G2c$	10	2	9	$12_{6,3}$	9	0.05	1.20	0.83_{100}	1.52_{100}	0.92_{100}	0.62_{100}	1.05_{100}	178	12
平均值				$7_{4,2}$	6.0	0.06	0.89	0.82_{100}	2.08_{100}	1.17_{100}	0.67_{100}	1.76_{100}	198	10.7
$G3a$	17	8	20	$2_{1,1}$	3	0.13	1.01	3.39_{50}	10.71_{50}	4.16_{0}	2.97_{11}	8.75_{0}	244	14
$G3b$	18	9	21	$3_{2,1}$	6	0.10	1.01	1.63_{50}	8.23_{50}	2.54_{0}	1.49_{11}	8.09_{0}	209	12.5
平均值				$2_{2,1}$	4.5	0.12	1.01	2.51_{50}	9.47_{50}	3.35_{0}	2.23_{11}	8.42_{0}	226	13.2
$G4a$	25	10	26	$3_{1,2}$	3	0.17	1.10	2.64_{100}	15.20_{100}	3.51_{100}	2.63_{100}	8.23_{100}	123	3
$G4b$	25	8	24	$12_{6,3}$	9	0.12	1.36	1.81_{100}	7.47_{100}	2.52_{100}	1.36_{100}	7.05_{100}	176	10.5
$G4c$	25	8	24	$16_{6,4}$	10	0.11	1.45	1.67_{100}	7.58_{100}	2.35_{100}	1.17_{100}	7.13_{100}	154	10
平均值				$10_{4,3}$	7.3	0.13	1.30	2.04_{100}	10.09_{100}	2.80_{100}	1.72_{100}	7.47_{100}	151	7.8
$G5a$	8	4	9	$2_{1,1}$	2	0.06	0.47	0.64_{100}	2.09_{100}	0.90_{100}	0.49_{100}	1.93_{100}	80	4
$G5b$	8	4	9	$2_{1,1}$	3	0.06	0.61	0.73_{100}	3.13_{100}	0.82_{100}	0.58_{100}	2.34_{100}	56	5
$G5c$	8	2	7	$6_{3,2}$	5	0.04	0.71	0.49_{100}	0.84_{100}	0.58_{100}	0.41_{100}	0.71_{100}	86	8
平均值				$3_{2,1}$	3.3	0.05	0.60	0.62_{100}	2.02_{100}	0.76_{100}	0.50_{100}	1.66_{100}	74	5.7
$G6a$	21	5	23	$2_{1,1}$	2	0.19	1.03	2.24_{100}	4.80_{100}	2.88_{100}	2.33_{100}	5.02_{100}	84	5
$G6b$	21	5	23	$14_{7,4}$	11	0.16	1.75	1.96_{100}	8.22_{100}	2.72_{100}	1.38_{100}	6.59_{100}	220	14.5
$G6c$	21	5	23	$18_{8,6}$	14	0.13	2.03	2.28_{100}	8.32_{100}	3.21_{100}	1.54_{100}	6.71_{100}	266	17
平均值				$11_{5,4}$	9.0	0.16	1.60	2.16_{100}	7.12_{100}	2.94_{100}	1.75_{100}	6.11_{100}	190	12.2
$G7$	56	12	77	$3_{2,1}$	3	0.51	2.47	4.79_{100}	24.67_{100}	5.60_{100}	4.01_{100}	19.45_{100}	337	8.5
$G8$	37	32	72	$9_{3,3}$	6	116.19	127.02	1.92_{100}	5828.16_{100}	2.89_{100}	1.51_{100}	242.55_{100}	305	6.5
$G9$	29	18	42	$23_{7,9}$	16	0.24	2.81	5.14_{100}	155.44_{100}	6.45_{100}	3.26_{100}	23.77_{100}	549	25

Kaitai 文法是一种声明式语言，用于描述 Tezos 中的二进制结构。[6] $G7$ 文法取自公开可用的 Michelson 文法子集，[7] 并结合了更多来自 Michelson 参考资料的构造。[8] 最后，$G9$ 它是 SQL 语言的一个子集。SQL 是一种用于访问和操作数据库的标准语言。

4.3 实验结果

Greta 已完成运行的汇总结果见表 1。各基准来源中的文法按规模递增排列，规模可通过以下指标近似衡量：

[6] https://gitlab.com/tezos/tezos/-/tree/master/client-libs/kaitai-ocaml [7] https://github.com/aigarashi/ocaml_的_michelson [8] https://tezos.gitlab.io/michelson-reference/

产生式数量 $|P|$，以及终结符 $|\Sigma|$和非终结符 $|V|$的数量。若某个文法存在多个变体，则按歧义数量递增排序，如 A_p、$_a$ 列所示，其中 p 表示与优先级顺序相关的数量，a 表示与结合性相关的数量。除默认配置（$I^{\text{def}}_{\%}$）外，我们还报告了交集算法在若干消融设置下的结果，见表 1 中的 $I^1_{\%}$、$I^2_{\%}$、$I^3_{\%}$和 $I^{123}_{\%}$列。[9] 这些消融分别禁用了算法 3.3 中引入的优化：(i) I^1 禁用对所有转移计算原始笛卡尔积，但不限制为通过学习到的 Δ可达的状态；(ii) I^2禁用重复状态消除；(iii) I^3 禁用 ϵ引入；(iv) I^{123} 禁用全部三项优化。这样的细分有助于我们分别评估每项优化对 Greta 有效性和效率的影响。为评估结果，我们探讨以下研究问题：

- RQ1：Greta 在消除文法中的所有 歧义方面有多有效？哪些因素会影响其性能？
- RQ2：歧义数量或文法规模的增加会如何影响 Greta 的效率？
- RQ3：对于输入文法规模或歧义数量，Greta 的可扩展性如何？

4.3.1 RQ1：有效性。总体而言，Greta 成功修复了二义文法，平均修复率为 85%。十种文法中有七种的修复率达到 100%；最低修复率来自 $G1$，其大多数失败主要是因为 Menhir（一种 LR(1) 语法分析器）每次只能向前查看一个词元，无法区分语法分析选项之间的差异，而这些差异若能向前多查看几个词元便可明确区分。换句话说，由于 Greta 依赖 Menhir 来识别歧义，因此有时即使底层的上下文无关文法中不存在歧义，Greta 仍会报告修复后的文法中存在剩余歧义。在这种情况下，Greta 尝试移除禁用树也不会进一步改变文法。

这些情况被报告为失败，导致 $G0$、$G1$ 和 $G3$ 在表 1中的固定率较低。例如，对于 $G0$，如果选中一棵指定 `(IF,6) < (IF,4)` 的树，Greta会生成一个包含以下片段的文法，见图 12。给定这些产生式，假设 Menhir 尝试

```
x2 → 如果 cond 那么 x2 否则 x2 x
2 → x3 x3 → 如果 cond 那么 x3 :
```

图 12. 产生式关于(IF,6) < (IF,4)

解析类似 `IF TRUE THEN IFTRUE THEN 1 ELSE 2` 这样的表达式。可以先应用规则 `x2 →IF cond THEN x2 ELSE x2`，然后在非终结符 `x2` 处，即THEN之后，应用 (1) 同一条规则或(2) ϵ-转移到 `x3`，接着应用`x3 →IF cond THEN x3`，尽管 (1) 会在程序中产生数量不同的ELSE。另一方面，如果选中 `(IF,4) < (IF,6)`，生成图 13中的转移，那么 Menhir 不再报告歧义，因为解析只有一种方式：首先应用`x2 →IF cond x2`，然后在非终结符`x2`处，即 THEN之后，ϵ-转移到`x3`，并应用`x3 → IF cond THEN x3 ELSE x3`。

在所有基准测试中，%）的 $I^1_{\%}$成功率与默认配置（$I^{\text{def}}_{\%}$）相同。尽管 $I^1_{\%}$ 使用不受限制的交叉乘积执行交集，但后续的重复状态消除和ϵ引入步骤确保生成的自动机和文法与

```
x2 → 如果 条件 那么 x2 x2 →
x3 x3 → 如果 条件 那么 x3 否则
x3:
```

图 13。 产生式关于(如果,4) < (如果,6)

所有优化。这表明优化 1 主要提升的是效率，而非正确性。相比之下，禁用重复状态消除（$I^2_{\%}$）或 ϵ 引入（$I^3{}_{\%}$）会显著降低若干文法的有效性，尤其是 $G0$、$G1$s 和 $G3$s。在这些情况下，生成的树自动机和上下文无关文法在语义上仍然正确，但包含冗余或重叠状态

[9]带下标的百分比% 表示每种方法下的成功率。

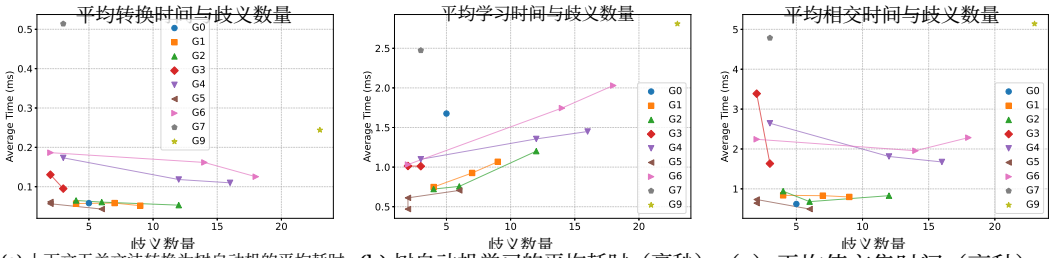
以及非终结符，使得该文法不适合作为确定性 LR 解析器生成器的输入。这凸显了一个重要区别：树自动机和上下文无关文法可以容忍冗余，但确定性解析不能。因此，重复项消除和 ϵ 引入等优化不仅是性能改进，也是确保兼容 LR 解析所必需的；Greta 通过 Menhir 依赖 LR 解析。此外，禁用所有优化（$I_{\%}^{123}$）会使这些问题叠加，导致同一文法的成功率始终偏低。

对于几乎所有基准测试数据，至少存在一个场景，即一种用户选择的组合，可以消除所有歧义。这意味着，Greta 可以用来提供语法分析选项，最终使文法无冲突；对于正在寻找消除文法歧义方法的用户而言，这可能很有用。要完全修复一个平均含有 8 个歧义的文法，大约需要 6.3 条提示。这体现了我们交互设计的优势，因为平均而言，解决 N 个歧义确实只需要少于 N 条提示。

4.3.2 研究问题 *2*：效率。Greta 能够快速地对不同文法进行消歧，平均运行时间不到 20ms （不包括用户交互时间），用户几乎无法察觉。测得的时间是成功案例所用时间的平均值，即，在这些案例中，所测试的场景能够消除所有歧义。如表 1所示，运行总时间分为三个部分，分别是转换（Conv）、树自动机学习（Learn），以及启用所有优化后的树自动机交集（$I_{\%}^{\text{def}}$）。在大多数文法中，转换耗时最少，其次是学习，而交集通常耗时最多。通过算法 3.3中的优化，算法的复杂度从通常严格随产生式数量呈二次增长，提升为更高效的算法。对于产生式数量最多且需要计算大量可达状态的 $G7$，交集耗时最长。如图14a 和图14c所示，转换和交集耗时似乎基本不受歧义数量影响。 图14b 显示，学习耗时随相对于歧义数量的增加而显著变慢，呈现出更明显的上升趋势。

$G8$ 是规模较大的文法之一，其学习和转换时间明显长于其他文法。我们认为，这很可能是因为它的状态数量较多，同时产生式数量也很大，而我们的实现中，转换和学习所用的记账过程具有二次复杂度。相比之下， $G8$ 的默认交集时间仍然相对较短。这是因为交集只针对可达状态对进行计算，从而大幅缩小了有效搜索空间。消融结果证实了这一优化的效果：禁用基于可达性的剪枝（I^1 和 I^{123}）后， $G8$ 的交集时间会增加几个数量级。这表明，对于状态和产生式数量较多的文法，基于可达性的优化对于控制交集开销至关重要。也就是说，虽然大型文法可能导致转换和学习开销高昂，但交集优化能够有效防止状态爆炸主导总运行时间。

4.3.3 RQ3：可扩展性。我们考察输入文法规模（即，产生式数量 $|P|$、终结符数量 $|\Sigma|$以及非终结符数量 $|V|$）和歧义数量增加时的性能趋势，以评估可扩展性。在各个文法族中 $G1$-$G6$，歧义数量增加对转换或交集时间的影响适中，而学习时间则呈现低于线性的增长趋势，如图14所示。这表明，歧义数量增加对学习时间的影响适中，说明Greta 仅就歧义数量而言具有良好的可扩展性。另一方面，产生式和符号数量庞大的文法，或非终结符结构复杂的文法（例如，非终结符深度嵌套或相互递归）会带来更高的成本，这反映出所构造的自动机规模有所增大。例如，即使对于歧义数量超过 20 的文法（例如， $G9$），Greta 仍能在几毫秒内成功消除所有受支持的歧义。此外，Greta 平均修复耗时最长的是 $G8$，



(a) 上下文无关文法转换为树自动机的平均耗时（毫秒）与歧义数量的关系适用于不同文法类型。

(b) 树自动机学习的平均耗时（毫秒）相对于不同歧义数量的不同文法类型。

（c）平均值交集时间（毫秒）相对于不同歧义数量的不同文法类型。

图14。不同歧义数量下各种文法类型的性能指标。每个点代表一个文法变体，标记表示文法族。

尽管其歧义数量少于 *G*2c、 *G*4b-c、 *G*6b-c 和 *G*9等文法，但由于其产生式集规模庞大，且其中的产生式包含大量相互递归的非终结符。这些结果表明，Greta 的可扩展性与其说受歧义数量本身影响，不如说取决于文法的规模和结构复杂度。

尽管这些趋势可能表明，Greta 在歧义数量方面具有良好的扩展性，在文法规模方面也具备相当不错的扩展性，但仍有一个注意事项：测得的时间来自成功运行，而规模更大、复杂度更高的文法可能包含超出Greta 或Menhir 的Greta 所依赖的范围的边缘场景。这意味着，Greta的可扩展性受其范围与局限性制约，我们将在第 4.4 节进一步讨论。

4.3.4 手动消歧。为评估在不使用*Greta*的情况下修复文法所需的工作量，两位作者分别仅使用 Menhir 对每个文法进行手动消歧。手动消歧的过程包括反复检查 Menhir 的错误消息和冲突报告，编辑文法产生式以解决报告的歧义，然后重新运行 Menhir，直至不再报告冲突。两位作者对 Menhir 的输入格式以及冲突诊断信息的解读都同样熟悉。

对于每个文法，我们记录了手动修复所用的总时间和所做的产生式编辑次数；T_{man} 和 Δ_{man} 的表 1中的数值分别为耗时和编辑次数的平均值。实际上，即使对于歧义数量不多的文法，手动消歧通常也需要数分钟，而非（毫）秒，并且需要编辑数十个产生式。此外，手动修复有时会出错或限制过严，导致有效解析被意外排除。这类情况需要撤销或修改先前的改动，进一步增加了总体工作量。造成这一成本的一个重要因素，是向用户呈现歧义的形式：Menhir 在 LR 自动机状态和动作层级报告冲突，因此用户必须重构相互竞争的解析结构，并在修改文法之前推断预期的消歧方式。补充材料中提供了一个示例，展示 Menhir 与Greta 的用户界面，突显了两者向用户呈现歧义的方式有何不同。

4.4 范围与局限

4.4.1 Menhir 与 *Greta*：所处理歧义的范围。 虽然 Greta 依赖 Menhir 的歧义检测机制，但 Menhir 会报告 LR(1) 解析器构造过程中出现的歧义的过度近似，形式为移进/归约冲突和归约/归约冲突。这些冲突可能由多种因素引起，例如运算符优先级和结合性、悬空 else 歧义、文法欠规范，或与真正的句法歧义无关的 LR(1) 特有限制。在 Menhir 检测到的这些 LR(1) 冲突中，Greta 处理其中一部分

：也就是说，解析偏好可以通过选择两个备选树示例之一来表达的歧义，详见 第 3.1 节。换言之，Menhir 报告的冲突并非都能由 Greta处理；我们将在 第 4.4.3 节详细讨论这些不可处理的冲突。因此，Menhir 充当冲突检测后端，而 Greta 则专注于修复那些可通过示例驱动的文法约束解决的冲突。

4.4.2 终止行为。 Greta 通过迭代修复输入文法，并在生成的文法上重新调用 Menhir 来运行。该过程会在以下三种情况之一发生时终止：(i)Menhir 报告不存在剩余冲突；(ii) 剩余冲突不属于 Greta基于示例的歧义类别（第 3.1 节）；或 (iii) Menhir报告的冲突是假阳性，无法通过进一步限制文法来消除。在情况 (ii) 和 (iii) 中，Greta 会检测到无法取得进一步进展，随即停止，不再尝试后续迭代，并向用户报告剩余冲突无法处理。

用户不会陷入无限循环。每当 Greta 应用一个修复步骤时，生成的文法所允许的解析树集合都是前一个文法所允许解析树集合的严格子集。由于原始文法产出有限的解析树和冲突集合，这种对可接受解析树的单调缩减保证了过程终止。下面我们将讨论属于情况 (ii) 和 (iii) 的冲突类型，因此这些冲突构成了 Greta 的固有限制。

4.4.3 局限性与不可处理的冲突。 Greta 旨在解决能够表示为基于示例的二元选择(第 3.1 节, 第 4.4.1 节)的歧义，从而提供轻量且直观的交互模型。然而，这种设计选择也对可处理的歧义类别施加了固有的限制。如第 4.4.2 节所述，在两种情况下，Greta 可能会在尚未解决所有已报告冲突的情况下终止：(1) 由于Menhir的局限性而报告的假阳性；(2) 由于底层解析偏好无法表示为两个备选项，因而无法构造树示例的歧义。在这两种情况下，剩余冲突都超出了Greta 的处理范围，因此会被报告为不可处理的冲突。

首先，情况 (1) 的出现源于Greta依赖于Menhir，如第 4.3.1 节所述。解决这一问题的自然方式是采用不受前瞻标记数量限制的解析器。例如，dypgen [25] 是一个不受固定数量前瞻标记限制的 GLR 解析器生成器。另一种选择是，Earley [27]允许构造一个具有无限数量前瞻标记的解析器；该解析器基于一种通过动态规划生成可能解析结果图表的算法。虽然这些都是可行的选择，但潜在问题在于效率低下（两者的时间复杂度均为立方时间复杂度，而Menhir的时间复杂度为线性时间），此外，对于Earley，输入上下文无关文法需要用函数组合的方式编写，因此定义解析器需要更多工作。此外，情况 (2) 可能发生于无法按照第 3.1 节构造出树示例时。简单来说，这可以理解为有三个或更多不同的产生式共同造成歧义，因此难以将解析偏好表示为两个不同的备选项。

5 相关工作

语法修复。 我们的工作受到了 Adams 和 Might 的方法启发 [2]。在他们的工作中，将用于描述不期望的解析树集合的树正则表达式作为输入，并与对应于输入二义性上下文无关文法的树自动机求交。这一过程的时间复杂度会随着负例数量的增加而呈指数级增长，并且 文法规模也会导致这种增长（这是取反操作造成的）。我们的方法除了为用户提供比编写形式化树正则表达式更简单的界面之外，第 3.1 节 中的学习算法还会构造一个 编码所有用户偏好的树自动机，并执行一次交集操作；该操作的复杂度相对于自动机的大小为二次，与示例数量无关。此外，我们的

自动机学习步骤的复杂度与示例数量呈线性关系，因此这种方法的效率和可扩展性都显著更高。

其他已有研究提出了解决文法歧义的方法 [11, 17, 32] 或修复文 [28]法的方法。文法可以通过声明式消歧规则进行修复，这些规则称为过滤器 [11, 17, 32]，它们会在解析后作为一个步骤，或嵌入生成的解析器中，移除不需要的解析树。与我们的方法不同，过滤器无法更新文法以纳入消歧规则。这限制了人们对解析限制与原始上下文无关文法之间交互的理解，因为要弄清上下文无关文法实际接受哪些解析树，就需要经常查阅消歧规则。另一种自动化语法修复技术 [28]旨在修复未通过给定测试套件中测试的文法，并假定正确的文法确实存在。该技术通过迭代应用一组定制补丁来修正错误文法。Greta 不依赖固定的一组重写规则，而是利用一种基于用户所选示例构建的通用树自动机机制。

通过 *Angluin* 算法进行树自动机学习。 学习确定性有限自动机的一种著名方法是 Angluin 的 L* 算法 [4, 5], , 它依赖预言机，并通过预言机查询维护观察表，随后用于构造确定性有限自动机。尽管已有工作将 L* 算法的思想应用于树自动机学习 [7], , 但我们发现在本设定中难以采用该方法。这是因为 L* 算法假定存在一个能够回答成员查询的预言机（即，某棵树是否属于某个树自动机），而在我们的示例编程设定中提供这样的预言机将颇具挑战。

我们最初考虑将与用户的交互视为预言机，并在学习循环中引入用户提示。然而，这种方法既不可取也无必要，因为用于树自动机学习的 L* 算法需要一组有代表性的示例树，以便每个树自动机转移至少被一个示例所触发。这将要求：(a) 构造涉及未参与任何冲突的符号的示例；(b) 与用户进行大量交互以正确模拟预言机，从而背离本工具的初衷，尤其是在给定文法规模较大时。另一方面，我们的学习算法除了在要求用户指定其偏好的树示例关于歧义时之外，无需让用户参与。或者，我们可以将输入上下文无关文法——更准确地说，是由该上下文无关文法转换而来的树自动机——视为预言机。然而，这样的预言机无法正确回答关于额外的用户提供的限制的成员查询。

示例编程。 示例编程（基于示例的编程）是指根据一组输入输出示例进行程序合成。基于示例的编程已被用于 Leung等人的工作 [21], 中的文法合成，该工作根据用户提供的解析树示例构造解析器。这项工作通过指定结合性和优先级顺序规则来编码解析限制，类似于 Greta 中语法限制规则的构造。这些规则随后被用于移除不需要的解析树。该方法要求用户提供能够完整刻画目标语言的示例，因为该方法不允许将现有文法作为上下文提供。对于大型语言，或对于扩展现有文法，这很快就会变得不切实际。我们的技术以现有文法作为上下文，并以用户偏好作为额外约束，从而生成更新后的文法。这简化了从零编写文法的过程，同时也支持对现有文法进行增量更新和维护。

Wang 等人 [34]针对表格数据补全任务，采用基于示例的编程来学习树自动机。与其他基于示例的编程方法类似，这需要用户提供示例，并通过用于推理表格数据的领域特定语言（DSL）中的公式加以约束。在 Wang 等人的工作中，树自动机被用于紧凑地表示用户示例，但其所解决的问题

与我们的根本不同：该方法专门针对表格数据补全任务，并且高度依赖用户提供正确且充分的示例。

6 结论

在这项工作中，我们提出了一种新颖的思路来解决一个几乎与编程语言领域本身一样古老的问题——如何以确定且不存在解析歧义的方式指定编程语言的语法 [3]。具体而言，我们将消除上下文无关文法歧义的问题视为示例编程的一个实例，并将语法修复（即消除歧义）的过程组织成一系列与文法设计者进行的轻量级交互，由文法设计者通过选择自己偏好的解析树来引导修复。

我们认为，我们方法的核心思想——将成对互补的正/负树示例编译为树自动机，并用其细化最初提供的“基础”文法——其应用并不局限于消除上下文无关文法中的歧义。具体而言，基于树自动机的示例表示可用作指导形式文法设计的工具，使编程语言设计者和用于文法学习的自动化工具都能从中受益。

致谢

我们感谢 Hila Peleg 和 George Pîrlea 对本文草稿提出的意见，也感谢匿名的 OOPSLA’26 审稿人。本研究部分得到了新加坡教育部（MoE）Tier 3 项目“自动程序修复”（MOE-MOET32021-0001）的资助。

数据可用性

本文附带的软件制品可在线获取 [20]。其中包含 Greta 的 OCaml 实现，包括 GenTA (第 3.1 节) 和 IntersectTA (第 3.2 节)，以及基准测试数据和构建脚本，用于复现第 4 节 中报告的评估结果。

参考文献

[1] Paul W. Abrahams。1966。关于 ALGOL 及相关语言中悬空 else 问题的最终解决方案。Commun. ACM（），–。doi: . / .[2] 2017 Michael D. Adams 和 Matthew Might。2017。使用树自动机限制文法。ACM 程序设计语言会议论文集 1，OOPSLA（2017），82:1–82:25。doi:10.1145/3133906[3] Alfred V. Aho、Stephen C. Johnson 和 Jeffrey D. Ullman。1973。二义文法的确定性解析。载于 POPL。ACM Press，1–21。doi:10.1145/512927.512928[4] Dana Angluin。1981。关于识别正则语言所需查询次数的注记。Inf. Control. 51，1（1981），76 87 10 1016 0019 9958 81 90090 5 –。doi: . /S - ( ) - Dana Angluin。。通过查询和反例学习正则集合。Inf. Comput. 75，2（1987），87–106。doi: 10 1016 0890 5401 87 90052 6 . / - ( ) - Osbert Bastani、Rahul Sharma、Alex Aiken 和 Percy Liang。2017。合成程序输入文法。载于 PLDI。ACM，95–110。doi:10.1145/3062341.3062349[7] Jérôme Besombes 和 Jean-Yves Marion。2007。通过正例和成员查询学习树语言。Theor. Comput. Sci. 382, 3 2007 183 197 10 1016 2007 03 038 （），–。doi: . /J.TCS. . . Leon Bettscheider 和 Andreas Zeller。。看，妈妈，没有输入样本！通过符号解析从代码中挖掘输入文法。载于 FSE。ACM，522 526 10 1145 3663529 3663790 –。doi: . / . John Boyland 和 Daniel Spiewak。。工具论文：ScalaBison 递归式升降分析器生成器。253，7（2009），65–74。doi:10.1016/J.ENTCS.2010.08.032[10] Hubert Comon、Max Dauchet、Rémi Gilleron、Florent Jacquemard、Denis Lugiez、Christof Löding、Sophie Tison 和 Marc Tommasi。2008。树自动机技术与应用。https://inria.hal.science/hal-03367725v1/file/tata.pdf。[11] Luís Eduardo de Souza Amorim、Michael J. Steindorfer 和 Eelco Visser。2018。迈向深度优先级冲突的零开销消歧。《编程的艺术、科学与工程》2，3（2018年3月）。doi:10.22152/programming- journal.org/ 2018 2 13 / / Alexander Demenchuk。2008。Beaver——LALR 分析器生成器。https://beaver.sourceforge.net/。

[13] Joost Engelfriet。2015。树自动机与树文法。CoRR abs/1510.02036（2015）。arXiv:1510.02036 http: // arxiv.org/abs/ 1510 02036 。Rahul Gopinath、Björn Mathis 和 Andreas Zeller。2020。从动态控制流中挖掘输入文法。[14] 载于 ESEC/FSE。ACM，172–183。doi:10.1145/3368089.3409679[15] Sumit Gulwani。2011。使用输入输出示例自动化电子表格中的字符串处理。载于 POPL，Thomas Ball 和 Mooly Sagiv（编）。ACM，317 330 10 1145 1926385 1926423 –。doi: . / . Stephen C Johnson 等。Yacc：又一个编译器生成器。第[16] 卷。1975 贝尔实验室，Murray Hill，新泽西州。[17] 32 Paul Klint 和 Eelco Visser。1994。使用过滤器消解上下文无关文法的歧义。载于 ASMICS 解析理论研讨会论文集。意大利米兰，1–20。[18] Vu Le 和 Sumit Gulwani。2014。FlashExtract：一个通过示例进行数据提取的框架。载于 PLDI。ACM，542–553。doi:10.1145/2594291.2594333[19] Xuan-Bach Dinh Le、Duc-Hiep Chu、David Lo、Claire Le Goues 和 Willem Visser。2017。S3：通过示例编程进行语法与语义引导的修复合成。载于 ESEC/FSE。ACM，593 604 10 1145 3106237 3106309 –。doi: . / . Yunjeong Lee、Gokul Rajiv 和 Ilya Sergey。[20] 2026 。Greta：使用示例和树自动机进行语法修复（制品）。doi:10.5281/zenodo.17322761 工作仓库位于 https://github.com/verse-lab/greta。[21] Alan Leung、John Sarracino 和 Sorin Lerner。2015。通过示例进行交互式解析器合成。载于 PLDI。ACM，565–574。doi:10.1145/2737924.2738002[22] Don Libes。1995。探索 Expect——用于自动化交互式程序的 Tcl 工具包（第 2 版）。O'Reilly。[23] José Nuno Macedo 和 João Saraiva。2020。将消歧过滤器表示为组合子。载于 SAC。ACM，1348–1351。doi:10.1145/3341105.3374123[24] Anil Madhavapeddy 和 Yaron Minsky。2022。使用 OCamllex 和 Menhir 进行解析（第 2 版）。剑桥大学出版社，英国剑桥，第 19 章，361–373。[25] Emmanuel Onzon。2012。dypgen：用户手册。http://dypgen.free.fr/。[26] Hila Peleg、Sharon Shoham 和 Eran Yahav。2018。不只通过示例编程。载于 ICSE。ACM，1114–1124。doi:10.1145/3180155.3180189[27] Christophe Raffalli 和 Rodolphe Lepigre。2020。earley：基于 Earley 算法的解析库。https://opam. ocaml.org/packages/earley/。[28] Moeketsi Raselimo 和 Bernd Fischer。2021。自动语法修复。载于 SLE。ACM，126–142。doi:10.1145/3486608. 3486910[29] Yann Régis-Gianas 和 François Pottier。2016。Menhir 参考手册。https://gallium.inria.fr/~fpottier/menhir/。[30] Eugene Syriani、Lechanceux Luhunu 和 Houari A. Sahraoui。2018。基于模板的代码生成系统映射研究。计算语言、系统与结构 52 2018 43 62 10 1016 2017 11 003 （），–。doi: . /J.CL. . . Mark van den Brand、Jeroen Scheerder、Jurgen J. Vinju 和 Eelco Visser。[31] 2002 。无扫描器广义 LR 解析器的消歧过滤器。载于 CC（LNCS，第 *2304* 143 158 10 1007 3 540 45937 5_12 卷）。施普林格，–。doi: . / - - - E. Visser。[32] 1997 。通过消歧过滤器优化解析模式的案例研究。载于国际解析技术研讨会。麻省理工学院，波士顿，210–224。[33] Chenglong Wang、Alvin Cheung 和 Rastislav Bodík。2017。根据输入输出示例合成高表达能力的 SQL 查询。载于 PLDI。ACM，452 466 10 1145 3062341 3062365 –。doi: . / . Xinyu Wang、Isil Dillig 和 Rishabh Singh。[34] 2017 。使用有限树自动机合成数据补全脚本。ACM 程序设计语言会议论文集 1, 2017 62 1 62 26 10 1145 3133886 OOPSLA（），：–：。doi: . / R. Michael Wharton。[35] 1976 。解析中的歧义消解。Acta Informatica 6（1976），387–395。doi:10.1007/ BF00268139[36] Shanchan Wu、Jerry Liu 和 Jian Fan。2015。结合学习与分组的自动网页内容提取。载于 WWW。ACM，1264–1274。doi:10.1145/2736277.2741659

一种形式化方法

本节将对 Greta 中使用的核心概念给出形式化定义：上下文无关文法、树自动机、树接受，以及从上下文无关文法到树自动机的转换。

A.1 上下文无关文法

定义 *A.1*（上下文无关文法）。一个上下文无关文法（CFG）是一个元组$\mathcal{G} = (V, \Sigma, S, P)$，其中：

- V 是一个有限的非终结符集合，
- Σ 是一个有限的终结符集合，其中 $V \cap \Sigma = \emptyset$，
- $S \in V$ 是开始符号</style}，并且
- $P \subseteq V \times (V \cup \Sigma)^*$ 是一个有限的产生式集合。

我们将产生式 $(A, \beta) \in P$ 写作 $A \to \beta$，其中 $A \in V$ 是该产生式的左部，而 $\beta \in (V \cup \Sigma)^*$ 是该产生式的右部。

定义 *A.2*（解析树）。给定一个 CFG $G = (V, \Sigma, S, P)$，解析树是一个有限有序树 t，其中：

- 每个内部节点都标记为一个非终结符 $A \in V$，
- 每个叶节点都标记为一个终结符 $a \in \Sigma$，
- 对于每个标记为 A、子节点从左到右标记为 $X_1, \ldots, X_n$ 的内部节点，都存在一个产生式 $A \to X_1 \cdots X_n$，属于 P。

当解析树的根节点标记为开始符号 S 时，该解析树称为完整。我们用 $L_{\mathcal{G}}$ 表示 G 的所有完整解析树组成的集合。

定义 *A.3*（二义文法）。如果存在一个字符串 $w \in \Sigma^*$，使得 w 是 $L_{\mathcal{G}}$ 中两个或更多不同解析树的产出，则上下文无关文法 G 是二义的。

A.2 树自动机

我们使用一种经过调整的有限树自动机变体，用于表示上下文无关文法语法分析树。在此形式化中，转移同时消耗状态（对应于非终结符）和终结符号，从而保留上下文无关文法产生式的结构。

定义 *A.4*（带秩字母表）。A 个带秩字母表是一个有限符号集 F，其中每个符号都关联一个称为秩（或元数）的非负整数。也就是说，$F \triangleq \{(\text{符号}(p), \text{秩}(p)) \mid p \in P\}$。我们将 $F_{(n)}$ 写作 F 中秩为 n 的符号集合。

对于一个上下文无关文法产生式 p：$A \to X_1 \cdots X_n$，我们定义：

- 符号(p) 是与该产生式关联的唯一符号。为便于本文阐述，我们使用 $X_1 \cdots X_n$ 中的第一个终结符号；如果不存在终结符号，则使用一个特殊符号 δ，
- 秩$(p) = n$，即右部的长度。

定义 *A.5*（树自动机）。一个（有限）自底向上树自动机是一个元组$\mathcal{A} = (Q, \mathcal{F}, \Sigma, Q_f, \Delta)$，其中：

- Q 是一个有限的状态集合，
- F 是构造器标签的带秩字母表；我们假定 $(\epsilon, 1) \in F$，以便允许 ϵ-转移，
- Σ 是一个有限的终结符集合，且 $Q \cap \Sigma = \emptyset$，
- $Qf \subseteq Q$ 是一组终止（接受）状态，
- Δ 是一个有限的转移规则集合。对于有秩符号 $f \in F_{(n)}$，由 f 标记的转移是 $(Q \cup \Sigma)^n \times \{f\} \to Q$ 中的函数。

我们将转移写作：

$$q \leftarrow_f k_1, k_2, \ldots, k_n$$

其中 $f \in \mathrm{F}_{(n)}$、$q \in Q$，且每个 $k_i \in Q \cup \Sigma$。我们将 q 称为目标状态，将 f 称为构造子标签，并将 k_1、$\ldots$、k_n 称为该转移的子节点。

转移 $q \leftarrow_f k_1$、$\ldots$、k_n 表示：处理一个由构造子 f 标记的节点时，如果它的子节点（从左到右）是状态或终结符 k_1、$\ldots$、k_n，则该节点转移到状态 q。

此外，上述树自动机定义与通常树自动机定义的略微修改版本相对应；后者 Δ 由以下转移规则组成：从终结符和/或状态的组合（对应于上下文无关文法产生式右部中的终结符和/或非终结符）以及一个构造子标签到一个状态（对应于上下文无关文法产生式的左部）。[10] 因此，上下文无关文法中产生式的结构 P 在树自动机的转移规则中得以保留 Δ。这种表述较为轻量，因为与传统定义相比，它所需的转移和状态更少，因此有助于树自动机交集。

定义 *A.6*（ϵ-转移）。An ϵ-转移 是形如 $q \leftarrow_{(\epsilon,1)} q'$ 的转移，其中 q，$q' \in Q$。此转移允许将状态 q' 视为状态 q，而不消耗任何树节点。

树自动机可以包含 ϵ-转移，这类转移不消耗任何符号。本文中，当 Δ 包含 ϵ-转移时，我们会在 F 中加入一个 ε 符号（ϵ，1），以明确表示这种转移。

A.3 树接受

我们定义自底向上树自动机的接受过程。这类自动机从叶节点开始处理树，并逐步向根节点移动。

定义 *A.7*（运行）。给定一个树自动机 $\mathrm{A} = (Q, \mathrm{F}, \Sigma, Q_\mathrm{f}, \Delta)$ 和一棵树 t，A 在 t 上的运行 是一个映射 $\rho : \text{节点}(t) \to Q$，它为 t 的每个节点分配一个状态，并满足以下条件：

(1) 对于每个内部节点 v，其构造子标签为 $f \in \mathrm{F}_{(n)}$，子节点为 v_1、……、v_n，都存在一个转移 $\rho(v) \leftarrow_f k_1$、……、$k_n \in \Delta$，其中对每个 i：

- 如果 v_i 是一个内部节点，那么 $k_i = \rho(v_i)$，或者存在一系列从 $\rho(v_i)$ 到 k_i 的 ϵ 转移，
- 如果 v_i 是一个标记为终结符 a 的叶，那么 $k_i = a$。

定义 *A.8*（接受）。如果存在一个运行 ρ，使得自动机 $\mathrm{A} = (Q, \mathrm{F}, \Sigma, Q_\mathrm{f}, \Delta)$ 在树 t 上的运行满足 $\rho(r) \in Q_\mathrm{f}$，则树 t 被树自动机 A 接受，其中 r 是 t 的根节点。A 的语言记作 $L(\mathrm{A})$，是所有被 A 接受的树的集合。

直观而言，要检查一棵树是否被接受：
(1) 从叶节点开始；直接匹配终结符。(2) 对于每个内部节点，查找与构造子标签以及子节点的状态/终结符相匹配的转移规则。(3) 如果存在这样的规则，就将其目标状态赋给该节点。(4) 如果根节点可以被赋予一个终态，则该树被接受。(5) ϵ-转移允许在不消耗节点的情况下，将一个状态“提升”为另一个状态。

[10] We 将上下文无关文法转换为自底向上树自动机。对于自顶向下树自动机，规则的转移方向相反。

例如，图 15 展示了论文中讨论的 A_g 所接受的一棵示例树，该树表示一种解析表达式 整数类型 标识符 等号 整数 分号 的方式。该树被接受，是因为自动机 A_g 存在一条在这棵树上成功运行的路径，并按自底向上的顺序应用以下四条规则：(i) 标识符 $\leftarrow_{(\text{IDENT},\,1)}$ 标识符，(ii) 表达式 $\leftarrow_{(\text{INT},\,1)}$整数，(iii)声明 $\leftarrow_{(\text{TINT},\,4)}$ 整数类型 标识符 等号 表达式，以及 (iv) 语句 $\leftarrow_{(\text{SEMI},\,2)}$声明 分号。

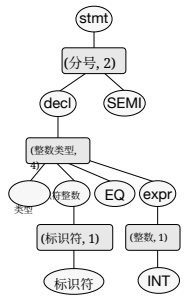


图 15. 树示例。

A.4 将上下文无关文法转换为树自动机

任何上下文无关文法都可以转换为一个恰好接受其解析树的树自动机。现在，我们将这一构造精确定义。

定义 *A.9*（上下文无关文法到树自动机的转换）。给定一个上下文无关文法 $G = (V, \Sigma, S, P)$，我们按如下方式构造一个树自动机 $A_G = (Q, F, \Sigma, Q_f, \Delta)$：

(1) 状态： $Q = V$（每个非终结符都变为一个状态）。**(2) 终态：** $Q_f = \{S\}$（开始符号是唯一的接受状态）。

(3) 带秩字母表： 对于每个产生式 $p \in P$，定义有秩符号（符号(p)，秩(p)），并令

$$\mathcal{F} = \{(Sym(p), Rank(p)) \mid p \in P\}$$

我写 为函数产生式： $F \to P$，它将每个有秩符号映射回其产生式 n.

(4) 转移： 对于每个产生式 p： $A \to X_1 \cdots X_n$，其有秩符号为 $f =$（符号(p) n），添加转移:

$$A \leftarrow_f X_1, X_2, \ldots, X_n$$

其中每个 X_i要么是一个状态（如果 $X_i \in V$），要么是一个终结符（如果 $X_i \in \Sigma$）。

定理 A.10（翻译的正确性）。对于任意上下文无关文法 $\mathcal{G}$，由定义 $\mathcal{A}_\mathcal{G}$ *A.9* 构造的树自动机满足:

$$L(\mathcal{A}_\mathcal{G}) = \mathcal{L}_\mathcal{G}$$

也就是说，$\mathcal{A}_\mathcal{G}$ 恰好接受$\mathcal{G}$的完整解析树。

证明。由构造可直接得出。 □

B 定理的证明

引理 B.1（O_p与$\mathcal{L}_r$的关系）。对于某些（s_1, o_1）、（s_2, o_2）$\in O_p$，其中 s_1 和 s_2 涉及优先级冲突，我们有:

$$o_1 \le o_2 \iff \mathcal{L}_r \textit{ includes trees with } s_1 \textit{ directly above } s_2$$

证明。

(⇒)在所构造的树自动机 L_r 中，与 s_1 对应的转移将导向状态 e_{o_1}，与 s_2 对应的转移将导向状态 e_{o_2}。由于 $o_1 \le o_2$，s_1 转移右部上的任意状态e_{o_1} 都有可能仅通过 ϵ-转移来自状态 e_{o_2}。由于初始状态也可通过 ε 转移从 e_{o_1} 到达，因此这样一棵 s_1 直接位于 s_2之上的树将属于 L_r。

(⇐) 如果 $o_1 > o_2$，那么由于 ε 转移仅从较高层级的状态引入到较低层级的状态，且层级 i 上转移的右部仅提及状态 $\ge i$（引入的环除外），因此状态 s_2 不可能在来自 s_1的树中直接出现在其下方。相邻层级上参与冲突的符号（即它们参与一个环）的情形明确不由该算法处理。□

引理 B.2（未涉及冲突的符号顺序保持）。对于某些(s_1, o_1)，$(s_2, o_2) \in O_{bp}$，其中 s_1 或 s_2 未涉及优先级冲突，且 s_1 可以出现在$\mathcal{L}_g$ 中的某棵解析树里，位于 s_2 之上，以及 $o_1 \leq o_2$。那么存在(s_1, o_1')，$(s_2, o_2') \in O_p$，使得$o_1' \leq o_2'$

证明。如果 $o_1 < o_2$，那么这两个符号都不涉及冲突，因此在 O_p 学习过程中，它们在 O_p中的次序得以保留。也就是说，存在新的次序 O_p、 o_1' 和 o_2'，使得 $o_1' < o_2'$。

如果 $o_1 = o_2$，并且由于根据假设，这些符号中至少有一个不涉及冲突，再根据 O_p的构造（该构造会将非冲突符号复制到每个新插入的次序中），仍存在次序 o_1' 和 o_2'，使得 $o_1' = o_2'$。□

定理 3.1（GenTA 的可靠性）设 A_r 为 GenTA 返回的有限树自动机，A_g 为从上下文无关文法 G 推导出的树自动机，T^- 为负树示例。此外，$L_r = L(A_r)$、$L_g = L(A_g)$，且 $L^- = \bigcup_{t \in T^-} P^-(t)$。那么，以下陈述成立：

(1) $L_r \supseteq L_g \setminus L^-$

(2) $L_r \cap L^- = \emptyset$.

证明。我们为这两个论断给出证明。

(1) 由引理 B.1 和引理 B.2结合证明。对于所有的树 $t \in \mathcal{L}_g \setminus \mathcal{L}^-$，对于每一种直接父节点—子符号（$s_1$， s_2）关系，以下情况必有其一：(1) 两个符号都涉及 T^-，但方向正确；(2) s_1 直接位于 s_2上方，但 $o_1 > o_2$（存在一个环）；或 (3) 它们满足引理 B.2的前提。

在情况 (1) 中，由于 O_p 学习算法保持冲突符号的全序，因此 s_1 和 s_2 在 O_p中的正确次序得以维持。因此，这类关系在 L_r中得以保留。这可以通过与$\Rightarrow$方向引理 B.1类似的论证看出。

对于情形 (2)，GenTA 所引入的循环产生式维持着消耗此类父子关系的转移。

最后，在至少一个符号未参与 T^-，且 $o_1 \leq o_2$ 在 O_{bp}中的情形下，根据引理 B.2，存在 o_1'， o_2' 属于 O_p，使得 $o_1' \leq o_2'$。然后，采用与（$\nrightarrow$ 方向类似的论证来处理引理 B.1，从而在$\mathcal{A}_r$ 中存在消耗树中此类节点的转移。

由于对于树 t中的所有直接父子节点，我们的树自动机都具有消耗它们的转移，因此树 t必定被该自动机接受，从而意味着 $t \in L_r$

(2) 考虑 $T^- = T_a^- \cup T_p^-$。首先，$\forall t_p \in T_p^-$，其中 s_1为顶部符号，s_2为底部符号，O_p连同 (s_1, o_1) 和 (s_2, o_2) 被加入，其中 $o_1 > o_2$。然后，根据引理 B.1，$P^-(t_p) \cap L_r = \emptyset$。接下来，$\forall t_a \in T_a^-$，其中 s是 t中的一个符号，$idx = t_{idx}$，并且 $(s, i) \in O_p, \delta_s \triangleq \delta(e_i, [\ldots; e_{i+1}; \ldots], s)$ 被加入到 Δ_r，其中 e_{i+1}出现在 idx。这意味着 $\text{ParseTrees}(t) \cap L_r = \emptyset$。因此，我们有 $L_r \cap L^- = \emptyset$。

□

定理 3.2（Greta 的正确性）自动机 A_r（GenTA 的结果）与 A_g（由上下文无关文法 G 推导出的自动机）的交集生成了一个能够识别语言 $L_g \setminus L^-$ 的树自动机。

证明。交集得到的自动机接受树语言 $L_r \cap L_g$，结合定理 3.1 可知，它所识别的语言恰好是 $L_g \setminus L^-$。□

C 算法复杂度分析

C.1 算法 3.1 LearnOaOp

算法 3.1 本质上是在不同次序下重新插入冲突符号（大小至多为 $|\mathbb{F}|$），同时更新 F 中其他符号的次序，因此在最坏情况下，时间复杂度可能为 $O(|\mathbb{F}|^2)$，空间复杂度为 $O(|\mathbb{F}|)$。

C.2 算法 3.2 GenTA

• 设 $|\mathbb{F}|$、$|O_p|$、$|O_a|$ 是带秩符号的大小，集合 O_p 和 O_a，并设 $r_{\text{最大}}$ 是 F 中的最大秩。

• **状态** Q：该算法用 m—— O_p 的最大次序——个状态填充 Q，因此 $|Q| = m$。
• **构建** $\delta_{\mathcal{F}}$ **从** P：增量生成器从预定义产生式映射中检索一个模板，该映射在算法外部构建一次，空间开销为 $O(r_{\text{最大}} \cdot |\mathbb{F}|)$。由于产生式的右部是文法的一部分，与 $\delta_{\mathcal{F}}$ 相关的唯一空间开销是产生式映射，因此 $O(|\mathbb{F}|)$。此外，在映射上查找一个符号的复杂度为 $O(1)$，遍历相应产生式的右部与产生式长度成正比，而该长度至多为带秩符号的最大元数，因此时间复杂度为 $O(r_{\text{最大}})$。

• **遍历** O_p/O_a：在对 $(s, i) \in O_p$ 的循环内部，它在 O_a 中执行常数时间的成员资格检查，并调用 δ-生成器。由此得到的时间复杂度为 $O(r_{\text{最大}} \cdot |O_p|)$，空间复杂度为 $O(|O_p|)$。

• **遍历** $\mathbb{F}_{tr}$ **和** $[1..m-1]$：这一部分对 $\mathbb{F}_{tr}$ 进行一次遍历，并对层级 1、……、$m-1$ 进行一次遍历，从而得到 $O(|\mathbb{F}_{tr}| + m)$ 的时间和空间复杂度。
• **遍历** $HighToLow(G,\ O_p)$ 中的符号：$HighToLow(G,\ O_p)$ 的上界为 $O(|\mathbb{F}|)$。并且这一部分在循环内调用 $\delta_{\mathcal{F}}$，从而得到 $O(r_{\max} \cdot |\mathbb{F}|)$ 的时间复杂度，空间开销为$O(|\mathbb{F}|)$。

• 综合以上各项，算法 3.2 的时间和空间复杂度为 $O(|\mathbb{F}|)$。

C.3 算法 3.3IntersectTA

• 设 $|\Delta_g|$、$|\Delta_r|$ 以及 $|\mathbb{F}|$ 分别为输入树自动机的转移规模、带秩符号的规模，并设 $r_{\max}$ 为 F 中的最大秩。
• **初始化** Q_f、Q、Q_{tmp}：每个输入自动机都只有一个接受状态，计算所需时间为常数。

• **遍历可达状态**：– 每个积状态最多进入 Q_{tmp} 一次，迭代次数上限为$|Q| \le |Q_g| \cdot |Q_r|$。– 然后，为每个积状态查找可达符号需要 $|\mathbb{F}|$ 时间，而遍历这些符号的时间复杂度为 $O(|\mathbb{F}|)$，因为在最坏情况下所有符号都可能可达。– 对于一个可达符号和一个可达积状态（e_g, e_r），候选转移对的数量在最坏情况下为 $|\Delta_g| \cdot |\Delta_r|$。请注意，尽管遍历可达符号的 for 循环位于可达积状态循环之内，但转移对的总数仍然不超过 $|\Delta_g| \cdot |\Delta_r|$，因此无需再乘以 $|\mathbb{F}|$。– 对于每对转移，该算法会对这些转移取笛卡尔积，其规模受最大元数限制，因此耗时为 $O(r_{\text{最大值}})$。– 在最大元数$r_{\text{最大值}}$有界的情况下（大多数上下文无关文法通常如此），遍历可达状态的时间和空间复杂度为 $O(|\Delta_g| \cdot |\Delta_r|)$。

• **查找并移除重复状态**（算法 3.4FindDupStates）：

– 在算法 3.4 中，对于每一对状态，比较其转移集合的运行时间上界为 $O(|Q|^2 \cdot |\Delta|)$，其中 Q和 Δ分别指算法 3.3 主循环之后的状态数与转移数。– 至多存在 $O(|Q|^2)$ 个状态对，存储每个状态的转移集合的空间开销为$O(|\Delta|)$，因此算法 3.4 的空间复杂度为 $O(|Q|^2 + |\Delta|)$。（回到算法 3.3：）– 那么，对所有重复状态对（Q_{dup}）执行合并/移除操作的时间开销为 $O(|Q|^2)$，空间开销为 $O(|Q|)$。– 通过遍历所有转移来重写转移的时间和空间开销均为 $O(|\Delta|)$。– 因此，这一部分——以检测开销为主——的时间复杂度和空间复杂度分别为 $O(|Q|^2 \cdot |\Delta|)$ 和 $O(|Q|^2 + |\Delta|)$。

- **ε-转移引入**：这一部分遍历有序状态，并检查每一对状态对应的转移。与重复项检测过程相比，这一部分可以忽略不计。
- 综合以上内容，算法 3.3 的时间复杂度为 $O((|Q_g|\cdot|Q_r|)^2\cdot|\Delta_g|\cdot|\Delta_r|)$，空间复杂度为 $O((|Q_g|\cdot|Q_r|)^2 + |\Delta_g|\cdot|\Delta_r|)$。

D 用户交互：Menhir 与 Greta

本节通过一个具体示例，说明 Menhir 提供的用户交互与 Greta 在解决文法歧义时采用的基于示例的交互有何不同；示例展示了手动消歧过程中呈现给用户的信息形式及其复杂程度。该示例取材于我们修复论文基准测试中的 Michelson 文法 *G7* 的实践经验。本节还指出，手动消歧的工作量不仅取决于歧义数量，也取决于理解自动机级冲突报告并预判文法修改效果所需的认知成本。

Menhir 冲突报告。 应用于文法 *G7*时，Menhir 会生成一份冲突报告，如图 16所示。我们大体上逐字收录这份报告，以展示向用户提供的细节程度。要手动解决此冲突，用户需要检查自动机状态，找出相关产生式，重建相互竞争的解析树，并推断哪种文法修改符合预期的解析偏好。这个过程既要求熟悉文法，也要求具备 LR 解析器诊断方面的经验。

Greta 用户界面。 对于相同的歧义，Greta 会向用户展示 图 17 中的提示，呈现冲突所涉及的解析结构。这里，歧义被明确表示为具体解析备选项之间的选择。然后，用户选择符合其意图的解析方式来解决冲突，而无需检查解析器状态或冲突报告。这种基于示例的交互可避免意外的过度消歧和回溯，并带来更可预测的修复行为。

** 状态 134 中存在冲突（移进/归约）。** 涉及的词元：
ELT ** 读取以下内容后，将从顶层到达此状态：

CODE 左花括号 MNEMONIC tyy SOME 字面量

** 下面出现的推导具有以下共同因子：**（问号符号 (?) 表示推导开始出现分歧的位置。）

顶层 script EOF CODE 左花括号
instlist 右花括号 singleinst
MNEMONIC tyy 字面量 (?)

** 在状态 134 中，向前查看 ELT 时，归约产生式……

字面量 -> SOME 字面量

** 由于以下子推导，因此允许：

字面量 ELT 字面量 // 向前看符号出现SOME 字面量 。

** 在状态 134 中，向前看 ELT 时，允许移入。。

图 16. Michelson 文法 G7 的 Menhir 冲突报告摘录。

选择你的偏好！（输入 0 或 1。）选项 0：
（ SOME（字面量 ELT 字面量））选项 1：
（（SOME 字面量）ELT 字面量）

图 17. Greta 为同一歧义呈现的用户界面，如图 16所示。