arXiv:2301.11220v2 [cs.PL] 6 Mar 2023

User-脚本语言的可定制转译

Bo Wang，新加坡国立大学，新加坡Aashish Kolluri，新加坡国立大学，新加坡Ivica Nikolić，新加坡国立大学，新加坡Teodora Baluta，新加坡国立大学，新加坡Prateek Saxena，新加坡国立大学，新加坡

转译器可将代码从一种编程语言转换为另一种编程语言。转译器的许多实际用途都要求用户能够引导或定制根据给定源程序生成的程序。这种可定制性对于实现生成代码的诸多特定于应用的目标十分重要，例如确保性能、可读性、易于阐释或可维护性，以及与外部环境或分析工具的兼容性等。传统转译器是由规则驱动的确定性系统，通常在设计时并未提供针对不同用户和不同程序的定制能力。近年来，基于神经网络的转译器取得了进展，为用户提供了一定程度的定制能力，例如通过交互式提示实现，但用户仍难以精准控制所生成的目标输出。传统转译和神经网络转译还都面临“最后一公里”问题：它们平均而言能够生成正确代码，即能够正确处理给定程序的大多数部分，但不一定能正确处理其中的所有部分。我们提出了一种新的转译方法，在无需用户理解给定源程序全局语义的情况下，提供对他人创建的转译规则进行细粒度定制和复用的能力。我们的方法大多是自动的且渐进式的，即根据用户的指导，逐步构建转译给定程序所需的语法结构翻译规则。用户可以依赖现有转译规则正确翻译程序的大部分内容，同时将精力集中在局部，仅处理不正确或需要定制的部分。这提高了最终结果的正确性。我们将该转译器实现为名为 DuoGlot 的工具，用于将 Python 程序翻译为 JavaScript 程序，并在广受欢迎的 GeeksForGeeks 基准测试上对其进行评估。DuoGlot 达到了 90% 的翻译准确率，因此优于所有现有转译器（包括手工编写的转译器和基于神经网络的转译器），同时生成的代码具有良好的可读性。我们还在另外两个包含更具挑战性且更长程序的基准测试上评估了 DuoGlot，并同样观察到，与其他转译器相比，其准确率有所提高。

1 引言

转译器旨在自动将以一种编程语言编写的源代码转换为另一种编程语言中与之等效的代码。转译器有许多用途，包括将代码库迁移到较新的语言，以及将以新设计的语言编写的代码编译为支持完善的语言。据报道，澳大利亚联邦银行曾花费7.5亿美元，历时5 年，替换其以 COBOL 编写的代码库 [Irrera 2017]。当语言频繁更新至较新版本，且新版本有选择地破坏向后兼容性时，这一问题会更加严重，例如 Python v2 到 v3 [vanRossum 2009] 以及 Tensorflow v1 到 v2 [Google 2021]的情况。在所有这些情形下，借助转译器都能提高生产效率并降低成本。转译器还有其他用途，例如在教育领域，学生可以通过编写自己熟悉的语言中的示例代码来学习新的编程语言 [Pasternak 等人 2017]。

已经有人提出了多种设计转译器的方法。传统方法是由人类专家手动编写转译器，将其构建为一组翻译规则 [Cordy 2006]。另一方面，神经机器翻译（NMT）可用于从大量数据中学习翻译模式或模型

作者地址：Bo Wang，新加坡国立大学，新加坡；Aashish Kolluri，新加坡国立大学，新加坡；Ivica Nikolić，新加坡国立大学，新加坡；Teodora Baluta，新加坡国立大学，新加坡；Prateek Saxena，新加坡国立大学，新加坡。

2023. XXXX-XXXX/2023/3-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

这些数据是源语言和目标语言的代码语料库——这是一种很有前景的方法，可减少编写转译器所需的人工投入 [Mariano et al. 2022；Roziere 等，2020，2021年；Xinyun et al.2018]。现有方法都至少存在两项主要缺陷之一。第一项与开发转译器所需的整体性成本有关。对于传统转译器而言，要编写足够完备的翻译规则，以便将源语言中的大量程序转译出来，需要投入大量时间和专业知识。设计这类整体性转译器并将其集成到开发流水线中的前期成本，可能会超过其带来的收益。用户往往只想翻译某些特定程序，不愿投入精力编写通用转译器。神经机器翻译转译器也会产生高昂的整体性成本。它们需要数十万行源语言和目标语言代码作为示例，需要访问 GPT-3/Codex 等大型通用语言模型 [Brown et al. 2020年；Chen et al. 2021]，，并且训练时计算开销巨大。如果训练数据不足，且计算成本高昂，神经机器翻译器的转译准确率就可能较低。第二项问题是缺乏灵活性。现有转译器假定所有用户都希望同一源程序得到相同的翻译。同一段代码有多种翻译方式；开发者在选择具体翻译方式时，往往需要满足多项软约束（如库依赖、偏好的编程范式、可读性、性能等）。现有转译器是为通用场景而非用户所需的特定场景设计的，因此其生成的翻译往往与用户意图存在差距。

我们的方法。 本文提出一种设计转译器的新方法，以解决上述缺点。我们主张，应将用户可定制性 作为首要设计目标。在我们提出的方法中，用户可以选择编写或重用足以正确转译其特定目标程序的翻译规则。该方法的优势来自两个子特性：规则以渐进式方式创建，而且大多由系统自动完成。渐进式是指用户可以选择一个起始翻译规则集（该规则集可能由他人编写），并逐步扩充，直到给定程序得到正确转译。大多由系统自动完成是指转译器会从用户作为“提示”提供的代码片段中学习转译规则，以尽量减少用户的工作量。这个过程应具备数据效率，即学习一条新规则只需极少的源-目标代码片段示例对。

我们构建了一个名为 DuoGlot 的原型系统，用于在两种流行的脚本语言之间转译程序：从 Python 转译为 JavaScript。我们选择 Python 和 JavaScript 来展示这一更广泛概念及其实际可行性，不过我们预计，支持其他无类型脚本语言同样可行。DuoGlot 接收一个 Python 程序作为输入，并输出一个 JavaScript 程序。其目标是生成正确翻译的程序，也就是说，在给定的单元测试下，生成与源程序具有相同输入输出行为的目标代码。几个设计决策使这种方法具备了实用性。首先，DuoGlot 将一个程序的抽象语法树（AST）表示转换为另一种语言中的 AST 表示，这与一些现有转译器类似。使用标准编译器将代码转换为 AST，再通过代码美化器反向转换，在实践中切实可行，因为许多流行语言都有现成的此类工具。其次，我们的转译器是一个“无状态”转换器。通过避免显式的状态转换规则，DuoGlot 减少了过拟合，并提高了从代码示例中学习到的规则的可用性；也就是说，使用 DuoGlot 创建的规则可应用于同一程序其他位置的类似语句，也可在转译类似基准测试或库时导入。最后，该转换器经过设计，允许非确定性，即在转译过程中的特定节点提供多个选项。这对于让转译适应正在转译的程序至关重要，因为系统必须动态地从多个翻译规则中进行选择。此外，所有创建的规则都可供用户检查，因此与神经机器翻译相比，这种方法更具可解释性。

我们展示DuoGlot通过在一个现有基准测试和两个新基准测试上将 Python 代码转译为 JavaScript，评估其性能。首先，我们在 TransCoder 项目维护的热门基准测试上进行评估 [MetaResearch2022；Roziere 等人 2020, 2021]。这些基准测试包含699个带有单元测试的 Python 程序（每个程序 1–40 行）。我们将DuoGlot 与3 种公开可用的手工编写、基于规则的转译器，以及 Codex（一种支持将 Python 转译为 JavaScript 的现代神经网络转译器）进行比较。DuoGlot 仅凭用户提供的代码片段对（每条规则一到两个），学习得到的142 条规则，就能正确转译90% 的基准测试程序。其他所有受评估工具正确转译的程序都至少少14% 。此外，为展示规则如何针对软约束进行定制，我们使用代码膨胀度量指标（字符数的增加）衡量转译后代码的可读性。代码膨胀越少越好 [Alkhatib1992]。DuoGlot 生成的代码具有良好的可读性，与 Codex 相当，并且 1.6× 比8× 我们研究的手工编写转译器更好。从定性角度来看，我们分析了使用DuoGlot所需的人工工作量，并给出示例和粗略分类。我们认为，大多数规则都相当简单，所需的代码片段也同样简单。

我们进一步通过两项案例研究评估了DuoGlot的优势与局限性，每项研究都侧重于在新的基准测试上运行该工具。第一项案例研究涉及翻译更真实且更复杂的程序，其中的单元测试可能不足。我们收集了用 Python 编写的 LeetCode 编程挑战题解法[1]（每个题目描述中都包含几个单元测试），并将其转译为 JavaScript。最终我们获得了 1067 个此类 Python 程序，平均长度为 15 行。由于程序复杂度提高，其他转译器均无法达到超过 50% 的准确率。DuoGlot 在此前 142 条规则的基础上额外增加 106 条规则后，达到了 75.4% 的准确率，并且每个程序需要 1.75 秒。在第二项案例研究中，我们测试了对更长代码的性能。我们从《Cracking the Coding Interview》一书中收集了基准测试[McDowell2015] ——共计 25 个 Python 程序，每个程序有 54 至 160 行代码。再次，DuoGlot 在各转译器中具有最高的准确率，但其平均翻译时间增加至每个程序 6.8 秒。它使用 197 条规则（此前的 142 条及额外的 55 条）实现了 56% 的准确率，高于其他转译器的准确率（Codex 为 12%，Transcrypt 为 44%）。这两项案例研究也揭示了 DuoGlot 的一些局限性。首先，尽管比其他转译器更正确，DuoGlot 仍未达到理想的 100% 准确率。其次，当单元测试未能提供全面覆盖时，DuoGlot 在未覆盖代码上的准确率可能难以预测。第三，DuoGlot 具有可扩展性，但准确率和效率会有所下降。尽管上述局限性大多具有普遍性，但它们表明 DuoGlot 仍有改进空间，我们将此留作一个开放问题。

2 问题设置与 DuoGlot 概述

用户使用转译器，将以源语言编写的程序翻译成以目标语言编写的程序。我们设计了一种名为DuoGlot 的新型转译器，可将 Python（源语言）程序翻译成 JavaScript（目标语言）程序。我们假设源程序和目标程序都有单元测试，可用于验证翻译的正确性。在实践中，基于测试的验证已得到应用，例如 LeetCode 等编程竞赛。如果只有源代码的单元测试，那么如前文所示 [Roziere 等人 2020]，将这些测试翻译成目标语言相对容易。当转译器无法翻译某个源程序时，通常会中止运行或输出错误的目标程序。不过，还有一种有趣的替代方案：通过请用户提供修正方案，推断出一条能够处理程序中问题部分的新规则。这正是我们在DuoGlot 中采用的方法：请用户提供两种语言的代码片段，作为修正翻译的提示。通过与用户交互来实现所需功能这一范式

[1] https://leetcode.com/problemset/all/

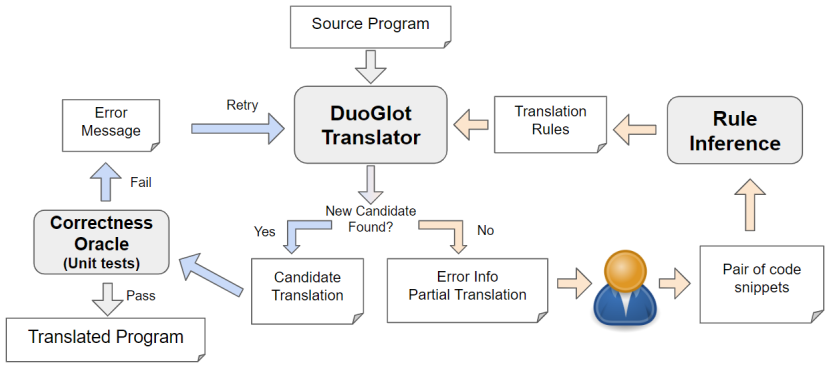

图 1. 增量翻译的工作流程。左侧蓝色箭头所示的循环是机器循环（常见情况），右侧黄色箭头所示的循环是用户循环（较少见的增量式情况）。

应用于程序合成 [Gulwani 等人 2017] 和验证 [Bader 等人 2018]。例如，广受欢迎的基于示例编程综合工具 Flashfill [Polozov 和 Gulwani 2015], 会接收用户提供的输入 — 输出示例，从而使搜索正确程序的过程变得可行。

我们的转译器 DuoGlot 的工作流程如图1所示。我们的转译器让用户掌控转译过程。DuoGlot 是一种增量式转译器。每当它无法翻译代码的某一部分时，就会尝试学习一条新规则，因此其翻译规则集会逐渐扩展。为了学习每条新规则，它会与用户交互，提示用户提供源语言和目标语言中的代码片段对，以帮助推断规则。这条规则将用户的注意力集中在代码的某一局部，因此用户无需理解源程序的全局语义。这些由用户提供的代码片段有助于规则推断，同时也让用户能够直接通过所提供的代码片段自定义翻译。请注意，我们并不期望用户是专家——他们不需要学习第三种语言，也不需要了解 DuoGlot 的内部机制。

```python
def f_gold(lines, word_list):
  trwords = [[y.upper() for y in x.split(" ")] for x in lines]
  count_dict = {}
  for line_split in trwords:
    for word in line_split:
      if word in word_list:
        if word not in count_dict:
          count_dict[word] = 0
          word_list.append(word)
        count_dict[word] += 1
  for i in range(len(word_list)):
    word = word_list[i]
    msg = "found" if word in count_dict else "not found"
    print(i, word, msg)
  count_sum = 0
  for word in count_dict: count_sum += count_dict[word]
  return count_sum // len(word_list)
```

图 2. 待翻译的 Python 程序。

```python
# Python
[x for x in []]
[y+y for y in a]
```

```javascript
// JavaScript
Array.from([]).map(x => x);
Array.from(a).map(y => y+y);
```

图 3. 用于翻译列表推导式的源-目标代码片段对。

```python
# Python
[x for x in []]
[y+y for y in a]
```

```javascript
// JavaScript
[].reduce(function (arr, x)
{arr.push(x); return arr;}, []);
a.reduce(function (arr, y)
{arr.push(y+y); return arr;}, []);
```

图 4. 列表推导式的另一种翻译对应的源-目标代码片段对。

一开始，DuoGlot 使用一组通用翻译规则启动源程序的翻译过程，以引导翻译流程。这些规则捕获了语言中最主要的语法结构，例如标识符、字面量、基本赋值语句、if 语句等。先前翻译过程中学习到的所有规则也会被添加到规则集中，并且用户可以完全控制所选择的规则。

DuoGlot n 使用提供的规则搜索候选源程序翻译。如果

搜索结束时仍未找到正确的翻译，DuoGlot 会定位翻译失败的代码位置，并提示用户协助处理翻译失败的语句或表达式。随后，用户会提供简短的代码片段，展示此类语句或表达式的正确翻译以及偏好的翻译风格。此时，DuoGlot 会通过查看所提供代码片段的语法模式，执行规则推断过程以学习一条规则。它会将这条新学习的规则添加到规则集中，以便今后在其他位置出现相同语法模式时直接应用该规则。DuoGlot 使用更新后的规则集重新搜索更多候选译文，直到翻译任务完成。

DuoGlot 大部分情况下 是自动运行的。当用户提供代码片段作为修复提示时，DuoGlot 会根据这些提示，在内部学习翻译规则，从而完成大部分繁重工作。此外，在转译过程中生成新的候选译文时，DuoGlot 会通过运行单元测试，逐步检查其正确性。如果翻译结果不正确，系统便会自动反馈错误消息及错误位置（如果有）。因此，正确性判定器不仅负责验证，还充当测试修复循环。这有助于尽早剔除错误的翻译，从而高效生成正确的翻译。据我们所知，现有的转译器设计缺少这种能够确保翻译正确的循环。

示例说明。 为了说明我们的转译器的工作流程，请看图中给出的 Python 程序2。该程序输出一组单词在一组行中出现的平均频率。假设我们从一个规模较小、无法翻译整个程序的规则集开始使用 DuoGlot 。例如，没有规则可以翻译第 2 行使用的 Python 列表推导式。我们的工具会检测到这一点，通知用户第 2 行存在问题，并请求提供代码片段。用户可以提供类似图 3 中所示的简单代码片段。请注意，列表推导式在 JavaScript 中可以有不同的翻译方式，用户也可以选择并提供图 4 中的另一种风格。（这就是转译器的定制功能——用户通过提供的代码决定偏好的风格。）随后，DuoGlot 的规则学习过程会根据这些代码片段自动推断出一条翻译规则，并将其添加到规则集中。用户也可以根据需要直接检查该规则。

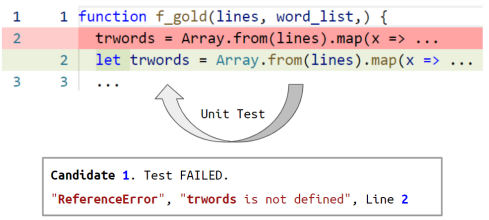

图 5. 候选译文 1 在第 2 行存在错误。DuoGlot 将自动修复此类错误。

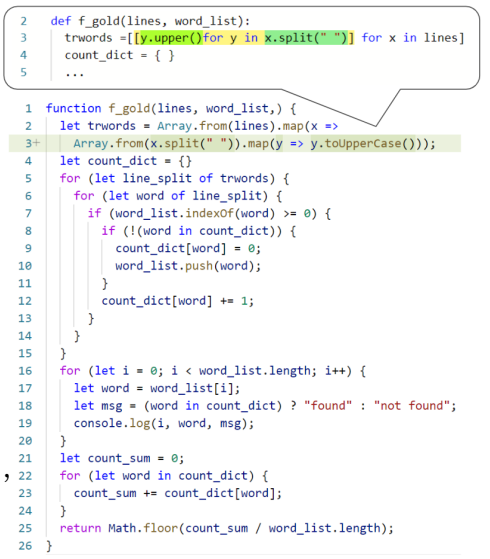

图 6. 转译后的程序（候选译文 13）。用户界面中展示了代码片段之间的映射。

接着，DuoGlot 会重试翻译，并找到列表推导式的候选译文。应用此规则后，它会生成候选译文 1 （图 5 中展示了前两行），其中它将列表推导式赋值给trwords 变量，但没有声明该变量。[illegible]对于带有 = operator 的 Python 表达式，bootstrap 得到的规则集中有两条相互竞争的规则

例如 a=[]第一条规则会在 JavaScript 翻译中保持表达式原样。第二条规则会将其翻译为 JavaScript 中的声明，let a=b。之所以有这两条规则，是因为 Python 没有将赋值和声明分别表示的机制，而 JavaScript 有。不过，应用第一条规则会得到候选译文 1，该译文未能通过测试，正确性判定器报告 trwords 在第 2 行未定义。DuoGlot 随后会查找其规则集，并发现第 2 行还有另一条可应用的规则（将 Python 赋值翻译为 JavaScript 声明）。因此，它会生成候选译文 2。

在我们的示例中，这种迭代式测试驱动流程将持续 13 轮[2]，直到找到通过所有单元测试的候选翻译。与候选方案 1不同，候选方案 13正确翻译了所有语法模式，包括将 for x in y 翻译为 for (let x of y)，for (let x in y) 或 for(let i = 0; i < ...; i++)。最终的 JavaScript 程序如图 6所示。翻译结果显示在用户界面（UI）中，用户可将鼠标悬停在一段已翻译的代码上，查看它所对应的源代码部分以及所用的翻译规则。



图 7. Codex 对图 2 中代码的翻译存在多个错误，主要是因为混淆了语义相近的运算符和表达式。Transcrypt 的翻译模拟了许多 Python 内置函数，但仍有一个错误，原因是 对象 与 数组 的类型不同，而且翻译结果冗长得多。

与 *Codex* 和 *Transcrypt* 的比较。为了凸显我们的方法的优势，我们还比较了图2 中的程序经两种最先进的转译器 Codex [Chen 等人 2021] 和 Transcrypt [3]（参见图7，其中展示了它们转译后的 JavaScript 程序）。Codex 是一种用于代码相关任务的机器学习语言模型，被 GitHub Copilot 使用；而 Transcrypt 是一种广受欢迎、由人工编写的基于规则的转译器，据称已应用于生产环境 [Cloudflare 2020]。我们观察到

[2]在这个特定示例中，恰好是 13 轮。[3]Python 3.7 到 JavaScript 编译器。
https://github.com/qquick/Transcrypt

Codex 经常混淆语义相似的表达式、运算符和变量，或者遗漏部分源代码。下面来看 Codex 翻译中用红色标出的 5 个错误。在第 2 行，Codex 忽略了嵌套列表推导式中的 `toupper()` 函数调用。在第 [illegible] 行，Codex 将表达式 `word in word_list` 原样翻译，但 `word_list` 并非字典，因此正确的翻译应为 [illegible] `= 0`。在第 19 行，翻译在语法上无效，因为这里应使用三元表达式，而不是先写一个表达式再接一条语句。在第 20 行，它将 `print` 原样翻译；而在第 26 行，[illegible] 了 `//` 和 `/`。由于没有可解释的规则或翻译说明，我们很难预测 Codex 是否会出错，以及它为何会出错。总体而言，Codex 的输出易于阅读，但它经常混淆语义相似的运算符。使用 Codex 的人或许能够修正这些问题，但 Codex 方法需要庞大的训练数据集和大量计算资源才能达到这一效果。

与 Codex 或 DuoGlot不同，Transcrypt 会导入一个自定义运行时库，以模拟许多常见的 Python 函数和运算符，如图 7右上方所示。然而，它在第 36 行仍然存在错误，因为它无法推断出 `count_dict` 是对象而非数组，因此 [illegible] `for (let x in y)` 而不是 `for (let x of y)`。这凸显了一个关键问题：试图一次性解决通用转译问题的转译器难以准确判断变量的类型（或可能的取值），而这对于在不同运行时环境中将代码转译到另一种语言至关重要。相反，我们提出的增量转译范式甚至不需要显式的类型推断。DuoGlot 的测试驱动反馈循环会引导它得出正确的翻译。

总而言之，我们的转译范式具有以下优势：

(1) **可定制性**。DuoGlot 采用用户提供的翻译规则，可针对不同的编码风格、API、语言版本等进行定制。此外，DuoGlot 在从代码片段中学习规则时，支持细粒度定制，允许用户为所有语言结构指定翻译规则。用户还可以修改翻译规则，并通过拆分或合并规则集来尝试不同的规则组合。此外，定制是可选的，用户可以选择部分或全部复用他人提供的规则。现有转译器都不具备这种可定制性，部分原因在于它们并非增量式转译器。人工编写的一体化转译器需要专业知识，而基于机器学习的转译器则可能需要重新训练。

(2) **人工工作量低**。 DuoGlot 自动 从用户提供的小型代码片段中学习规则，从而减轻手动编写规则的工作量。此外，只有在需要学习知识库中尚不存在的新规则时，DuoGlot 才会要求用户提供代码片段。它采用增量式方法，只需学习完成当前翻译任务所需的规则。因此，从设计上说，与训练通用机器学习模型等方法相比，它所需的代码片段要少得多。翻译后的人工工作量也很低，因为 DuoGlot 经过优化，能够生成正确、易读的代码，并附带说明。在我们的实验中，DuoGlot 仅使用总计 360 行用户提供的代码片段，就在主要基准测试中实现了 90% 的正确翻译，涉及的代码总量为 5297 行。

3 DUOGLOT：挑战与方法

DuoGlot 的核心组件是一个树转换器（[Comon 等人 2008], 第 6 章）。它读取源程序的输入抽象语法树（AST），并将其转译为目标抽象语法树。目标抽象语法树包含足够的信息，可通过美化打印器将其转换回语法有效的目标程序。可以将此转换器视为匹配输入抽象语法树中的每棵子树，并将其

映射到目标抽象语法树中的一棵子树。因此，翻译以一系列步骤进行，每一步中，转换器都会使用一条翻译规则来输出已转译抽象语法树。转换以自顶向下的递归方式进行，也就是说，它从输入抽象语法树的根节点开始，将翻译规则应用于该树，然后向下移动以转译其中的一些子树。这类转换器称为扩展左侧（扩展 LHS）树转换器 [Graehl 等人 2008]。翻译规则以领域专用语言（DSL）编写，我们将在第4.1节详细介绍。DuoGlot 通过应对三项关键挑战，使转译切实可行。

首先，DuoGlot需要学习通用且可复用的规则，从而避免对用户提供的代码样例过拟合。假设提供了代码样例 `a = 1; a = 2` (Python) 和`let a = 1; a = 2`，用于学习处理赋值的规则。DuoGlot不应学习这样的规则：“只有在变量已声明且赋值为 *1*之后，才将值赋给该变量”。这是过拟合的一种形式，类似的例子还有很多。为应对这一挑战，DuoGlot的扩展左侧树转换器仅使用一个状态[4]。这样的转换器不会在一系列翻译步骤中保留任何上下文或历史记录；也就是说，无论输入抽象语法树中的其他子树如何被翻译，它都会对输入抽象语法树中的某个子树应用相同的翻译。因此，DuoGlot 必须对例如`a=2`始终使用相同的翻译，而不管`a=1`之前是如何翻译的。因此，它无法制定这样的规则：“只有当变量已声明为值 1 时，才将 Python 赋值子表达式翻译为 JavaScript 赋值；否则进行其他翻译”。这从根本上解决了部分过拟合问题，即学习到的规则可能会对给定的代码片段过拟合。

第二个挑战是在转换过程中处理歧义。歧义有两个来源：1）转换器缺少足够的历史信息或上下文，无法在规则之间进行选择；2）要将给定的代码片段从 Python 翻译为 JavaScript，功能上正确的映射本身就不止一种。为了解决第一个歧义来源，转换器需要具有非确定性的，也就是说，对于同一个输入抽象语法树子树，转换器最终可以应用不止一条规则。不过，由于转换器的单状态属性不允许这样做，因此不会在转换器的状态机中手动指定使用哪条规则。相反，转换器将不得不在每一步动态尝试不同的规则，并反复尝试，直到找到可行的选择。

单靠非确定性可能导致组合爆炸，因为需要尝试的规则组合呈指数级增长——在转换器对输入抽象语法树进行自顶向下遍历的每一步中，都有多个选项。关键在于，在遍历的每一步中，众多语法候选项里只有少数在语法上正确，而最终能通过给定测试的候选项则更少。因此，DuoGlot 会在每一步使用语法检查器主动验证候选项，并在一系列步骤之后使用基于单元测试的语义正确性检查器进行验证。这种增量式验证显著减少了经过几个转导步骤后仍然有效的翻译规则候选项数量。例如，将`a[0] = 5`翻译为`let . = .`，再将`a[0]`翻译为`[`的转导步骤会被语法检查器主动拒绝，因为`let a[0] = 5`在语法上无效。同样，语义正确性检查器也能发挥作用。图8 （中）展示了将 Python 代码`if x in y`翻译为 JavaScript 的过程。在 JavaScript 中，有三种语法上有效的译法：`y.indexOf(x)>= 0`、`y.has(x)`和`x in y`。然而，假设动态类型（它必须是一个集合），其中只有一种在语义上可能正确。DuoGlot的语义检查器采用测试单元过程，剔除这类无效的翻译选项。每当生成一个语法上有效的候选译文时，都会调用该过程运行单元测试，以验证其正确性。因此，转换器会通过选择第一个选项并运行代码和单元测试，生成完整的

[4]为简洁起见，从现在起，在描述我们的树转换器时，我们省略“扩展左侧”。

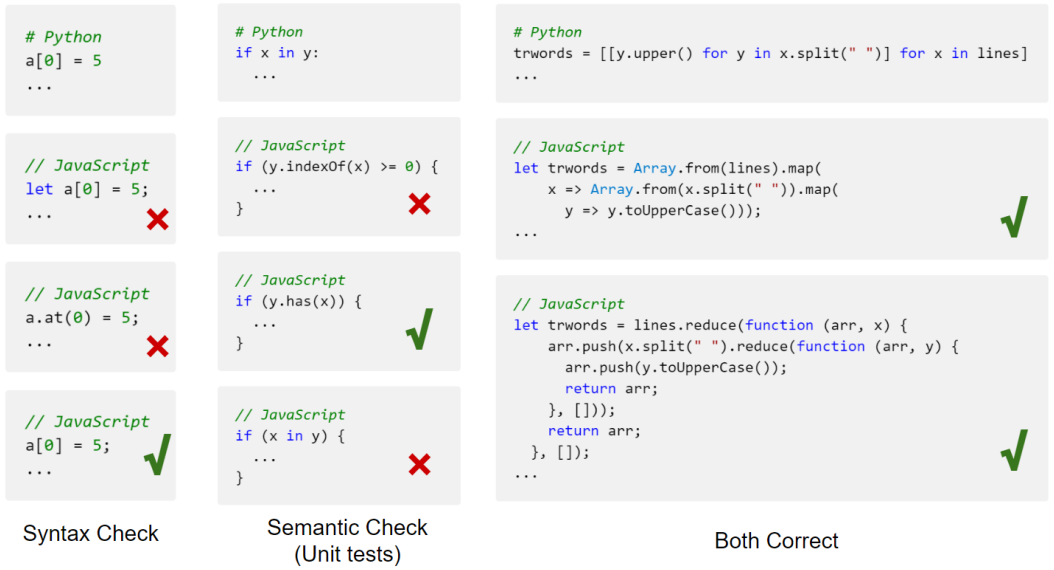

图 8。（左）语法有效与无效的翻译。DuoGlot 会检查每个转导步骤的有效性，并即时拒绝无效翻译。（中）语法有效，但不一定语义有效的翻译。当 y 的动态类型为集合时，第二个翻译才被视为正确。DuoGlot会在每次生成语法有效的候选译文时运行单元测试，以检查其语义有效性。（右）语法和语义均有效的翻译。在这种情况下，DuoGlot 只需根据预先确定的规则顺序（该顺序是任意选择的）选出一个译文，以尽量减少用户的工作。

翻译。如果 y 的运行时类型不是数组，那么该行产生的错误会反馈给 DuoGlot，使其立即转而尝试下一个选项。我们强调，这一过程会自动完成，无需用户进行任何交互；平均只需几秒到十几秒即可修复十几个错误。

第二种歧义（多个功能上正确的翻译）如图 8（右）所示。这种歧义有两种解决方式：询问用户倾向于应用哪条规则，或者按照任意预先设定的顺序选择规则。DuoGlot 两种方式都支持。为尽量减少与用户的交互，默认情况下采用后一种方式。具体来说，它会在每个转导步骤中按预先确定的顺序选择翻译规则，然后继续执行，直到将整个程序转换为语义正确的译文，或者陷入停滞并回溯。

当DuoGlot 已没有可用于翻译输入抽象语法树的有效规则可尝试时，它会与用户交互以协助推导规则。它要求用户提供源语言和目标语言的代码片段。 DuoGlot 所应对的第三项挑战是，如何完全自动地或仅需最少人工工作量地推断新规则。例如，在图3 和4 中，我们展示了用户针对图2 第2行的规则推断查询所提供的回复。关键在于理解，该查询被局限于源程序中的单一上下文，即DuoGlot 会提示用户指定如何翻译代码中发现的一个语法结构。用户可以轻松提供代码片段，因为无需理解该语法结构在程序中使用时的全局上下文。

DuoGlot通过匹配代码片段的抽象语法树中观察到的模式，自动推断翻译规则。该过程具有确定性，并且在实践中表现良好。图9 展示了一个DuoGlot的规则推断引擎如何从图3所示的代码片段中学习翻译规则。回顾一下，在我们的运行示例中，有两个 Python 列表推导式，我们希望获得其对应的 JavaScript 翻译。简而言之，DuoGlot识别出一个

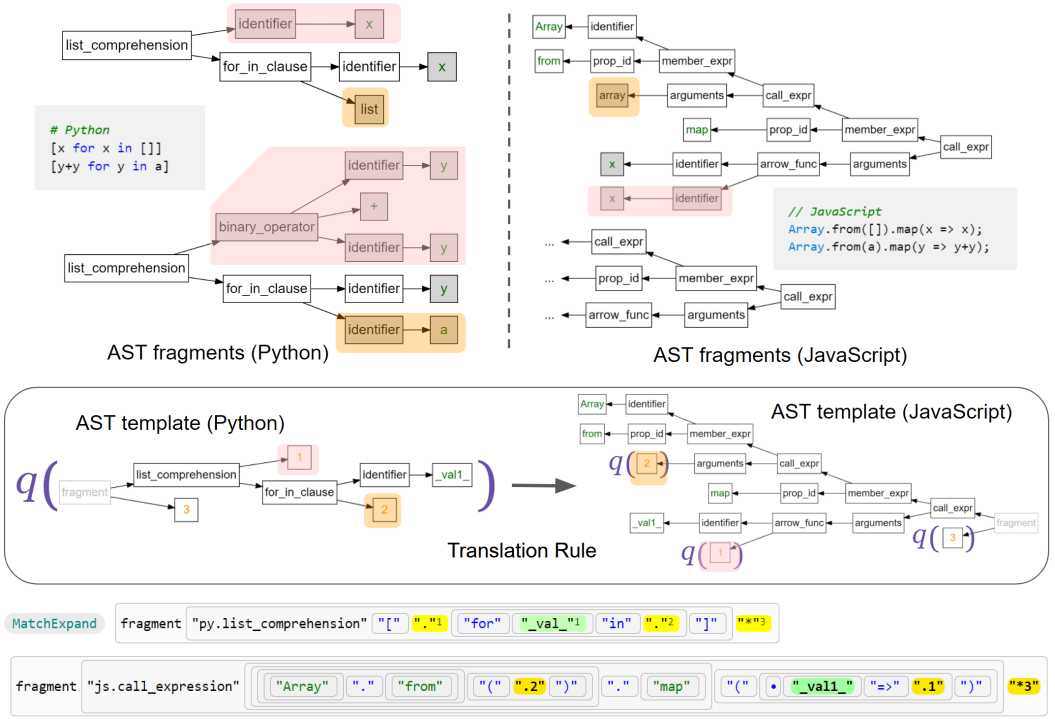

图 9. 用于翻译列表推导式（下方）的翻译规则，是从用户提供的 Python（左）和 JavaScript（右）代码片段中获得的抽象语法树片段推断而来的。灰色阴影节点是抽象语法树片段中的非公共子树，而未着色的节点则是公共子树。粉色阴影子树在各自语言中均匹配到所得抽象语法树模板中的同一空位（1）。同样，黄色阴影子树在各自的抽象语法树中匹配到同一空位（2）。q(...) 指源抽象语法树中将被转导的子树。

抽象语法树模板，用于概括 Python 代码片段中与列表推导式对应的两个抽象语法树片段。学习得到的抽象语法树模板由两个抽象语法树片段共有的节点组成（图9中未着色的方框），以及对两个抽象语法树片段之间不相同的部分进行抽象的空位组成（图9中的灰色方框）。

如图右侧所示，同样的过程会为所提供的 JavaScript 代码片段创建抽象语法树模板，见图 9。抽象语法树模板中的空位由图下方的数字表示。翻译规则就是将 Python 抽象语法树模板转换为 JavaScript 抽象语法树模板，即源抽象语法树中的固定抽象语法树节点将被翻译为目标抽象语法树中的固定节点。翻译会保留与空位匹配的 Python 片段，以便在转换器的后续步骤中进行翻译。关于如何创建抽象语法树模板以及如何处理空位的细节，我们留待第 4 节说明。图 9 还展示了以我们的领域特定语言表示的学习到的规则，用户可对其进行检查，并可选择直接编辑以自定义推断出的规则。

4 DUOGLOT：关键组件

DuoGlot的四个关键组件——转换器（第 4.1 节）、语法和语义检查器（第 4.2 节）以及规则推断过程（第 4.3 节）——通过从小型代码片段中逐步学习所需规则，实现了对程序的高效翻译。所有关键组件

均旨在适用于任意一对动态类型脚本语言，而 DuoGlot 以 Python 和 JavaScript 展示了这一点。

4.1 非确定性单状态树转换器

转换器通过应用翻译规则，递归地翻译源抽象语法树的不同部分。如图 10 所示，我们设计了一种用于编写代码翻译规则的领域专用语言（DSL）。DuoGlot 需要一组与起始符号相对应的翻译规则**规则集**。**规则集** 由一系列规则组成，每条规则都包含一个源模式和一个目标模式。源模式（**源模式**）匹配源抽象语法树片段，该片段可以是整棵树，也可以是源抽象语法树的子树。同理，目标模式（**目标模式**）匹配目标抽象语法树片段。因此，当源抽象语法树片段与当前应用的**规则**中的源模式匹配时，就可以使用目标模式将其翻译为相应的目标抽象语法树。每个模式都是由一个或多个**节点匹配器**组成的序列，用于描述相应抽象语法树片段中的“节点”。节点可以是非终结符（例如for 语句和if 语句）、终结符（任意字符串），也可以是另一个片段。因此，当前规则中的片段可以通过适用于该片段的规则递归翻译。递归片段由占位符“.” 或“*”表示。前者匹配一个表示非终结符的任意抽象语法树节点，后者则表示由非终结符和终结符组成的抽象语法树节点序列。在源模式中，这些占位符由**捕获** 规则表示；在目标模式中，则由**参照项** 规则表示。目标模式中的每个**参照项**都对应于源模式中的一个**捕获**。因此，**参照项**只是**捕获**，当前规则会原样保留它们，直到下一条规则可以继续匹配并展开（翻译）它们。

形式上，转换器可以用元组 $(Q, \Sigma, \Gamma, I, \Delta)$ 描述，其中 Q 是转换器状态的集合，Σ 和 Γ 分别是源语言和目标语言的排序字母表，$I \subset Q$ 是初始状态的集合，Δ 是翻译规则的集合。由于我们的转换器是单状态的，因此只有一个状态 $Q = q$ 和 $I = q$。该树转换器以字母表 Σ 中的抽象语法树作为输入，并使用 Δ 指定的规则生成字母表 Γ 中的抽象语法树。Δ 中的每条规则都采用以下形式：$q(F(x_1, x_2, ..., x_n)) \rightarrow G(q(x_{i_1})\ q(x_{i_2}) ..., q(x_{i_m}))$，其中 F 是由 Σ 构成的多层树，其 n 个子树表示为 $x_1, \cdots x_n$；G 是由 $\Gamma \cup (Q \times \{x_1, \ldots, x_n\})$ 构成的多层树，其中 m 个子树将在状态 q 下进行翻译，并表示为 $q(x_{i_k})$ 和 $i_k \in \{1, .., n\}$。每个 x_i 都是由 Capture 匹配的占位符，而 $q(x_{i_j})$ 则是由 Reference 匹配的占位符。

规则集 ::= 规则+规则 ::= (匹配展开 源模式 目标模式) 源模式 ::= (片段 节点匹配器+) 目标模式 ::= (片段 节点展开器+) 节点匹配器 ::= 非终结符匹配器 | 终结符匹配器 | 非终结符匹配模板 | 终结符匹配模板 | 捕获 节点展开器 ::= 非终结符展开器 | 终结符展开器 | 非终结符展开模板 | 终结符展开模板 | 参照项 非终结符匹配器 ::= (非终结符符号 节点匹配器+) 非终结符展开器 ::= (非终结符符号 节点展开器+) 终结符匹配器 ::= (str 字符串) | (nostr) | (val 字符串) 终结符展开器 ::= (str 字符串) | (nostr) | (val 字符串) 非终结符匹配模板 ::= (“_nt_” 节点匹配器+) 非终结符展开模板 ::= (“_ntIdx_”节点展开器+) 终结符匹配模板 ::= “_str_” | “_val¯” 终结符展开模板 ::= “_strIdx_” | “_valIdx¯” 捕获 ::= “.” | “*” 参照项 ::= “.Idx” | “*Idx” 非终结符号 ::= “Lang.NTName” 字符串：任意字符串 Idx：正整数 Lang：源编程语言或目标编程语言的名称 NTName：对应语言中有效的非终结符

图 10. 树转换器的翻译规则领域专用语言。

例如，考虑两种简单的算术表达式语言：源语言的字母表为 $\Sigma = \{“\times”, “+”\ a, b, c\}$，目标语言的字母表为 $\Gamma = \{\text{Add, Mult}\ A, B, C\}$。给定规则 $q(“\times”(“+”(x_1, x_2)\ x_3)) \rightarrow \text{Add}(\text{Mult}(q(x_1)\ q(x_3)), \text{Mult}(q(x_2)\ q(x_3)))$，该规则可直接匹配多个节点。转换器可以一步将源语言中的语法树“×”(“+”(a, b) c) Σ转换为 Γ中的转导语法树，其形式为 $\text{Add}(\text{Mult}(q(a)\ q(c)), \text{Mult}(q(b)\ q(c)))$。当 3

提供其他规则 $q(a) \rightarrow A, q(b) \rightarrow B$ 和 $q(c) \rightarrow C$后，转换便会完成，结果为字母表 Γ中的 Add(Mult(A, C)，Mult(B, C)。请注意，我们的领域特定语言具有非确定性，因为它支持学习其他规则（例如，针对相同或被涵盖左部的规则 $\delta_2 = q(\text{“}\times\text{”}(x_1, x_2)) \rightarrow \text{Mult}(...)$)。因此，由于处理语言结构的方式不同，以及可根据每位用户的编码风格定制规则，它支持产生歧义的翻译。

我们的转换器旨在尽量减少学习新规则的需求。首先，如前例所示，我们的转换器可以同时将规则应用于源模式中的多个节点，而不是一次只处理一个：规则会同时直接应用于“×”和“ +”这两个节点，而不是先应用于“ ×”，然后再应用于“ +”。若采用后一种方式，最终翻译将是 Mult(Add(A, B) C)，并且相比前一种方式需要额外增加一条规则。其次，要翻译源字母表 Σ中的所有符号，就必须为每个符号指定某条翻译规则。如果字母表很大，这会十分繁琐。为此，我们可以编写（或推断）一条模板规则，使其涵盖许多相似的运算符、标识符、字面量等。例如，可以为所有运算符“$<, >, \leq, \geq$”创建模板节点“`_str_`”（见图 **Tpl**”后缀的节点），用一条包含该模板节点的统括规则，取代分别对应每个运算符的规则。

4.2 语法和语义检查器

如果 R 是获得有效翻译所必需且充分的规则集合，那么目标就是搜索一条规则轨迹 τ：$[r_1, r_2, \cdots\ r_m]$，$r_i \in R$，使其能够应用于源抽象语法树。这条轨迹通常称为最左推导。如果同一抽象语法树片段可以应用多条规则（通常确实如此），搜索这样的轨迹就会更加复杂。即使抽象语法树中的每个节点只有 2 条规则，规则轨迹的总数也会迅速增长到 $2^{\#\text{nodes}}$。此外，DuoGlot 在开始时只有一小组初始规则 $R_I \subset R$，因此事先并不知道 R。所以，DuoGlot 的增量过程必须在搜索正确的最左推导 τ 的同时，搜索要应用的 R。

为了简化这一搜索，DuoGlot 使用语法检查器，仅枚举能够生成语法正确翻译的抽象语法树；并使用语义检查器进一步引导搜索，以找到语义正确的翻译。当DuoGlot 找不到更多可应用的规则选项时，它会提示用户提供代码片段，以便学习新规则（将在第4.3节中介绍）。

语法检查器。 此组件使用下推自动机跟踪每次转换的结果，并拒绝导致语法无效译文的转换。具体而言，该自动机以目标语法为参数，每个中间步骤的转换结果都会作为输入提供给自动机。自动机读取提供的输入，并检查部分翻译的子树能否用目标语法表示。如果自动机进入错误状态，则拒绝当前（最新）的转换，以及所有基于当前转换继续构建的后续转换。如果自动机接受了完全转导的抽象语法树，则可以根据自动机的日志重建出一棵包含比抽象语法树更多信息的解析树。随后，这棵解析树会经过美化排版，并作为候选译文输出。

我们使用解析表达式文法（PEG） [Ford 2004]来定义目标语言的语法。PEG 可以看作是带有产生式规则优先级的上下文无关文法。此类语法通常易于编写，并且在实践中得到广泛应用 [vanRossum etal. 2020]。结合 PEG 语法以及关于如何将抽象语法树映射到解析树的附加注释，自动机能够自动且增量地检查抽象语法树的语法有效性。形式上，DuoGlot 中使用的非确定性下推自动机 M 定义为一个元组（$Q', \Sigma', \Gamma', \Delta', q_0', Z, F$），其中 $Q' = \{\text{Start}, \text{Accept}, \text{Error}\}$ 是有限状态集，$q_0' = \text{Start}$ 是初始状态，$F = \{\text{Accept}\}$ 是接受状态，Z' 是包含目标 PEG 语法起始符号的初始栈，Σ' 是用于表示

由词元序列构成的抽象语法树，Γ' 是包含终结符和带有这些符号注释信息的非终结符的栈字母表，并且 Δ' 是一组转移指令。自动机的输入序列是部分已转译抽象语法树的先序遍历结果。表示终结符的抽象语法树叶节点可以缺失于输入序列中。只有当输入结束时栈为空，自动机才会进入接受状态。我们省略了构造此类自动机所需的字母表和转移指令的具体细节，因为这遵循标准教材中的过程 [Sipser 1996]。

语义检查器。 语义检查器通过实现测试重试过程，进一步帮助寻找正确的翻译。一种从转换器生成的语法有效的候选翻译中找出正确翻译的简单策略，是使用单元测试逐一测试所有候选译文。这种策略存在两个问题。首先，语法有效的翻译数量也可能随代码规模呈指数级增长。其次，如果枚举出的候选译文都不正确，转换器可能必须学习一条新规则才能进行正确翻译。但只有在测试完所有枚举出的候选译文之后，它才能知道这一点。为了解决这些问题，DuoGlot 使用提供的单元测试，在生成候选译文的同时对其进行测试。当某个候选译文未通过测试时，在许多情况下，例如存在未声明变量、动态类型处理不当或使用了不支持的操作，错误位置是显而易见的。在这些情况下，DuoGlot 会在该位置应用一条新规则，生成新的候选译文。实际上，DuoGlot 会依次修复候选译文中的一部分代码，然后继续测试，查找代码中的下一个错误部分。实践中我们发现，这种策略往往能快速找到正确的翻译。

也可能不存在运行时错误，但翻译后的代码的单元测试输出与源代码对应的输出不同。这类问题可能由内置函数之间细微的语义差异（例如 `int(x)` 与 `parseInt(x)` 在处理字符串时的差异）、内置运算符（例如整数溢出）、变量作用域（引用了错误的变量）、运行时行为（例如字典的迭代顺序）等因素导致。在这种情况下，如果没有行号可供调试，搜索器就会与核心协同，重试所有可能的翻译，直到达到测试重试循环预先设定的重试上限，或者返回第一个通过单元测试的翻译。作为未来工作，其他故障定位技术 [Artzi 等人 2010；Ocariza Jr 等人 2016；Shen 等人 2021] 或许能够改进我们的方法，以处理此类情况。

搜索优化。为提高搜索和检查效率，我们使用按需构建的转导树、动态规划，以及对下推自动机状态的缓存，以实现高效回溯。通过这些优化，搜索新的候选译文所需的执行时间与测试重试过程的运行时间相比微不足道。优化的具体细节见补充材料。

4.3 推断翻译规则

完成语法和DuoGlot 语义检查后，如果 DuoGlot 找不到适用的规则，就会继续推断新规则。规则推断问题包括：根据用户提供的源语言-目标语言代码片段对，生成一对语法模式以及相应的翻译规则。 DuoGlot 使用一对或最多两对代码片段。推断出的规则左侧是源模式（抽象语法树模板），右侧是目标模式（抽象语法树模板），如图9所示。

规则推断问题由三个子问题组成。第一个问题是提取出现在源代码和目标代码片段中的模式实例所对应的抽象语法树。第二个问题是提取一个抽象语法树模板，用来概括每个代码片段中的两棵抽象语法树。

算法 1规则推断算法

```
1: 过程RuleInfer(SrcASTFragments, TarASTFragments)
2:     左部 =同时遍历(SrcASTFragments)                    ▷ 源代码的抽象语法树模板
3:     src_占位符, src_tpl_占位符 = getPlaceholders(LHS)
4:     右部 =同时遍历(TarASTFragments)                    ▷ 目标代码的抽象语法树模板
5:     tar_PHs、tar_tpl_PHs =获取占位符(RHS)
6:     对于tar PH 在tar PHs执行_ _                        ▷ 计算源代码与目标代码的成对树编辑距离
7:         参考_idx = argmin_i(树编辑距离(src_PHs[i], tar_PH))
8:         tar_PH.setReferenceIdx(refer_idx)              ▷ 查找每个目标代码 PH 的最小距离
9:     对tar_tpl_PH 在tar_tpl_PHs 执行                    ▷ 匹配终结符
10:        tpl_idx = argmin_i((src_tpl_PHs[i] == tar_tpl_PH) ? 0 : 1)
11:        tar_PH.setTemplateIdx(tpl_idx == None ? “意外情况” : tpl_idx)
12:    返回createRule(LHS, RHS)                           ▷ 将 LHS、RHS 包装为 DSL 格式的规则
```

第三个问题是生成一条规则，将源代码片段中的抽象语法树模板转换为目标代码片段中与之对应的模板。

从代码片段到抽象语法树。为解决第一个问题，DuoGlot 假设用户每行只提供一个捕获目标语法模式的实例，且任何与该语法模式不匹配的额外代码都很少。此外，它还假设源代码和目标代码片段中的所有实例要么是表达式，要么是语句，因为表达式和语句之间不太可能发生翻译。DuoGlot 随后会为每个代码片段自动提取每个实例的抽象语法树，以表示一个表达式或语句。例如，在图9中，系统会从每一行的 Python 和 JavaScript 表达式中自动提取抽象语法树。如果因识别到非预期实例而导致提取失败，DuoGlot 也允许用户突出显示用于捕获语法模式的预期实例。

提取抽象语法树模板。根据从源代码和目标代码片段中得到的抽象语法树，我们的规则推断过程会推断出一条翻译规则。规则推断算法见算法1。为了从源代码片段中的抽象语法树中找到抽象语法树模板，该算法会寻找源抽象语法树之间最大的公共抽象语法树片段。具体而言，我们的算法在第2行调用同时遍历子过程[5]，该过程通过对两棵抽象语法树同时进行前序遍历来匹配它们的节点。不同的子树会被抽象化，并在提取出的抽象语法树模板中替换为占位符（如果不同的子树是终端，则替换为模板占位符）（见第3行）。源抽象语法树模板就是正在推断的规则的左部（见图9中翻译规则的左侧）。之后，会对目标抽象语法树应用相同的同时遍历过程（第4行），以找到规则右部的抽象语法树模板。

将抽象语法树模板转换为规则。 回顾一下，根据我们的规则领域特定语言（见第 4.1 节），左部中的占位符是 **捕获**，右部中的占位符是 **参照项**。右部中的 **参照项** 和模板扩展器仍是占位符，尚未与左部中的 **捕获** 和模板匹配器建立关联。对于每个模板扩展器，与模板匹配器的关联只需通过字符串比较即可计算得出（见第 9-11 行）。接下来，我们将模板抽象语法树中的 **参照项** 与 **捕获** 关联起来（见第 6-8 行）。其思路是根据所编码子树的相似度，将 **参照项** 与最相似的 **捕获**关联起来。每个 **参照项** 编码两个子树，

[5]详情请参阅补充材料。

分别来自目标抽象语法树，而每个 **捕获** 也编码两个子树，分别来自源抽象语法树。 **捕获** 与 **参照项** 之间的相似度，通过对其对应子树的成对相似度求和计算得出。我们使用树编辑距离作为相似度度量，来计算两个子树之间的相似度。树编辑距离受子树结构、节点名称以及终端中的节点值影响。树编辑距离使用 APTED 算法计算 [Pawlik 和 Augsten 2016] ，节点之间的距离则定义为节点名称和终端节点值的 Levenshtein 字符串编辑距离 [Levenshtein 等人1966]。我们运行示例所得到的规则见图 9。 **参照项**与 **捕获**之间的关联，是基于这样一个假设推断得出的：与源抽象语法树中特定子树对应的翻译结果，在所有其他已转译子树中，与已转译抽象语法树中的对应子树具有最小的树编辑距离。实践中，我们发现，对于所提供的几乎所有代码片段，这一假设都成立。若假设不成立，用户可以在规则编辑器中手动选择 **参照项**与 **捕获**的对应关系。

总而言之，规则推断过程是自动进行还是由人工引导，取决于用户的决定。此外，规则推断过程并非专门针对 Python 和 JavaScript；同样地，它也可以用于其他语言，因为它只需要能够根据这两种语言提供的代码片段构建抽象语法树。

5 实现

DuoGlot 是开源的 [Wang 2023]。它采用模块化方式实现。DuoGlot的核心模块实现了树转换器和语法检查器，并采用了补充材料中提到的优化。核心模块提供了一个接口，可与外部过程交互，以便在搜索正确翻译的过程中选择规则。DuoGlot 还包含用于自动执行测试驱动的增量规则学习的辅助组件。此外，它支持即插即用的用户界面，让用户能够控制和定制翻译的所有阶段。

5.1 DuoGlot Core

输入和输出。DuoGlot的核心专注于将源代码的抽象语法树转译为目标语言的抽象语法树。它需要两个输入：源代码的抽象语法树，以及用于启动翻译过程的一组规则（见图11）。源代码的抽象语法树由源语言的 Tree-sitter 解析器生成 [6]。DuoGlot 还需要目标语言的 PEG 语法定义，可从该目标语言的 Tree-sitter 软件包中获取。DuoGlot 会寻找一条规则轨迹，将输入抽象语法树转换为目标语言中语法有效的抽象语法树。它会输出这样的候选抽象语法树，以及用于生成该树的规则映射。

规则选择钩子。核心实现了一个非确定性树转换器，因此在同一个翻译步骤中，可能有多条规则适用。根据源程序、用户的意图以及单元测试，有些应用可能是错误的，或者并非首选。因此，该钩子允许任意外部模块或搜索启发式方法与 DuoGlot 核心交互，从而选择或优先排列在同一翻译步骤中适用的规则。用户也可以选择能够实现预期翻译的规则。

参数化。核心中的语法检查器由针对目标语言的解析表达式文法（PEG）定义进行参数化。我们实现了该检查器，使其能够接受 Tree-sitter 的文法领域专用语言。许多常见编程语言的 tree-sitter 语法定义已用于生产环境 [Brunsfeld 2018]。因此，DuoGlot的核心可以轻松扩展，借助现有的 Tree-sitter 语法定义支持其他脚本语言。

[6] Tree-sitter是一种常用于生产环境的解析器生成器。它接受 PEG 语法定义。

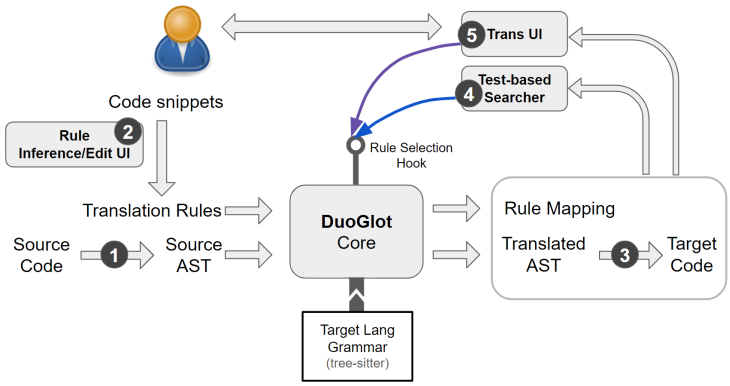

图 11. DuoGlot 的模块及输入/输出。

5.2 DuoGlot外围组件

DuoGlot的核心会输出一个语法有效的候选抽象语法树，但不包含测试修复流程，也不包含用于在抽象语法树与精确代码之间进行转换的模块。这些功能由外围模块实现（见图11）。

解析器和美化打印器。图11 中的圆圈 1 和 3 分别表示源语言的解析器和目标语言的美化打印器。对于解析器，我们使用现成的 Tree-sitter 解析器，从源代码获取抽象语法树。美化打印器用于将 Tree-sitter 格式的已转译抽象语法树转换为目标语言的代码。目前没有这类美化打印器的现有实现。因此，我们实现了一个元美化打印器，它以目标语言的 Tree-sitter PEG 为参数。此外，我们还处理了每种语言的一些边界情况，例如终结符和非上下文无关特殊情况（如 Python 的缩进）。

基于测试的搜索器。当DuoGlot的核心生成候选译文后，会将其传递给基于测试的搜索器。搜索器运行单元测试以验证其正确性。图11 中的圆圈 4 表示基于测试的搜索器，它使用单元测试和错误分析来测试候选译文。如果译文通过了所有单元测试，搜索器就会输出该译文。如果译文未通过单元测试，并且测试给出了故障行号，搜索器便会按以下方式引导搜索。它会先查看规则映射，计算推导历史中与代码故障行对应的片段。随后，搜索器会在该片段中选择其他可能的规则选择，并将这些选择发送回DuoGlot 核心。请注意，一行代码可能对应多个规则应用。因此，在最坏情况下，搜索器必须尝试该行上所有适用规则的各种可能组合。因此，搜索器会先选择每次只更改一条规则，然后再尝试更改两条规则，依此类推。核心收到更新后的选择后，会生成另一个遵循搜索器所选新规则的候选已转译抽象语法树。随后会再次测试新的候选译文。如果尝试故障行上的所有替代方案后仍无法修复该行的错误，搜索器就会陷入停滞，并提示用户提供新的代码片段。

我们对测试重试迭代次数设定了上限，称为重试上限。例如，在主要评估中，我们将重试上限设为 20。因此，在这 20 次重试中，可能出现以下情况：1）DuoGlot 生成通过所有单元测试的翻译；2）由于在某一行已耗尽所有可应用的规则而陷入停滞；3）全部 20 个候选译文都引发运行时错误；或 4）最后一个候选译文没有运行时错误，但其输出仍与源代码的输出不一致。只有当 DuoGlot 陷入停滞（第二种情况）时，系统才会提示用户提供新的代码片段，以学习一条新规则，然后重新开始测试重试过程。

Trans 用户界面（基于用户的规则选择）。图 11 中的圆圈 5 表示 DuoGlot用户界面的一个组件。该组件允许用户查看翻译过程。用户可以检查候选译文及其规则映射。在翻译过程中，用户还可以在每个翻译步骤更改规则，以获得不同的候选译文。补充材料中提供了该组件的屏幕截图。

规则推断/编辑用户界面。图 11 中的圆圈 2 表示DuoGlot用户界面中的规则推断/编辑组件。在此处，用户输入代码片段并检查自动推断出的规则。专家用户还可以根据需要手动编辑规则。补充材料中提供了该组件的屏幕截图。

6 评估

我们展示了 DuoGlot 是在 Python 和 JavaScript 等无类型脚本语言之间进行翻译的最佳实用工具之一。我们计划提交包含代码和基准测试的软件制品。我们的评估结构如下。

(1) **定量评估（第 6.2 和 6.3节）**：我们测量 DuoGlot能够正确解决的基准测试数量，以及解决这些基准测试所需的规则数量。随后，我们将DuoGlot与几种现有转译器在正确翻译及其可读性方面进行比较。(2) **定性评估（第 6.4节）**：我们根据规则复杂度对规则进行人工分类，并确定规则复杂度如何影响性能。我们还讨论了与其他最先进的转译器相比，使用DuoGlot 所需的额外人工投入。(3) **案例研究（第 6.5 和 6.6节）**：我们介绍两个案例研究，以评估在测试覆盖率不同、代码更复杂或更长的场景下，我们的工具所具备的能力与局限性。

6.1 评估设置

我们评估 在热门基准测试上，将代码从 Python 翻译为 JavaScript 的 DuoGlot。

基准测试。 我们使用 GeeksForGeeks (GFG) 基准测试 [MetaResearch 2022] Transcoder 项目中的 [Roziere 等人 2020] 的基准测试作为主要评估基准测试，并使用另外两组程序进行案例研究（参见第 6.5 节和 6.6）。GFG 基准测试包含 702 个带有单元测试的独立 Python 脚本，每个脚本都有一个待翻译的目标函数。我们删除了 3 个无法运行（存在错误）的脚本，最终得到699 个用于评估的 Python 脚本。目标函数的长度介于 1-40 行之间（平均 8.53 行），或介于 23-974 个非空格字符之间（平均 154 个字符）。

单元测试。 这些基准测试为源 Python 代码提供了单元测试（行覆盖率达到 99%），但没有为目标 JavaScript 提供单元测试。不过，Python 单元测试很容易翻译成 JavaScript：既可以像以往研究 [Roziere 等人 2020] 所做的那样，使用简单的临时单元测试转译器，也可以使用 DuoGlot 的简化版本（不使用测试修复策略）。我们采用后一种方法，并手动确认所有单元测试均已正确翻译。

DuoGlot 设置。 为了将 Python 转译为 JavaScript，我们使用上一节介绍的 DuoGlot Core 和 DuoGlot Peripherals 创建了一个流水线。我们关闭了用户界面中允许手动选择备选规则的部分，并将测试重试循环（运行单元测试）的重试上限设为 20 次迭代。我们选择 20 次，是因为我们希望 DuoGlot 能在 1–2 秒内完成源代码的翻译；如果失败，则要求提供新的代码片段。开始时，DuoGlot 使用 44 条基础规则，将最常见的抽象语法树节点（例如标识符、数值字面量、字符串字面量、赋值语句等）从 Python 直接映射到 JavaScript。我们默认包含这些基础规则（尽管最终用户也可以创建或修改它们），以避免在 DuoGlot 当前及未来的所有应用中反复重新定义这些规则。

库调用。 GFG 基准测试仅依赖 Python 内置库，不会调用第三方库。测试 Python 到 JavaScript 的翻译时，仅使用内置库是一种常见做法，因为这样可以有效比较转译器的核心能力。请注意，DuoGlot 在处理第三方库（如 NumPy 和 pandas）方面并无根本限制，不过这需要 JavaScript 中存在类似的库。

评估准确率。 为确认翻译的正确性，我们使用单元测试。我们在单元测试上比较源程序和转译程序的输出，包括其中的中间日志（打印输出）。只有当所有单元测试的输出都匹配时，我们才认为翻译是正确的。准确率即为已正确翻译的基准测试所占的百分比。

评估可读性。 我们根据转译代码大小来评估可读性 [Weinberg1971]，并假设大小越小越好，因为这能够保留高级抽象，同时排除包含许多额外不必要变量或冗余代码的翻译。相比其他可读性指标（例如 [Buse and Weimer 2008]），我们更倾向于采用这一指标，因为我们主要关注的是保持与源程序相同的抽象层级，而不是关注变量标识符命名、注释质量等人为因素。我们将可读性表示为转译代码与源代码之间非空白字符数量的膨胀比。

比较转译器。我们将DuoGlot 与其他最先进的转译器进行比较，包括基于机器学习的转译器和手工编写的转译器。在基于机器学习的转译器中，我们评估了 Codex[Chen 等人提出的 2021]（即 GitHub Copilot 使用的 API）以及 TransCoder[Roziere 等人提出的 2020]。对于手工编写的转译器，我们在 GitHub 上搜索与 Python 到 JavaScript 翻译相关的开源项目。我们选择了 Transcrypt（2.5千个星标[7]）、Javascripthon[8]（824个星标）和 py2js[9]（90 个星标），因为它们也旨在生成正确且易读的 JavaScript 代码。为避免比较不公平，我们采用以下标准，这些标准通常会高估部分工具的性能：高估：

- **TransCoder**：我们选取概率最高的 10 个翻译（即束宽为 10），并对这 10 个翻译全部运行测试。只要其中任意一个翻译通过所有单元测试，我们就将其计为成功。
- **Codex**：我们从 Codex API 获取一种可能的翻译。我们会删除翻译中第一个 JavaScript 函数之后多余的代码或文本，以清理翻译。如果翻译后的函数名有拼写错误，我们会通过字符串替换进行修正。清理并修正后的翻译若能通过所有单元测试，就计为成功。
- **Transcrypt、Javascripthon 和 py2js**：我们对每个基准测试分别运行一次这些工具，以获取翻译后的代码。翻译若能通过所有单元测试，就计为成功。

系统规格。我们使用一台配备 Intel i7-9700 八核心中央处理器和 32GB 内存的台式机。DuoGlot Core、基于测试的搜索器，以及非机器学习工具的评估脚本分别运行在一个中央处理器核心上。对于 TransCoder，我们使用一台配备 AMD Threadripper 3970X 32 核处理器、64GB 内存和 GeForce RTX 3090 图形处理器的机器。对于 Codex，我们直接使用 OpenAI 的 API，不进行任何本地计算。

6.2 DuoGlot 在 GFG 基准测试上的性能

DuoGlot 通过从用户提供的代码片段中学习并推断规则，提高了准确率。没有学习到的规则时（仅使用设置中提到的 44 条引导规则），其在 0% 的 699 项基准测试中的准确率为，但随着规则总数达到 142 条，准确率提升至 90%。我们在图 12 中绘制了规则数量与可由特定数量规则解决的基准测试百分比之间的关系。如图所示，随着0 条规则增加到80 条

[7]星标数量表示在 GitHub 上关注该项目的开发者人数。[8]Python 3 到 ES6 JavaScript 转换器。https://github.com/metapensiero/metapensiero.pj [9]Python 到 JavaScript 转换器。https://github.com/qsnake/py2js

，准确率迅速提升至约 60%，并在 98 条规则时达到 80%。此后，准确率缓慢上升，并在 142 条规则时达到最高的 90%。

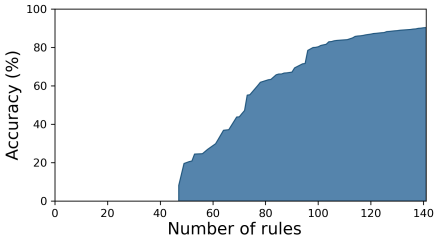

图 12. GeeksForGeeks 基准测试的转译准确率随所用规则数量的变化。前 80 条规则使准确率从 0% 提升至 60%。

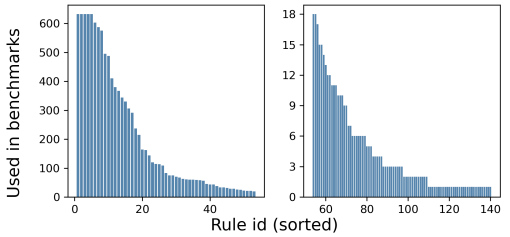

图 13. 142 条规则在转译后的基准测试中的分布情况。最常用的前 15 条规则出现在约半数基准测试中，且这些规则均为基础规则。

我们还展示了规则在转译后的基准测试中的分布情况，参见图 13。最常用的前 15 条规则出现在多达半数的基准测试中。正如预期，这些规则均来自引导集合（基础规则）。其余规则的使用情况各不相同：较常用的规则用于数百个基准测试，而较少使用的规则则只用于少数几个。

6.3 与其他转译器在 GFG 基准测试上的比较

此外，我们还比较了所有转译器的特性，并请读者参阅表1。**准确率比较**。表1展示了在所考虑的评估设置下所有转译器的准确率。DuoGlot的性能优于所有手工编写的转译器，例如 Transcrypt、Javascripthon 和 py2js。此外，即使修正翻译中的少量拼写错误，其性能仍优于 Codex。对于 TransCoder，无法进行直接比较，因此我们仅列出 TransCoder 将 Python 翻译为 C++ 或 Java 时的性能，以供参照。由于缺少数据和资源，无法针对 JavaScript 目标重新训练 TransCoder，因此我们只能推测，若使用 JavaScript 进行训练，TransCoder 在 Python 到 JavaScript 翻译上的性能将与其 Python 到 C++翻译的性能相当。

表 1. 不同转译器在准确率、可读性（膨胀比）和执行时间方面的比较。

将 Python 转换为 JavaScript	准确率	可读性	Time
DuoGlot（142 条规则）	90%	1.34x	0.46s
Codex [10]	35%	1.2x	7.33秒
Codex（修正拼写错误） [11]	74%	1.2x	7.33秒
Transcrypt	76%	10.7倍[12]	0.16s
Javascripthon	43%	2.1x	0.06s
py2js	21%	2.2x	0.03s
其他设置	**准确率**	**可读性**	**Time**
TransCoder（转换为 Java，束宽为 10）	67%	-	1.36 秒
TransCoder（转换为 C++，束宽为 10）	79%	-	2.85 秒

可读性比较。DuoGlot 的可读性（以膨胀比量化）在所有受测转译器中也名列前茅（参见表1）。在所有准确翻译的基准测试中，平均而言，DuoGlot会将原始 Python 程序的大小扩大 1.34× 倍（范围为1.06-2.22）。其表现仅次于 Codex 的 1.2× 可读性。另一方面，其余转译器的膨胀比则大得多： 10.7× 这是 Transcrypt 在不计 API 仿真库大小时的膨胀比， 2.1× 这是 JavaScripthon 的膨胀比， 2.2× 这是 py2js 的膨胀比。

时间比较。DuoGlot 获得翻译某个基准测试所需的全部规则后，翻译该基准测试的平均总耗时（包括测试）为 0.46 秒。其他手动转译器的平均耗时如下：py2js 需要 0.03 秒，Javascripthon 需要 0.06 秒，Transcrypt 需要 0.16 秒。TransCoder 将 Python 翻译为 C++/Java 时，束宽为 10，每个基准测试的平均耗时为 2.85/1.36 秒。最后，调用 OpenAI 的 Codex API 每个基准测试平均需要 7.43 秒（包括往返时间）。因此，DuoGlot 的执行时间介于手动转译器和基于机器学习的转译器之间。请注意，前者由于没有测试修复循环，翻译速度更快。不过，DuoGlot 中的这一循环实现了时间准确性权衡：虽然需要额外计算，但能显著提高翻译的正确性。因此，这种设计无疑是有益的，因为其执行时间仍然非常实用。

6.4 规则分类与人工工作量

表 2. 基础、简单、中等和复杂类别规则的代码片段示例

类别	Python 代码片段	JavaScript 代码片段
Base	`None`	`null;`
Base	`a = 5` `[c, d] = [e , f]`	`a = 5;` `[c, d] = [e , f];`
简单	`非 x` `非 y == 0`	`!(x);` `!(y === 0);`
简单	`a 在 xs 中` `2 在 [] 中`	`xs.indexOf(a) >= 0;` `[].indexOf(2) >= 0;`
中等	`list1.remove("bob")` `[1,2,3].remove(1)`	`list1.splice(list1.indexOf("bob"), 1);` `[1,2,3].splice([1,2,3].indexOf(1), 1);`

中等	`for x in range(10): pass`
	`for y in range(a):`
	`  break`
		`for (let x = 0; x < 10; x++) {}`
		`for (let y = 0; y < a; y++) {break;}`

复杂	`{x["id"]:x for x in xs}`
	`{y:y*y for y in [1,2,3]}`
		`Array.from(xs).map((x) => [x["id"], x])`
		`  .reduce((a, b) =>(a[b[0]] = b[1],a), {});`
		`Array.from([1,2,3]).map((y) => [y, y*y])`
		`  .reduce((a, b) =>(a[b[0]] = b[1], a), {});`

DuoGlot 在经过测试的基准测试中，使用 142 条规则达到了 90% 的准确率。除 44 条基础规则外，我们还可以根据复杂度将其余98 条规则分为三类：简单、中等和复杂。复杂度由**捕获**和**参照项**出现的次数决定（参见第3节）：出现次数越多，推断规则所需的代码片段（就长度而言）越复杂。简单规则有2-6 次出现，中等7-9次，而复杂规则中捕获/参照项出现次数超过**9**次。我们

[10]我们使用的模型是code-davinci-001，使用的提示词是 '##### 将此函数从 Python 翻译成 JavaScript\n### Python\n\n< Python 代码>\n \n### JavaScript\n"use strict";\n'[11]对于 Codex 翻译的许多基准测试，函数名称有误，我们通过字符串替换来修正。[12]Transcrypt 在其翻译结果中添加了许多“导入”，以便在 JavaScript 中模拟 Python 函数。

强调，规则的复杂度不会影响推断规则所需的代码片段数量，该数量始终为一到两个。在 98 条规则中，共有 82 条（84%）是简单规则，9 条（9%）是中等规则，另有 7 条（7%）是复杂规则。因此，绝大多数规则都很简单。表2中列出了各类别规则的示例，并通过用于推断规则的代码片段加以说明。

运行 DuoGlot 所需的人工工作量可分为两类。第一类占据了大部分工作量，包括提供前文提到的代码片段，用于推断增量式转译器的新规则。对于所有规则，我们总共只提供了 360 行代码片段（适用于 *Python* 和 JavaScript），其中包含 5389 个非空字符。第二类工作量是偶尔需要手动调整规则推断结果，即预期规则与推断结果不一致时进行调整。我们对 15 条规则进行了此类调整。每次调整都需要在规则编辑器中点击几下（具体次数取决于规则复杂度），以确定 **参照项**与**捕获**之间的关联，详见第 4.3节。采用更先进的实现技术即可完全消除此类工作量。最后需要强调的是，人工工作量是每条规则的一次性成本，而且不同的规则集可以合并和扩展。

6.5 案例研究：LeetCode 编程练习解答的转译

表3. DuoGlot 在案例研究中的性能

案例研究：LeetCode	准确率[13]	Time
DuoGlot（142 + 5条规则）	37.2% / -	-
DuoGlot（142 + 16条规则）	**51.7%**/ -	-
DuoGlot（142 + 41条规则）	**61.4%**/ -	-
DuoGlot（142 + 106条规则）	**75.4% / 71.6±5%**	1.75秒
Codex（修正拼写错误）	50.0% / -	5.23s
Transcrypt	48.7% / -	**0.24s**
Javascripthon	0% / -	-
py2js	0% / -	-

案例研究：CtCI	准确率	Time
DuoGlot（142 + 34规则）	28%	-
DuoGlot（142 + 55规则）	**56%**	6.78秒
Codex（修正拼写错误）	12%	37.6秒
Transcrypt	44%	**0.25秒**
Javascripthon	8%	-
py2js	0%	-

LeetCode[14] 是一个面向编程竞赛和问题的热门在线平台，拥有数百万用户。平台上有超过 2000 道题目，用户可以使用多种受支持的编程语言提交解决方案。LeetCode 会通过内部的大量单元测试检查解决方案的正确性，同时每道题只向公众开放少量单元测试。用户通常会在其他平台上分享自己的解决方案，例如 GitHub，尤其是https://github.com/doocs/leetcode拥有 15k+ 颗星标。这些用户提供的解决方案大多使用 Python 编写，而使用 JavaScript 编写的较少。因此，JavaScript 程序员可能处于不利地位，因为他们无法充分受益于大量的 Python 解决方案。

我们使用 DuoGlot 来提供这些解决方案的 JavaScript 版本。我们从上述 GitHub 页面获取了 1414 个 Python 解决方案，筛除了 347 个解决方案[15]，，最终得到 1067 个 Python 程序，每个程序都配有少量可用的单元测试。为了便于比较，我们也使用其他转译器翻译这些程序，并在表 3 中报告结果。由于 JavaScripthon 和 py2js 依赖最新的语法和 API（Python 3.9），而这两种转译器不支持这些语法和 API，因此无法翻译任何程序。Transcrypt 和 Codex 翻译了大约一半的

[13]离线准确率 / 在线准确率。离线准确率是示例测试的测试准确率。在线准确率根据手动提交至 LeetCode 的 100 个程序样本计算得出，为 98% 置信区间。 [14]https://leetcode.com/problemset/all/, https://www.linkedin.com/company/leet-code [15]它们并非自包含，也就是说，无法作为独立程序运行。

程序。使用 GFG 基准测试的 142 条规则初始化的 DuoGlot 无法翻译任何程序；不过，添加 16 条规则后，其准确率便迅速超过 Transcrypt 和 Codex。增加更多规则后，准确率进一步提高：额外添加 41 条规则时达到 61.4%，添加 106 条规则时达到 75.4%。其平均翻译时间为每个程序 1.75 秒。

DuoGlot 提供的上述 JavaScript 翻译仅基于少量可用的离线单元测试。为了确保这些翻译能够通过实际的 LeetCode 在线评测，我们从 805 个离线正确的翻译中随机抽取 100 个 JavaScript 程序，并手动提交到 LeetCode。该网站报告称，其中 95 个通过了所有内部单元测试。这意味着，我们有 98% 的信心认为，所有程序的在线准确率为 (71.6 ± 4.7) %，远高于其他任何受比较转译器的离线准确率。我们可以推测，在线准确率与离线准确率非常接近，是因为增量范式产生的搜索空间足够小，且受到源代码的高度约束，因此几个示例或许就足以筛除该空间中的所有错误翻译。

6.6 案例研究：翻译更长的代码

我们通过翻译更长的程序来测试DuoGlot 的可扩展性。为此，我们使用了 Python 程序[16]，这些程序选自《Cracking the Coding Interview》一书 [McDowell 2015]。我们收集了所有代码行数超过 50 行的程序，共 25 个，其中 19 个有 54–100 行，6 个有 100–160 行。这 25 个程序平均有 88 行（相比之下，LeetCode 程序平均有 15 行），其中一些包含多个函数和类定义。单元测试的总体行覆盖率为 88%。

与之前一样，我们从DuoGlot 开始，使用从 GFG 基准测试中获得的 142 条规则，并逐步添加新规则。最终结果及与其他转译器的比较见表 3。使用 197 条（142 + 55）规则后，DuoGlot 可以翻译 25 个程序中的 14 个。另一方面，Codex 只能翻译 3 个程序。DuoGlot 平均需要 6.78 秒来搜索能够通过测试的翻译结果，这在意料之中，因为代码越长，DuoGlot 需要检查的选项就越多，才能找到正确的翻译。对于DuoGlot 而言，翻译一个 114 行的 Python 程序耗时最长，为 25 秒。同时，Codex 平均需要 37.6 秒，因为需要解码大量词元。除了DuoGlot 和 Codex，Transcrypt 也能翻译 25 个程序中的 11 个，但它只是通过臃肿的 JavaScript 包装函数来模拟 Python 类，而不是将其转换为原生 JavaScript 类，因此牺牲了可读性和可维护性。

由于测试覆盖率为 88%，我们还手动检查了代码中未覆盖部分的翻译。结果发现，剩余的 12% 主要是用于调试输出的函数、被覆盖的函数桩，以及偶尔出现的未使用死代码。虽然这不会影响翻译后的功能，但 DuoGlot 无法自动修复这 12% 未覆盖部分中的错误。

7 讨论

让我们来　看看 DuoGlot 的应用领域、与编译器的比较以及可扩展性　.

7.1 DuoGlot 应用领域与使用模型

不同应用领域中的代码翻译在挑战和技术方面可能存在巨大差异。对于将代码从 Python 移植到 JavaScript 这类用例，翻译需要跨越不同的执行环境（例如，从 Python 虚拟机迁移到 Node.js 运行时），因此可用的 API 和库集合并不相同。这可能比在同一执行环境中进行翻译更具挑战性，例如 CoffeeScript 到 JavaScript（在 Node.js 中）、Visual Basic 到 C#（在 .NET 中）以及 Kotlin 到 Java（在 JVM 中）。在后者的情况下，当可用的 API 和

[16]程序取自该书的官方 GitHub 仓库。

运行时行为没有变化时，翻译主要聚焦于语言特性，而不是数据类型和 API，因此，使用少量规则的编译器可能非常适合此类任务。另一方面，对于跨执行环境和依赖项的代码迁移，现有工具大多是临时开发的，准确性也较低。两个应用领域中的转译器示例均列于表 4）。

表 4. 不同应用领域的代码翻译工具示例

跨环境	例如，DuoGlot、TSS[17]、Transcrypt、Codex、TransCoder
相同环境	例如，TypeScript / CoffeeScript（node.js）、.NET Reflector（.NET）、Fernflower（JVM）

DuoGlot 的使用模式包含符合第 6.5 节的应用，即从 Python 代码生成可用的 JavaScript 代码。我们强调，如果不需要定制，DuoGlot 的翻译规则只需创建一次，之后便可用于所有翻译。因此，使用该工具的门槛较低。例如，DuoGlot可用于为教程网站翻译编程挑战题。网站开发者只需创建一次翻译规则。只熟悉某些语言的最终用户可以使用DuoGlot 自动翻译他们不熟悉的其他语言编写的解答。因此，从这个例子可以看出，规则创建者与最终用户不必是同一方。

如果单元测试不足，可能需要额外的正确性判定器。在上述例子中，如果用户对翻译的正确性没有把握，可以将其提交到教程网站的在线评测系统获取反馈；如果系统报告失败，他们还可以使用额外的单元测试运行 DuoGlot。

7.2 对比DuoGlot的翻译规则与基于规则的转译器

传统转译器在所用的数据结构和规则抽象方面可能有所不同（详见补充材料中的讨论）。不过，它们通常会通过人工指定的状态机来驱动翻译过程，由状态机分析代码并检查转换规则的适用条件。这类状态机通常以抽象语法树的访问者形式实现[18] 。相比之下，DuoGlot 的翻译过程由语法和单元测试反馈驱动，而非手工编写的代码分析。其他转译器的规则组合依赖于抽象语法树访问者不透明的内部状态，而 DuoGlot 的行为透明，因为规则组合与任何人工指定的状态变化都无关。为了适应上下文并减少错误，DuoGlot 会根据测试驱动循环的反馈，自适应地撤销不正确的规则组合，而不是像编译器那样使用手工编写的确定性判定过程。

7.3 扩展到其他语言

DuoGlot 当前的设计可以适用于具有相似内存管理方式、数据类型和编程范式的语言对。例如，Python、JavaScript、Lua 和 ActionScript 在这三个方面有相似之处，尽管它们面向的是具有不同 API 的不同执行环境。若要将 DuoGlot 扩展到其他语言对，需要提供 tree-sitter 语法[19] （许多热门语言已有现成语法）以及用于该语言 tree-sitter 抽象语法树的美化打印器。

[17]Tangible Software Solutions：源代码转换器，https://www.tangiblesoftwaresolutions.com/converters.html [18] https://en.wikipedia.org/wiki/Visitor_模式 [19]如果使用外部扫描器，tree-sitter 语法可能需要进行一些调整。

我们推测，对于相似度较低的语言对，当前的领域特定语言和纯语法解决方案并不足够。例如，对于动态类型语言与静态类型语言组成的语言对，DuoGlot 可能难以生成类型一致的翻译；如果代码不适合即时编译（某个变量在运行时被赋予多种不同类型），正确的翻译也可能不在搜索空间中 [Gong 等人 2015]）。另一个棘手的情况是，将使用垃圾回收的语言（如 Go）翻译为使用手动内存管理的语言（如 C/C++）。若没有指针分析，DuoGlot可能会引入内存错误。

7.4 局限性与未来工作

让我们总结一下 DuoGlot 的局限性，并展望未来可能的工作 .

单元测试不足。 当单元测试只能提供部分代码覆盖率时，DuoGlot 使用基于测试的正确性判定器，无法获得未覆盖代码行的测试反馈来修复错误。Google 等公司会提供充足的单元测试 [Ivanković 等人 2019]（平均 ~90%），但开源项目并非总是如此。一些热门 Python 项目的覆盖率达到 90% [Zhai 等人 2019]，，而对于其他代码库，编写单元测试则更加困难（例如，用户界面代码） [Fard 和 Mesbah 2017]。要解决单元测试不足的问题，可以使用自动化测试生成器来提高覆盖率。另一种做法是使用静态分析器和形式化验证工具作为正确性判定器，以提供额外反馈给DuoGlot。然而，对于动态类型语言，这些技术可能存在局限，尤其是在缺少输入规范的情况下。在最坏的情形下，用户可能需要检查翻译中未覆盖的部分，并手动选择正确的规则。

执行环境的差异。 对于相似的数据类型，不同的执行环境可能会表现出不同的行为。例如，遍历字典时，Python 虚拟机可能会按插入顺序处理，而 JavaScript 的 Node.js 运行时的顺序则可能无法预测 [20]。因此，如果 Python 代码依赖迭代顺序，就无法直接进行翻译。Transcrypt 采用的解决方案是在 JavaScript 中模拟 Python 的数据类型，但由于生成的代码不符合惯用写法且会带来性能损失，这种方案可能不适用于代码迁移。

领域特定语言的表达能力与人工投入。 DuoGlot的规则领域特定语言直接映射到树转换器，无法匹配叶子抽象语法树节点内部的子词元。例如，它无法以通用方式将`"score: %d"` 翻译为`"score: {}"`，[illegible]叶子节[illegible]无法执行复杂的计算，例如根据函数定义重新排列函数调用中的关键字参数。一种无需修改系统的临时解决方案是针对这些边界情况使用一次性规则，但这种做法不会减少人工投入，还会产生许多无法复用的规则。表达能力更强的规则领域特定语言或许能够支持更复杂的转换，但规则推断也可能因此变得更具挑战性。我们将如何设计更好的翻译领域特定语言，并结合规则推断进一步减少总体人工投入，留作一个开放问题。

可扩展性。 如第 6.6 节中的 CtCI 案例研究所示，代码越长，所有 Python 到 JavaScript 的转译器准确率都会降低，这是累积的翻译错误和边界情况所致。具体而言，对于 DuoGlot，CtCI 中一半失败的翻译都包含无法自动修复的边界情况，而对于正确的翻译，重试次数会增加（例如，处理 114 行代码最多需要 25 秒和 52 次重试）。因此，可能需要将较长的代码拆分成子任务，以便 DuoGlot 高效处理。在某些极端情况下，如果程序运行时间很长（例如所谓的 `pytest` 单[illegible]框架），尽管重试次数不多，DuoGlot 仍需要更长的运行时间。改进分治策略、采用更智能的规则选择，或加快正确性检查，都可能提升DuoGlot的可扩展性，而这些都是极具价值的未来工作。

[20] https://stackoverflow.com/questions/5525795/does-javascript-guarantee-object-property-order

8 相关工作

Th以下总结了几种用于代码翻译的常见方法 w.

基于统计和神经机器学习的翻译。已有多种统计机器翻译技术被提出。 [Karaivanov 等人 2014], [Nguyen 等人 2013]和 [Nguyen 等人2015] 采用基于短语的统计翻译。 [Graehl 等人 2008] 提出了一种树转换器训练算法。 [Nguyen 等人 2016] 使用统计方法来寻找 API 之间的映射。近年来，基于神经网络的翻译器越来越受欢迎。 [Xinyun 等人 2018] 使用树到树神经网络将 JavaScript 转译为 CoffeeScript。 [Roziere 等人 2020] 提出了 TransCoder，它通过无监督学习来学习翻译代码。除了专用翻译器外，大语言模型（如 Codex [Chen 等人 2021]）也经过训练，可执行许多与代码相关的任务，包括翻译。然而，所有统计方法都需要大量数据进行训练。我们的方法将翻译的语法方面与语义方面分离开来。这使我们能够从少量简短的代码片段中学习定制的语法模式，并即时搜索语义正确的翻译。

项重写系统。 通常，源代码翻译可以通过手工编写基于规则的项重写系统和模式匹配来实现。StringTemplate[21] 是一个流行的、用于基于规则的代码生成和翻译的框架。TXL [Cordy 2006] 是一种通用的源代码转换语言，基于联合文法技术和手工编写的转换规则。SrcML [Collard 等人 2013] 是另一个用于在同一种语言内部进行轻量级源代码转换的系统。一些计算机代数系统（如 Mathematica [Wellin 等人2005]）具有表达能力很强的模式语言，可在不同字母表之间转换数学表达式，或将其导出为 LaTeX。这些转换工具都需要专家手动编写规则；而在语法和语义各异的语言之间进行翻译时，通常很难保证语法或语义上的正确性。

程序合成。 另一种方法是使用程序合成来解决代码翻译和转换问题。 [Kamil 等人 2016] 提出了经验证的提升方法，基于 SAT/SMT 求解，从一段低级顺序代码中合成语义等价的并行代码。[Mariano 等人 2022] 提出了一种具有类似功能的神经网络引导的程序合成器，使用同源文法网络。然而，由于典型程序合成器存在可扩展性问题，这种方法只能转换较短的程序。 [Rolim 等人 2017],[Miltner 等人 2019] 合成用于对同一语言中的若干语句进行自动代码重构的短代码转换程序。我们的工作与此正交，因为我们要在不同语言之间翻译完整程序。

9 结论

我们提出了一种新的转译设计，旨在让构建和使用此类转译器的过程更加大众化。借助我们的设计，最终用户只需付出少量人工工作量，就能构建适合自身代码风格、不同 API 和偏好语言特性的自定义转译器。其核心设计原则是：从用户提供的代码片段中，逐步学习仅为翻译当前程序所需的规则。通过这些代码片段，用户可以完全自定义翻译规则，并根据自身需求调整这些规则。人工工作量也仅限于为每条规则指定少量简短的代码片段，且每条规则只需指定一次，之后便可重复使用。在内部，该设计将翻译问题形式化为在程序空间中的搜索

[21] https://www.stringtemplate.org/index.html

该程序空间由非确定性树转换器构成，通过转换规则进行自定义，并受语法和单元测试约束。

我们还提供了名为 DuoGlot 的实用实现，它可以将 Python 转译为 JavaScript。DuoGlot 从 360 行代码片段中学习了 142 条规则，在 GeeksForGeeks 基准测试中达到了 90% 的准确率；该基准测试总计包含 5297 行代码（ 14.7× 的代码片段）。相比之下，其他转译器的准确率均未超过 76%。同样，其他转译器都不支持灵活的定制，也无法让非专家轻松升级翻译规则。DuoGlot 可以相当轻松地调整，以便在其他无类型脚本语言之间进行转译。

引用

Ghazi Alkhatib。1992年。应用软件的维护问题：一项实证分析。《软件维护：研究与实践》4, 2（1992年），83–104。Shay Artzi、Julian Dolby、Frank Tip 和 Marco Pistoia。2010年。动态 Web 应用的实用故障定位。载于《第32届 ACM/IEEE 国际软件工程会议 2010》，第1卷。IEEE，265–274。Johannes Bader、Jonathan Aldrich 和 Éric Tanter。2018年。渐进式程序验证。载于《国际验证、模型检测与抽象解释会议》。Springer，25–46。Tom B. Brown、Benjamin Mann、Nick Ryder、Melanie Subbiah、Jared Kaplan、Prafulla Dhariwal、Arvind Neelakantan、Pranav Shyam、Girish Sastry、Amanda Askell、Sandhini Agarwal、Ariel Herbert-Voss、Gretchen Krueger、Tom Henighan、Rewon Child、Aditya Ramesh、Daniel M. Ziegler、Jeffrey Wu、Clemens Winter、Christopher Hesse、Mark Chen、Eric Sigler、Mateusz Litwin、Scott Gray、Benjamin Chess、Jack Clark、Christopher Berner、Sam McCandlish、Alec Radford、Ilya Sutskever 和 Dario Amodei。2020年。语言模型是少样本学习者。（2020年）。arXiv:2005.14165 [cs.CL] Max Brunsfeld。2018年。Atom 对你的代码有了前所未有的理解。https://github.blog/2018-10-31-atoms-new- parsing-system/ Raymond PL Buse 和 Westley R Weimer。2008年。一种软件可读性度量。载于《2008年国际软件测试与分析研讨会论文集》。121–130。Mark Chen、Jerry Tworek、Heewoo Jun、Qiming Yuan、Henrique Ponde de Oliveira Pinto、Jared Kaplan、Harri Edwards、Yuri Burda、Nicholas Joseph、Greg Brockman 等。2021年。评估基于代码训练的大语言模型。arXiv 预印本 arXiv:2107.03374（2021年）。Cloudflare 博客。2020年。Cloudflare Workers 宣布广泛支持多种语言。https://blog.cloudflare.com/cloudflare- workers-announces-broad-language-support/ Michael L Collard、Michael John Decker 和 Jonathan I Maletic。2013年。srcml：用于探索、分析和操作源代码的基础设施：工具演示。载于《2013年 IEEE 国际软件维护会议》。IEEE，516–519。Hubert Comon、Max Dauchet、Rémi Gilleron、Florent Jacquemard、Denis Lugiez、Christof Löding、Sophie Tison 和 Marc Tommasi。2008年。树自动机技术与应用。James R Cordy。2006年。TXL 源代码转换语言。《计算机科学与编程》61, 3（2006年），190–210。Amin Milani Fard 和 Ali Mesbah。2017年。JavaScript：尚未（完全）涵盖的部分。载于《2017年 IEEE 国际软件测试、验证与确认会议》（ICST）。IEEE，230–240。Bryan Ford。2004年。解析表达式文法：基于识别的句法基础。载于《第31届 ACM SIGPLAN-SIGACT 编程语言原理研讨会论文集》。111–122。Liang Gong、Michael Pradel 和 Koushik Sen。2015年。JITProf：定位不利于 JIT 的 JavaScript 代码。载于 ESEC/FSE（ESEC/FSE 2015）。美国纽约州纽约市计算机协会，357–368。https://doi.org/10.1145/2786805. 2786831 Google。2021年。TensorFlow 1.x 与 TensorFlow 2——行为和 API。https://www.tensorflow.org/guide/migrate/tf1_vs_tf2 Jonathan Graehl、Kevin Knight 和 Jonathan May。2008年。训练树转换器。《计算语言学》34, 3（2008年），391–427。Sumit Gulwani、Oleksandr Polozov、Rishabh Singh 等。2017年。程序合成。《编程语言基础与趋势》4, 1-2（2017年），1–119。Anna Irrera。2017年。银行忙于修复旧系统，IT“牛仔”则逐渐退出舞台。https://www.reuters.com/article/us-usa-banks-cobol-idUSKBN17C0D8 Marko Ivanković、Goran Petrović、René Just 和 Gordon Fraser。2019年。Google 的代码覆盖率。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》。955–963。Shoaib Kamil、Alvin Cheung、Shachar Itzhaky 和 Armando Solar-Lezama。2016年。模板计算的经验证的提升。《ACM SIGPLAN 通知》51, 6（2016年），711–726。

用户可定制的脚本语言转译

Svetoslav Karaivanov、Veselin Raychev 和 Martin Vechev。2014年。基于短语的编程语言统计翻译。载于《2014年 ACM 编程与软件新思想、新范式与新思考国际研讨会论文集》。173–184。Vladimir I Levenshtein 等。1966年。能够纠正删除、插入和反转的二进制码。载于《苏联物理学文献》，第10卷。苏联，707–710。Benjamin Mariano、Yanju Chen、Yu Feng、Greg Durrett 和 Isil Dillig。2022年。使用神经网络引导的程序合成将命令式代码自动转译为函数式代码（扩展版）。arXiv 预印本 arXiv:2203.09452（2022年）。Gayle Laakmann McDowell。2015年。《Cracking the Coding Interview：189道编程问题及解答》。CareerCup, LLC。MetaResearch。2022年。GeeksForGeeks 基准测试。
https://github.com/facebookresearch/CodeGen/tree/
659963b195f5edaa42502d97605386b7fecbaa62/data/transcoder_评估_gfg Anders Miltner、Sumit Gulwani、Vu Le、Alan Leung、Arjun Radhakrishna、Gustavo Soares、Ashish Tiwari 和 Abhishek Udupa。2019年。即时合成编辑建议。《ACM 编程语言论文集》3, OOPSLA（2019年），1–29。Anh Tuan Nguyen、Tung Thanh Nguyen 和 Tien N Nguyen。2013年。用于语言迁移的词汇统计机器翻译。载于《2013年第9届软件工程基础联合会议论文集》。651–654。Anh Tuan Nguyen、Tung Thanh Nguyen 和 Tien N Nguyen。2015年。用于源代码多阶段统计迁移的分治方法（T）。载于《2015年第30届 IEEE/ACM 自动化软件工程国际会议》（ASE）。IEEE，585–596。Trong Duc Nguyen、Anh Tuan Nguyen 和 Tien N Nguyen。2016年。使用向量表示映射用于代码迁移的 API 元素。载于《2016年第38届 IEEE/ACM 国际软件工程会议伴随会议》（ICSE-C）。IEEE，756–758。Frolin S Ocariza Jr、Guanpeng Li、Karthik Pattabiraman 和 Ali Mesbah。2016年。客户端 JavaScript 的自动故障定位。《软件测试、验证与可靠性》26, 1（2016年），69–88。Erik Pasternak、Rachel Fenichel 和 Andrew N. Marshall。2017年。使用 Blockly 创建积木式语言的技巧。载于《2017年 IEEE Blocks and Beyond 研讨会》（B&B）。21–24。
https://doi.org/10.1109/BLOCKS.2017.8120404 Mateusz Pawlik 和 Nikolaus Augsten。2016年。树编辑距离：稳健且节省内存。《信息系统》56（2016年），157–173。Oleksandr Polozov 和 Sumit Gulwani。2015年。FlashMeta：归纳式程序合成框架。载于《面向对象编程、系统、语言与应用》（OOPSLA），第50卷。107–126。https://doi.org/10.1145/2858965.2814310 Reudismam Rolim、Gustavo Soares、Loris D'Antoni、Oleksandr Polozov、Sumit Gulwani、Rohit Gheyi、Ryo Suzuki 和 Björn Hartmann。2017年。从示例中学习句法程序转换。载于《2017年第39届 IEEE/ACM 国际软件工程会议》（ICSE）。IEEE，404–415。Baptiste Roziere、Marie-Anne Lachaux、Lowik Chanussot 和 Guillaume Lample。2020年。编程语言的无监督翻译。《神经信息处理系统进展》33（2020年），20601–20611。Baptiste Roziere、Jie M Zhang、Francois Charton、Mark Harman、Gabriel Synnaeve 和 Guillaume Lample。2021年。利用自动化单元测试进行无监督代码翻译。arXiv 预印本 arXiv:2110.06773（2021年）。Shiqi Shen、Aashish Kolluri、Zhen Dong、Prateek Saxena 和 Abhik Roychoudhury。2021年。从单个漏洞利用中统计定位漏洞。载于《2021年 ACM 亚洲计算机与通信安全会议论文集》。537–549。Michael Sipser。1996年。《计算理论导论》。《ACM SIGACT 新闻》27, 1（1996年），27–29。Guido van Rossum。2009年。Python 3.0 有哪些新特性。
https://docs.python.org/release/3.0.1/whatsnew/3.0.html Guido van Rossum、Pablo Galindo 和 Lysandros Nikolaou。2020年。PEP 617——CPython 的新 PEG 解析器 | peps.python.org。
https://peps.python.org/pep-0617/ Bo Wang。2023年。. https://github.com/HALOCORE/DuoGlot Gerald M Weinberg。1971年。《计算机编程心理学》。第29卷。Van Nostrand Reinhold，纽约。Paul R Wellin、Richard J Gaylord 和 Samuel N Kamin。2005年。《Mathematica 编程导论》。剑桥大学出版社。Chen Xinyun、Liu Chang、Song Dawn 等。2018年。用于程序翻译的树到树神经网络。NeurIPS（2018年）。Hongyu Zhai、Casey Casalnuovo 和 Prem Devanbu。2019年。Python 程序中的测试覆盖率。载于《2019年 IEEE/ACM 第16届国际软件仓库挖掘会议》（MSR）。IEEE，116–120。