arXiv:2303.08574v1 [cs.LG] 15 Mar 2023

WikiCoder：学习编写知识驱动代码

Théo Matricon[1][0000−0002−5043−3221]、
Nathanaël Fijalkow[1, 2][0000−0002−6576−4680]和
Gaëtan Margueritte[1][0009−0006−1742−4688]

[1] 法国国家科学研究中心、波尔多计算机科学研究实验室、波尔多大学，法国
{theomatricon,gamargueritte}@gmail.com[2] 华沙大学数学、计算机与力学学院、华沙大学，波兰 nathanael.fijalkow@gmail.com

摘要。 我们研究如何根据少量输入输出示例自动生成计算机程序。本文的出发点是这样一个观察：在许多应用中，解决方案程序必须使用示例中未包含的外部知识。我们称这类程序为知识驱动程序，因为它们可以引用从维基百科等知识图谱中收集的信息。本文迈出了知识驱动的程序合成的第一步。我们介绍了 WikiCoder，这是一个基于最先进的机器学习程序合成器并集成知识图谱的系统。我们对其进行评估，展示它在不同领域的广泛适用性，并讨论其局限性。借助知识图谱，WikiCoder 能够解决此前任何程序合成器都无法解决的任务，同时融合该领域的最新进展，实现大规模运行。

关键词： 程序合成 · 知识图谱 · 代码生成。

1 引言

任务自动化在我们所生活的数字世界中变得日益重要，然而，编写软件仍只有少数人能够做到。程序合成旨在通过开发相关方法，使编码更加可靠且易于普及，以自动生成代码 [10]。例如，FlashFill Microsoft Excel 中的系统 [9] 让非技术用户能够通过提供输入输出示例来合成电子表格程序，而 TF-coder [20] 则帮助开发者编写用于操作张量的 TensorFlow 代码。

过去五年中，运用机器学习方法推动程序合成，已经取得了令人瞩目的成果。近期的研究探索了如何设计神经网络来引导程序搜索 [1,6,17,24,19,16,27,5,7,8]，方法是训练网络，使其能够对源代码进行语言模型建模。该研究方向最为显著的成功案例，是为 GitHub 的 Copilot 提供支持的 OpenAI Codex 系统 [4]，它基于超大型语言模型。然而，

由于它只在纯粹的语法层面上运作，程序合成在许多应用中都难以奏效。

让我们以以下任务为例：

```
f("17", "美国") = "17 美元"f("42", "法
国") ="42 欧元"
```

该任务采用示例编程设定：目标是构造一个函数 f，将输入映射到相应的输出。解决此任务需要理解第二个输入是国家，并将其映射到对应的货币。示例中并未提供这条信息，因此，若不依赖外部信息，任何程序合成工具都无法解决该任务。换句话说，解决方案程序必须是知识驱动的！下面给出一个类似 Python 语法的知识驱动程序，可解决上述任务：

```
def f(x,y):
    返回 x  +" "  + CurrencyOf(y)
```

它使用了一个从外部知识源获取的函数 CurrencyOf。

知识驱动的程序合成通过面向知识驱动程序扩展了经典程序合成。 [22] 最近提出了将程序合成中的语法操作与语义信息相结合这一挑战。他们讨论了多种应用：字符串操作、代码重构和字符串分析，并构建了一种基于超大型语言模型的算法（参见相关工作一节）。

我们的方法有所不同：我们开发的方法依赖于知识图谱。知识图谱是规模极大的结构化数据库，利用本体组织知识，以便进行高效、精准的浏览和推理。目前，公开可用的知识图谱日益增多，例如 Wiki-data.org，其中包含并整理了维基百科数据；还有 Yago [21,12], ，它基于维基数据和 schema.org。关于近期出版的教科书，请参阅 [14]；关于知识图谱及其应用的优秀综述，请参阅 [13]；图 1 展示了相关示例。

程序合成和知识图谱近期取得的成功表明，现在正是将二者结合起来、以实现知识驱动的程序合成这一研究目标的良机。

我们的贡献：

– 我们提出知识驱动的程序合成，它扩展了程序合成，使程序能够引用从知识图谱中收集的外部信息。– 我们确定了知识驱动的程序合成的若干里程碑，并提出了一个由人工生成且公开可用的任务数据集，包含 46 项任务，用于评估这一研究目标的未来进展。– 我们构建了一种算法，将最先进的机器学习程序合成器与知识图谱查询相结合，并可部署于任何知识图谱。

– 我们实现了通用知识程序合成工具 Wiki-Coder，并在多个领域对其进行了评估。WikiCoder 能够解决以往任何程序合成工具都无法解决的任务，同时通过整合程序合成领域的最先进技术，实现大规模运行。

2 知识驱动的示例编程

2.1 目标

示例编程，又称归纳综合，是指这样一个问题：给定少量示例，构造一个满足这些示例的程序。程序合成中的这一特定设置允许用户提供非常不完整的规范，在自动化终端用户任务方面极为实用且成效显著；其中的典型例子是 FlashFill，它可在 Excel 中执行字符串转换 [9]。

在考虑知识驱动的示例编程时，我们改变的不是问题，而只是解决方案：我们不再使用执行句法操作的传统程序，而是纳入知识驱动程序。为说明两者的区别，我们来看下面两个任务。

```
f("巴黎") ="我爱 P"f("柏林")
 =" 我爱 B"

g("唐纳德·克努特") ="DK 是美国人"g("艾达·
洛夫莱斯") ="AL 是英国人"
```

第一个任务是经典的程序合成任务，因为它纯粹是句法层面的：只需用一个两行程序将“I love ”与输入的首字母连接起来即可解决。另一方面，第二个任务需要了解输入人物的信息，此处是他们的国籍：要解决这个任务，就需要一个知识驱动程序。由于几乎所有程序综合工具都只对示例进行句法操作（深入讨论请参阅相关工作一节），因此它们无法解决第二个任务。

知识驱动的示例编程远不止于查询应答：目标并非回答某个特定查询，而是生成一个能够针对任意输入回答该查询的程序。这在计算上和概念上都是一个困难得多的问题。

为便于说明，我们先介绍一些有关知识图谱的术语，并请参阅图 1 了解示意图。节点称为实体，边则以关系标记。实体被划分为类别：“埃马纽埃尔·马克龙”属于人物类别，“法国”属于国家类别。类别和关系受本体约束，本体规定了实体之间可以成立哪些关系。事实上，查询知识图谱的标准方式是使用 SPARQL 查询；这是一种功能强大且用途广泛的查询语言。

2.2 里程碑

我们确定了使用语义信息的三种独立方式。它们对应于解决任务的不同阶段：

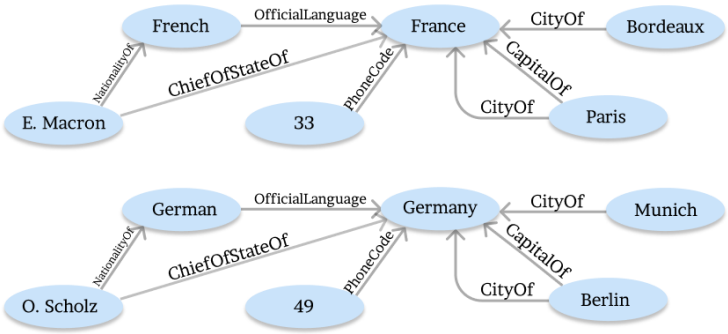

图 1. 知识图谱部分内容的示意图。

– 预处理：第一步是从示例中提取实体；– 关联：第二步是在知识图谱中关联实体；– 后处理：第三步是处理在知识图谱中找到的信息。

需要多少预处理才能提取实体？让我们考虑两个任务。

`f`("艾克斯、巴黎、波尔多") = "巴黎"`f`("汉堡、柏林、慕尼黑") = "柏林"

`g`("奥巴马总统") = "奥巴马"`g`("首相德·费弗尔·约翰逊") = "德·费弗尔·约翰逊"

在函数f中，目标是提取逗号分隔的第二个词：这纯粹是一种句法操作。在函数g中，我们需要从输入中去掉职务头衔：这需要语义知识，例如，仅使用第二个词或最后一个词都不够。

实体之间的关系有多复杂？我们再考察两项任务。

`f`("巴黎") ="法国很美丽"`f`("柏林") ="德国很美丽"

`g`("巴黎") ="电话国家代码：33"`g`("柏林") = "电话国家代码：49"

函数 f 关联两个实体：一个城市和一个国家。可以预期，知识图谱中包含关系 CapitalOf，该关系会在“巴黎”和“法国”之间以及“柏林”和“德国”之间生成一条带标签的边（如图 1 所示）。请注意，也可能是关系城市所属。更复杂的是，函数 g 需要跨关系信息：事实上，要将一个城市与其国家电话区号关联起来，可能需要组合两个关系：CapitalOf 和电话区号。换言之，

这些实体在知识图谱中由一条长度为 2 的路径关联，我们将其记为 CapitalOf-PhoneCode。更一般地说，关联这些实体的路径长度是衡量任务复杂度的一项指标。

需要对外部知识进行多少后处理？让我们再看看两个任务。

```
f("巴黎") ="国家'的首字母：F"f("柏林") ="国
家'的首字母：G"

g("奥巴马总统") = "B@r@ck"g("约翰逊首相")
 = "B0r1s"
```

对于函数 f，难点在于找出要搜索的外部知识，即“巴黎”与“法国”、“柏林”与“德国”之间的关系，然后只返回结果的首字母。类似地，对于 g，在进行 leet 转换之前，我们需要检索作为外部知识的名字。在这两种情况下，难点都在于示例中没有包含这些中间知识。

这三个步骤可能都是必需的；最复杂的任务会同时涉及精细的预处理以提取实体、实体之间复杂的关系，以及对外部知识进行大量后处理。

2.3 动机示例

我们通过三个应用领域展示知识驱动的程序合成所带来的颠覆性力量。

常识 第一个示例贯穿全文，用于说明数据集和实验中的做法：利用知识图谱获取关于世界的一般事实，例如地理、电影和人物方面的事实。维基数据和 Yago 是适用于这一场景的自然选择。查询回答大量应用于这一领域；将其与程序合成结合，便能提升到新的层次，因为程序可以泛化到任意输入。

语法练习 第二个示例受 [22], 启发，涉及语言学习：任务是语法练习，目标是写出语法正确的句子。此处的知识图谱包含语法形式及其关联，例如动词及其不同的变位形式、代词、形容词等。生成用于解答练习的程序带来了多种可能性，包括生成新练习以及自动解答练习。

高级数据库查询 在第三个示例中，知识驱动的程序合成成为了强大的查询引擎。这一场景已针对 SQL 查询 [23,26],进行了深入研究，但研究仅停留在语法层面。依靠数据的语义属性能够带来诸多可能性，下面

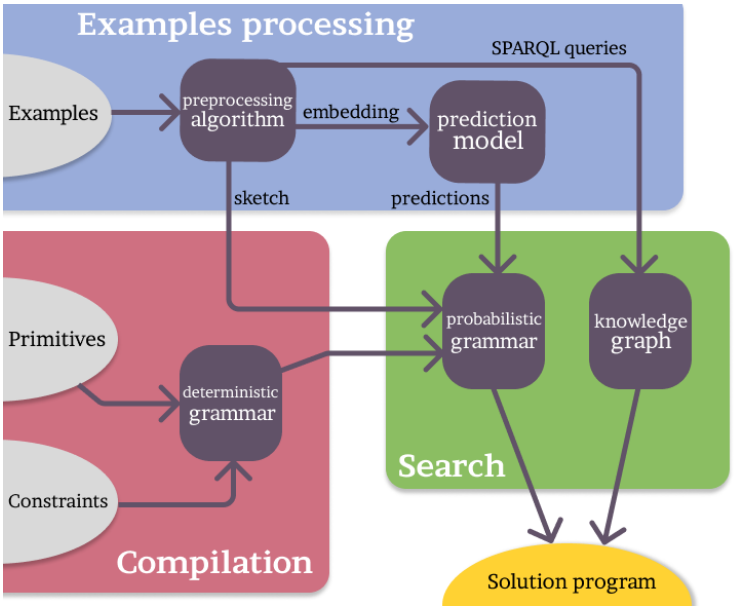

图 2. WikiCoder 流程。

我们通过一个示例场景来说明这些可能性。知识图谱由一家公司拥有并构建，其中包含有关产品的不可变数据。数据库中则包含客户数据。将被查询数据库与知识图谱中的语义信息交叉使用，可以让用户生成复杂查询，从数据库中提取更多信息，例如复杂统计。

3 WikiCoder

WikiCoder 是一种通用的基于语法的程序合成工具，使用 Python 开发，集成了最先进的程序合成技术。它已在 GitHub 上公开提供[3]。WikiCoder 的主要功能是解决示例驱动编程任务：用户输入几个示例，WikiCoder 便会合成一个满足这些示例的知识驱动程序。编程语言被指定为一种领域专用语言（DSL），旨在解决一组常见任务。WikiCoder 支持多种经典 DSL，包括塔构建 [7], 整数列表操作 [1], 正则表达式，以及类似 FlashFill 的字符串操作任务 [9]：我们的实验报告聚焦于后一种 DSL。遵循机器学习程序合成器的最新进展 [7,8]，WikiCoder 分为三个组件：编译、示例处理和搜索，如图2所示，并将在

[3] https://github.com/nathanael-fijalkow/ProgSynth/tree/WikiCoder

接下来的三个章节中讨论。这与 Codex 的架构截然不同，后者基于自回归超大型模型，直接生成代码。

3.1 编译

领域专用语言（DSL）通过一个基本操作列表来定义，并附有各自的类型和语义。编译阶段的工作是将程序集合转换为一种高效表示形式，即上下文无关文法。理解文法的正确方式是将其视为生成模型：它会生成代码。得益于上下文无关文法的表达能力，我们可以确保许多语法属性；其中首要且最重要的是，所有生成的程序都具有正确的类型。

我们在实验中使用的领域专用语言（DSL）专为 FlashFill 式的字符串处理任务量身定制。为便于介绍，我们对其做了少许简化。我们使用两种基本类型：字符串（STRING）和正则表达式（REGEXP），以及一个类型构造器 Arrow。下面列出这些基本操作，它们具有预期的语义。

$ ： 正则表达式 # 字符串结尾． ： 正则表达式 # 全部[^_]+ ： Arrow(字符串, 正则表达式) # 除 X 之外的全部[^_]+$ ： Arrow(字符串,正则表达式) # 结尾处除 X 之外的全部compose：Arrow(正则表达式, Arrow(正则表达式, 正则表达式))

连接 :Arrow(字符串, Arrow(字符串, 字符串)) 匹配 : Arrow(字符串, Arrow(正则表达式, 字符串))

连接_*if:*如果第二个参数（常量）# 不存在于第一个参数中，则连接连接_if : Arrow(字符串, Arrow(字符串, 字符串))# *split_fst:*使用正则表达式拆分，返回第一个结果split_fst : Arrow(字符串, Arrow(正则表达式, 字符串))# *split_snd:*使用正则表达式拆分，返回第二个结果split_snd :Arrow(字符串, Arrow(正则表达式, 字符串))

在实现中，我们还使用了两种基本类型：CONSANT_IN 和 CONSTANT_OUT，它们分别对应输入和输出中的常量。这只是为了提升性能，并不会增加表达能力。部分基本操作会重复定义，以便使用这两种新的基本类型。

3.2 示例处理

预处理算法从示例中提取三类信息：草图（即将当前任务分解为若干子任务）、针对知识图谱的一组 SPARQL 查询，以及供预测模型使用的示例嵌入。

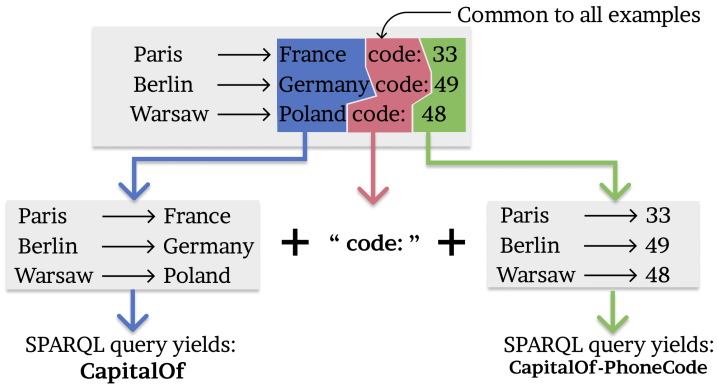

图 3. 预处理算法示例。

预处理算法 图 3 展示了预处理算法的运行过程。其核心思路是从不同示例中寻找共享模式。在上面的任务中，“ code: ”是所有示例共有的，因此将其提取出来。朴素实现方法是寻找字符串之间最长的公共因子，然后分别递归处理左侧和右侧。算法 1 展示了这一过程：它从字符串列表中生成常量列表。该过程分别应用于输入和输出，以生成草图。借助这些常量，任务被拆分为若干子任务，如图 3 所示，我们将任务拆分成了两个子任务：CapitalOf 和 CapitalOf-PhoneCode。为了解决每个子任务，我们使用输入和输出查询知识图谱。如果找不到路径，我们就运行常规的——基于语法的——程序合成算法。

算法 1常量提取

过程 GetConstants($S = (S_k)_{k \in [1,n]}$: 字符串)
  如果 S 中存在空字符串**则**
    返回空列表
  结束 if
  因子 $\leftarrow$ S中所有字符串的最长公共因子
  如果 len(factor) ≤ 2 **那么**
    返回空列表
  else
    $S_{\text{left}} \leftarrow$ S中因子的前缀
    $L_{\text{left}} \leftarrow$ GetConstants(S_{left})
    $S_{\text{right}} \leftarrow$ 因子在 S 中的后缀
    $L_{\text{right}} \leftarrow$ GetConstants(S_{right})
    返回 $L_{\text{left}} +$ 因子 $+ L_{\text{right}}$
  结束 if
结束过程

生成的*SPARQL* 查询SPARQL 查询是在对示例进行预处理后生成的。使用算法 1 从输入和输出中提取常量后，我们可以将输入和输出拆分为常量部分和非常量部分，只有非常量部分才相关。对于输出中的每个非常量部分，我们根据输入中的非常量部分生成查询，这些输入部分应映射到输出中的这个非常量部分。由于关系可能很复杂，也就是说，“巴黎”与“法国”相距 1 个关系，而“33”与“巴黎”相距 2 个关系，因此我们会针对递增的距离生成 SPARQL 查询，直到达到固定的上限。以下是距离为 2 的查询，我们使用 CapitalOf-PhoneCode 对图 3 中的示例执行该查询：

```
PREFIX w: <https://en.wikipedia.org/wiki/>
SELECT ?p0 ?p1 WHERE { w:巴黎 ?p0 ?o_1_0 . ?
o_1_0 ?p1 w:33 . w:柏林 ?p0?o_2_0 . ?o_2_0 ?
p1 w:49 . w:华沙 ?p0?o_3_0 . ?o_3_0 ?p1
w:48 . }
```

请注意，为了适应更长的路径长度，会出现中间实体。上述查询的输出将包含两条路径：城市-电话区号和 CapitalOf-PhoneCode。

作为一种消歧策略（受 [25] 启发），我们选择在所有示例中命中次数最少的路径，称为歧义最少路径。在此示例中，没有首选路径，因为两条路径对每个示例都只指向一个实体。作为这种消歧策略的示例，我们来考虑从“33”到“巴黎”的路径：有两条路径，PhoneCodeOf-Capital 和 PhoneCodeOf-City。歧义最少路径是 PhoneCodeOf-Capital，因为 PhoneCodeOf-City 会指向该国家的所有城市。

要找到歧义最少路径，我们需要统计命中次数，这可以通过更多 SPARQL 查询来完成。以下是一个示例查询，用于获取从起始实体“巴黎”出发，沿 CapitalOf-PhoneCode 路径到达的所有实体：

```
PREFIX w: <https://en.wikipedia.org/wiki/>
SELECT ?dstWHERE { w:巴黎 w:CapitalOf ?e0 .
?e0 w: 电话区号 ?dst . }
```

我们统计所有示例的结果数量，以获取某条路径的命中次数。

预测模型高效搜索对于程序合成至关重要，因为搜索空间会呈组合式爆炸。我们相信，更广泛的研究界也认同，学习是应对这一挑战的正确方法。接下来 [7,8],

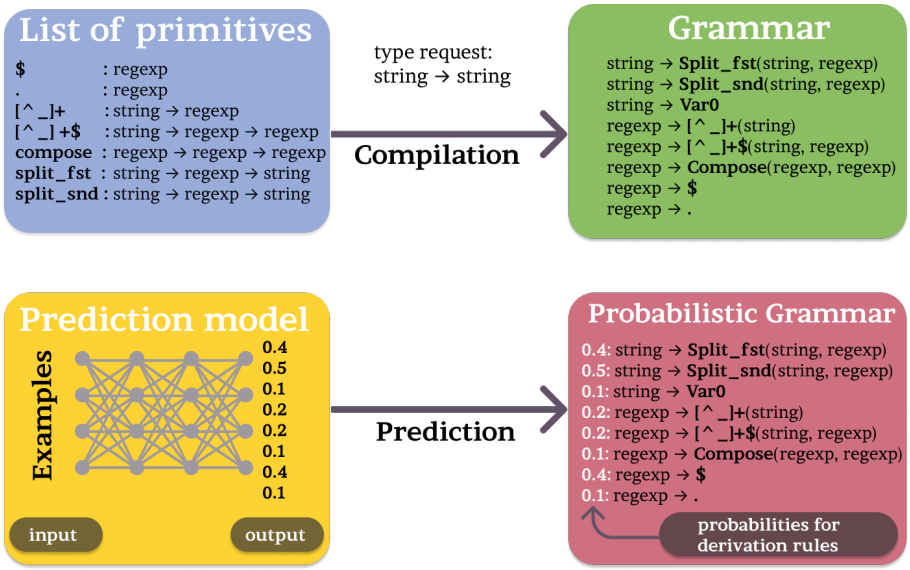

图 4. 预测模型示意图。

我们构建了一个神经网络形式的预测模型：它读取示例的嵌入，并输出文法推导规则的概率，这些规则表示了所有程序。这样一来，表示程序的上下文无关文法就转变为概率上下文无关文法，后者是一种生成程序的随机过程。换句话说，它定义了程序上的概率分布，而预测模型经过训练，旨在最大化生成解决方案程序的概率。

图 4 展示了这一预测过程。该方法有效利用了神经网络的预测能力，同时不牺牲正确性：预测模型会优先搜索最有可能的程序，但不会从搜索树中删除任何部分，因此——从理论上讲——只要存在解，就一定能找到。

3.3 搜索

预测模型为每个候选程序分配一个可能性值，表示它是当前任务解决方案的概率。WikiCoder 使用 HeapSearch 算法，这是一种自底向上枚举搜索算法 [8]，按照预测模型给出的可能性顺序输出程序。事实证明，它优于其他枚举方法（如 A* 或束搜索），且易于部署在并行计算环境中。候选程序使用知识图谱上的 SPARQL 查询结果，并在示例上运行。

4 评估

我们通过实验来回答以下问题：

(Q1) 第 2.2 节所述的哪些里程碑可以通过我们的算法实现？(Q2) WikiCoder 与 GPT-3 和 Codex 相比表现如何？Codex 是 Copilot 的底层算法，基于超大型语言模型。(Q3) 知识驱动的程序合成工具能否大规模用于经典的纯语法任务？

4.1 环境

实验所用的数据集以及代码已在 GitHub[4] 上公开。WikiCoder 使用 Python 编写，并采用了一个以 PyTorch 编写的预测模型。如上所述，我们优先考虑简洁性而非性能，并将系统清晰地划分为相互独立的组件。

基准套件 由于目前没有可用于测试第 2.2 节所述新特性的现有数据集，我们自行创建了一个。该数据集包含 46 项任务，其设计理念借鉴了 FlashFill 数据集 [9]。其中一些任务的灵感来自 [22], ，或从中提取；这些任务在我们的代码中均有相应标记。每项任务都需要借助外部知识才能解决，并带有 3 项元数据：

– 实体抽取的预处理：如果输入已经是目标实体，则为 0；如果只需语法程序即可抽取，则为 1；否则为 2；– 实体间关系的复杂程度：如果无需使用知识图谱中的关系，则为 0；如果是简单关系（知识图谱中的单条边），则为 1；如果是复合关系，则为 2；– 外部知识的后处理：如果无需对知识进行后处理，则为 0；否则为 1。

这形成了 8 个类别，我们为每个类别至少提供 4 个任务。

知识图谱 为简化流程并确保可复现性，我们在实验中使用了一个自制的小型知识图谱。我们提供了用于构建该知识图谱的 SPARQL 查询。

实验设置 所有实验均在一台消费级笔记本电脑（MSI GF65 Thin 9SE）上进行，配备英特尔 i7-9750H 处理器，单线程运行频率最高可达 2.60GHz。操作系统为 Manjaro 21.3，使用 Linux 内核 5.17.5-1。代码使用 Python 3.8.10 编写。SPARQL 数据库所用的框架为 BlazeGraph 2.1.6。提供的代码包含复现实验所需的全部细节，也包括所用 Python 库的确切版本。

[4] https://github.com/nathanael-fijalkow/ProgSynth

预测模型 在预测模型中，我们使用独热编码对示例进行嵌入。由于不同任务的示例数量可能不同，因此将示例输入单层循环神经网络，以构建固定形状的中间表示。随后接入 4 个线性层，并使用 ReLU 激活函数；最后一层除外，不使用激活函数。最终张量会被拆分，并对文法中的每个非终结符应用对数 softmax 运算，从而生成对数空间中的概率上下文无关文法。

预测模型在包含 2500 个任务的数据集上训练 2 个 epoch，每批包含 16 个任务。任务通过以下过程生成：从均匀概率上下文无关文法中随机采样一个程序，然后随机生成输入并在该程序上运行。我们使用 Adam 优化器和交叉熵损失，最大化从文法中生成解答程序的概率。

4.2 新数据集上的结果

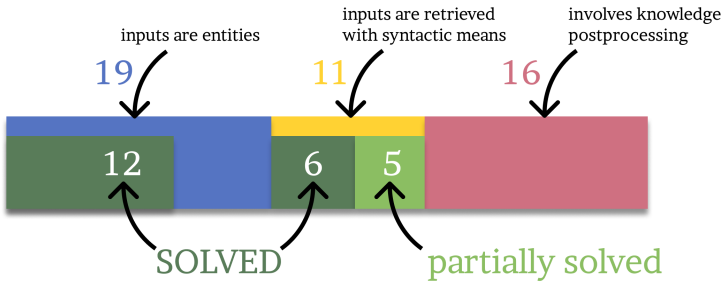

图 5. WikiCoder 在新数据集上的表现

WikiCoder 在 60 秒超时限制内解决了 46 个任务中的 18 个。让我们进一步理解这个数字：

– 由于我们的算法不对知识进行后处理，因此涉及知识后处理的16项任务均未能解决。因此，我们的算法能够处理的任务只有30项。– 在这30项任务中，有19项的输入本身就是要查找的实体。WikiCoder解决了其中12项；大多数未能解决的任务都无法通过我们的领域专用语言解决。– 在剩余的11项任务中，可以通过句法操作检索实体。WikiCoder解决了其中6项。– 进一步分析最后一种情况发现，剩余5项任务的问题都出在预处理算法上：它未能正确分解任务并构造适当的 SPARQL 查询。不过，如果提供正确的分解方式，WikiCoder就能解决全部5项任务。

结果如图5所示。核心结论是：WikiCoder几乎解决了所有不涉及知识后处理的任务；对于未能解决的任务，主要问题在于预处理算法的实体抽取能力。

4.3 与 Codex 和 GPT-3 的比较

由于 Codex 和 GPT-3 解决的问题各不相同，因此很难进行公平的比较。这些模型有两种方式可以解决示例驱动编程任务。实验使用 OpenAI 的测试版平台，采用 GPT-3 和 Codex 的达芬奇模型（能力最强的模型），并使用默认参数。

使用 Codex：将其用作代码补全工具，并以文档字符串的形式提供示例。这使得 Codex 面临的任务非常困难，原因有二：首先，它受训于解析自然语言规格说明，这些说明可能辅以示例。仅提供示例是一种非常不完整的规格说明，Codex 并不习惯这种形式。其次，Codex 不使用领域专用语言，而是使用 Python 和 C 等通用编程语言。这意味着程序可能以不同方式出错：语法错误、类型问题或编译错误。基于上述原因，在这种对 Codex 不利的比较中，它只能解决最简单的 5 个任务。

使用 GPT-3 有两种方式：查询回答或代码补全。对于查询回答，我们将除最后一个示例之外的所有示例提供给 GPT-3，并要求它给出最后一个示例的输入所对应的输出：

查询：

```
f("法国") ="我住在法国"f("德国") =
"我住在德国"f("波兰") ="我住在波兰"
f("新西兰") = ?
```

输出：f("新西兰") = "我住在新西兰"

这种情形的弱点在于，GPT-3 不会输出程序，只会回答查询。因此，正确性测试非常薄弱：在单个查询上得到正确答案，并不能保证它对任何输入都能给出正确答案。更糟的是，不输出程序意味着整个过程成了一个黑箱，放弃了我们的框架（参见相关工作部分）以及程序合成通常所具有的优势。

将 GPT-3/ChatGPT 用作代码补全：提示内容是生成一个满足以下示例的 Python 函数。这解决了无法输出具有查询回答功能的程序这一弱点。不过，对于需要知识图谱的任务，答案要么是一连串 if 语句，要么是使用在语义上等同于这些 if 语句的字典。从某种意义上说，由于系统只掌握部分知识，因此不会尝试超越给定示例进行泛化。

GPT-3 在查询回答场景中表现出色，在 46 项任务中解决了 31 项。只有 2 项任务由 WikiCoder 解决而未由 GPT-3 解决；反过来，有 11 项由 GPT-3 解决而未由 WikiCoder 解决。GPT-3 只解决了 3 项涉及知识后处理的任务：从这个意义上说，WikiCoder 和 GPT-3 面临着相同的局限，它们都难以处理知识后处理。

4.4 FlashFill 上的结果

为展示 WikiCoder 在经典程序合成任务中具备良好的扩展性，我们使用经典且成熟的 FlashFill 数据集对其进行测试。结果如图 6 所示（展示累计耗时）：在每项任务限时 60 秒的情况下，WikiCoder 成功解决了 101 项任务中的 70 项，性能与最先进的通用程序综合工具相当。使用我们的领域专用语言只能解决 85 项任务。

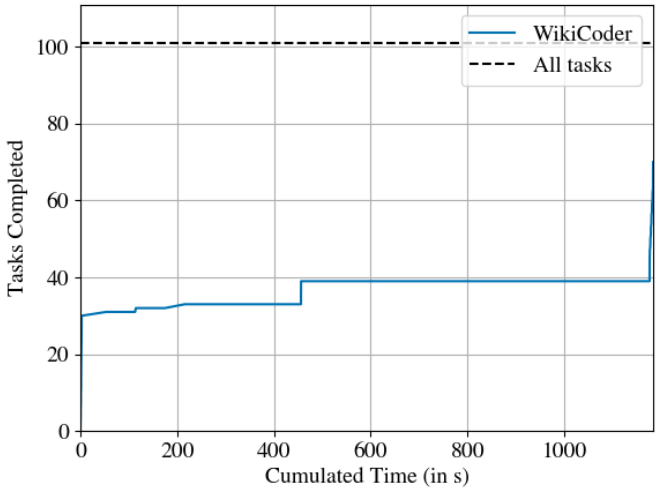

图 6. WikiCoder 可解决 70% 的 FlashFill 任务

4.5 局限性

为展示我们方法的局限性，我们讨论一些反例。

实体抽取 我们的预处理算法会在示例中寻找长度至少为 2 的最长共有模式。来看下面这个示例：

```
f("法国") ="我住在法国"f("德国") =
"我住在德国"f("波兰") ="我住在波兰"
```

最长的模式自然是“我住在”，然而，另一个模式却意外地出现了：“an”出现在每个国家名称中。使用我们的预处理算法时，这会导致草图错误。如果国家名称中不包含“an”，再添加第四个示例就可以消除这种歧义。一个相关的例子是：如果所有示例都包含同一千年的年份日期，例如都采用“20XX”这种形式，那么“20”就会出现在每个示例中，但不应将它与 XX 分开。

知识后处理 没有解决任何涉及知识后处理的任务。事实上，如果示例中没有要使用的实体，就很难猜出应该使用哪一个。自然的候选对象是起始实体邻域中的实体。我们应对这一挑战的方法已被证明效率低下。

4.6 程序示例

为了展示生成的程序清晰易懂且可供人类解读，我们展示其中几个程序转换为 Python 等效代码后的示例。下面是第一个需要两种关系的示例：

```
// 示例：f("法国") ="法国人，首都：巴黎"f("德国") ="德国人，
首都：柏林"f("中国") ="中国人，首都：北京"f("新西兰") = "新
西兰人，首都：惠灵顿"


// 生成的程序：
定义 f(x：字符串) ->字符串：a = 标签(跟随_边_来自(x,
"居民称谓"))b = "，首都："c = 标签(跟随_边_来自(x，"
是……的首都"))返回 a +b + c
```

我们再展示一个需要距离为 2 的关系的示例：

```
// 示例：
    f("巴黎") ="电话国家代码是 33"f("柏林") ="电话国
    家代码是 49"f("底特律") ="电话国家代码是 1"f("奇
    瓦瓦") ="电话国家代码是 52"

// 生成的程序：
def f(x:str) ->str:a = “电话国家代码是”b = 标签（沿着_边_
从(x,“城市所属”，“电话区号”))返回 a +b
```

5 讨论

5.1 相关工作

最相关的工作是 [22]: 工具 FlashGPT3 采用截然不同的方法解决知识驱动的程序合成问题：FlashGPT3 不查询知识图谱，而是使用大语言模型（GPT3 [3]）获取外部知识。研究表明，FlashGPT3 的表现优于单独使用程序综合工具或 GPT3，并在不同领域取得了令人瞩目的成果。与大语言模型相比，使用知识图谱有三大优势：

– **可靠性**：合成程序的可靠性取决于它所使用的知识来源。为说明这一点，我们来考虑以下任务
  `f(“巴黎”)` =“法国”`f(“柏林”)`
  =“ 德国 ”
  GPT3 只需很少的示例就能预测
  `f("华盛顿") = "美国"`
  但对于更特殊的输入，它可能会失败。另一方面，查询知识图谱会得到 f = CountryOf 的赋值，这意味着程序能够正确复现图谱中的知识。尽管 GPT3 在查询回答方面取得了非凡的成果，但依靠维基百科这类成熟且可能经过认证的知识来源仍然更令人满意。

– **可解释性**：构造出的程序具有良好的可解释性：它所使用的确切知识来源可以追溯到知识图谱，因此能够得到有效验证。

– **适应性**：知识驱动的程序合成可以部署于任何知识图谱，也可以收集大型语言模型可能并不掌握的专业信息或私人信息。

其他知识驱动的程序综合工具 大多数程序综合工具都只在语法层面运作。然而，有些工具在特定领域任务中具备有限的语义能力：例如，示例驱动的数据转换（TDE）使用代码库和网页表单中的函数，以便在归纳综合中执行语义操作 [11],；API 则用于数据转换 [2]。

知识图谱越来越多的研究人员致力于创建、维护和查询知识图谱。与我们的研究设定最相关的问题是示例查询，其目标是根据一组示例 [15,18], 构建查询；另一个是实体集扩展，旨在将一小组示例扩展为一组更完整、具有共同特征的实体 [25]。

5.2 贡献与展望

我们提出了知识驱动的程序合成，通过允许程序依赖知识图谱，扩展了程序合成的范畴。我们介绍了这一令人振奋且尚未得到充分探索的研究目标所涉及的若干里程碑，并提出了一个用于评估该领域进展的数据集。我们设计了一种算法，并实现了通用的知识驱动程序合成工具 WikiCoder，它能够解决此前无法解决的任务。我们的工具目前只能处理数据集约三分之一的内容；我们认为，解决整个数据集将是推动程序合成在现实世界中落地的重要一步。

这项工作的自然延续方向是利用超大型语言模型进行知识驱动的程序合成。如上文所述，在借助超大型语言模型强大能力的同时，我们该如何保留知识图谱框架的特性：可靠性、可解释性和适应性?

参考文献

1. M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow. Deepcoder：学习编写程序。发表于国际学习表征会议，*ICLR*，2017。2. S. Bhupatiraju, R. Singh, A. Mohamed, and P. Kohli. Deep API programmer：学习使用 API 编程。$CoRR$，abs/1704.04327，2017。3. T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Nee- lakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. Mc- Candlish, A. Radford, I. Sutskever, and D. Amodei. 语言模型是少样本学习者。发表于神经信息处理系统年会，*NeurIPS*，2020。4. M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Ed- wards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plap- pert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. Mc-Candlish, I. Sutskever, and W. Zaremba. 对使用代码训练的大语言模型进行评估。$CoRR$，abs/2107.03374，2021。5. X. Chen, C. Liu, and D. Song. 执行引导的神经程序合成。发表于国际学习表征会议，*ICLR*，2018。6. J. Devlin, J. Uesato, S. Bhupatiraju, R. Singh, A. Mohamed, and P. Kohli. Robustfill：在含噪输入/输出条件下进行神经程序学习。发表于国际机器学习会议，*ICML*，第 70 卷机器学习研究论文集，2017。

7. K. Ellis, C. Wong, M. I. Nye, M. Sablé-Meyer, L. Morales, L. B. Hewitt, L. Cary, A. Solar-Lezama, and J. B. Tenenbaum. Dreamcoder：通过唤醒-睡眠库学习引导归纳式程序综合。发表于国际编程语言设计与实现会议，*PLDI*，2021。8. N. Fijalkow, G. Lagarde, T. Matricon, K. Ellis, P. Ohlmann, and A. Potta. 通过基于分布的搜索扩展神经程序综合。发表于人工智能国际会议，*AAAI*，2022。9. S. Gulwani. 使用输入输出示例自动处理电子表格中的字符串。发表于*ACM SIGPLAN-SIGACT* 编程语言原理研讨会，*POPL*，2011。10. S. Gulwani, O. Polozov, and R. Singh. 程序合成。编程语言基础与趋势，4(1-2)，2017。11. Y. He, X. Chu, K. Ganjam, Y. Zheng, V. R. Narasayya, and S. Chaudhuri. Transform-data-by-example (TDE)：一种可扩展的数据转换搜索引擎。*VLDB* 会刊，11(10):1165–1177，2018。12. J. Hoffart, F. M. Suchanek, K. Berberich, and G. Weikum. YAGO2：从维基百科构建的时空增强型知识库。人工智能，194:28–61，2013。13. A. Hogan. 知识图谱：导览（特邀论文）。载于 C. Bourgaux, A. Ozaki, and R. Peñaloza 编，卑尔根人工智能国际研究学院，*AIB 2022*，*2022*年*6*月*7*日至*11*日，卑尔根大学，挪威，OASIcs 第 99 卷，页码 1:1–1:21。Schloss Dagstuhl - Leibniz-Zentrum für Informatik，2022。14. A. Hogan, E. Blomqvist, M. Cochez, C. d'Amato, G. de Melo, C. Gutiérrez, S. Kir- rane, J. E. L. Gayo, R. Navigli, S. Neumaier, A. N. Ngomo, A. Polleres, S. M. Rashid, A. Rula, L. Schmelzeisen, J. Sequeda, S. Staab, and A. Zimmermann。知识图谱。《数据、语义与知识综合讲座》。Morgan & Claypool Publishers，2021。15. N. Jayaram, A. Khan, C. Li, X. Yan, and R. Elmasri. 基于示例实体元组查询知识图谱。发表于第*32*届 *IEEE* 国际数据工程会议，*ICDE 2016*，赫尔辛基，芬兰，*2016*年*5*月*16*日至*20*日，页码 1494–1495。IEEE 计算机学会，2016。16. A. Kalyan, A. Mohta, O. Polozov, D. Batra, P. Jain, and S. Gulwani. 面向实时程序综合的神经引导演绎搜索。发表于国际学习表征会议，*ICLR*，2018。17. W. Lee, K. Heo, R. Alur, and M. Naik. 利用学习得到的概率模型加速基于搜索的程序综合。发表于*ACM* 编程语言设计与实现会议，*PLDI*，2018。18. S. Metzger, R. Schenkel, and M. Sydow. QBEES：基于最大方面、多样性感知和松弛，在语义知识图谱中进行基于示例查询的实体搜索。*J. Intell. Inf. Syst.*，49(3):333–366，2017。19. I. Polosukhin and A. Skidanov. 神经程序搜索：根据描述和示例解决编程任务。发表于国际学习表征会议，*ICLR*，2018。20. K. Shi, D. Bieber, and R. Singh. TF-coder：用于张量操作的程序合成。*ACM* 编程语言与系统汇刊，44(2):10:1–10:36，2022。21. T. P. Tanon, G. Weikum, and F. M. Suchanek. YAGO 4：一个合理的知识库。发表于语义网——第*17*届国际会议，*ESWC*，计算机科学讲义第 12123 卷，页码 583–596。Springer，2020。

22. G. Verbruggen、V. Le 和 S. Gulwani。利用预训练模型进行基于示例的语义编程。*ACM* 编程语言论文集，5(OOPSLA)：1–25，2021。23. C. Wang、A. Cheung 和 R. Bodík。根据输入输出示例合成表达能力极强的 SQL 查询。载于 A. Cohen 和 M. T. Vechev 编，《style id='3'>编程语言设计与实现会议，PLDI》，第 452–466 页。ACM，2017。24. L. Zhang、G. Rosenblatt、E. Fetaya、R. Liao、W. E. Byrd、R. Urtasun 和 R. S. Zemel。利用约束逻辑编程实现神经网络引导的程序合成。载于 国际学习表征会议，*ICLR*，2018。25. Y. Zheng、C. Shi、X. Cao、X. Li 和 B. Wu。一种基于元路径的知识图谱实体集扩展方法。*IEEE* 大数据汇刊，8 629，2022 3 616( ): – 。26. X. Zhou、R. Bodík、A. Cheung 和 C. Wang。通过计算演示合成分析型 SQL 查询。载于 R. Jhala 和 I. Dillig 编，《style id='9'>国际编程语言设计与实现会议，PLDI》，第 168–182 页。ACM，2022。27. A. Zohar 和 L. Wolf。利用学习到的垃圾回收器自动合成长程序。载于 神经信息处理系统大会，*NeurIPS*，2018。