arXiv:2302.03927v1 [cs.PL] 8 Feb 2023

论语言模型对基于块的程序的适用性

Elisabeth Griebl*、Benedikt Fein*、Florian Oberm¨uller*、Gordon Fraser*、Ren´e Just†
*帕绍大学，德国帕绍
†华盛顿大学，西雅图，美国{elisabeth.griebl, benedikt.fein, florian.obermueller, gordon.fraser}
@uni-passau.de, rjust@cs.washington.edu

摘要—像 SCRATCH 这样的基于代码块的编程语言在编程教育和终端用户编程中日益流行。近期的程序分析建立在这样一种认识之上：可以使用自然语言处理技术对源代码进行建模。支持这一方法的许多源代码规律，都源于文本编程语言所带来的语法开销。然而，基于代码块的语言正是为了简化编程而移除了这种语法开销。因此，这种建模方法在基于代码块的编程语言上的表现究竟如何，尚不明确。本文研究语言模型对流行的基于积木的编程语言 SCRATCH 的适用性。我们使用 n 元模型（语言模型中最基本的一类）和 Transformer（一种流行的深度学习模型）对 SCRATCH 程序进行建模。在代码补全和缺陷发现这两个示例任务上的评估证实，代码块会抑制可预测性，但使用语言模型仍然是可行的。我们的研究结果为改进基于代码块的语言的工具和分析奠定了基础。

索引词—**基于块的程序、Scratch、自然语言模型、代码补全、BUGRAM**

I. 引言

基于代码块的编程语言在教育领域 [1] 以及需要终端用户编程的工业应用中 [2]–[4]越来越受欢迎。这类编程语言的显著特点是，能够减少文本编程语言中常见的语法开销，转而使用图形化积木表示编程结构，并且这些积木只能以符合语法的方式组合。图 1a 展示了一个 Java 函数，它会打印 10 次“Hello world!”；而在广受欢迎的基于代码块的编程语言 SCRATCH 中，只需三个代码块即可实现相同的功能 [5]（图 1b）。通常还会通过在用户界面中明确列出所有可用代码块，进一步简化编程，使程序员既无需记忆语法，也无需记忆可用的命令和 API（依赖识别而非召回率）。

与使用基于文本的编程语言编写的程序一样，我们也需要对基于块的程序进行程序分析：学习者可以从自动生成的提示和反馈中受益，程序员则可以从代码补全或错误检测中受益。实现此类分析的一种常见方法是将源代码视为自然语言，从而借助近期自然语言处理（NLP）方法研究的蓬勃发展。这些方法的核心是语言模型这一概念，它能够捕捉序列上的概率分布

```
public static void main(
    String[] args) {
  对于 (整数i = 0; i < 10; i++) {
    System.out.println(
      "Hello World!");
  }
}
```

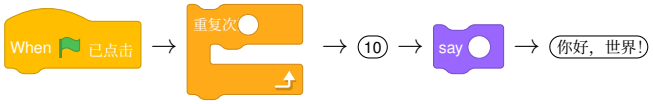
(a) Java 函数。



(b) SCRATCH 代码。

public → static → void → main → (→ String → [→] →args →) → { → for → (→ int → i → = → 0 → ; → i → <→ 10 → ; → i → ++ →) → { → System → . → out → . →println → (→ ¨ → 你好，世界！ → ¨ →) → ; → } → }

(c) Java令牌序列。



(d) SCRATCH 词元序列。

图 1：文本格式和基于积木的格式中的示例代码。

词语。人们发现，源代码呈现出规律性，因此适用于自然语言处理 [6],；同样的模型也可用于捕捉源代码的概率分布。例如，这些模型可以预测常见的代码序列以实现代码补全 [7], ，或识别异常的代码序列以进行错误检测 [8], [9]。

语言模型通过将源代码解释为由词元组成的序列来构建，即由空白字符或分隔符分开的单个词语或符号。因此，图 1a 中的 Java 程序会被解释为图 1c 所示的由 39 个词元组成的序列。同一程序的 SCRATCH 版本（图 1b）则只会产生一个由五个词元组成的简单流（图 1d）。一方面，这种差异有力地印证了基于块的编程在降低认知开销方面远胜于基于文本的编程语言 [10]。另一方面，这种简化会如何改变最终生成的语言模型，目前尚不清楚。即使移除“语法”词元 [11],，JAVA 示例中剩余的词元直观上仍处于比 SCRATCH 程序词元更低的抽象层次；后者所包含的修饰符或类型等技术细节较少。因此，语言模型是否适用于基于块的程序分析任务，仍不明确。

为了探究能否使用语言模型对基于代码块的程序进行建模和分析，我们通过实证研究

了基于使用 SCRATCH 编程语言编写的程序的模型 [5], 。SCRATCH 是最受欢迎的基于代码块的编程语言，面向年轻学习者。一个由数百万用户组成的活跃社区会分享他们的程序，从而提供大量代码，使我们能够对语言模型进行外部评估。具体而言，本文的贡献如下：

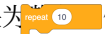
- 我们描述并实现了从 SCRATCH 程序创建 n 元模型的过程。尽管存在各种替代性的神经模型，但研究表明，n 元模型在许多任务中表现良好；而要获得可靠的理解，则需要使用具有可解释性的模型。
- 我们使用一个由公开分享的 SCRATCH 项目构成的数据集，评估 n 元模型是否适用于常见的代码补全任务，即预测令牌流中下一个将使用的代码块。
- 我们研究 n 元模型识别 SCRATCH 编程作业中错误解答的能力。
- 我们探究作为一种流行深度学习模型的 Transformer，能否相比 n 元模型提升补全任务的性能。

我们的实验确认，基于块的程序与基于文本的程序存在根本差异，这种差异会对其可预测性产生不利影响。然而，语法和重复性方面仍存在一些特征，使代码块具有足够的可预测性，从而能够在基于代码块的编程语言中使用自然语言模型。

II. 背景

近年来，基于代码块的编程语言在教授新手编程概念方面受到越来越多的关注 [1]，也应用于需要终端用户编程的工业领域 [2]–[4]。本文聚焦于广受欢迎的教育编程语言 SCRATCH [5]。

A. Scratch编程语言

SCRATCH [5] 是一种面向年轻学习者的基于代码块的编程语言。Scratch 程序控制环境（舞台）中角色的行为；每个角色都可以执行多个脚本。图 1b 展示了一个这样的脚本：脚本以事件处理程序（例如，）开头，后接在事件发生后执行的代码块。为了支持识别和召回，代码块根据类别以不同颜色标示：控制结构为橙色，如图 1b 中的代码块；影响角色视觉外观的代码块为例如，）等。代码块还分为不同形状，例如可堆叠积木块（语句），以及圆形报告器积木块，它们可以嵌入其他代码块的空槽中（表达式）。代码块可以包含用于输入数字和字符串的自由文本区域，例如，也可以包含用于选择预设选项的下拉菜单。SCRATCH 促进了混编文化 [10]：用户可以分享自己的程序，其他人则可以克隆并改进这些程序。

尽管代码块形状能够避免语法错误，但构建程序仍可能颇具挑战：学习者可能难以实现所需功能，或存在误解 [12]–[15]，程序员也可能错失现代编程环境带来的便利与支持。因此，人们提出了各种分析工具，主要用于实现传统的程序分析，例如代码检查（如 [16], [17]）或自动化测试（如 [18], [19]）。然而，据我们所知，目前尚无使用自然语言处理方法的分析工具。

B. n元模型

概率语言模型用于为给定语言中的词元序列分配概率。n元语言模型基于马尔可夫假设，该假设认为，句子 s 的概率可以根据其所有词元 $w_1 \dots w_n$ 出现的概率链来估算。n 元模型进一步简化了这一思路，假设每个词实际上只依赖于它前面的 $n-1$ 个词，也就是说，只依赖于其局部上下文。给定 $n=3$ 和 $s=\langle w_1 w_2 w_3 w_4 \rangle$，n元模型据此按如下方式估算 s 的概率：

$$P(s) \approx P(w_1) \times P(w_2|w_1) \times P(w_3|w_1 w_2) \times P(w_4|w_2 w_3)$$

与常规马尔可夫链不同，w_4 的概率估算只考虑上下文 $w_2 w_3$，而不考虑 $w_1 w_2 w_3$。乘积中的各项是条件概率，根据训练数据中词元的出现次数估算。例如，如果训练数据中只有 c 出现在 ab 之后，那么概率 $P(c|ab)$ 在局部上下文 ab 下为 1。

在实际应用中，概率为 1 的情况并不现实：训练数据很可能没有包含局部上下文之后可能出现的所有词元。因此，平滑算法会根据 n 元语法的计数调整原始概率，使训练数据中未出现的其他 n 元语法也能被赋予概率 >0。这样一来，模型就不会因遇到训练数据集中未出现的 n 元语法而“无限惊讶”。一种常用的平滑机制是改进的 Kneser-Ney 平滑 [20]，据报道，它在自然语言场景中表现最佳 [21]，此前也曾用于编程语言方面的研究 [6]。

C. 深度学习模型：*Transformer*

尽管 n 元模型因其简洁易懂而备受重视，近期研究已转向神经网络方法。Vaswani 等人 [22] 提出了 Transformer 架构，以便在自动化自然语言翻译过程中捕捉长程信息。Transformer 的设计采用了编码器-解码器架构 [22]：在编码部分，模型会学习用于表示不同词语关系的权重矩阵，以编码序列中其他每个词对某个词的编码所产生的影响程度 [22]。在解码过程中，下一个生成的词元不仅受先前生成的输出影响，也受整个输入序列对应的权重矩阵影响 [22]。Transformer 支持自监督学习，例如，可以对输入序列中的随机词元进行掩码，并训练 Transformer 预测缺失的词语 [23]。与 n 元语法模型相比，这些模型的训练过程通常需要更多计算资源，但已有研究表明，在大型数据集上训练的 Transformer，

（例如 BERT [23], 、CodeBERT [24]），无需微调或仅需少量微调，即可用于辅助源代码处理领域的任务 [25]–[28]。

D. 基于语言模型的程序分析

Hindle 等人基于“自然性假设”提出使用 n 元模型对源代码建模 [6]。作为一项初步应用，他们展示了一种基于 n 元语法模型的简单代码补全方法，该方法根据补全内容的局部上下文，对最可能的词元进行排序。也就是说，根据给定的局部上下文选择概率最高的 n 元组，从而最大化完整序列的概率。

利用自然语言处理的概念并将其与其他技术结合的想法，已在其他工作中得到成功应用，例如代码补全领域 [7],[29]。然而，对自然性假设的进一步研究也表明，代码的自然性很大程度上源于括号或分号等语法元素 [11]。尽管源代码的自然性不如先前认为的那么强，即使移除某些语法元素，仍然可以在源代码中发现规律。特别是，尽管自然性假设存在局限，研究发现 n 元模型能够很好地表示源代码，在配置得当时，其效果往往优于深度神经网络 [30], 。

如果源代码具有规律性，那么其中的不规则之处就值得怀疑：Ray 等人证明，含缺陷代码的熵高于正确的源代码 [9]。这一发现使得我们可以应用 n 元模型来识别源代码中的漏洞。具体而言，Wang 等人提出了 BUGRAM，这是一种基于 n 元模型自动查找漏洞的方法 [8]。对于给定的项目，BUGRAM 会在源代码上训练一个 n 元模型，计算源代码中所有序列的概率，并将概率较低的序列报告为潜在漏洞。

由于 Transformer 具有特殊的结构，它们为代码分析带来了更多可能性。通过在同时包含词元化代码及其文档的序列上训练模型，可以让模型联合学习代码和自然语言，从而根据自然语言描述进行代码搜索，或生成文档 [24]。这些预训练模型还可用于仅针对代码的任务，例如识别含缺陷代码 [31]并为其生成可能的修复方案 [32]。

III. 用于 SCRATCH 的语言模型

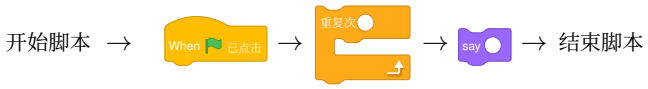
SCRATCH 与此前应用过语言模型的基于文本的编程语言不同。因此，我们首先需要定义如何对 SCRATCH 程序进行分词。随后，我们将说明如何生成 n 元模型，以及如何将其用于代码补全和缺陷发现。我们还将介绍如何获得 Transformer，以及如何将其用于代码补全。

A. 对 SCRATCH程序进行分词

对基于文本的编程语言进行词元化很直接，例如，直接对源代码进行词法分析即可。但如何对 SCRATCH 程序进行词元化，则不那么显而易见：SCRATCH 程序由一个 ZIP 文件组成，其中包含资源（图像、声音）以及一个采用 JavaScript 对象表示法（JSON）格式的文本文件，用于描述代码。JSON 文件以目标 （即舞台和角色）为单位描述程序；每个目标都包含其名称、变量、列表、消息、声音、造型、脚本、过程（即自定义代码块）和代码块。代码块以任意顺序列出（例如，按它们插入程序的顺序），每个代码块都包含一个唯一标识符，以及其父代码块、后继代码块和任意参数积木的标识符。在 SCRATCH 虚拟机中，代码块标识符及其父子关系用于创建类似语法树的表示。尽管 JSON 格式是 SCRATCH 特有的，其他基于块的语言也会以类似方式表示程序；例如，SNAP! 使用 XML 对代码块进行编码 [33]。为了对程序进行词元化，我们首先使用 LITTERBOX [16] 分析框架提供的解析器，并为该程序创建抽象语法树。随后，我们以前序遍历的方式遍历语法树，将遍历到的每个表示具体代码块的节点添加到令牌流中。最终得到的词元序列如图 1c 所示。

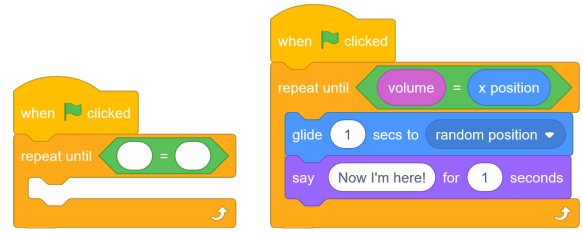
为了减小词汇表大小并避免未登录词问题 [34], ，我们按以下方式处理字面量 [35]：首先，不纳入字符串和数字字面量。一方面，预测字面量非常困难；另一方面，用户输入的文本和数字通常高度依赖具体使用场景。其次，借鉴先前的研究 [36], [37], ，我们将具体变量的出现统一处理为通用变量积木 的出现，并将对自定义代码块的调用视为过程调用 。由于大多数程序实际上只定义少数几个变量和过程，这种做法一方面可能不会造成太大的信息损失，另一方面也简化了跨项目的泛化。

尽管 SCRATCH 在其内部表示中会将某些代码块包含的下拉菜单视为独立的代码块，但我们不会将其纳入词元，因为它们不可分割，且针对特定代码块和使用场景定制。因此，总体而言，我们仅将语句和表达式积木作为词元，词汇表大小由此为 137 个代码块。与自然语言处理中的惯例相同，我们引入用于标记脚本或角色起始和结束位置的结构积木。因此，图 1b 中脚本剩余的词元序列如下所示：



B. 使用 *n* 元模型进行代码补全

n 元模型是最基础的一类语言模型。与较新的深度学习方法相比，它们所需的训练数据相对较少，并且已成功应用于各种软件工程任务（参见第 II-D 节），因此我们将其作为本研究中使用语言模型的基线。我们首先将 n 元模型应用于代码补全任务。其思路是在按顺序（即按照词元序列的顺序）解释前面的代码时，为下一个代码块



(a) 开发过程中的不完整脚本。 (b) 使用一种罕见条件的完整脚本，该条件位于重复代码块中。

图 2：SCRATCH 脚本示例。

提供代码补全。图 2a 展示了一个处于开发过程中的简单 SCRATCH 程序。现有的代码词元将按以下顺序进行解释：



对于三元语法模型，[illegible]模型需[illegible]的局部上下文。因此，在此情况下，代码补全应建议通常用于 while 循环中相等比较的代码块，例如 [illegible] 、 [illegible] 或 [illegible] 。为此，我们按照 III-A 节所述，在一大批 SCRATCH 程序上采用改进的 Kneser-Ney 平滑构建一个“通用”n元语法模型。

SCRATCH 中的代码补全有一种特殊情况，涉及过程定义代码块：与 Java 中的方法头类似，这些代码块构成了新建自定义代码块的头部。由于过程定义并不像其他代码块那样通过拖放添加，而是通过专门的对话框设置名称并选择可能的参数，因此我们不将过程定义纳入预测。另一种特殊情况是，当模型根据训练期间观察到的结束脚本代码块预测代码块序列的末端时，代码补全的行为。在这种情况下，补全不会返回这个对用户毫无价值的预测，而是预测一个新的首个代码块，即补全脚本开头上下文对应的内容。

据我们所知，这是首个针对基于代码块的编程语言实现的代码补全。因此，我们实现了一种纯粹基于 n 元模型概率的简单代码补全方法，使我们能够评估语言模型本身对该语言的表征效果，并为后续研究提供基线。

C. 使用 n 元模型进行缺陷发现

语言模型能够反映出含缺陷代码比无缺陷代码的规律性更低这一特点 [9]。例如，图 2b 展示了一个 SCRATCH 脚本，其中循环代码块内有一个极不可能出现的条件：将音量 [illegible] 与角色的 x 坐标 [illegible] 进行比较，几乎不可能是有意义的比较。理想情况下，缺陷发现方法应能够捕捉源代码中的这些及其他不规则之处。

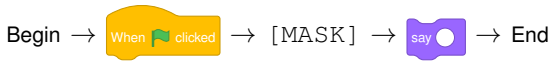
这一原理已应用于 BUGRAM 方法 [8]（参见第 II-D 节）。由于 SCRATCH 项目远小于 Java 或 Python 项目，而且可能包含的重复 API 用法更少，因此 BUGRAM 并不直接适用。然而，在教育环境中，常见的情况是学生要完成一项任务，而这项任务有一个或多个标准解答。因此，我们在标准解答或已知正确的学生解答上训练一个 n元语法模型，然后评估学生解答中所有序列的概率，并报告概率特别低的序列。由于该模型并非在整个代码库（即所有学生解答）上训练，与 BUGRAM 方法不同，因此需要使用平滑算法；我们同样采用改进的 Kneser-Ney 平滑。此外，鉴于 SCRATCH 结构简单，我们跳过 BUGRAM 使用的任何预处理，例如在解析源代码时将完整路径纳入方法名称，或跳过 Wang 等人 [8] 提出的计数特别低的词元。因此，在结构上，我们对代码补全和错误查找任务使用相同的模型。

D. 使用 *Transformer* 进行代码补全

虽然 n 元模型简单、轻量且易于解释，但 Transformer 更为先进，通常能取得更好的结果 [25], [38]。不过，它们仅适用于拥有大量训练数据的任务；在我们的研究中，这类任务只有代码补全任务。

使用脚本分词方法（第 III-A 节），我们将每个角色的一个词元序列 $[t_0, \ldots, t_n]$ 生成为：先连接所有过程定义中的词元序列，再连接脚本中的词元序列。由于用户可以在画布上自由放置代码块，我们按照脚本在项目文件中的出现顺序对其进行分词。这通常反映了用户创建这些脚本的顺序。训练时，我们使用了 PYTHON*transformers* 库 [40] 提供的 RoBERTa [39] 实现。由于词元序列中已移除标识符，模型无需处理词汇表外的情况，词汇表也缩减为 137 个不同的词元。因此，我们使用一种简单的词级分词器，为每个词元分配一个数字标识符。

该模型采用默认的 RoBERTa 方法进行训练，使用掩码语言模型 [23]随机遮蔽词元 [39]。在此方法中，词元会被随机替换为占位符 ［MASK］，然后根据上下文预测原始词元：



使用训练好的模型进行代码补全时，会提取现有代码末尾最多 $m-1$ 个词元的序列，并追加额外的 ［MASK］ 词元。随后，模型会预测每个词元替换此掩码的概率，以便向用户展示排名前 x 的建议。与使用 n 元模型进行补全类似，按照第 III-B 节所述，程序定义会被排除；对于脚本末端的预测，则会替换为新脚本的建议。

IV. 评估

为了了解语言模型在基于代码块的编程语言中的适用性，我们以基线为参照，从两个相反的角度对 SCRATCH 程序中的 n 元模型进行实验考察：首先，我们通过代码补全任务，考察模型所编码的最高概率，以了解 n 元模型对 SCRATCH 程序规律的捕捉效果。其次，我们通过缺陷发现任务，考察模型所编码的最低概率，以了解 n 元模型检测常见模式偏差的效果。最后，我们研究能否利用深度学习模型改进预测。由此提出以下研究问题：

- **RQ1，代码补全：**基于 n 元模型的代码补全在 SCRATCH 源代码上的表现如何?
- **RQ2，缺陷发现：**基于 n 元模型的缺陷发现在 SCRATCH 源代码上的表现如何?
- **RQ3，模型比较：**基于Transformer的深度学习模型能否优于 n 元模型?

A. RQ1：使用 *n* 元语法模型进行代码补全

1）实验设置：为了构建用于代码补全的通用模型，我们在随机选取的 100 000 个 SCRATCH 程序上训练了一个 n 元语法模型。2021 年 5 月至 2022 年 2 月期间，我们每天使用 SCRATCH网站的 REST API 获取最近公开分享的 10 000 个 SCRATCH 程序，总计获得 270 万个项目。随后，我们筛除了代码块少于 10 个的项目，因为这类项目往往是孩子专注于艺术和绘画（例如绘制背景并在其上摆放角色），而非编写代码的项目。此外，我们还排除了改编作品（即复制并修改的程序）。从剩余的 110 万个项目中，我们随机抽取了 110 000 个，并将其划分为包含 100 000 个项目的训练集和包含 10 000 个项目的评估集。比较训练集和评估集中项目的代码块数量后发现，两者之间没有显著差异（$p = 0.191$ 使用曼-惠特尼U检验 [41]），这证实了这两个数据集来自相同的总体分布。

为了选择合适的序列长度 n，我们训练了模型 $n = \{1, 2, 3, 4\}$。我们将 4 设为上界，原因有二：首先，我们发现更大的 n带来的提升微乎其微，这与先前研究 [6], [30]的发现一致。其次，随着 n 增大，模型所需的内存和计算时间会显著增加。这一点在漏洞发现的应用场景中尤为重要：例如，为了识别异常序列，可以针对一组给定程序训练定制模型。对于发现编程作业中的漏洞而言，由于这类作业通常使用相对小巧、简单的 SCRATCH 程序，训练复杂模型可能并不值得。我们为每个 n 元语法模型使用相同的 100000 个随机选取的 SCRATCH 程序。

评估数据集中的项目被划分为多个局部上下文集合，每个集合由 $n-1$ 个代码块组成。我们将每个上下文输入代码补全引擎，并要求它预测该上下文的下一个代码块。代码补全引擎会按概率从高到低返回前 $x = \{1, 2, 3, 5, 10\}$ 个代码块，我们则通过前x 准确率来评估代码补全建议，即建议中包含原始程序实际代码块的比例。我们考察不同 x 和 n下的前x准确率，并评估代码块频率、类别和形状对前x 准确率的影响。

表 I：不同 n 值下代码补全在前 x 个建议中的准确率。

	前 1 个	前 2 个	前 3 个	前 5 个	前 10 名
一元语法	6.24 %	12.27 %	17.17 %	25.88 %	44.38 %
二元语法	23.22 %	34.56 %	42.54 %	54.03 %	69.44 %
三元语法	31.41 %	43.87 %	52.08 %	62.82 %	76.07 %
四元语法	36.31 %	49.05 %	56.77 %	66.76 %	78.35 %
Transformer	33.83 %	43.78 %	49.91 %	57.79 %	69.32 %

2) 有效性威胁：外部效度威胁源于结果可能无法推广到数据集之外的项目。我们确认，就规模而言，该样本能够有效代表公开共享项目；不过，尚未完成或未共享的程序可能具有其他属性。为避免非常相似的代码使结果产生偏差，我们只使用原始项目，并排除了改编作品。内部效度威胁可能源于我们的实现：尽管我们对所有代码进行了全面测试和验证，但我们的实现可能会混淆所研究的测量指标和关系。例如，测试期间未遇到的 SCRATCH 程序表示中的罕见情况可能会被错误表示。构念效度威胁可能源于我们选择以 top-x准确率而非精确率或召回率作为指标。这一选择基于具体用例：我们假设 top-x 中的每条建议同样有用。实际上，部署的代码补全引擎可以抑制低置信度预测，而且用户不必接受错误的建议。

3) 结果：表 I 展示了针对 $n = \{1, 2, 3, 4\}$ 的代码补全 top-x 准确率。top-x 准确率定义为每个代码块的所有真正例预测之和除以预测总数。如果 top-x 建议集合包含待预测的实际代码块，则该预测被视为真正例。根据定义，top-x 准确率会随着 x 的增加而单调上升。我们还观察到，随着 n的增加，准确率也会提高。由于四元语法表现最佳，因此在 RQ1 的其余部分中我们使用四元语法。此外，在后续评估中，我们使用前三准确率，因为 3 是 SCRATCH 用户界面中建议数量的合理值（例如代码片段的“背包”中），符合 SCRATCH 降低认知负荷的设计理念 [10]。

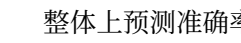
整体上预测准确率最高的代码块前三准确率为 95.52 %。它的出现率为 6.03 %，是评估数据中出现频率第二高的代码块。因此，出现频率很可能会影响预测性能。我们注意到，其他类别中排名最高的代码块，其前三准确率明显较低，这表明类别似乎也会产生影响。最后，我们观察到，通常椭圆形和菱形代码块的准确率都

表 I：n=4、x=3 各类别的完成准确率。

组	出现次数	准确率	准确率变换器
声音	2.5 %	32.9 %	44.7 %
pen	0.9 %	34.0 %	25.1 %
myblocks	0.9 %	41.3 %	60.6 %
事件	15.3 %	48.0 %	30.4 %
运动	12.2 %	51.5 %	48.9 %
外观	20.7 %	53.9 %	64.2 %
控制	21.2 %	58.8 %	31.0 %
算子	9.6 %	66.3 %	53.2 %
data	11.4 %	68.0 %	66.3 %
侦测	5.3 %	73.6 %	75.9 %

表 III：n=4、x=3 时按形状划分的完成准确率。

形状	出现次数	准确率	准确率变换器
end	1.3 %	29.9 %	58.9 %
hat	13.7 %	47.6 %	16.9 %
堆叠	52.0 %	51.8 %	56.3 %
c	12.2 %	59.9 %	14.0%
oval	11.5%	74.9%	67.6%
菱形	9.4%	75.2 %	70.7 %

特别高，这表明代码块的形状也会影响预测。为了更好地了解哪些因素决定了整体预测性能，我们因此研究以下三个方面的影响：（1）代码块在实际应用中的出现频率；（2）代码块所属的类别（例如运动、外观等）；以及（3）代码块的形状（例如常规的可堆叠代码块、事件处理程序代码块等）。

图 3a 总结了这三个方面及其对 $n = 4$ 前三准确率的影响：图表按 SCRATCH 用户界面中代码块排序所在的 10 个不同代码块类别划分为多个子图。对于每个类别，根据代码块在训练数据中的出现次数（x 轴）及相应的前三准确率（y 轴）绘制代码块。数据点根据其形状用不同颜色标示。具体而言，帽子积木 表示事件处理程序，堆叠形程序块 是常规语句，椭圆形 代码块表示返回数值或文本数据的报告器，菱形 代码块表示布尔值，c形代码块是循环和条件判断等控制结构，而停止 代码块是没有后继程序块的语句积木。“myblocks”类别只包含一个代码块，因为我们会泛化标识符（见第 III-B 节），并且只预测对这些自定义程序块的调用，而不预测它们的创建。因此，过程定义代码块及其可能参数的数据点不在其中。为了更好地理解代码块频率、类别和形状对准确率的影响，我们使用多元线性回归对这种关系进行建模。我们将出现次数作为连续变量，将类别和形状作为分类变量纳入模型。由于我们关注的是特定类别和/或形状的准确率在不考虑出现次数的情况下是否显著偏离平均准确率，因此我们采用偏差编码，将各个水平与总均值进行比较。图 3b 展示了回归分析的结果。

表 IV：图 2a 中示例的前 3 个预测。

代码块	置信度
var	80.31 %
造型编号	5.67 %
答案	5.44 %

图 3a 显示不同类别之间存在差异，表 II 对此进行了总结：侦测、数据、算子和控制类别的代码块，其预测准确率显著高于声音或画笔类别的代码块。例如，造成这种差异的原因之一是出现频率不同。与运动或外观类别的代码块相比，画笔或声音类别的代码块出现频率要低得多。出现次数的重要性不仅体现在不同类别之间，也体现在类别内部。图 3a 中的拟合线非常清楚地表明，在所有类别和形状中，出现次数对准确率都有正向影响：代码块在实际使用中出现得越频繁，被正确预测的概率就越高。例如，事件类别中预测效果最佳的代码块（参见表 II） 的出现次数和准确率都很高，这证实了这一点。图 3b 证实，出现次数对预测准确率有显著影响，不过回归模型的系数较小，表明这种影响并不大。

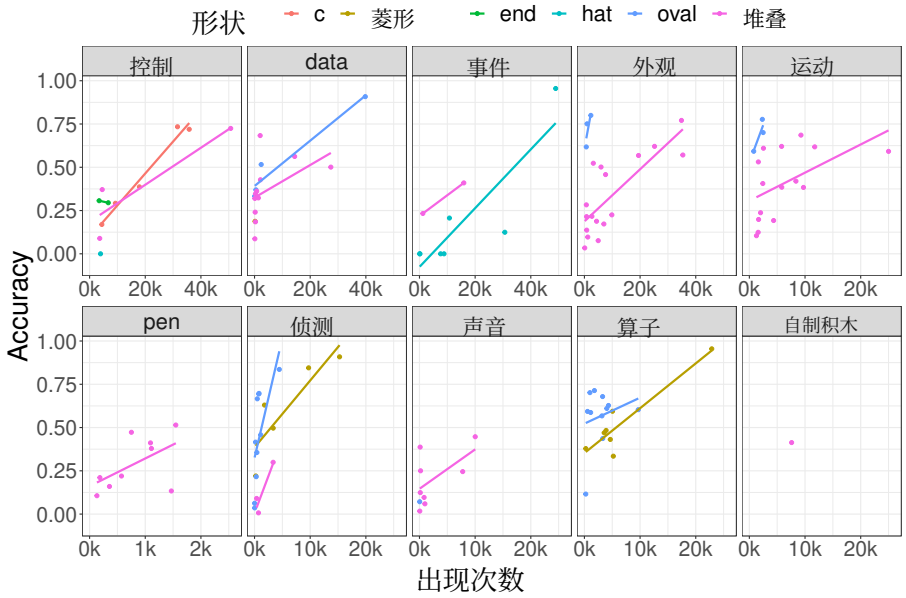


然而，并非所有类别之间的差异都能通过出现次数来解释。例如，尽管侦测和算子类别中的代码块总体上比运动代码块等出现得少，但这两个类别仍然能够得到较好的预测（表 II）。图 3b 证实了类别会产生一定影响；尤其是运动类别会显著影响预测准确率。不过，我们观察到，易于预测的侦测和算子类别在一个重要属性上与其他类别不同：它们都包含特别高比例的菱形积木和椭圆形积木，这些积木表示表达式而非语句（算子类别中表达式积木占 100%，侦测类别中占 83%）。例如，画笔和声音代码块几乎全是常规的堆叠形程序块。图 3a 通过不同颜色和拟合线总体表明，与椭圆形代码块（参见数据、外观、运动和侦测类别）或菱形积木（参见侦测类别）相比，堆叠形程序块似乎更难预测。表 III 使用按代码块形状分组的前三准确率值支持了这一观察。图 3b 证实，椭圆形和菱形形状会显著提升预测效果，而帽子积木则会显著降低预测效果。

只要局部上下文中有一个通常后接表达式积木的代码块，实际可选的匹配代码块就会大幅减少。图 2a 中的代码示例展示了这一现象：这里只能放入菱形表达式积木。圆形表达式积木应放入 代码块的圆形占位符中，这会减少后继积木。顶部代码块





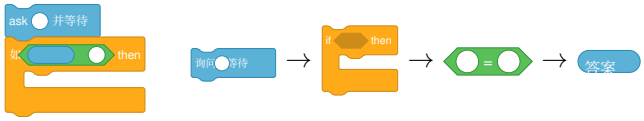


(a) 按代码块的类别和形状分组，前三准确率与代码块出现次数的关系。（请注意，x轴的刻度不同。）

变量	系数	p
截距	2.94×10^{-1}	< 0.001
出现次数	1.49×10^{-5}	< 0.001
类别		
控制	-1.05×10^{-1}	0.216
data	9.81×10^{-3}	0.834
事件	-4.62×10^{-2}	0.634
外观	5.69×10^{-3}	0.897
运动	1.08×10^{-1}	0.019
pen	5.72×10^{-2}	0.342
侦测	-2.68×10^{-2}	0.622
声音	-9.49×10^{-2}	0.108
算子	1.28×10^{-2}	0.831
自制积木	7.96×10^{-2}	0.612
形状		
c	-2.34×10^{-2}	0.797
菱形	1.55×10^{-1}	0.034
end	3.62×10^{-2}	0.763
hat	-2.86×10^{-1}	0.003
oval	1.92×10^{-1}	0.002
堆叠	-7.35×10^{-2}	0.146

(b) 一般线性模型。

图3：块出现、类别和形状对代码补全准确率的影响。



（a）处理用户输入的代码（左）及其对应的四元语法（右）。该四元语法的概率为 97.56%。



(b) 将角色隐藏的独立脚本，在背景切换时不包含任何后续代码块。四元语法的概率为 95.48%。

图 4：模型捕捉到的常见语言习惯表达。

模型为该场景给出的建议如表 IV 所示。模型有 80.58 % 的把握认为应在 代码块中插入一个 。与变量积木相比， 和 的概率相对较低，分别为 5.69 % 和 5.29 %。虽然 是该场景下合理的建议，但只有在前面有一个 代码块时 代码块才有用，因为它可以引用该代码块 示例见图 4a）；然而，图 2a 中并没有这样的代码块。尽管如此，这 3 个建议都有一个共同点：它们的形状使其成为语法正确的构建块。这一现象与先前的观察相似，即语法元素对源代码的可预测性有显著贡献 [11]。因此，对于画笔、声音或运动等语法约束不多的类别，补全效果较差。

最后，常见 SCRATCH 代码中反复出现的惯用表达也体现出了一定的规律性。考察模型赋予极高概率的 n 元语法，并排除以 代码块开头等已知模式后，我们发现模型学会了一些超越纯句法规律的惯用表达。与传统编程语言类似，某些词元序列会跨越特定程序的边界反复出现。例如，图 4a 中的四元语法概率为 97.56 %，描述了请求用户输入并根据比较结果作出反应的过程。虽然这一序列的出现频率远低于以 开头的程序（7 465 次，而后者出现了 499 146 次），但对于这一局部上下文，模型非常确 作为最后一个代码块的是 。出现次数、类别和形状只能部分解释这一点：尽管 是一个椭圆积木，在句法上十分契合，但 SCRATCH 中还有许多其他椭圆形代码块。然而，对于所有其他形状的代码块而言，在这一局部上下文之后出现的概率总计不到 2.5%。

我们观察到的其他惯用模式包括将功能隔离在非常短的脚本中（参见图 4b），以及重复使用相同的代码块（例如，将元素插入列表）。这些惯用模式表明代码块在语义层面存在联系，因此除了纯粹的语法连接（即形状和代码块频率，也就是出现次数）之外，还存在更多重复且可预测的结构。这些惯用模式与在传统编程语言中发现的模式相当。

总结（RQ1，补全）： 最佳模型（四元语法）的前三准确率为 56.77%。代码块频率、形状和类别都会影响预测质量，但我们还发现，反复出现的惯用模式也会影响规律性。

B. RQ2：使用 *n* 元模型进行缺陷发现

对于 RQ2，我们以与 RQ1 相反的方式使用 n元语法模型：我们关注的不是预测最可能出现的代码块，而是最不可能出现的代码块序列。

与 BUGRAM 论文中的评估 [8],不同，我们假设，对于规模较小的学生解答，可以通过测试识别程序中的所有实际错误。因此，在我们的论文中，测试套件充当真实情况，用于评估最不可能出现的序列是否确实包含错误代码。在原论文中，作者在大型项目中寻找未检测到的 漏洞和重构可能性。他们没有将代码与真实情况（例如现有测试）进行比较，而是通过人工检查寻找改进可能性。

1）实验设置：错误查找任务的应用场景是教育环境中的一项编程作业。我们使用了 WHISKER [19] 测试 SCRATCH 程序论文复现包中提供的数据集，其中包含 41 份学生解答和一份接水果游戏的标准解答（参见图 5a）。游戏的目标是在 30 秒内，通过移动屏幕底部的碗，尽可能多地接住苹果和香蕉。香蕉碰到地面时，玩家会失分。如果苹果掉到地面上，游戏就会失败。

我们在标准解答和一份经自动化测试判定为几乎正确的学生解答上，使用改进的 Kneser-Ney 平滑训练了一个三元语法模型 [19]。我们采用了 $n = 3$ ，这是基于 Wang 等人的先前结果；他们发现，三元语法模型在发现漏洞和重构机会方面表现最佳 [8]。

我们使用该模型，确定了 41 个学生程序中所有出现序列的概率（也就是说，包括最佳学生解答）。直观而言，模型赋予概率较低的序列更可能包含漏洞。提取序列时，我们排除了“松散”代码，即未连接到事件处理程序、因而永远不会执行的代码块和脚本。由于此前尚未在我们的环境中研究过此类分析的理想序列长度，我们评估了序列长度为 3 至 6 的情况。对于通常规模较小的 SCRATCH 程序而言，较长的序列长度不太可能有用，因为这类序列往往会超出脚本边界。

为探究低概率序列是否意味着存在漏洞，我们从 41 个学生解答中随机选取了 10 个进行人工验证，仅考虑至少包含 10 个序列的解答，因为其他解答不太可能完整实现任何功能。对于这 10 个程序，我们针对长度为 3 到 6 的每种序列长度，人工分类概率最低的 10 个序列。两位作者独立评估相应序列是否包含漏洞。我们使用 Stahlbauer 等人提供的详尽 WHISKER 测试套件作为判断漏洞是否存在的真实情况，该套件完整覆盖程序行为。我们将导致一个或多个测试用例失败或被跳过的序列视为包含漏洞。因此，对于每个序列，我们确定了 (1) 该序列是否至少包含一个漏洞；(2) 对于每个失败的测试用例，该序列是否导致了测试失败。若两位作者的独立分类存在分歧，则重新讨论这些个案，直至达成共识。因此，总共对 400 个序列进行了人工分类。

表 V：各序列长度下发现的漏洞百分比。

序列长度	3	4	5	6
发现的漏洞（%）	57.58	62.88	64.39	56.06

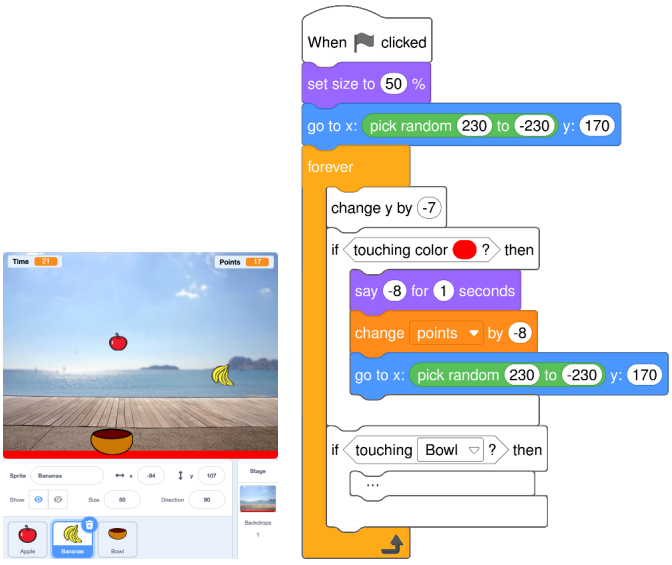
表 VI：对于长度为 4 的序列，列出最低排名的 10 个最不可能序列和 10 个随机序列，给出每个序列至少包含一个漏洞的 Precision@10、发现的漏洞占漏洞总数的比例，以及程序中的漏洞总数。

	Precision@10		发现的漏洞百分比		Bugs
	最低排名	随机	最低排名	随机	总计
K6 S01	90.0	40.0	96.3	37.04	27.0
K6 S12	60.0	90.0	80.8	26.9	26.0
K6 S15	60.0	30.0	62.5	31.3	16.0
K6 S18	60.0	50.0	50.0	50.0	6.0
K6 S31	30.0	10.0	75.0	12.5	8.0
K7 S03	60.0	40.0	27.3	45.5	11.0
K7 S10	20.0	0.0	28.6	0.0	7.0
K7 S14	30.0	20.0	60.0	20.0	5.0
K7 S17	10.0	10.0	50.0	50.0	2.0
K7 S24	70.0	50.0	33.33	20.8	24.0
平均值	49.0	34.0	62.88	28.79	13.2
p	0.073		0.003		
$\hat{A}_{12}$	0.69		0.84		

作为基线，我们还使用根据低概率序列分类所确定的最佳序列长度，为每个程序随机选取并分类了 10 个随机序列，以便确定最不可能的 10 个序列是否比随机序列更可能是漏洞。

2）有效性威胁：由于我们的实验基于一项任务以及两个班级学生的解答，因此可能存在外部效度威胁，结果未必能推广到其他任务或班级。为避免内部效度威胁，我们随机抽取项目，以避免偏差。人工分类可能受到主观解读的影响，因此本文的两位作者分别独立对每个序列进行分类，以尽量降低其对结果的影响（评定者间信度：88.8 %）。此外，两位作者对所有序列进行了分类，以确保结果彼此一致且具有可比性。为确保评估的构念效度，我们采用公认的缺陷发现衡量指标，同时考虑存在漏洞的序列数量以及独特漏洞的数量。

3) 结果：表 V 列出了不同序列长度下发现的漏洞总体百分比（b = #发现的漏洞/#漏洞总数）。长度为 5 的序列总共发现的漏洞最多，长度为 4 的序列紧随其后；长度为 6 的序列发现的漏洞最少。长度为 4 和 5 的序列之间的细微差异仅源于一个程序：对该程序而言，长度为 5 的序列发现的漏洞显著更多（K7 S03：27.27 % 对 90.91 %）。在所有其他程序中，长度为 4 的序列结果相同或更好。由于长度为 4 的序列比长度为 5 的序列更能缩小问题/漏洞来源的范围，因此后续所有结果均基于长度为 4 个词元的序列。



(a) 接水果游戏。

(b) 香蕉角色的学生解法 K7 S10。彩色代码块是所报告序列的示例，这些序列会导致测试失败。

图5：包含有缺陷学生解法的接水果游戏示例。

表 VI 列出了基于概率最低的序列和随机序列得出的各程序结果。Precision@10 根据最低排名的 10 个序列中至少包含一个漏洞的序列数量给出，即最低排名的 10 个序列包含实际缺陷的概率。该表还展示了每个程序中发现的所有漏洞所占比例。由于程序相对较小，一个序列中往往包含多个漏洞；作为参考，表中还列出了漏洞总数，该数量对应失败测试的数量。

就有缺陷序列的百分比而言，平均来看，最不可能的序列中有一半至少包含一个漏洞，而随机选取的序列中这一比例仅为 34 %。在 $\alpha = 0.05 (p = 0.073)$下，这一差异不具有统计显著性；但请注意，由于这些序列是随机选取的，因此与最不可能的 10 个序列存在一定重叠：平均有 16% 的随机选取序列也属于最低排名的 10 个序列。然而，长度为 4 的随机选取序列总共只能发现 29 % 的漏洞，并且在任何程序中发现的漏洞都不超过 50 %。相比之下，最不可能的序列平均能发现总计 63 % 的漏洞。与随机序列相比，最不可能序列的改进具有统计显著性，且 Vargha-Delaney 效应量较大，ˆ 为 $A_{12} = 0.84$；使用曼-惠特尼U检验得到 $p = 0.003$。这证实了 n元语法模型确实捕捉到了预期的解法结构。

例如，图 5b 展示了香蕉角色的学生解答 K7 S10。源代码中着色的部分是两个长度为 3 的序列，模型认为它们出现的可能性最低，位列倒数 10 名。在这两个例子中，代码实际上都会导致测试用例失败：对于第一个序列，该解答违反了任务规范，即香蕉应在一

秒后 旗帜被点击。可是，程序立即开始执行移动香蕉的 for循环。对于第二个代码序列，扣分作为时间惩罚后，香蕉应当消失一秒，但有缺陷的实现会导致香蕉立即再次从顶部开始下落。

我们通常观察到，训练数据中未包含的代码块会强烈影响一个序列的概率。具体而言，这类序列不仅会被赋予特别低的概率，而且同一个异常代码块往往会影响其周围多个序列的概率。因此，在最坏情况下，概率最低的 10 个序列可能只覆盖程序的一小部分。例如，解法 K6 S01 通过在游戏开始时（即每个角色中的每个脚本）按下向上箭头键而不是绿旗，违反了任务规范。模型在训练数据中从未见过这个代码块。因此，模型报告的概率最低的 10 个序列中，有 9 个包含未知的 代码块。我们还注意到，不常见但正确的代码 同的情况。

最初的 BUGRAM 论文建议过滤稀有词元 [8]。由于 SCRATCH 的词汇表要小得多，我们决定不采用这种方法；然而，实验结果表明，在实际实施这种方法时，这可能是一项有用的补充，不过其中的一个挑战是不能丢弃规模本就相当小的学生解答中的过大部分，否则也可能导致无法检测到潜在漏洞。

程序 K7 S17 是整体表现最佳的学生解答，它已被纳入模型的训练数据。该程序中的两个漏洞之一被识别出来这一事实（表 VI）表明，与 BUGRAM [8] 方法类似，即使是用作训练数据的程序也能发现错误。

总结（RQ2，缺陷发现）： 在 10 个程序中，有 9 个程序使用该模型发现的漏洞数量大于或等于通过随机选择的序列发现的数量；这一提升具有统计显著性。

C. RQ3，对比：使用 *Transformer* 进行代码补全

1) 实验设置： 我们使用通过与第 IV-A1 节所述相同流程获取的数据集训练了 RoBERTa 模型 [39]，该数据集通过抽样获得，包含 517 431 个角色的 500 000 个程序。为将序列限制在模型可处理的最大长度 m 以内，我们将其拆分为子序列：对于包含 n 个词元的程序，第一个生成的序列为$[t_0, \ldots, t_{\min(n,m)}]$。随后，将未完全包含在该序列中的第一个脚本作为下一个序列的起始点。为了在嵌入脚本时尽可能利用上下文，将该脚本前后的词元对称地添加进去，直到序列长度达到 m。重复此过程，直至所有词元都至少包含在一个子序列中。不在脚本层级拆分序列，也使得模型能够预测何时应结束一个脚本并开始一个新脚本。序列拆分后，训练集共包含 4 493833 个序列。评估使用了与 RQ1 相同的 10 000 个程序。

我们以 RoBERTa 的默认超参数作为进一步调优的起点 [39]。由于待建模的语言规模较小，程序通常也较短，因此调优时将最大序列长度降至 256，并减少隐藏层数量（从 12 层降至 2 层）、隐藏层维度（隐藏维度为 256，中间层维度为 512）以及注意力头数量（从 12 个降至 4 个）。其他参数保持不变。

2) 有效性威胁： 适用的有效性威胁与第 IV-A2 节所述相同。将序列拆分为更短的子序列会带来额外的构念效度威胁。为降低这一风险，我们比较了两种拆分方式的结果：一种是每个脚本对应一个填充后的子序列，另一种是使用长度等于模型序列长度、彼此不重叠的词元块。我们选择了第 III-D 节所述的策略，因为其他策略并未提高预测准确率。使用 RoBERTa 模型 [39]获得的结果未必能推广至其他基于 Transformer 的模型。这符合我们的目标：不追求最大化某一特定模型性能指标，而是提供一个基线，并证明有必要将不同模型纳入未来的研究。

3) 结果： 表 I 显示了代码补全的前 x 个预测准确率。前三个建议词元的准确率与三元语法模型相近，也就是说，与 RQ1 的结果一致，该模型是表现第二好的 n 元语法模型。

表 II 和表 III 更详细地展示了哪些类型的代码块能够被准确预测。对于结束代码块（ 和 ）的预测，Transformer 优于 4 元语法模型（即表现最佳的 n元语法模型）；对于常规语句（堆叠）、布尔表达式（菱形）和占位符（椭圆形），两者表现相近。对于脚本起始块（帽子）和分支代码块（c），其准确率则低于 n元语法模型。

能够预测脚本的结束，却难以预测新脚本的起始，这一差异值得注意。SCRATCH 中的脚本不一定要使用“end”代码块作为终止语句。因此，在大多数情况下，并没有明确的标志表明何时应开始一个新脚本。在“帽子”组的代码块中，准确率最高的 （35.3 %）、 （25.3 %）和 （22.5 %）。在这些情况下，很可能是用于更改背景或发送消息的相应语句被放在了前一个脚本的末尾，因此可以将其视为新脚本即将开始的提示。请注意，“end”代码块不一定放在角色序列的末尾。相反，序列中的脚本按照用户创建时的顺序保存。因此，模型无法利用“end”代码块之后的词元数量来学习脚本是否应该结束。

对于分支代码块（C 形代码块），Transformer 的准确率（14.0%）显著低于 n元语法模型（59.5%）。这可能是由训练期间采用的双向注意力造成的：模型利用被遮蔽 C 形代码块两侧的周围上下文，学会了除无限循环之外，这些代码块在大多数情况下后面都会跟着一个布尔条件词元。由于在代码补全任务中缺少这一信息，因此没有足够的上下文来可靠地预测正确的词元。然而，修改注意力机制，使其仅允许关注位于被遮蔽词元之前的词元，反而导致准确率更低。

总体而言，Transformer 的预测准确率不如表现最好的 n 元语法模型。Transformer 可以处理规模达数万个不同词语的大词汇表 [23]，但这一优势几乎派不上用场，因为词元化的 SCRATCH 语言只有 137 个词。此外，长程信息对下一词元预测的作用尚不明确。脚本按照用户创建程序时插入其中第一个代码块的顺序排列；除相互传递的消息外，它们在程序流程中彼此独立。例如，在同一角色的上下文中，有些脚本可能负责处理用户输入及由此产生的移动，而另一些脚本则会在与其他角色交互时触发，以播放声音或改变外观。因此，我们预计下一个词元主要取决于短程局部上下文，而其他脚本中无关的代码块会起到干扰作用。

总结（RQ3，模型比较）： Transformer 模型的表现与三元语法模型相当，因此不如我们表现最好的 n 元语法模型（$n = 4$）。我们推测，这是由较小的词汇表以及在代码补全中短程信息比长程信息更重要所导致的。

V. 讨论

近期的程序分析经常基于“自然假设” [6]，，该假设认为编程语言具有与自然语言相似的规律性，因此源代码适用于自然语言处理技术。研究表明，源代码中观察到的这种规律性有一定程度源于基于文本的编程语言所具有的语法开销。具体而言，Rahman 等人发现，过滤掉分隔符或嵌套词元等常见语法词元后，规律性会降低 [11]。我们的研究与此相关，因为基于代码块的编程语言明确避免使用这类语法词元，因此也可能使语言的重复性和可预测性降低。

与 Java 相比，SCRATCH 的代码补全表现似乎稍差。例如，在一项相关研究中， [29] 针对 Java 使用三元语法模型时，top-1 准确率比 SCRATCH 中四元语法模型对应的 top-1 准确率高出近 20%。该研究将源代码建模为词法词元流，其中包括由编程语言 [29]指定的标识符、关键字或符号。此外，该研究评估了一项补全任务，使用局部上下文（如 <if、(、node> ）以及补全建议，例如“!= null”、“== null”和“.isRoot()” [29]。该任务的设计与我们的补全任务相当。因此，我们假设，在编程语言边界允许的范围内，两者的结果具有可比性。因此，预测准确率相差近 20%，表明基于代码块的编程语言与基于文本的编程语言在属性方面，或其程序的特征方面，存在显著差异。

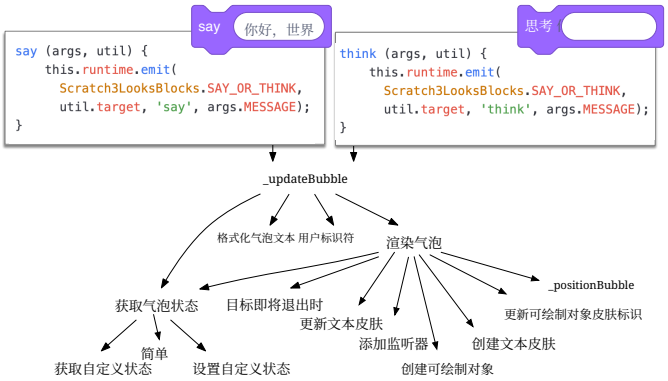


图6：两个不同的 SCRATCH 代码块及其在 JavaScript 中共享的实现和调用树。

根据自然语言处理和软件工程领域近期的发展趋势，人们可能会预期深度学习模型会在此发挥作用，但我们的结果表明事实并非如此（参见第IV-C3节）。在 Ciniselli 等人针对 Java 代码进行的一项类似实验中，Transformer模型的预测准确率优于他们表现最好的 n 元语法模型 [25]。这表明，在基于代码块的编程语言与基于文本的编程语言中，模型利用长程信息的方式有所不同。为缓解这一问题，可以尝试将更多信息融入深度学习模型 [38], [42]。例如，将扁平序列作为 Transformer 的输入，会丢失代码中的所有结构信息。Guo 等人 [42] 调整了其 GraphCodeBERT 模型的注意力机制，使其聚焦于由数据流图确定的相关代码元素。同样，对于 SCRATCH，可以提取有关脚本之间关系的信息（例如，传递的消息、发生变化的角色属性） [16] ，帮助 Transformer 聚焦于与待补全脚本相关的脚本，并忽略其他脚本。这或许能提升模型从包含多个短脚本的长序列中捕获长程信息的能力。

作为RQ1和RQ3（第IV-A节和第IV-C节）的一部分，我们讨论了基于代码块的程序及其对可预测性影响的若干方面。除这些方面之外，传统的基于文本的编程语言与基于代码块的编程语言之间还有一个可能影响重复性和可预测性的重要差异：源代码所定义的抽象层次。图6展示了由 或 代码块执行的实际JavaScript代码，以及SCRATCH虚拟机代码中相应回调函数的调用树。一方面，该示例展示了与传统编程语言相比，SCRATCH代码的抽象程度有多高。不仅去除了语法开销，在某种程度上代码也得到了精简。另一方面，这会导致JavaScript代码中出现重复，因此可能被理解为引入了代码规律性：在JavaScript实现中， 和 代码块都会创建、格式化并渲染一个对话气泡，而对_updateBubble的调用对于这两个代码块而言本质上是相同的。然而，在SCRATCH代码中，这两个代码块彼此独立，不存在任何重复性。因此，JavaScript代码更加重复，也更具可预测性。

VI. 结论

软件工程研究近期的一个趋势，是将语言模型和自然语言处理技术应用于基于文本的编程语言，以完成各种不同的任务。迄今为止，这一趋势尚未涵盖一个较为小众的编程语言领域，即基于块的语言；它们与基于文本的语言在某些特性上有所不同，而正是这些特性使自然语言处理技术能够应用于源代码。为了探究语言模型对基于块的程序的适用性，我们对 SCRATCH 的 n-gram 模型和 Transformer 模型进行了实证研究。尽管结果表明，基于块的语言更难预测，但也证明了这种方法切实可行。

关于代码补全的先前研究提出了多种可能改进我们基线模型的方法。例如，其他模型（如统计图模型 [7], [11] 或神经模型（例如， [43], [44]）） 有可能提升性能。进一步筛选词汇表，例如过滤掉罕见代码块， [8]，或许也能提升性能 [34], [35]。此外，纳入更多上下文信息也可能改善性能 [45]；例如，对于 SCRATCH，可以利用正在编辑的角色或舞台提供上下文。类似地，也可以针对不同类型的程序构建模型；例如，SCRATCH 中的游戏可能与动画或艺术项目有着根本的差异，因此可以采用不同的模型。

除了模型的性能之外，未来工作还需要关注这些模型的应用。例如，与基于文本的编程不同，代码块没有可显示代码补全的文本光标，因此带来了可用性方面的挑战。尽管如此，将代码补全系统整合到 SCRATCH 用户界面中应该是可行的，因为代码块已经按照类型分组显示。修改界面后，可以将推荐的代码块作为一个组展示。不过，教育应用领域也可能意味着需要定制模型，将教育水平纳入考量；例如，代码补全不应推荐学习者尚未掌握相关概念的代码块，而这可以通过 SCRATCH 的计算思维指标来确定 [46]。此外，除了最初的错误查找任务之外，我们还设想了语言模型在教育领域中的许多潜在应用。

为支持研究复现和未来工作，分词器与语言模型的源代码、数据集、分析脚本以及原始结果可在以下位置找到

https://doi.org/10.6084/m9.figshare.19382588.v1

致谢

本研究获得BMWK（50RM2100B，“ANUKI”）、巴伐利亚研究基金会（AZ-1520-21，“DeepCode”）以及德国研究联合会（FR2955/4-1，“STUNT”）的支持。

参考文献

[1] M. M. McGill 和 A. Decker，《中小学计算机教育中使用的工具、语言和环境》，载于《2020年ACM计算机科学教育创新与技术会议论文集》，ITiCSE '20系列。美国纽约州纽约市：计算机协会，2020年，第103–109页。 [在线]。可访问：https://doi.org/ 10 1145 3341525 3387365 。/。. D. Weintrop 、 D. C. Shepherd 、 P. Francis 和 D. Franklin，《 Blockly 投入工作：用于工业机器人的基于代码块的编程》，载于《2017年IEEE Blocks and Beyond研讨会（B&B）》。IEEE，2017年，第29–36页。[3] N. Ritschel、V. Kovalenko、R. Holmes、R. Garcia 和 D. C. Shepherd，《双臂机器人基于代码块的编程模型比较》，《IEEE软件工程汇刊》，2020年。[4] C. Mayr-Dorn、M. Winterer、C. Salomon、D. Hohensinger 和 R. Ramler，《使用基于代码块的语言进行工业机器人编程的考量——案例研究》，载于《第三届IEEE/ACM机器人软件工程国际研讨会》，RoSE@ICSE 2021，西班牙马德里，2021年6月2日。IEEE，2021年，第5–12页。 [在线]。可访问：https://doi.org/ 10 1109 52553 2021 00008 。/RoSE。。J. Maloney、M. Resnick、N. Rusk、B. Silverman 和 E. Eastmond，《Scratch编程语言与环境》，《ACM计算教育汇刊》，第10卷，第4期，第16:1–16:15页，2010年。 [在线]。可访问：https://doi.org/ 10 1145 1868358 1868363 。/。. A. Hindle、E. T. Barr、Z. Su、M. Gabel 和 P. T. Devanbu，《软件的自然性》，载于《第34届国际软件工程会议》，ICSE 2012，2012年6月2日至9日，瑞士苏黎世，M. Glinz、G. C. Murphy 和 M. Pezz`e，编。IEEE计算机学会，2012年，第 837 847 [ ] 10 1109 2012 6227135 –页。在线。可访问：https://doi.org/。/ICSE。。V. Raychev、M. T. Vechev 和 E. Yahav，《使用统计语言模型进行代码补全》，载于ACM SIGPLAN编程语言设计与实现会议，PLDI '14，英国爱丁堡，2014年6月9日至11日，M. F. P. O'Boyle 和 K. Pingali，编。ACM，2014年，第419–428页。 [在线]。可访问：https://doi.org/ 10 1145 2594291 2594321 。/。. S. Wang、D. Chollak、D. Movshovitz-Attias 和 L. Tan，《BUGRAM：使用n元语言模型进行错误检测》，载于《第31届IEEE/ACM自动化软件工程国际会议论文集》，ASE 2016，新加坡，2016年9月3日至7日，D. Lo、S. Apel 和 S. Khurshid，编。ACM，2016年，第708–719页。 [在线]。可访问：https://doi.org/ 10 1145 2970276 2970341 。/。. B. Ray、V. Hellendoorn、S. Godhane、Z. Tu、A. Bacchelli 和 P. Devanbu，《含缺陷代码的"自然性"》，载于《2016年IEEE/ACM第38届国际软件工程会议（ICSE）》。IEEE，2016年，第428–439页。[10] D. Bau、J. Gray、C. Kelleher、J. Sheldon 和 F. A. Turbak，《可学习编程：代码块及其他》，《ACM通讯》，第60卷，第6期，第 72 80, 2017 [ ] 10 1145 3015455 –页。在线。可访问：https://doi.org/。/ M. Rahman、D. Palani 和 P. C. Rigby，《重新审视自然软件》，载于《2019年IEEE/ACM第41届国际软件工程会议（ICSE）》，2019年，第37–48页。[12] F. Hermans、K. T. Stolee 和 D. Hoepelman，《基于代码块的编程语言中的代码异味》，载于《2016年IEEE可视化语言与以人为中心的计算研讨会（VL/HCC）》。IEEE，2016年，第68–72页。[在线]。可访问：http://ieeexplore.ieee.org/document/7739666/ [13] F. Hermans 和 E. Aivaloglou，《代码异味会妨碍初学者编程吗？关于Scratch程序的对照实验》，载于《2016年IEEE第24届国际程序理解会议（ICPC）》，2016年5月，第1–10页。[14] P. Techapalokul 和 E. Tilevich，《理解基于代码块的软件中反复出现的质量问题及其对代码共享的影响》，载于《2017年IEEE可视化语言与以人为中心的计算研讨会（VL/HCC）》。IEEE，2017年，第43–51页。 [在线]。可访问：http://ieeexplore.ieee.org/document/8103449/ [15] C. Fr¨adrich、F. Obermˆuller、N. K¨orber、U. Heuer 和 G. Fraser，《Scratch程序中的常见错误》，载于《2020年ACM计算机科学教育创新与技术会议论文集》，ITiCSE '20系列。美国纽约州纽约市：计算机协会，2020年，第89–95页。 [在线]。可访问：https://doi.org/ 10 1145 3341525 3387389 。/。. G. Fraser、U. Heuer、N. Korber、F. Obermˆuller 和 E. Wasmeier，《LITTERBOX：用于Scratch程序的静态检查工具》，载于《2021年IEEE/ACM第43届国际软件工程会议：软件工程教育与培训（ICSE-SEET）》，2021年，第183–188页。

[17] P. Techapalokul 和 E. Tilevich，《Quality Hound——面向Scratch程序的在线代码异味分析器》，载于*2017*年*IEEE*可视化语言与以人为中心的计算研讨会（*VL/HCC*），2017年10月，第337–338页。[18] D. E. Johnson，《ITCH：针对Scratch作业的计算机作业个性化测试》，载于第*47*届*ACM*计算机科学教育技术研讨会论文集。ACM，2016年，第223–227页。[19] A. Stahlbauer、M. Kreis 和 G. Fraser，《自动测试Scratch程序》，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，2019年，第165–175页。[20] S. F. Chen 和 J. Goodman，《语言建模平滑技术的实证研究》，《计算机语音与语言》，第13卷，第4期，第359– 393, 1999 [ ] 10 1006 1999 0128 页。在线。可访问：https://doi.org/。/csla.。C. Chelba、T. Mikolov、M. Schuster、Q. Ge、T. Brants、P. Koehn 和 T. Robinson，《用于衡量统计语言建模进展的十亿词基准》，载于*INTERSPEECH 2014*，国际语音通信协会第*15*届年会，新加坡，*2014*年*9*月*14*日至*18*日，H. Li、H. M. Meng、B. Ma、E. Chng 和 L. Xie 编。ISCA，2014年，第2635–2639页。 [在线]。可访问：http://www.isca-speech.org/archive/interspeech 2014/i14 2635.html [22] A. Vaswani、N. Shazeer、N. Parmar、J. Uszkoreit、L. Jones、A. N. Gomez、L. Kaiser 和 I. Polosukhin，《注意力就是你所需的一切》，载于神经信息处理系统进展，I. Guyon、U. V. Luxburg、S. Bengio、H. Wallach、R. Fergus、S. Vishwanathan 和 R. Garnett 编，第30卷。Curran Associates, Inc.，2017年。[23] J. Devlin、M.-W. Chang、K. Lee 和 K. Toutanova，《BERT：面向语言理解的深度双向Transformer预训练》，载于美国计算语言学协会分会*2019*年会议论文集：人类语言技术，第*1*卷（长篇与短篇论文）。明尼阿波利斯，明尼苏达州：计算语言学协会，2019年6月，第 4171 4186 [ ] 19 1423 –页。在线。可访问：https://aclanthology.org/N - Z. Feng、D. Guo、D. Tang、N. Duan、X. Feng、M. Gong、L. Shou、B. Qin、T. Liu、D. Jiang 和 M. Zhou，《CodeBERT：面向编程语言与自然语言的预训练模型》，载于计算语言学协会论文集：*EMNLP 2020*。在线：计算语言学协会，2020年11月，第1536–1547页。 [ ] 2020 139 在线。可访问：https://aclanthology.org/。findings-emnlp。M. Ciniselli、N. Cooper、L. Pascarella、D. Poshyvanyk、M. Di Penta 和 G. Bavota，《BERT模型用于代码补全的实证研究》，载于*2021*年*IEEE/ACM*第*18*届国际软件仓库挖掘会议（*MSR*）。IEEE，2021年5月，第108–119页。 [ ] 10 1109 52588 2021 00024 在线。可访问：https://doi.org/。/msr。。N. Chirkova 和 S. Troshin，《Transformer用于源代码的实证研究》，载于*ESEC/FSE '21*：第*29*届*ACM*欧洲软件工程会议与软件工程基础研讨会联合会议，希腊雅典，*2021*年*8*月*23*日至*28*日，D. Spinellis、G. Gousios、M. Chechik 和 M. D. Penta 编。ACM，2021年，第 703 715 [ ] 10 1145 3468264 3468611 –页。在线。可访问：https://doi.org/。/。R. Degiovanni 和 M. Papadakis，《bert：使用预训练语言模型进行变异测试》，载于*2022*年*IEEE*国际软件测试、验证与确认会议研讨会（*ICSTW*），2022年，第160–169页。[28] A. Svyatkovskiy、S. K. Deng、S. Fu 和 N. Sundaresan，《Intellicode Compose：使用Transformer进行代码生成》，载于*ESEC/FSE '20*：第*28*届*ACM*欧洲软件工程会议与软件工程基础研讨会联合会议，线上会议，美国，*2020*年*11*月*8*日至*13*日，P. Devanbu、M. B. Cohen 和 T. Zimmermann 编。ACM，2020年，第1433–1443页。 [在线]。可访问：https://doi.org/ 10 1145 3368089 3417058 。/。T. T. Nguyen、A. T. Nguyen、H. A. Nguyen 和 T. N. Nguyen，《面向源代码的统计语义语言模型》，载于欧洲软件工程会议与*ACM SIGSOFT*软件工程基础研讨会联合会议，*ESEC/FSE'13*，俄罗斯圣彼得堡，*2013*年*8*月*18*日至*26*日，B. Meyer、L. Baresi 和 M. Mezini 编。ACM，2013年，第532–542页。 [在线]。可访问：https://doi.org/ 10 1145 2491411 2491458 。/。V. J. Hellendoorn 和 P. Devanbu，《深度神经网络是建模源代码的最佳选择吗？》，载于*2017*年第*11*届软件工程基础联合会议论文集，2017年，第763–773页。[31] C. Pan、M. Lu 和 B. Xu，《使用CodeBERT模型进行软件缺陷预测的实证研究》，《应用科学》，第 11, 11, 卷，2021 [ ] 2076 3417 11 11 4793 第。在线。可访问：https://www.mdpi.com/ - / / /

[32] E. Mashhadi 和 H. Hemmati，《应用CodeBERT自动修复Java简单漏洞》，载于*2021*年*IEEE/ACM*第*18*届国际软件仓库挖掘会议（*MSR*），2021年，第505–509页。[33] B. Harvey、D. D. Garcia、T. Barnes、N. Titterton、D. Armendariz、L. Segars、E. Lemon、S. Morris 和 J. Paley，《Snap！（创建自己的代码块）》，载于第*44*届*ACM*计算机科学教育技术研讨会论文集，2013年，第759–759页。[34] R. Karampatsis、H. Babii、R. Robbes、C. Sutton 和 A. Janes，《大代码！= 大词汇表：面向源代码的开放词汇模型》，载于*ICSE ' 20*：第*42*届国际软件工程会议，韩国首尔，*2020*年*6*月*27*日至7月*19*日，G. Rothermel 和 D. Bae 编。ACM，2020年，第1073–1085页。 [在线]。可访问：https://doi.org/ 10 1145 3377811 3380342 。/。H. Babii、A. Janes 和 R. Robbes，《大规模代码机器学习的词汇建模》，*arXiv*预印本*arXiv:1904.01873*，2019年。[36] U. Z. Ahmed、P. Kumar、A. Karkare、P. Kar 和 S. Gulwani，《编译错误修复：为学生程序而来，源自学生程序》，载于第*40*届国际软件工程会议论文集：软件工程教育与培训，2018年，第78–87页。[37] S. Xu、Y. Yao、F. Xu、T. Gu、H. Tong 和 J. Lu，《为源代码变更生成提交消息》，载于*IJCAI*，2019年。[38] S. Kim、J. Zhao、Y. Tian 和 S. Chandra，《将树输入Transformer以进行代码预测》，载于*2021*年*IEEE/ACM*第*43*届国际软件工程会议（*ICSE*），2021年，第150–162页。[39] Y. Liu、M. Ott、N. Goyal、J. Du、M. Joshi、D. Chen、O. Levy、M. Lewis、L. Zettlemoyer 和 V. Stoyanov，《RoBERTa：一种稳健优化的BERT预训练方法》，2019年。[40] T. Wolf、L. Debut、V. Sanh、J. Chaumond、C. Delangue、A. Moi、P. Cistac、T. Rault、R. Louf、M. Funtowicz、J. Davison、S. Shleifer、P. von Platen，[35] C. Ma, Y. Jernite, J. Plu, C. Xu, T. Le Scao, S. Gugger, M. Drame, Q. Lhoest, 和 A. Rush，“Transformers：最先进的自然语言处理”，载于*2020*年自然语言处理实证方法会议论文集：系统演示。计算语言学协会，2020年10月，第38–45 页。 [ ] 10 18653 1 2020 6 在线。可访问：https://doi.org/ 。/v / .emnlp-demos。H. B. Mann 和 D. R. Whitney，“关于两个随机变量中一个是否随机占优于另一个的检验”，《数学统计年刊》，第 50–60 页，1947年。[42] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, M. Tufano, S. K. Deng, C. Clement, D. Drain, N. Sundaresan, J. Yin, D. Jiang, 和 M. Zhou，“GraphCodeBERT：利用数据流预训练代码表示”，2020年。 [在线]。可访问：https://arxiv.org/abs/2009 08366 。J. Li, Y. Wang, M. R. Lyu, 和 I. King，“使用神经注意力和指针网络进行代码补全”，载于第*27*届国际人工智能联合会议论文集，丛书 IJCAI’18。AAAI出版社，2018年，第 4159–25 页。[44] F. Liu, G. Li, Y. Zhao, 和 Z. Jin，“基于多任务学习的代码补全预训练语言模型”，载于第*35*届*IEEE/ACM*自动化软件工程国际会议论文集，2020年，第 473–485 页。[45] M. Asaduzzaman, C. K. Roy, K. A. Schneider, 和 D. Hou，“提升API可用性的上下文感知代码补全工具”，载于*2014*年*IEEE*软件维护与演化国际会议。IEEE，2014年，第 621–624 页。[46] J. Moreno-Le´on, G. Robles, 和 M. Rom´an-Gonz´alez，“Dr. Scratch：自动分析 Scratch 项目以评估并培养计算思维”，*RED.* 远程教育杂志 ´ 46， 1 23，（下转），2015无，第 – 页。[41]