北京大学

博士研究生学位论文

题目：**语言定义感知的深度代码学习技术及应用**

姓　　名：	朱琪豪
学　　号：	1901111307
院　　系：	计算机学院
专　　业：	计算机软件与理论
研究方向：	软件工程与系统软件
导　　师：	熊英飞 长聘副教授

☑ 学术学位　　☐ 专业学位

二〇二四 年 五 月

版权声明

摘要

随着信息化时代的加速发展，软件已经成为支撑社会进步的重要基础，不仅贯穿于工业生产、商业管理等多个领域，也影响着人们的日常生活。软件的广泛应用使得其开发过程愈加复杂和多样，尤其是代码规模的不断扩大和系统结构的日益复杂，提高了软件开发和维护的难度。在这样的背景下，如何提高软件开发的效率和质量成为了研究者关注的焦点。随着近年来深度学习技术在图像识别、自然语言处理等领域取得了显著成就，其同样在软件工程领域中展现出了巨大潜力，特别是在代码理解和生成方面，通过利用深度学习模型来分析和处理代码，可以提高软件开发者的研发效率。

尽管深度学习技术为软件工程领域的研究提供了新的技术手段，但当前的研究和应用还存在局限性。大多数现有方法在处理程序语言时，将代码视为一种特殊的自然语言文本，采用与处理自然语言相似的技术方法。这些方法忽略了程序语言的一项关键性质——语言的形式定义（如语法和语义）。程序语言是为了精确地表达计算机执行的指令而设计的，其包含严格的语言定义，这与自然语言的模糊性和多样性形成鲜明对比。因此，简单地将程序语言当作自然语言来处理，会忽视掉程序代码的这些本质特性，从而限制了模型在理解和生成代码方面的准确性。例如，模型可能生成语法错误的代码，或者生成的代码虽然语法正确但无法通过编译。

为了应对上述挑战，神经网络需要在学习代码语料的同时，能够感知程序语言的形式定义，以此来增加生成程序的准确性。然而通过神经网络来学习编程语言定义时，由于其具有不同于自然语言文本的形式特征，同时也没有较好的适合神经网络学习的相应技术，现有的深度代码学习技术难以对其进行理解和表征。因此如何让神经网络能够学习语言定义所包含的信息面临着新的技术难题。

本文针对通过神经网络学习编程语言定义面临的挑战，将以学习各类语言定义的关键信息为起点，从整合为综合性的预训练模型到特定下游任务依序展开研究。本文的研究路线为：

本文首先提出了三项引导神经网络学习程序语言定义的技术，分别是：

- 上下文无关文法规则感知的学习（相关工作发表于 IJCAI2022）：通过图神经网络来表征上下文无关文法规则之间的关系，基于图神经网络中的节点和边之间的信息传递机制，使模型深入理解程序语法的结构信息；引入门控层技术来融合图结构信息和语法规则的内容信息，增加了程序生成的准确性。
- 类型规则感知的学习（相关工作发表于 ICSE2023）：通过单条类型规则的学习目标让神经网络学习生成代码时所需要满足的类型约束，确保程序语义一致性

和减少运行时错误；使用图来表征类型规则和基于自注意力机制的网络架构来处理规则间的制约关系，提高了对类型信息识别的精确度与效率。

- 声明-使用关系感知的学习（相关工作应用于目前最强的开源代码模型 DeepSeek Coder，且该技术报告以本人为共同第一作者公开）：通过拓扑排序将项目级别的代码组织成相互依赖关系的最大团，使模型从训练上下文中自发学习对应的引用信息，提升了模型理解和处理代码中的函数调用等引用的对应关系。

其次，本文提出了一个适用于多种编程语言且能感知语言定义的程序预训练技术（相关工作发表于 ICSE2024）。该技术通过整合上下文无关文法的图表示、类型规则的图表示及声明-使用保留的语料组织方式，针对多语言的应用场景，开创性地解决多语言定义信息冲突并使用了多种预训练任务。这一成果不仅推动了多语言程序理解技术的发展，同时也为多项下游任务领域的研究提供了重要的基础。

本文最后针对上述形成的深度代码模型在程序搜索和程序修复这两个具体下游任务上的应用，引入了若干创新性的先进编码策略与架构设计，不仅提升了模型处理这些复杂场景的能力，还突显了深度学习技术在软件工程应用领域的广泛潜力与价值：

- 程序搜索中的字符级别重叠矩阵特征（相关工作发表于 ASE2020）：在字符层面对代码片段进行细致深入的比较分析，有效识别出代码之间的微小差异和模式变化，从而提升查找、匹配和识别代码片段时的精确度和效率。
- 程序修复中的基于编辑操作的扩展语法（相关工作发表于 ESEC/FSE2021）：通过扩展现有的语法编码机制，直接支持添加、删除或替换代码段等编辑操作，为模型提供了丰富的上下文信息和操作指令，从而增强了模型在制定修复策略时的灵活性和精确度。

通过应用上述技术，本文作者构建了一系列模型与工具，并在多个领域得到应用。例如通过应用第一部分和第二部分技术，构建了不同参数规模的当前效果最好的代码生成模型（千万参数级别的 Grape，亿参数级别的 GrammarT5 以及百亿参数级别 DeepSeekCoder-33B），并在多个第三方榜单（EvalPlus，CodeXGlue 等）取得领先地位。通过应用第三部分技术，构建了缺陷修复工具 Recoder 和 ET。其中 Recoder 是首个超过传统技术的深度学习缺陷修复工具，在多个第三方在不同设定下的测试中均达到最优效果，应用于中兴公司；ET 修复工具在国际程序修复大赛取得 Java 赛道第一名。

关键词：神经网络，程序理解，程序自动化

Deep Learning for Code with Language Definition Awareness: Techniques and Applications

Qihao Zhu (Computer Software and Theory)
Mentor: Associate Prof. Yingfei Xiong

ABSTRACT

With the accelerated development of the information age, software has become a vital foundation supporting societal progress in various fields such as industrial production and business management. The widespread application of software has made its development process increasingly complex and diverse, raising the difficulty of both development and maintenance. Therefore, both academic and industrial circles have focused on improving the efficiency and quality of software development. In recent years, deep learning technology has achieved remarkable success in fields such as image recognition and natural language processing and has also shown tremendous potential in the realm of software engineering. Specifically, in program understanding and code generation, utilizing deep learning models to analyze and process code can significantly improve the efficiency of software development.

Although deep learning technology has provided new technical means for research in the field of software engineering, there are still limitations in current research and applications. Most existing methods treat programming languages as a special form of natural language. This approach overlooks a key property of programming languages: their formal language definition (syntax and semantic etc.). Programming languages are designed to express instructions for computers to execute precisely, containing strict syntax rules and clear semantics, in stark contrast to the ambiguity and diversity of natural languages. Thus, treating programming languages as natural languages directly neglects these essential characteristics, limiting the reliability and accuracy of models in understanding and generating code. For example, models might generate code with syntax errors, or the generated code might be syntactically correct but fail to pass the typing check.

To address these challenges, neural networks need to comprehend the formal definitions of programming languages while learning from code corpora. However, the distinct formal characteristics from natural language text and the lack of suitable techniques make it difficult for

neural network to learn the definitions of programming languages. Existing deep code learning technologies struggle to understand and represent these definitions. Thus, it is challenging for neural networks to learn the language definitions.

This thesis focuses on the opportunities and challenges of learning programming language definitions through neural networks. Firstly, it delves into techniques for learning key information of various language definitions. Subsequently, it aims to integrate these techniques with pre-trained code models. Finally, it applies the code models to specific downstream tasks. The main content of this thesis includes:

Firstly, this thesis introduces three techniques guiding neural networks to learn programming language definitions:

- Context-free grammar rules aware learning (published at IJCAI2022): Representing the structure and hierarchical relationships of context-free grammar through graph neural networks. Based on the information transfer mechanism between nodes and edges in graph neural networks, this enables the model to understand the structural information of program syntax. The gated layers merge the structural information and content of grammar rules, significantly increasing the grammatical accuracy and logical coherence of program generation.
- Typing rules aware learning (published at ICSE2023): Reducing the neural network's burden through learning of individual typing rules, ensuring program semantic consistency and reducing runtime errors. Representing typing rules with graphs, and employing a neural network based on the self-attention mechanism to handle the constraints between rules, significantly enhances the precision and efficiency of type information learning.
- Declaration-use relationship-aware learning (building the currently strongest open-source code model, DeepSeek Coder): Organizing project-level code into maximal cliques with interdependency relationships through topological sorting, enabling the model to learn corresponding semantic information from the training context, significantly improving the model's ability to understand and process complex semantics and logical relationships in code.

Then, this thesis proposes a program pre-training technique suitable for multiple programming languages and aware of language definitions (published at ICSE2024). This technique, by integrating graph representations of context-free grammar, typing rules, and structured representations of declaration-use constraints, constructs a decoding framework with profound

insights into programming language specifications. For multi-language application scenarios, it innovatively resolves conflicts in multi-language definition information and employs various pre-training tasks. This achievement not only promotes the development of multi-language program understanding technology but also provides an important foundation for research in multiple downstream tasks.

Finally, this thesis focuses on the application of deep code models in two specific downstream tasks, program search, and program repair, introducing several innovative coding strategies and architectural designs. This not only enhances the models' capabilities in handling these complex scenarios, but it also highlights the broad potential and value of deep learning technology in the field of software engineering:

- Character-level overlapping matrix features in program search (published at ASE2020): Conducting detailed and deep comparisons of code fragments at the character level, effectively identifying subtle differences and pattern changes between codes, thereby significantly improving the precision and efficiency of searching code fragments.
- Edit-operation integrated syntax in program repair (published at ESEC/FSE2021): By extending the existing syntax encoding mechanism to directly support editing operations such as adding, deleting, or replacing code segments, the model is provided with rich context information and operation instructions, greatly enhancing the model's flexibility and precision in program repair.

By applying the aforementioned techniques, the author has developed a series of models and tools, which have been applied in multiple fields. Firstly, the application of this study produced the strongest code generation models of different parameter sizes: Grape with tens of millions of parameters, GrammarT5 with hundreds of millions, and DeepSeekCoder-33B with tens of billions of parameters, achieving leading positions in multiple third-party leaderboards (such as EvalPlus and CodeXGlue). By applying the technology from the third section, the defect repair tools Recoder and ET were developed. Recoder is the first deep learning-based defect repair tool that surpasses traditional technologies, achieving optimal results in tests conducted by multiple third parties under different settings, and has been applied at ZTE Corporation. The ET repair tool won the first place in the Java track at the international program repair competition.

KEY WORDS: Neural Network, Program Understanding, Software Automation

目录

表格索引

插图索引

第一章　引言

1.1　问题的提出

随着人类社会不断进步，信息化已经成为推动社会发展的关键因素之一，而软件则逐步演变为信息化基础设施的核心组成部分。在这个过程中，软件所承载的功能日益复杂；与此同时，相关的代码规模也在不断扩大。由于软件开发主要依赖开发者的人工劳动投入，如何在实践中有效提高开发者的开发效率、减轻其负担，已经成为软件工程研究领域关注的核心问题。

近年快速发展的深度学习技术在自然语言处理、计算机视觉等领域取得了显著成果。在软件工程领域，深度学习也为程序理解和程序生成带来了新的机遇。首先，在程序理解方面，深度学习可以帮助解析和理解代码的语义。传统的程序理解方法通常基于规则、模板和启发式等方法，这些方法在处理复杂和多样化的编程语言时往往受到限制。而深度学习模型可以自动捕捉程序中的模式和结构，通过模型训练可以实现更准确的源代码表示和语义分析，从而提高程序理解的质量。其中几个代表性的深度程序理解的应用为：（1）缺陷检测，深度学习可以用卷积神经网络（CNN）和循环神经网络（RNN）识别代码中的语法错误[1]和生成补丁建议[2]；（2）克隆检测，深度学习技术可以用孪生神经网络[3]计算代码片段的相似度来识别代码重复或克隆现象；(3) 程序分析，深度学习模型可以分析程序的性能特征和预测程序的执行时间[4]来帮助开发者找到性能瓶颈并进行优化。

其次，在程序生成方面，深度学习技术同样具有重要价值。深度学习技术可以自动生成代码，大规模预训练模型（如 GPT 系列模型）在自然语言处理任务中的成功表明，深度学习模型能有效地学习理解知识和模式，为特定的编程语言和任务生成高质量的代码；深度学习技术也可以用于预测和推荐代码片段，TabNine[5]和 Copilot[6] 等智能代码补全工具可以根据上下文提供实时的代码建议，提高开发者的编码效率；深度学习技术也可以用于代码搜索，通过对输入的自然语言描述进行编码，与预先编码的代码库进行比较来找到与描述最匹配的代码片段，这种方法可以提高开发者在查找代码示例和解决问题时的效率，进一步缩短开发周期。

尽管已有工作在一些任务上取得了显著的效果，但它们大多是将程序语言看作是一种特殊的自然语言，直接将自然语言处理的相关技术和模型迁移到对程序的处理过程中。这种方法虽然具有一定的合理性和简洁性，但是忽略了程序语言的特定性质——语言定义。每一种特定的编程语言，必然伴随着人类专家所赋予的语言定义，来表征特定程序片段的语义，使得编译器能够将其转换为机器码来执行特定的逻辑流程。如

果直接忽视语言定义的信息，会使得深度程序编解码模型无法解析程序中的特定约束信息，进而导致错误：表1.1展示了主流预训练程序模型在程序自动生成的基准数据集MBPP 上的错误分布。即使是有 20 亿参数的模型 CodeGen，在不考虑语言定义的情况下，仍有 6.03% 的概率生成语法不正确的程序。另一方面，即使生成的程序符合语法，也很可能无法通过编译器检查。因此，如何将程序语言的语言定义和神经网络模型结合，以提高程序生成和程序理解的准确性，是本文主要解决的问题。

表 1.1　数据集 MBPP 上的预训练模型的错误分布

错误类型 / 模型名称	**语法错误**	**命名错误**	**缩进错误**	**错误率**
CodeT5(220M)	5142	389	15	13.87%
CodeT5+(220M)	4392	294	5	11.73%
CodeT5-large(770M)	4123	229	2	10.89%
CodeGen-multi(2B)	2312	92	7	6.03%

总的来说，在软件开发过程中，深度代码学习技术扮演着重要的角色，旨在自动化编程任务，提高开发效率和代码质量。然而，这些技术在处理编程语言时，如果没有妥善考虑语言定义包含的三个主要内容：语法、类型和语义，往往会导致生成的程序质量不达标。首先，语法是编程语言的基础，规定了代码的结构和排列方式。如果模型忽略了这一点，就可能生成语法错误的代码，这样的代码无法被编译器理解，从而无法编译执行。其次，类型系统定义了变量和表达式的数据类型，是保证代码安全性和效率的重要机制。深度代码学习技术对类型系统的忽略可能导致类型不匹配的问题，这不仅会引发运行时错误，还可能导致潜在的安全隐患。最后，语义定义了程序的行为和意图。如果模型无法正确理解或应用程序的语义，即使代码能够编译通过，最终执行的结果也可能与预期大相径庭，导致逻辑错误或功能缺陷。

因此，当深度代码学习技术在处理代码时忽视了这些关键信息，其结果很可能是生成的代码充满了语法错误、类型错误以及逻辑错误，这不仅降低了软件开发的效率，还严重影响了最终产品的质量和可靠性。为了克服这些问题，本文旨在提出对应的解决方案，在设计深度代码学习技术时，更加注重对编程语言语言定义的理解，确保生成的代码不仅在形式上正确，而且能够准确实现既定的功能和逻辑，从而真正提高软件开发的效率和质量。

1.2　本文的主要内容和创新点

针对现有深度代码学习技术在捕获编程语言语言定义方面的不足，本文的核心观点是——**语言定义感知能力是深度代码学习技术必要的基石**。如图 1.1所示，本文的研

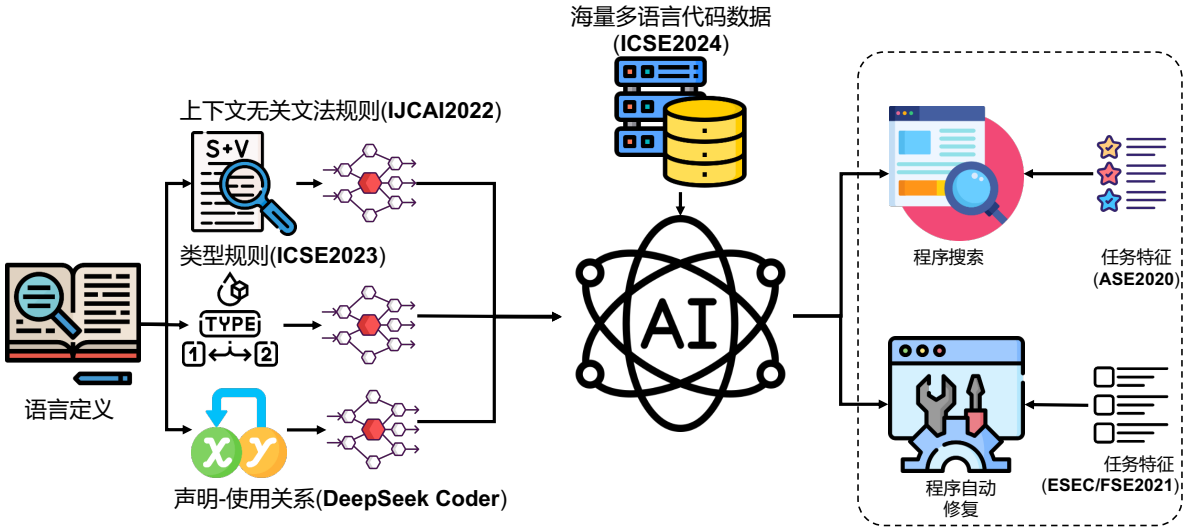


图 1.1　本文的研究思路

究路线主要分为三个部分。第一部分从语法、类型和语义这三个编程语言定义的核心要素出发，针对现有技术中经常忽略这些核心要素的现状，本研究提出了三项引导学习对应内容的创新技术。第二部分，本文讨论了这些技术在面对多语言混合应用场景时的挑战，特别是不同编程语言之间语言定义差异的问题。为解决这一问题，提出了一种整合多编程语言定义的方法，利用预训练和大规模多语言代码数据的训练，实现模型对不同编程语言定义信息的整合学习。第三部分，针对深度代码学习技术在特定下游任务（如程序搜索和自动修复）中的应用，本文根据任务特征提出了相应的优化策略，旨在提高深度代码学习技术在这些领域内的表现。通过这一系列的研究工作，本文为深度代码学习技术的发展和应用提供了新的视角和技术路径。

1.2.1　语言定义感知的学习

正如前文所说，在应用深度学习方法解析程序时，必须考虑程序语言定义本身的一些特性和约束，以确保生成的程序片段具有高质量和可维护性。语言定义中的三个信息主要包括：语法、类型以及语义。**本文的第一部分主要内容是引导神经网络学习语言定义，分别是针对语言定义的主要信息，提出特定的技术引导神经网络学习对应的约束信息**。这三种方法具体如下文所述：

1.2.1.1　语法：上下文无关文法规则感知的学习

在传统方法中，程序编写的语法约束通常与搜索算法和后处理步骤结合使用来生成代码。然而，一种更高效的方法是直接将程序表示为文法规则的序列，通过迭代地预测需要使用的文法规则，来确保生成的程序语法正确。但是，当使用神经网络模型

将文法规则转换成语义的实数向量时，现有的方法仅使用简单的词向量映射技术，而没有考虑语法中的约束信息。具体而言，语法中不仅包含了丰富的结构信息，如不同文法规则之间的层级关系和在抽象语法树（Abstract Syntax Tree, AST）中互相作为父子节点，还包含大量的内容信息，如规则定义中使用的符号及其排列顺序。简单地采用传统的词编码技术往往无法充分捕捉到这两类重要信息。

为了解决这一问题,本文创新地提出通过一种图结构来表示上下文无关文法(Context-Free Grammar,CFG)：图结构的引入不仅能明确地保存规则之间的结构关系，还能通过图神经网络（Graph Neural Networks, GNN）来有效提取这些结构信息；GNN 能够通过边和节点间的信息传递与聚合，深入捕捉结构之间的联系和层次关系。

为了整合 CFG 中的内容信息，本文进一步设计了一个门控层机制，对 GNN 提取的结构信息和直接从规则定义中得到的内容信息进行有效的整合；通过对这两种信息进行加权融合，门控层能够生成一个同时包含结构和内容特征的综合性嵌入向量。

通过将 CFG 以图结构表示，并结合 GNN 和门控层技术，能够在神经网络中有效地编码程序语法的深层次信息。这种方法不仅提高了模型生成代码的准确性，也增强了其对程序逻辑的理解能力。

1.2.1.2 类型：类型规则感知的学习

在程序设计中，类型系统的约束对于确保程序的语义一致性和减少运行时错误至关重要。这些约束通过规定变量、表达式和函数等元素的类型及其相互作用，帮助维护代码的逻辑正确性。然而，对于神经网络模型来说，直接从程序语言的复杂类型系统中学习并准确理解这些类型信息是一个巨大的挑战。程序语言的类型系统包括了类型推导、变量之间的相互作用、类型兼容性等多个复杂的形式化约束，这些约束的复杂性使得模型很难完整地掌握整个类型系统。

尽管学习整个类型系统可能过于复杂，但本文主要关注单个类型规则，这些规则通常更简单且易于理解。以赋值语句的类型规则为例，考虑一个形式为 v=t 的赋值语句，其中 v 是一个变量，t 是一个表达式。判断这个赋值语句是否合法的关键在于检验表达式 t 的类型与变量 v 的类型是否兼容，可以通过一个简化的类型规则来描述，该规则的形式化表示如下：

$$\frac{\Gamma \vdash v : D \quad \Gamma \vdash t : C \quad C <: D}{\Gamma \vdash v = t : \text{Void}}$$

其中 Γ 代表当前的上下文，包含了已声明变量的类型信息；⊢ 是推导关系，表示在给定的类型环境下某个表达式的类型判断；C 和 D 分别是表达式 t 和变量 v 的类型。规则的前提是，只有当 C 是 D 的子类型（记作 C <: D），即表达式的类型兼容于变量的类型时，赋值才被视为合法。

通过对这样的简化类型规则的学习，神经网络模型可以更精确地把握程序分析任务中的类型信息，从而提升模型在处理程序代码时的性能和准确度。基于这种前提，本文设计了一种新颖的图表示法来表征类型规则中所包含的约束信息，并且基于关系编码的自注意力机制设计了对应的神经网络结构，从而显著增强了模型对于类型系统约束的学习能力。

1.2.1.3 语义：声明-使用关系感知的学习

程序的语义信息通常包含许多内容，例如程序的特定逻辑功能语义、程序语言定义的特定规则语义以及标识符的引用约束等。由于预训练模型在自然语言处理领域取得了优秀的性能，能够理解自然语言中所包含的复杂语义，本文认为神经网络能够从训练的上下文中自发地学习对应的语义信息。但是，在程序模型训练中，当前的方法还仅限于给定一个文件级别的约束信息，但是大部分程序包含复杂的跨文件声明-使用关系的信息，这使得即使模型有学习约束信息的能力，但基于给定的上下文，还是难以学会复杂的跨文件标识符引用的约束。因此，本文旨在解决当前模型无法从有限的文件级别语料学习声明-使用关系信息的问题，针对性地提出使用基于文件级别依赖来组织训练语料，通过拓扑排序的方式，将项目级别的代码组织成相互依赖关系的最大团，通过以约束为基础的团为粒度进行程序模型的训练，使得深度模型能够捕捉和理解这些复杂的上下文引用信息。

本文的这项工作填补了模型对于声明-使用关系学习的研究空白，为深度程序模型的发展开辟了新的道路。通过这种扩展训练粒度的方式，期望模型能够更好地理解和处理程序代码中的复杂语义和逻辑关系，进而提高在自动编码、代码补全、错误检测等领域的性能。

1.2.2 语言定义感知的多程序语言预训练技术

1.2.2.1 语言定义感知的多语言程序预训练模型

本文首先探讨了三种引导程序模型学习语言定义信息的方法，旨在让模型能够深入理解程序语言中的语法的信息、类型规则的信息以及标识符引用约束的信息。这些技术虽然在单个程序语言的场景下具有良好的适用性，但无法将其融合并应用于多语言混合的场景中。因此，为了高效地融合这些不同方面的理解，同时将不同程序语言的语言定义信息相结合，本文提出了一个语言定义感知的多程序语言预训练技术。这个技术旨在整合上述三种语言定义感知的学习方式，同时对于多种程序语言的语言定义的信息进行整合和适配，提出相应技术相结合，以此产生一个全面的、对语言定义有深刻洞察的多程序语言理解的程序编解码模型。该模型可应用于各种程序相关的下

游任务，如程序生成、缺陷检测、程序摘要和程序搜索等。

本文的第二部分主要内容在于提出了适配多种编程语言、语言定义感知的程序预训练技术。该技术采取了一种编码器和解码器相结合的网络架构，能够在学习程序编码的同时，学习多种编程语言的语言定义，生成符合对应语言约束的程序。其中包括利用图神经网络来描述语法信息、通过门控层来融合结构和内容信息、采用类型推导规则来精确捕获类型信息，以及使用字符级别编码和字节对编码（Byte Pair Encoding,BPE）技术来处理标识符的多样性和复杂性。此外，通过预训练模型学习项目级别的上下文引用约束，本文进一步加深了模型对代码结构和语义的理解。

其次，适配多语言的一项技术挑战便是如何融合不同编程语言的上下文无关文法。针对这个难题，本文通过实验性探究实现了一个目前效果最好的适配策略，即为每一种语言添加特定的语言后缀来使得网络能够区分不同程序语言的语法信息。同时，本文针对不同的约束信息，提出创新的预训练策略，即在预训练阶段，利用大量的程序语料库联合五种特定的无监督预训练任务，对深度程序模型进行训练。

预训练完成后，所产生的深度程序模型可以被应用于各种下游任务中，通过使用特定任务中的少量数据进行微调，该模型就可以将预训练中所学习到的语言定义的知识应用于不同的下游任务中，大幅提高现有神经网络技术在不同的程序相关的下游任务中的性能。

综上所述，本文通过提出的预训练技术产出了一个全面理解语言定义的深度程序模型，旨在提高深度程序模型的性能；通过综合应用多种技术手段，生成符合编程语言定义的高质量代码，展现了对程序语言深层次理解和应用的可能。

1.2.3 深度代码学习技术在下游任务的应用

尽管基于预训练的深度程序模型可以帮助生成符合约束的程序候选并加速搜索过程，但是在面对更加复杂的程序相关任务时，仅使用上述技术可能会存在一定的局限性。在这种情况下，需要引入更加复杂的特定特征来提高模型的性能和鲁棒性。**本文的第三部分主要内容是针对程序搜索任务和程序修复任务引入了两个创新性的技术，字符级别的重叠矩阵和结合编辑操作的扩展语法**。对应的内容分别如下文所述。

1.2.3.1 程序搜索：字符级别的重叠矩阵

尽管上述的技术能够补充具有普适性的三种程序语言定义的信息，但仍未能完全捕捉程序搜索问题中的一个重要的约束信息：字符重叠信息。不同开发者使用的名称之间的重叠表明两个不同的名称可能是相关的（例如 “message” 和 “msg”），而代码中的标识符与自然语言描述中的单词之间的重叠表明代码片段和描述可能是相关的。为了解决这个问题，本文创新地提出了一种针对程序搜索任务的表示特征——重叠矩阵，

以表示每个自然语言单词和标识符之间的重叠程度。通过这个技术，模型能够更好地捕捉不同名称和自然语言描述中单词之间的关系，从而提高程序搜索和推荐系统的准确性和效果。

1.2.3.2 程序修复：结合编辑操作的扩展语法

针对程序修复任务，本文注意到一个现象：很多修复补丁通常只需对原有代码作出极其细微的调整。基于这一观察，本文指出，如果修复技术尝试从头开始重新构建整个代码语句，这会导致潜在的搜索空间异常巨大。针对这一挑战，本文提出了一种新颖的解决方案：一种结合代码编辑操作的扩展语法方法。与传统方法不同，这种方法允许模型在解码阶段生成一连串的编辑操作序列，而不是完整的新代码语句。通过这种方式，能够显著缩减修复补丁的搜索范围，从而在保证效率的同时减少资源消耗。这种扩展语法的引入，不仅优化了程序修复的过程，也为后续相关领域的研究提供了新的视角。

1.2.4 本文产出的技术工具

综上所述，基于本文的研究内容产出的技术工具如下：

- 语法感知的程序生成模型 Grape（第三章）。Grape 对应的学术论文发表于国际会议 IJCAI2022，并且现在是千万参数级效果最好的神经网络代码生成模型 (超过 GPT-2 等亿级参数模型)。
- 类型感知的程序修复工具 Tare（第四章）。Tare 对应的学术论文发表于国际会议 ICSE2023，并且在 Defects4J 2.0 上成功修复 132 个缺陷，超过了其他采用 10 倍于 Tare 参数量的预训练模型（CodeT5[7]，CodeLLama[8]，CodeBERT[9]）的修复技术。基于 Tare 的修复工具 ET 也获得了国际缺陷修复大赛 Java 功能性缺陷修复赛道第一名①。
- 声明-引用关系保留的预训练程序模型 DeepSeek Coder（第五章）。DeepSeek Coder 对应的技术报告已经发表于 arxiv，在多个公开的排行榜上（EvalPlus[10], EvoEval[11]，bigcode-models-leaderboard[12]）均处于开源模型的第一位，也是许多代码生成工具的基座模型[13-16]。
- 语言定义感知的多语言预训练程序模型 GrammarT5（第六章）。GrammarT5 对应的学术论文发表于国际会议 ICSE2024，是五亿参数以下效果最优的神经网络程序合成模型，在条件一致的情况下，在十几个主流数据集上均超过了之前最好的模型 CodeT5，甚至超过了 20 亿参数的模型 CodeGen-multi，在评估程序模型的 CodeXGlue 排行榜上现在是程序生成和程序修复任务的第一名。

①https://apr-comp.github.io/results.html

- 基于重叠矩阵的程序搜索技术 OCoR（第七章）。OCoR 对应的学术论文发表于国际会议 ASE2024，其在当时取得了超过已有最好的代码搜索技术 CoaCor20% 的性能，并在 4 个数据集上取得最佳效果。
- 基于编辑操作的深度程序修复技术 Recoder（第八章）。Recoder 对应的学术论文发表于国际会议 ESEC/FSE2021，获得杰出论文提名奖，是目前该届会议引用数量最多的论文。同时，Recoder 工具还是在 Defects4J 数据集修复数量第一个超过传统修复工具的深度程序修复技术，其也被第三方在不同场景下开展实验，均展现出最优的性能[17-19]。在工业界中，Recoder 也被应用于中兴通讯公司，对应的 C 语言深度学习修复工具在来自公司业务部门的 32 个缺陷中成功修复了 21 个。

1.3 论文的组织结构

本文的结构划分为九个主要部分，具体如下：

- **第一章 引言**。本章主要提出了研究的问题，并介绍了本文的研究目标与主要创新之处。
- **第二章 相关研究现状**。本章详细探讨了与本文相关的现有研究和文献。
- **第三章 上下文无关文法规则感知的学习**。在这一章中，针对程序语言定义的上下文无关文法，提出了一种图结构来编码对应的结构信息，增强模型对于上下文无法文法约束的学习。
- **第四章 类型规则感知的学习**。在这一章中，针对程序语言定义的类型系统，提出了将其化简为单条的类型规则，基于类型规则设计对应的图结构来编码类型信息的约束，通过显式的结构学习，使得模型能够感知程序语言的类型约束。
- **第五章 声明-使用关系感知的学习**。在这一章中，本文观察到已有的技术并没有考虑上下文引用的约束关系，通常以文件级别的语料进行预训练，忽视了全局上下文的约束关系。因此，本文提出基于文件级别的引用关系解析，构造出项目级别的预训练粒度，从而帮助模型能够感知到上下文的引用约束。
- **第六章 语言定义感知的多语言深度程序预训练技术**。将前面三章提出的技术延伸应用于多种程序语言的场景，本章旨在通过预训练技术，将三种语言定义的信息通过特定的结构和大量程序语料的训练，产生一个性能优越的语言定义感知的多语言程序预训练模型，能够在多种任务中取得优越的性能。
- **第七章 深度代码学习技术在程序搜索上的应用**。本章基于提出的学习技术，针对程序搜索作了进一步的技术适配，使得本文的模型能够在该任务上取得优秀的性能。

- **第八章 深度代码学习技术在程序修复上的应用**。本章针对程序修复中的程序补丁表示方法的不足，提出利用编辑操作和程序语法来表示程序补丁，大大提升了本文在程序修复上的性能。
- **第九章 结论及展望**。本章总结了本文的研究成果并对未来的研究方向提出了展望。

第二章　相关研究现状

本章归纳了已有的利用语言定义信息的相关技术的研究发展。开篇回顾了结合上下文无关文法的相关技术（第 2.1 节）；其次总结了基于类型规则的程序分析技术（第 2.2 节）；最后，总结了基于语义信息的程序分析技术（第 2.3 节）。

通过对现有技术的详细分析，本章旨在揭示这些技术在指导神经网络理解和学习语言定义方面存在的局限性。这一分析不仅识别了当前方法的不足，也为本研究的焦点——将语言定义视为深度代码学习技术的核心——提供了理论基础。

2.1　基于上下文无关文法规则的相关工作

基于上下文无关文法规则的相关工作在深度学习领域内逐渐增多，主要是因为这种方法能够有效处理程序语言中的语法和结构问题。深度学习模型，特别是循环神经网络（RNNs）和长短期记忆网络（LSTMs），已被证明在捕捉长距离依赖和复杂语法结构方面非常有效。此外 Transformers 模型通过其自注意力（self-attention）机制，进一步加强了模型处理复杂语法结构的能力，这些研究主要可以分为以下两个部分。

2.1.1　基于上下文无关文法规则的程序理解技术

2.1.1.1　基于抽象语法树的程序理解技术

编程语言作为一种特殊的语言体系，其不同于自然语言的独特性主要体现在具有严格的结构和约束信息。传统的自然语言处理方法虽然在代码相关任务中取得了一定的成果，但往往因忽视了编程语言这一特性而在理解和生成代码方面遇到瓶颈。为了克服这一挑战，近年来很多研究工作开始探索更加贴近程序语言本质的编码方法，即树编码方法，这种方法考虑了程序代码的层次结构和文法规则，通过将代码表示为抽象语法树（AST），从而使模型能够理解代码的结构和语义。

例如，牟等人[20]在他们的工作中针对树的结构特性，对传统的卷积神经网络（CNN）进行了改造，提出了 TreeCNN 网络结构。该结构是专门用于处理带有层次结构的数据，如 AST。通过这种方式，TreeCNN 能够更精准地捕捉到代码中的结构特征，从而在代码克隆识别等任务上实现了更高的准确性。

另一方面，基于长短期记忆网络（LSTM）的 TreeLSTM[21]模型对原有 LSTM 进行了改进，使其能够在处理树状数据时考虑到父节点与子节点间的依赖关系。这种设计增强了模型对于代码的结构依赖信息的编码能力，使其在程序翻译等复杂任务中表现出更好的性能。

此外，Jiang 等人[22]提出的 TreeBERT 模型则是在流行的 Transformer 架构基础上进行修改，通过引入显式的树结构依赖，进一步提高模型对程序结构和语义信息的理解能力。TreeBERT 的提出不仅展示了树编码方法在深度学习模型中的应用潜力，也为后续的研究提供了新的方向，即如何更好地将程序语言的结构特性融入到模型设计中，以提升模型在各种代码相关任务上的表现。

2.1.2 基于上下文无关文法规则的程序生成技术

在程序生成方面，不同于自然语言生成，程序语言是一种自带强结构约束的一种语法序列，每一个代码片段都能根据预定义的文法被解析成相对应的抽象语法树，基于词牌序列生成的程序通常情况下很难完全满足语法约束，因此准确率有限。针对这一问题，许多研究者致力于对程序语法进行建模并实现程序生成。与基于序列的建模方式不同, 结构化的建模通常描述结构（如, 抽象语法树）生成过程的概率分布。因此, 许多工作开始探索语法制导的程序生成技术。

与 Rabinovich 等人[23] 根据树的语法结构设计不同的模块不同，Dong 和 Lapata[24-25] 提出了一种层级的树解码器 SEQ2TREE。解码器根据编码器得到的结果之后, 利用循环神经网络生成树深度为 1 的词牌, 当预测为非终结节点时, 将非终结节点的状态作为输入预测树的下一层, 直到没有非终结节点。Sun 等人[26] 提出了一种基于卷积神经网络的树形网络，有效的解决了程序生成中所面临的长依赖问题，并利用抽象语法树进行程序生成。在此基础上，Sun 等人[27] 将 Transformer 引入到程序生成，并提出了树形结构的框架，进一步解决了长程依赖和树形结构编码的问题。

2.1.3 已有工作的不足

已有这些工作在将文法规则转化为神经网络模型能够处理的语义实数向量时，主要依赖于基本的词向量映射技术，未能充分考虑文法规则中的约束性信息。具体来说，语法不仅蕴含了诸如不同文法规则之间的层次结构以及在抽象语法树（AST）中的父子节点关系等结构性信息，还包括了大量的内容性信息，例如规则定义中所用符号及其组织序列。简单地应用传统词编码方法往往无法有效捕捉这些结构性与内容性信息。为了解决这一问题，需要开发更高级的编码技术，这些技术能够综合考虑文法规则中的结构性约束和内容性信息，从而更准确地将文法规则的丰富信息转换为神经网络模型可以有效处理的形式。

2.2　基于类型约束的相关工作

类型约束在深度学习处理软件工程任务中能够为模型提供了关于代码如何互相交互的信息，从而增强了模型对代码结构的理解。具体来说，类型约束有助于模型准确识别和处理各种编程语言中的数据类型和函数签名，这能够提高代码自动生成、补全、以及缺陷检测等任务的准确性。

首先，类型约束使得深度学习模型能够理解变量和表达式的赋值规则，这有助于预测代码行为、生成类型一致的代码片段。例如，在自动代码生成和补全任务中，如果模型能够基于当前上下文和类型规则预测出下一个最合适的代码片段，那么生成的代码不仅语法正确，而且在运行时也更不易出错。

其次，利用类型约束，模型可以更好地识别潜在的类型错误和不一致，从而在缺陷检测和代码审核过程中提高准确率。通过理解变量和函数的期望类型，深度学习模型可以识别出可能导致运行时错误的类型不匹配问题，帮助开发者提前修正这些问题，从而减少运行时的故障率。已有的工作主要包含以下两种利用类型约束的方式。

2.2.1　基于类型约束的解码技术

基于类型约束的解码技术是一种利用程序的类型信息来指导深度学习模型生成代码的方法。该技术的核心思想在于将类型约束加入到模型的解码过程中，从而确保生成的代码不仅在语法上正确，而且在类型上一致。这种方法对于提高代码生成、补全和自动修复等任务的质量和准确度具有重要意义。

在基于类型约束的解码过程中，深度模型首先根据自身的概率分布进行程序的生成，然后利用一个类型检查工具在生成过程中分析已生成代码的上下文，包括变量定义、函数签名以及其他类型声明等信息。然后利用这些信息来排除不符合类型约束的候选代码，对其他搜索空间中符合类型定义的代码继续让模型进行生成。这一过程遵循编程语言的类型系统和规则，以确保每一步生成的代码都满足类型约束，从而避免类型错误或不一致。

如 Mukherjee 等人[28]提出一种宽松的类型检查方式，通过改写已有的类型检查工具，使其能够对部分生成的程序进行类型检查，以此来减小模型的搜索空间，该技术能够在特定的条件表达式生成任务上超过规模较大的程序预训练模型。

基于类型约束的解码技术还包括一系列优化策略，如动态类型推断、类型兼容性检查和基于类型的搜索空间剪枝等。这些策略进一步提高了模型生成代码的准确性和效率。例如，通过动态类型推断，模型可以在解码过程中不断更新对变量类型的预测，以适应代码的变化；通过类型兼容性检查，可以避免生成类型不匹配的代码；而基于类型的搜索空间剪枝则可以减少模型探索无效或错误解的时间，加快解码速度。

如 Agrawal 等人[29]提出了一种名为监视器引导解码（MGD）的方法，作为深度程序模型与静态分析之间的有状态接口，用于解决代码生成过程中的全局上下文缺失问题。通过在预定义的触发点查询静态分析，并将返回的建议转化为用于重塑程序模型后续解码步骤中的概率向量的掩码，MGD 能够确保生成的代码不仅语法正确，还与仓库中其他代码在类型上保持一致。这种方法克服了模型在缺乏对其他模块中定义的类型、功能或 API 的全局认识时产生的“幻觉”问题，如错误地使用其他文件中定义的类型。通过结合静态分析，MGD 能够有效引导模型生成符合类型规则约束的代码，从而提高代码的编译率和与基准事实的一致性，同时增强了模型在不同编程语言、编码场景以及满足丰富语义属性方面的泛化能力。

基于类型约束的解码技术为深度学习在软件工程中的应用开辟了新的路径，特别是在代码自动生成和补全等领域。它不仅提高了生成代码的准确性和可靠性，也为研究人员和开发者提供了工具，以支持更高效和智能的编程工作。

2.2.2 基于类型约束的强化学习技术

基于类型约束的强化学习技术是一种利用强化学习方法，以类型约束指导和优化代码生成、补全和修复等软件工程任务的技术。在这一框架下，强化学习模型不仅学习从给定的代码上下文中生成语法上正确的代码，而且还确保生成的代码满足特定的类型约束和语义要求。这种方法的目的是通过奖励机制来强化模型生成类型一致且逻辑正确的代码的能力，从而提高代码质量和开发效率。

在基于类型约束的强化学习过程中，代理（即强化学习模型）通过与环境（即代码库和类型系统）的交互来学习如何生成代码。在每一步操作中，代理根据当前状态（例如，当前代码片段和类型环境）选择一个动作（即生成特定的代码片段），然后环境提供一个反馈（奖励或惩罚），以指示该动作的好坏。奖励通常基于生成代码的类型一致性、语义正确性以及对完成任务的贡献度。

该技术的挑战之一在于设计有效的奖励机制。奖励机制需要精细地平衡各种因素，如代码的类型正确性、执行效率、以及其他可能的代码质量指标。此外，奖励设计还需考虑长期依赖性，即当前的动作如何影响后续代码的生成和整体项目的质量。

如 Jain 等人[30]提出了一种名为强化学习与协调反馈（RLCF）的方法，旨在通过强化学习训练深度程序模型生成代码。该方法在传统的预训练之后和任务特定的微调之前进行一种称为粗调的训练过程，利用两部分的奖励机制：一是利用编译器对生成的代码进行静态分析以检测语法错误和语义错误，并据此给予负奖励；二是在缺少测试用例的情况下，通过一个有访问示例解决方案的第二个程序模型（充当鉴别器）来评估生成代码是否接近问题的合理解，若能“欺骗”鉴别器，则给予正奖励。这种结合

编译器和鉴别器的奖励机制为定义有效的奖励函数提供了新途径。该方法显著提高了深度程序模型生成编译无误、可执行且输出正确的程序的能力，使模型在编程任务上的性能得到显著提升，超越了基于监督学习和强化学习的方法。此项工作不仅引入了利用 RL 反馈进行模型粗调的概念，还展示了通过编译器反馈和鉴别器模型协调反馈，平衡生成“自然”代码与通过正确性检查的目标的新方法，并证明了该方法能够在下游任务中显著提高模型生成代码的质量。

同样的，Shojaee 等人[31]提出了 PPOCoder，一种基于 PPO 算法的强化学习框架，旨在通过利用来自代码执行和结构对齐的非可微反馈优化代码生成任务。PPOCoder 结合了编译器反馈和生成代码与目标执行代码之间的语法与语义匹配得分，通过减少奖励函数的稀疏性，指导策略生成更高质量的代码。此外，为了减少记忆化的风险并防止从预训练过程中获得的先验知识发生灾难性偏离，PPOCoder 在奖励函数中加入了 KL 散度惩罚项，而不是之前工作中在微调阶段使用的基于词牌匹配的目标。PPOCoder 在多种代码生成任务（代码补全、代码翻译、程序合成）和编程语言上的性能均超过了现有最先进的模型。

窦等人[32]提出了 StepCoder，一种创新的强化学习框架，旨在通过课程代码完成子任务（CCCS）和细粒度优化（FGO）两个关键组件来提升代码生成性能。CCCS 通过逐步策略将复杂的探索问题分解为一系列更简单的子任务，以此降低探索难度，而 FGO 通过动态掩蔽单元测试中未执行的代码段，确保模型优化仅针对相关代码片段，同时利用编译器的信息来作为奖励信号的反馈。

在程序修复方面，Ye 等人[33]提出了基于程序修复的测试执行结果来提供奖励信号，将编译信息和程序执行结果反馈给模型，引导模型倾向于生成能够通过编译的程序补丁，以此来改进先前技术容易生成语法或者类型不正确的程序补丁的缺陷，提高了程序修复的缺陷数量。

总之，基于类型约束的强化学习技术为提高软件开发中代码生成和维护的自动化水平提供了一个新的视角。通过将类型约束作为学习过程中的奖励信息，这种方法有望生成更高质量、类型一致且符合项目需求的代码，从而支持更高效和智能的软件工程实践。

2.2.3 已有工作的不足

这些工作虽然利用了类型约束来帮助神经网络生成程序，但是均没有直接让神经网络来直接学习类型规则所包含的约束信息。直接让神经网络学习类型规则所包含的约束信息有助于进一步提升程序生成的质量和准确性。这种方法使得神经网络不仅是通过编译器反馈或是基于执行结果的奖励来调整其输出，更是能够在模型内部理解和

应用这些类型规则。为此，本文提出一个新的研究思路：通过在模型训练阶段引入类型规则的显式知识，使神经网络能够在生成代码时考虑到这些规则。

2.3 基于程序语义的相关工作

在如今的软件工程领域中，程序语义不仅关系到如何准确地理解和执行编写的代码，而且能提高代码分析的准确性。随着人工智能技术的飞速发展，对程序的语义学习主要包括两种技术，基于人工提取的特征进行学习的特征分析技术，以及基于不同结构代码表示方法的学习技术，各自针对编程语言的不同特性和应用场景进行优化。

2.3.1 基于程序语义的特征分析技术

这类方法主要依赖于从程序代码中提取的各种特征，用以代表和理解程序的语义内容。这些特征可以包括代码结构、执行路径、变量使用情况等，从而为分析提供丰富的信息。以 FLUCSS 系统[34]为例，该系统是专门为了定位软件中的缺陷而设计的。它不仅采用了频谱定位度量这种传统的方法，还集成了源代码的相关度量，例如代码的复杂性、变更历史等，以此来更准确地识别和定位软件中的错误和缺陷。

FLUCSS 系统通过整合这些度量，利用机器学习的方法来进一步提升缺陷定位的效率和准确性。具体而言，它采用了支持向量机和遗传算法这两种方法进行排序学习，优化了缺陷候选位置的排名。这意味着系统能够根据这些集成的语义特征，通过学习得到的模型，对可能的缺陷位置进行有效排序，从而帮助开发者快速定位问题。FLUCSS 在实际应用中表现出了较高的效能，其方法在排序软件缺陷方面达到了 50% 的平均准确率。

2.3.2 基于程序语义的学习分析技术

这类方法依赖于将程序表示为包含语义的特殊形式，通过神经网络的有监督学习来学习程序的语义，主要包含以下三种形式：

2.3.2.1 基于序列的形式

基于序列的表示形式将程序代码视为一系列的符号或指令序列。通过将这些符号序列转换为分布式的数值向量，可以使用各种机器学习算法来处理和分析代码，以支持多种复杂的任务，如代码缺陷检测、漏洞识别、代码生成等。

以 VulDeePecker 技术[35]为例，该技术专注于自动预测程序中是否存在安全漏洞。它首先通过分析程序的函数调用来生成代码片段，这些片段代表了程序的关键行为和

逻辑结构。然后，这些代码片段被转换成数值向量，作为机器学习模型的输入。VulDeePecker 采用了双向长短期记忆网络来处理序列数据。该网络能够捕获代码序列中的前后文信息，从而更准确地学习和理解程序的行为和潜在的安全漏洞。

2.3.2.2　基于树的形式

在程序分析领域，程序的结构化特性提供了独特的机会来理解和分析代码的语义。程序通常可以通过各种形式的结构化表示来模拟，其中抽象语法树（AST）是最常见的一种。AST 揭示了程序的语法结构，包括各种语句、表达式、控制流和更多复杂的结构。基于这一点，许多研究尝试通过利用 AST 这类结构化表示来深入学习和理解程序的语义和行为。

例如，code2vec[36]通过分析程序的抽象语法树来预测方法的名称。具体而言，code2vec 从 AST 中提取路径的上下文，这些路径上下文表示了树中从一个节点到另一个节点的唯一路径。通过学习这些路径上下文的向量表示，code2vec 能够捕获程序结构和语义的关键特征，从而预测方法名称。该方法在实际应用中取得了 58.4% 的 F1 分数，展现了利用程序结构化特性进行语义分析的有效性。

另一个例子是 CDLH[37]，这是一个使用抽象语法树来检测代码克隆的系统。代码克隆是软件开发中常见的现象，它涉及到复制和粘贴代码片段，可能会导致维护难度的增加和潜在的错误。CDLH 通过分析程序的抽象语法树，并将这些树表示成唯一的向量哈希，能够有效地识别 Type-3（在复制时修改了部分内容）和 Type-4（语义相似但语法不同）的代码克隆。这种基于深度学习的方法能够通过理解代码的结构和语义来识别代码克隆，为开发者提供了一个强大的工具来提高代码维护性。

2.3.2.3　基于图的形式

基于图的形式表示代码在程序分析领域内提供了一种挖掘程序语义的工具，这些方法超越了传统的抽象语法树表示，通过融合更丰富的程序结构信息，如数据流和控制流，构建出能够反映程序行为的复杂图结构。这种方法的核心思想是利用图的形式来模拟程序中各种元素之间的复杂关系，从而为程序的深度语义分析提供更为丰富和细致的视角。

以 Allamanis 等人[38]的研究为例，他们通过在增强的抽象语法树中嵌入不同类型的边来表现程序的数据流信息，创新地提出了一种基于图的程序表示方法。例如，他们构建边来连接变量节点和该变量上一次写入的位置，以及其他多种能够表示程序逻辑和数据流动态的边。这样的图表示不仅保留了程序的结构信息，还深化了对程序数据如何流动和变量如何相互作用的理解。

这种基于图的表示方法为程序的自动分析提供了新的可能，尤其是在自动化错误检测和代码理解方面显示出了显著的效力。在 Allamanis 及其团队的实践中，他们将这种基于图的表示方法应用于查找错误变量名称的任务上并取得了不错的性能。

基于图的形式表示代码通过模拟程序中的所有相关元素及其相互作用，为分析程序的语义提供了支持。这不仅推动了程序分析技术的进步，也为软件开发和维护中的自动化工具的设计和实现提供了新的思路。通过捕捉程序的数据流和控制流，这些方法有助于开发者更有效地识别代码中的潜在问题，优化程序设计，并提升软件质量。

2.3.3 已有工作的不足

已有的研究成果表明，神经网络和深度学习方法能够从大量的代码数据中学习到程序的丰富语义信息。这些方法通过训练神经网络模型来理解和预测代码的结构、功能和潜在的缺陷，展示了深度学习在软件工程领域的应用潜力。然而，大多数现有的方法主要关注在函数级别或文件级别的程序表示，这种做法虽然能够捕捉到单个函数内部或单个文件内部的程序行为和结构特征，但在处理更广泛的程序上下文时存在一定的局限性。

特别是当涉及到跨文件级别的声明-使用关系时，仅限于函数级别或文件级别的训练方法就显得不够充分。在现实世界的软件开发中，代码的模块化和重用是常见的现象，功能和逻辑经常被分布在不同的文件甚至不同的模块中。一个变量或函数的声明可能在一个文件中，而它的使用则可能散布在项目的多个其他文件中。这种跨文件的声明-使用关系有助于理解整个程序的行为，但是如果训练方法仅仅停留在单个文件或函数的级别，神经网络模型就难以学习到这些跨文件的复杂关系。

因此，本文提出一种新型的训练组织方式，基于文件的依赖关系分割训练单位，让模型能够在训练的上下文中学习完整的声明-引用关系。

第三章　上下文无关文法规则感知的学习

已有的基于文法的程序生成技术，在编码语法规则的嵌入表示时，通常简单的采用独热编码的方式，给每一个语法规则赋予一个唯一的标号，基于标号产生对应的词嵌入。然而，在处理上下文无关语言时，直接将语法规则与词嵌入结合会丢失两种信息：语法规则之间的结构关系和规则定义的内容信息。例如，在 $a \to b$ 之后可以使用 $b \to c$ 来进一步扩展 b。

针对上述问题，本章提出了一**种基于图结构的引导语法信息学习的方法，Grape**。与词嵌入方法类似，该方法引导产生的语法规则嵌入可用于下游应用；但与词嵌入不同的是，本方法保留了预定义的上下文无关文法规则的约束信息。保留这些约束信息面临以下挑战：首先，上下文无关文法包含丰富的结构信息，即不同语法规则之间的结构关系，如一个规则在 AST 中是否可以是另一个规则的父规则；在保留结构信息时，结构相似的规则应该具有相似的嵌入。例如，在图 3.1中，规则 4 或规则 5 都可以是规则 3 的子节点，同时也可以是规则 6 的父节点，所以它们的嵌入最好是接近的。其次，上下文无关文法还包含大量的内容信息，即规则定义本身的内容中的信息，如规则中使用的符号、符号的顺序等。例如，规则 6 和规则 9 的嵌入应该相似，因为它们共享 *orelse* 符号。传统的词嵌入方法无法保留这两种信息，因为规则是用独热编码表示的，这样规则的结构关系和内容都不会作为神经网络的输入。

为了保留结构信息，本章提出用图来表示上下文无关文法，称为*语法关系图*，其中图的每个节点对应于语法中的一个规则，每条边表示两个规则之间的父子关系；随后使用图神经网络（GNN）将每个节点映射到一个向量；为了保留内容信息，进一步使用门控层将规则定义与 GNN 中的节点嵌入相结合，将 Grape 用于替换现有的基座神经网络中的嵌入层，以此来产生对应的语法规则的嵌入。

本章在六个广泛使用的基准测试上进行了实验，这些基准测试包含四种上下文无关语言。Grape 使用这些基准上最先进的模型之一 TreeGen[39]作为本方法的基准模型，通过直接将 TreeGen 的嵌入层替换为 Grape 来验证本章方法的有效性。实验结果表明，本方法在六个主流基准数据集上使 TreeGen 的准确率提高了 0.8-6.4 个百分点。尽管本方法是为规则嵌入设计的，但它也可以应用于 AST 词牌嵌入的表示方法。此外，本章还通过结合 Sandwich Transformer[40]展示了 Grape 在方法命名上的通用性。总的来说，本章提出了以下创新点：

- 首次尝试让神经网络学习语法的嵌入，设计了一种图结构来表示上下文无关文法，并使用一种新颖的神经架构来编码这种图结构。

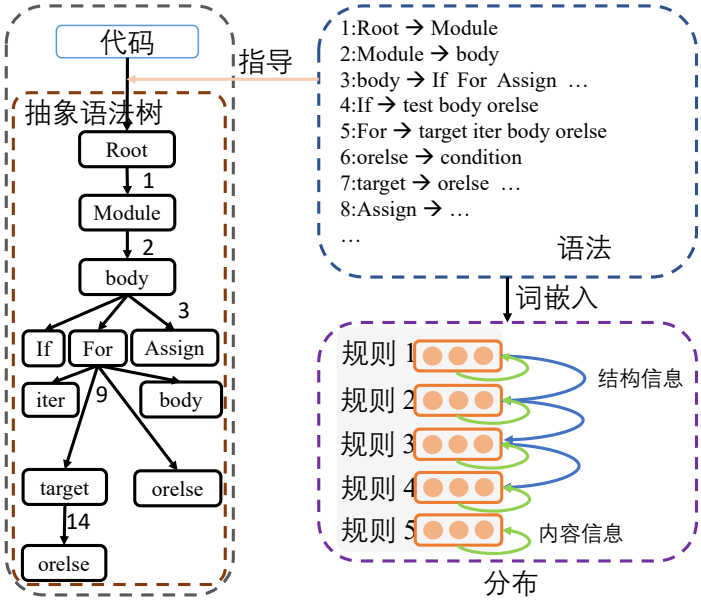


图 3.1　上下文无关文法和相应的 AST

- 在包含四种上下文无关语言的六个基准上评估了 Grape。实验结果显示，本方法在学习语法的语法和语义信息方面有效，并且在这些基准上比 TreeGen 取得了 0.8-6.4 个百分点的改进。本章还在另一个与代码相关的任务（生成方法对应的名称）上做了评估以展示 Grape 的通用性。实验结果显示，Grape 也同样提高了基础模型的性能，提升了 1.6 个百分点的 F1 分数。

3.1　方法细节

图 3.2展示了 Grape 的概览。如前所述，Grape 以上下文无关文法作为输入，输出每条规则的嵌入表示。为了实现这一组件，本章首先将语法映射到一个新颖的图结构中，然后使用图神经网络来产生嵌入。

3.1.1　语法关系图

在本节中，详细介绍语法关系图的详细结构以及将上下文无关文法映射到图中的方法。上下文无关文法可以定义为 $G^{(\mathrm{gra})} = \langle N, T, R, \lambda \rangle$，其中 N 表示非终结符的集合，T 表示终结符的集合，R 表示产生规则的集合，$\lambda \in N$ 表示一个特殊的起始符号。可以应用于非终结符 A 的产生规则表示为 $A \to B_1\ B_2\ \cdots\ B_n$，规则的应用将 A 替换为序列 $B_1\ B_2\ \cdots\ B_n$。

本节中提出*语法关系图*来表示上下文无关文法。形式上，给定一个上下文无关文法 $G^{(\mathrm{gra})}$，一个语法关系图是一个元组 $G = \langle V, E \rangle$，其中 $V = R$ 表示图中的顶点，

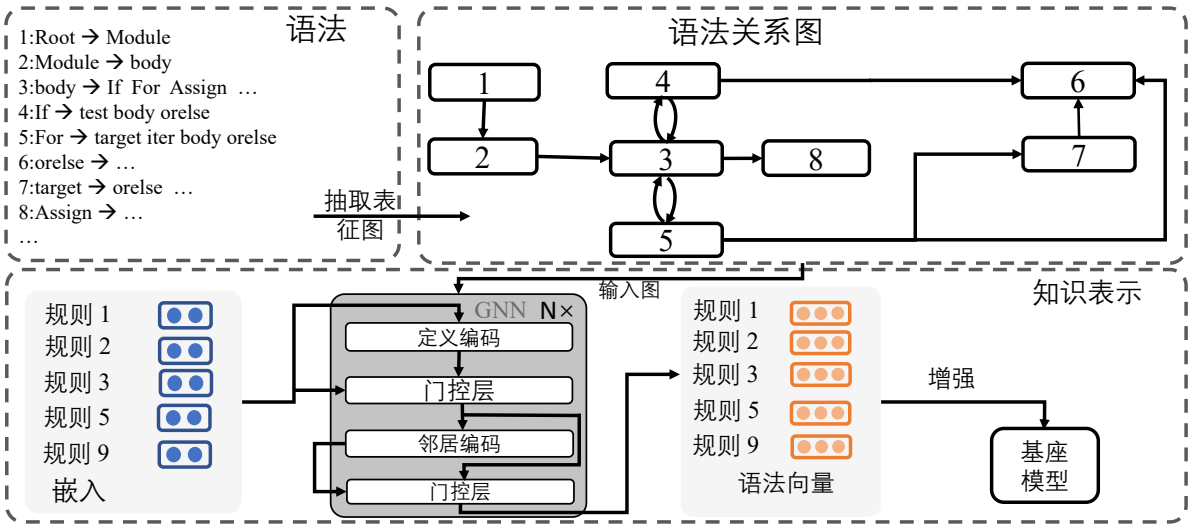


图 3.2　上下文无关文法和相应的语法关系图

$E \subseteq V \times V$ 表示边，并且是满足以下条件的最小集合：对于任意两条规则 $r_1 = A_1 \to B_{11}\ B_{12}\ \cdots\ B_{1m} \in V$ 和 $r_2 = A_2 \to B_{21}\ B_{22}\ \cdots\ B_{2n} \in V$，如果存在某个 j 使得 $A_2 = B_{1j}$ 且 $1 \le j \le m$，则 $(r_1, r_2) \in E$。

图 3.2展示了一个语法关系图的例子：左侧是一组语法规则，其中开头的数字是语法规则的编号；右侧是给定部分上下文无关文法的对应*语法关系图*；图中的每个节点对应于语法中的一个产生规则，并且用规则的 ID 标记。如图所示，规则 2 可以用来扩展由规则 1 产生的非终结符，因此在图中节点 1 有一个指向节点 2 的有向边。

3.1.2　神经网络结构

Grape 的第二部分是一个图神经网络（GNN）。该模型将语法关系图作为输入，并输出每条规则的向量，图 3.2是这部分的总览。在本节中，将介绍本方法中使用的 GNN 的详细结构。

嵌入层　Grape 的第一层是类似于大多数基础模型的一位有效编码层。Grape 将每个产生规则的唯一编号转换为 GNN 可以计算的固定大小的向量。

静态编码。Grape 首先需要将语法关系图中的每个节点编码为 GNN 的一个固定大小的向量。Grape 直接使用每个产生规则的编号作为一位有效编码标签。因此，本文通过词嵌入方法将 G 中的每个节点编码为 $\mathbf{n}_1, \mathbf{n}_2, \cdots, \mathbf{n}_P$，其中 P 表示 R 中规则的总数；并使用 r_i 的静态编码作为 GNN 的初始嵌入表示。

内容定义编码组件　如前所述，每个节点的静态编码仅编码规则的编号，但是这样就缺少了规则的定义信息。因此本节设计了将语法规则的定义整合到每个节点的嵌入的

组件，这个组件首先编码规则的内容定义，随后使用门控子层将编码后的规则内容嵌入与节点的语法向量结合起来。

内容定义编码子层。为了编码每个语法规则的内容信息，Grape 首先通过词嵌入方法将 $T \bigcup N$ 中的符号表示为向量 $\mathbf{s}$。然后，对于一个语法规则 $r: \alpha \rightarrow \beta_1\ \beta_2 \cdots \beta_n$，Grape 在相应的标记嵌入上采用一个全连接层。r 的规则编码计算为 $\mathbf{s}_r = \mathbf{W}[\mathbf{s}_\alpha; \mathbf{s}_{\beta 1}; ...; \mathbf{s}_{\beta n}]$，其中 $\mathbf{s}_\alpha$ 和 $\mathbf{s}_{\beta_j}$ 是父节点 α 和子节点 β_j 的词嵌入，$\mathbf{W}$ 是一个可训练参数。这些向量随后被送入门控子层，并在每次迭代时与每个节点的输入特征整合。

门控子层。由于每个节点的输入特征在几次迭代后包含了 G 的丰富结构信息。为了强调这些特征的重要性，本节采用了一种名为门控子层的机制来整合输入特征与语法规则内容编码。这种机制将三个向量 $\mathbf{q}, \mathbf{c}_1, \mathbf{c}_2$ 作为输入，并通过多头机制将 $\mathbf{c}_1$ 与 $\mathbf{c}_2$ 结合起来，将 $\mathbf{q}$ 作为控制向量。详细的计算方式可以在孙等人[39]的工作中找到。

对于第 t 次迭代，本节使用节点 r_i 的输入特征 $\mathbf{r}_t^i$ 作为控制向量，将其自身与相应的定义编码 $\mathbf{s}_{r_i}$ 整合。因此，节点 r_i 在第 t 次迭代的计算可以表示为 $\mathbf{m}_t^i = \text{Gating}(\mathbf{r}_t^i, \mathbf{r}_t^i, \mathbf{s}_{r_i})$. 向量 $\mathbf{m}_t^i$ 随后被送入第二个神经组件，以整合邻居节点的信息。

邻居编码组件　如同传统的图神经网络 (GNN) 一样，每个节点的编码信息应该与其邻居的编码信息整合，以提取图的结构信息。这个组件首先通过邻接矩阵计算邻居的编码，然后使用一个门控机制将节点的向量与其邻居的编码结合。

在邻居编码组件中，本节使用传统图神经网络 (GNN) 的方式来计算节点 r_i 在第 t 次迭代的邻居编码，可以表示为：$\mathbf{p}_t^i = \sum_{r^j \in G} A^n_{r^i r^j} \mathbf{m}_t^j$，其中 A^n 是图 G 的归一化邻接矩阵。本节对该矩阵使用由 Kipf 等人[41]提出的标准化操作：$A^n = S_1^{-1/2} A S_2^{-1/2}$，其中 A 是图 G 的邻接矩阵，S_1 和 S_2 是对 A 的列和行求和得到的对角矩阵。为了编码边的方向，本方法定义一个超参数 $\beta (\beta > 1)$ 来表示这一信息。例如，如果节点 r_i 有一个指向 r_j 的有向边，那么单元 $A_{i,j}$ 是 1，而单元 $A_{j,i}$ 是 β。

在门控子层中，为了将邻居编码与每个节点的输入向量整合，本方法应用同样的子层到输入向量 r_t^i 和邻居编码 p_t^i 上，用输入向量 r_t^i 作为控制向量 $\mathbf{q}$ 来合并这两个向量，这个计算可以表示为 $\mathbf{r}_{t+1}^i = \text{Gating}(\mathbf{m}_t^i, \mathbf{m}_t^i, \mathbf{p}_t^i)$。通过在每个节点的初始编码上应用 N 次迭代的 GNN 层，得到每个语法规则的向量，再将这些向量送到基座模型中。

3.2 实验结果

3.2.1 实验 I: 保持语法的规则嵌入

下文将报告 Grape 在几种不同的基准测试和上下文无关语言上的性能表现。

表 3.1　数据集的统计特征

	代码生成			语义解析		正则表达式生成
统计特征	**HS**	**DJANGO**	**CONCODE**	**ATIS**	**JOB**	**StrReg**
# 训练集	533	16,000	100,000	4,434	640	2173
# 验证集	66	10,000	2,000	491	-	351
# 测试集	66	1,805	2,000	448	140	996
平均词牌数（自然语言）	35.0	10.4	71.9	10.6	8.7	33.5
平均词牌数（代码）	83.2	8.4	26.3	33.9	17.9	15.1
图节点数目	772	668	477	180	50	193
度数的平均数	3.81	3.55	4.20	2.80	2.28	5.52
度数的中位数	3	2	5	2	1	4

数据集　本节评估了包括 HearthStone 基准[42]、两个语义解析基准[24]、Django 基准[43]、Concode 基准[44]和 StrReg 基准[45]在内的六个数据集。表 3.1汇总了这些数据集的统计信息。

HearthStone 基准包含了 665 张不同的 HearthStone 卡牌。每张卡牌由一份自然语言 (NL) 规范和一段用 Python 编写的程序组成。在处理 NL 时，本方法使用了孙等人[39]所描述的结构化预处理方法。语义解析任务包含两个基准。这项任务的输入是 NL 规范，而输出是一小段特定领域特定语言 (DSL) 中的 lambda 表达式。Django 基准包含了从 Django 网络框架中提取的 18,805 行 Python 源代码。每行代码都附有一份 NL 规范的注释。Concode 基准包含了 104,000 对 Java 代码和 NL 规范及其程序上下文。StrReg 基准包含了 3520 对结构复杂且现实的正则表达式和 NL 描述。

本章涉及的基准包括四种上下文无关语言：Java、Python、lambda 表达式和正则表达式。因此，对于 Java 和 Python 使用了从官方解析器中提取的语法①。对于 lambda 表达式，本章使用了 Kwiatkowskie 等人[46] 编写的语法。对于正则表达式使用了 Ye 等人[45]定义的语法。

指标与超参数　现有研究针对不同基准测试采用了不同的指标。StrAcc、Acc+、ExeAcc 和 DFAAcc 都是测量正确程序的百分比，但对正确性的定义不同。StrAcc[43]认为，当程序与真实结果的词牌序列完全相同时，该程序被视为正确。Acc+[26]进一步允许变量的重命名。ExeAcc[24]则进一步考虑了运算符的对称性。DFAAcc[45]认为，当一个正则表达式与真实结果的确定性有限自动机（DFA）等价时，该正则表达式被视为正确。为了与现有结果进行比较，本方法遵循了现有研究中的指标设置。

①对于 Python 和 Java 来说，解析器的链接分别是 https://docs.python.org/3/library/ast.html 和 https://github.com/c2nes/javalang 。

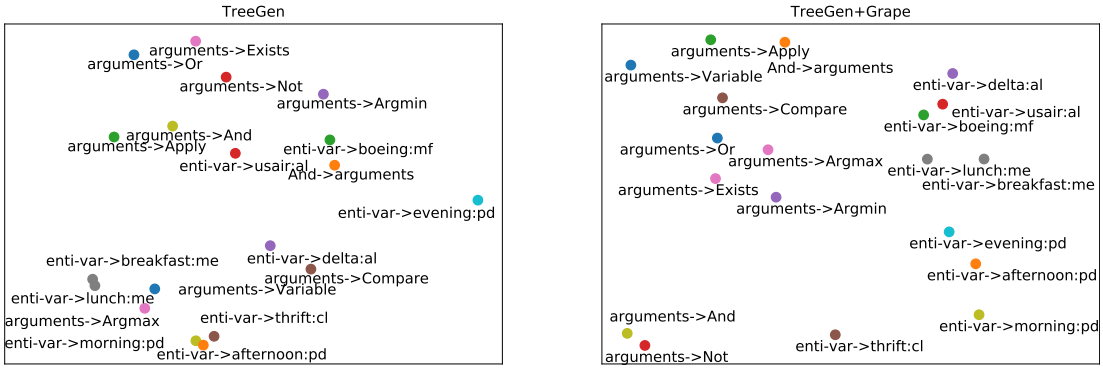


图 3.3　可视化结果

对于本方法模型的超参数，设置迭代次数 $N = 9$。隐藏层大小统一设置为 256。在图神经网络（GNN）层的每次迭代后，使用比例为 0.15 的 dropout。模型采用 Adam 优化器进行优化，学习率 $lr = 0.0001$。本方法基于 Concode 验证集的性能选择了超参数。迭代次数对结果的影响较小。当本方法将迭代次数从 6 改变到 11 时，模型在 Concode 上的性能变化小于 0.4%。由于这些基准测试的验证集较小，本方法在 Atis、Job 和 HearthStone 上使用不同的随机种子运行了 Grape 五次。在推理时，本方法采用了束搜索，束大小 $b = 5$，遵循孙等人[26]的设置。

基础模型与推理设置　本方法使用 TreeGen[39]作为基础模型，它是当前这些基准测试中的最先进模型之一。它使用基于树的 Transformer 模型来生成规则序列，具体是基于输出的概率分布选择下一个生成规则。在本方法中，生成规则的嵌入包含了丰富的结构和内容信息。因此，本方法引入了指针网络，它可以直接从语法关系图中选择一个规则。指针网络的计算公式为

$$\theta = \mathbf{v}^T \tanh(W_1\mathbf{h} + W_2\mathbf{r}) \tag{3.1}$$

$$\text{P(在步骤 i 中选择规则 s}|\cdot) = \frac{exp\theta_s}{\sum_{j=1}^{N_r} exp\theta_j} \tag{3.2}$$

其中，$\mathbf{h}$ 表示解码器的输出，$\mathbf{r}$ 表示输入到基础模型中的 Grape 的输出。

在训练过程中，本方法首先根据特定语言的语法构建语法关系图，再基于语法图构建相应的图神经网络（GNN）层，并将其与基础模型连接。上述过程可以端到端训练，其中 GNN 的输出直接输入到 TreeGen 中，并且梯度可以向后传递。

总体结果　表 3.2展示了本方法与之前的最优模型在几个基准测试中的性能比较。表3.2中的每一行对应一个现有的方法，并显示了其性能。第一部分表示使用基于规则的翻译但不使用神经模型的传统方法。第三部分表示使用大量额外数据预训练的神经模型。如

表 3.2　在不同基准测试集上的性能比较

		代码生成					语义解析		正则表达式生成
	方法名称	**HearthStone**			**Django**	**Concode**	**Atis**	**Job**	**StrReg**
	指标	**StrAcc**	**BLEU**	**Acc+**	**StrAcc**	**StrAcc**	**ExeAcc**	**ExeAcc**	**DFAAcc**
	KCAZ13[46]	-	-	-	-	-	89.0	-	-
	WKZ14[47]	-	-	-	-	-	91.3	90.7	-
神经网络	SEQ2TREE[24]	-	-	-	-	-	84.6	90.0	-
神经网络	ASN+SUPATT[23]	22.7	79.2	-	-	-	85.9	**92.9**	-
神经网络	TRANX[48]	-	-	-	73.7	-	86.3	90.0	-
神经网络	Iyer-Simp+200 idoms[44]	-	-	-	-	12.20	-	-	-
神经网络	GNN-Edge[49]	-	-	-	-	-	87.1	-	-
神经网络	SoftReGex[50]	-	-	-	-	-	-	-	28.2
神经网络	TreeGen[39]	30.3±1.061	80.8	33.3	76.4	16.6	89.6±0.329	91.5±0.586	22.5
	GPT-2[51]	16.7	71	18.2	62.3	17.3	84.4	92.1	24.6
	CodeGPT[52]	27.3	75.4	30.3	68.9	**18.3**	87.5	92.1	22.49
	TreeGen + Grape	**33.6±1.255**	**85.4**	**36.3**	**77.3**	17.6	**92.16±0.167**	**92.55±0.817**	**28.9**

表 3.3　模型在 ATIS 上的错误率

模型名称	**Top-1**	**Top-3**	**Top-5**
TreeGen	1.11%	31.61%	39.60%
TreeGen + Grape	**0.15%**	**27.09%**	**34.87%**

表所示，本方法在所有基准测试上都提高了 TreeGen 的性能，尤其是在 Atis 上的表现甚至优于传统方法。截至本方法发表时，这是神经方法首次在此基准测试上超过传统方法。此外，Grape 在 StrReg 上比 TreeGen 提高了 6.4 个百分点，可能的原因是 StrReg 的训练集不包含丰富的语法使用模式：当训练集包含丰富的使用模式时，神经网络可能从中学习结构和内容信息；但当使用模式不丰富时，编码语法定义变得至关重要。这些结果表明，学习语法嵌入能有效提高 TreeGen 的性能。

预测规则的错误率。　为了理解 Grape 是否帮助基础模型预测产生规则，因此计算了在 Atis 基准测试中模型预测的语法规则违反语法约束的概率①，如表3.3所示。本节计算了这些模型对前 1、前 3 和前 5 个预测的产生规则的错误率，定义为前 k 个预测中语法不正确的预测数量除以总预测数量。与 TreeGen 相比，加入 Grape 的 TreeGen 在所有方面都表现更好，特别是加入 Grape 的 TreeGen 的错误率从 1.11% 下降到了 0.15%。这些结果表明，Grape 帮助基础模型学习了语法的约束。

规则嵌入的可视化。　为了弄清楚 Grape 是否保留了语法的结构和内容信息，这里使用 t-Distributed Stochastic Neighbor Embedding（t-SNE）模型来展示嵌入表示的分布。本节随

①如果在前 k 个预测的规则中存在语法不正确的规则，则预测被认为是不正确的。前 k 个预测的错误率表示在生成过程中不正确预测的比率。

机选择了几条语义解析语言中的规则在图中展示。如图 3.3所示，内容相似的规则倾向于使用相似的表示，例如与节点 “arguments” 相关的规则主要位于左侧，而与 “enti-var” 相关的规则位于右侧；而 TreeGen 的规则编码则没有这个属性。同时，结构相似的规则也有相似的表示，如对于可以有共同父节点的规则 $entivar \rightarrow evening : pd, entivar \rightarrow afternoon : pd$ 和 $entivar \rightarrow morning : pd$，在带有 Grape 的右子图中有相似的表示；但在没有 Grape 的左子图中，$entivar \rightarrow evening : pd$ 与 $entivar \rightarrow morning : pd$ 的距离很远。这些观察表明，Grape 引导模型学习了更好的语法规则的嵌入表示。

时间效率和复杂度。 本节进一步评估了 Grape 的时间复杂度。平均而言，使用 Grape 在单个 Nvidia Titan RTX 上进行全部数据的训练需要 34.78 秒，而不使用 Grape 需要 30.74 秒。

3.2.2 实验 II：保留语法信息的 AST 词牌嵌入

除了语法规则序列，还有几种方法[26,36]使用抽象语法树（AST）遍历序列来代表 CFS。这些方法通常将 AST 词牌视为单词，并采用词嵌入来表示 AST 词牌。尽管 Grape 是为规则嵌入设计的，Grape 也可以用来表示 AST 词牌。为了使 Grape 适应 AST 词牌嵌入，本方法将第 i 个节点的相应规则 $r_i = \alpha \rightarrow \beta_1\ \beta_2 \cdots \beta_n$ 替换为在语法关系图中的 α，同时合并图中具有相同标签的节点。

为了评估有效性，本节在方法命名任务上进行了额外的实验。在这项任务中，模型需要预测给定其对应函数体的方法的名称。遵循 Alone 等人[53]的方法，模型预测目标方法名称为子词的序列，例如，getIndexOf 被预测为序列 “get index of”。正如 Alone[40]所描述，Sandwich Transformer（S-Transformer）采用 Transformer 来提取代码的长依赖，并采用 GNN 层学习代码的结构信息。因此使用 S-Transformer 作为基础模型，它能够处理长依赖和丰富的结构信息。

对于这项任务，这里使用了广泛使用的 Java 基准数据集[36,53]，即 *Java-small*，其中包含 11 个相对较大的 Java 项目。遵循 Alone 等人[53] 提出的验证方法，选取了 9 个项目进行训练，1 个项目进行验证和 1 个项目进行测试。这个数据集在训练集中包含 691,607 个示例，在验证集中包含 23,844 个示例，在测试集中包含 57,088 个示例。对于这个数据集，使用三个指标，精确度、召回率和 F1 分数来评估目标序列，按照 Alon 等人[36]提出的方法，基于开发集上的 F1 分数来选择最佳模型。

Grape 在方法命名任务上的表现如表3.4所示。表3.4中的每一行显示了相应方法在基准测试上的三个指标性能。与 S-Transformer 相比，Grape 在这三个指标上的表现分别提高了 0.58、2.47、1.60。这些结果展示了 Grape 的通用性。

表 3.4　Grape 在方法命名基准数据集的性能

Metric	Precision	Recall	F1
Code2Seq[53]	50.64	37.40	43.02
Code2Vec[36]	18.51	18.74	18.62
Transformer[54]	38.13	26.70	31.41
S-Transformer[40]	52.64	48.08	50.25
S-Transformer+Grape	**53.22**	**50.55**	**51.85**

3.3　本章小结

本章提出了一种学习上下文无关文法的嵌入表示的方法：首先引入了一种代表上下文无关文法的新型图结构，采用门控图神经网络来提取规则的结构信息，并通过门控层将内容信息与节点嵌入结合起来。为了确认本方法的有效性，本章在代码生成和语义解析的几个广泛使用的基准测试上进行了实验。结果显示，Grape 学习到了语法规则的良好嵌入，在结合基础模型 Treegen 的情况下，在这些基准测试上超越了现有的最好方法。

第四章　类型规则感知的学习

本章提出了一种引导类型信息的学习方法来帮助神经网络学习特定的类型约束。许多相关工作考虑到了程序语言的上下文文法的约束，提出语法引导生成[55-56]的方法：程序生成过程被简化为一系列生成步骤，每一步选择一个语法规则来扩展一个非终结符；神经网络用于估计每一步的语法规则的概率，而束搜索算法[57]被用来贪婪地选择具有较大概率的程序。通过上述方法来确保生成的程序语法正确。

然而，编程语言除了上下文无关语法之外还带有许多约束。例如，变量必须在声明后使用，参数类型的兼容性等。在现代编程语言中，这些约束通常通过类型系统被统一定义为类型规则。但由于神经模型不了解类型规则，语法引导生成器可能生成许多导致无法类型化代码的程序，称为*无法类型化的程序*。尽管这些无法类型化的程序可以通过编译器轻松过滤掉，但生成这些程序不可避免地降低了方法的性能。一方面，束搜索算法在每个生成步骤中贪婪地选择一组具有最大概率的候选项（即部分生成的程序），不了解类型的神经网络可能会为导致无法类型化的程序的候选项分配更高的概率，导致正确程序的候选项被排除；另一方面，过滤无法类型化的程序需要时间。基于本章对现有语法引导生成器的实验，生成的程序的可类型化率仅约为 30%- 40%，正确的程序经常被无法类型化的候选项排除在结果集之外。

一个直接的想法是在每一步立即过滤掉不会导致可类型化程序的候选项，尽管这一方法已被验证可行[58]，但其有效性有限。例如，在程序语句的生成过程中，由于赋值操作非常常见，不了解类型的神经网络可能会为语法规则 $\texttt{Stmt} \rightarrow \texttt{Var} = \texttt{Exp}$ 估计较高的概率；然而，如果局部变量和字段均为布尔类型，当前语句可能不是赋值操作，因为实际中给布尔变量赋值并不常见。过滤不能解决这个问题，因为这种并不常见的布尔变量赋值操作也是可行的。因此，对于神经网络来说，需要学习类型规则并在概率估计时考虑类型。

本章提出的技术旨在使神经网络能够学习类型规则并在推理过程中考虑类型信息，类型规则的学习和使用基于以下假设：如果（1）计算过程简单，并且（2）输入不包含太多无关信息以免干扰神经网络训练，则神经网络可以学习计算过程。在现有方法中，神经网络必须从训练集中学习整个类型系统，这对于神经网络来说可能过于复杂。

针对上述问题的解决思路是，尽管学习整个类型系统过于复杂，但单个类型规则往往并不复杂，且适合被神经网络学习。考虑如下的简化赋值类型规则：

$$\frac{\Gamma \vdash v : D \quad \Gamma \vdash t : C \quad C <: D}{\Gamma \vdash v = t : \text{Void}}$$

这条规则说明，给定变量 v，其类型为 D，类型上下文 Γ 记录了变量的类型信息，对类型为 C 的表达式 t，如果 C 是 D 的子类型，则赋值操作 $v = t$ 的类型正确，用类型 `Void` 表示。神经网络在学习类型规则时，需要获取类型规则的输入和输出。输入和输出之间存在三种关系：（1）子 AST（如一个表达式或一条语句）和一个类型之间的类型关系 :，（2）用户定义元素（如一个变量）和一个类型之间的类型上下文关系 Γ，以及（3）类型之间的子类型关系 <:。神经网络需要了解和使用这三种关系。然而，这三种关系十分庞大，直接编码它们将使得神经网络难以学习。

应对上述挑战的基本思路是，为了对程序进行类型检查，只需要与输入程序中的元素存在关联的部分关系。例如，如果一段程序用表达式 t' 来替换 $v = t$ 中的 t，只需要 t' 的类型、v 的类型以及两种类型之间的子类型关系。本章的第一个技术贡献是一种新颖的图表示法，`T-Graph`，用于关注程序的 AST 和与程序相关联的部分类型关系。T-Graph 中的节点附加了属性以表示相关元素的类型，并定义了几种类型的边来表示这些元素之间的类型关系。因此，T-Graph 代表了神经网络学习类型规则所需的关键信息。

尽管完整的程序可以轻松转换为 T-Graph，在语法引导生成中[59]，还需要对部分生成的程序进行编码，以推断下一个非终结符的语法规则。这样的部分程序经常具有模糊的类型，不能转换为 T-Graph。例如，给定一个部分程序 $\texttt{Var} = \texttt{Exp}_1 + \texttt{Exp}_2$，其中 `Var`、$\texttt{Exp}_1$、$\texttt{Exp}_2$ 是尚待进一步生成的非终结符，由于在 Java 中字符串和数值都可以被添加，因此难以推断 `Var` 的类型。为了解决这个问题，本章的第二个技术贡献是一种新颖的上下文无关语法形式，T-Grammar，它将类型信息集成到标准语法中，通过为每个非终结符符号附加一个类型，形成一组新符号，并使用带有类型信息的语法细化原始语法。例如，T-Grammar 不是使用语法规则 $\texttt{Exp} \rightarrow \texttt{Exp} + \texttt{Exp}$，而是有诸如 $\texttt{Exp}_{\texttt{Numeric}} \rightarrow \texttt{Exp}_{\texttt{Numeric}} + \texttt{Exp}_{\texttt{Numeric}}$ 和 $\texttt{Exp}_{\texttt{String}} \rightarrow \texttt{Exp}_{\texttt{String}} + \texttt{Exp}_{\texttt{String}}$ 的语法规则，其中 `Numeric` 是数值的超类型。这样，当神经网络预测一个语法规则时，也同时预测了其类型，使得可以从部分程序构建 T-Graph。为了确保所有正确的程序都被包含在内，T-Grammar 被设计为类型系统的上近似，即所有可类型化的程序都在 T-Grammar 的语言范围内。

基于 T-Grammar 和 T-Graph，为了验证神经网络结合类型系统信息的有效性，本章的第三个技术贡献是针对 Java 编程语言的新颖的类型感知的神经程序修复方法，Tare，即在程序自动修复（APR）下游任务上测试方法的有效性。Tare 是基于目前最先进的基于深度学习的 APR 方法之一 Recoder[56]构建的，将 Recoder 中的语法更改为 T-Grammar，并用编码 T-Graph 的神经组件替换了 Recoder 中编码 AST 的神经网络组件。为了编码具有属性的异构图 T-Graph，Tare 结合了两个之前为编码表格[60]和单词

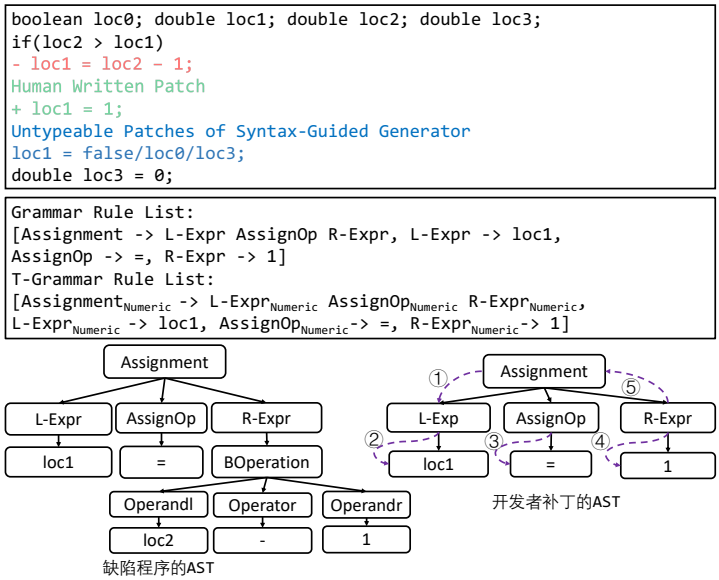


图 4.1　Defects4J 中的样例程序 Cli-25

序列[61]设计的神经层，作为一个新颖的神经组件。

本章对 Tare 性能的评估实验是在现有工作中广泛使用的三个基准上进行的，这些基准总共有 877 个程序缺陷，包括来自 *Defects4J v1.2* 的 393 个缺陷，来自 *Defects4J v2.0* 的 444 个程序缺陷，以及来自 *Quixbugs* 的 40 个程序缺陷。Tare 在 *Defects4J v1.2* 上成功修复了 62 个缺陷，超过了目前所有的 APR 方法。在另外两个基准测试中，Tare 也取得了现有方法中的最佳性能，在 *Defects4J v2.0* 上修复了 32 个缺陷，提高了 33.3%（8 个缺陷），在 *Quixbugs* 上修复了 27 个缺陷，提高了 42.1%（8 个缺陷）。此外，本章研究了 Tare 在正确补丁的排名和生成补丁的可编译率上的改进，发现：（1）Tare 在前一个指标上相比 Recoder 实现了 44.7% 的改进，而且在复杂程序中改进更高。（2）Tare 在可编译率上超过了现有的基于深度学习的 APR 方法，提高了 9.3-13.5 个百分点。这些结果表明 Tare 比现有的 APR 工具有更好的效果和泛化能力，同时也证明了其使用的编码类型系统信息的编码技术能大大提升所生成程序的准确性。

4.1　方法总览

4.1.1　展示动机的例子

本节通过真实样例程序来进一步说明本章提出的结合类型信息的方法的必要性。图4.1展示了一个来自广泛使用的基准 *Defects4J* 的缺陷 Cli-25，其中部分变量被重命名以简化程序。在这个例子中，有一个赋值语句是不正确的，正确的程序补丁是将语句的右侧替换为常量值 1。

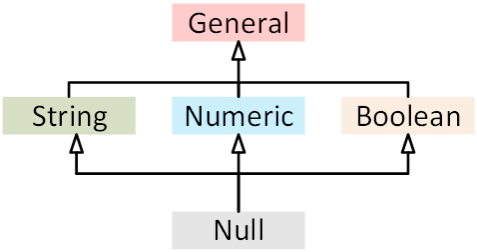


图 4.2　抽象类型的子类关系

已有的基于深度学习的 APR 有不同的方法来修复这个缺陷。假设缺陷定位方法能够正确地定位到错误代码行。一种方法是将缺陷行及其周围上下文（如具有缺陷的方法或文件）作为输入，训练神经网络输出修复后的代码行[55,62-64]。另一种方法是做出更精细的改变：首先训练神经网络预测哪部分需要被替换（在这个例子中是赋值语句的右侧），然后预测应该生成什么新内容[56]。即通过语法引导生成器[58,61]重复选择语法规则来扩展 AST 中的非终结符以生成代码。例如，为了生成人工编写的修复行，语法引导生成器从非终结符 `Assignment` 开始，生成图4.1中心部分所示的语法规则列表。这里我们假设语法引导生成器总是首先扩展最左边、最底层的非终结符。

为了生成语法规则序列，使用神经网络来估计每一步中使用的语法规则的概率，并使用束搜索算法找到具有最大概率的语法规则序列。算法维护一个候选的部分 AST 集合，候选集合大小不超过预定义的束宽 n。初始集合是一个只有根符号 `Assignment` 的 AST；在每次迭代中，算法选择池中概率最高的候选，并在完成时返回该候选，否则算法选择 AST 中一个未扩展的非终结符，并要求神经网络估计概率；神经网络以生成的部分 AST、待扩展的非终结符的位置和上下文代码作为输入，并产生扩展该非终结符的语法规则的概率作为输出。例如，在生成的最后一步，神经网络以部分程序的 AST`loc1 = R-Expr`、`R-Expr` 的位置和上下文代码作为输入，并估计扩展 `R-Expr` 的所有语法规则的概率。最后，算法计算所有扩展候选的概率，并从这些新扩展的候选和未选择的现有候选中保留 n 个最可能的候选在集合中。

当未考虑类型信息时，神经网络可能会错误地将高概率分配给无法类型化的候选或导致无法类型化补丁的候选，从而将正确的补丁排除在集合外。图4.1展示了一些生成的无法类型化补丁的例子，这些补丁要么用错误类型的字面量替换右侧，要么用未声明的变量或错误类型的变量替换。

4.1.2　Tare 的新颖组件

Tare 引入了三个新颖的组件，以指导神经网络感知类型系统的信息。本节将逐一介绍它们。

T-Grammar。如前所述，T-Grammar 通过引入类型信息来细化原始语法。在 T-Grammar 中，一个非终结符的形式为 N_T，其中 N 是原始语法中的一个非终结符，T 是一个类型，表示 N_T 生成所有由 N 可生成的类型为 T 的子 AST。然而，由于像 Java 这样的现代编程语言有大量的类型，直接使用这些类型会形成太多的非终结符，使神经网络不堪重负。为了避免这种情况，Tare 使用了抽象类型而不是目标编程语言的具体类型。在当前的 Java 实现中，使用如图 4.2所示的五种类型，其中 `General` 是所有事物的类型，`Null` 是特殊字面量 `null` 的类型，`Numeric`, `String` 和 `Boolean` 分别是数值、字符串和布尔表达式的类型。

Tare 扩展了 Java 编程语言的类型系统，以确保每个可类型化程序的子 AST 都有一个抽象类型；同时还将原始语法规则转换为 T-Grammar 规则，以确保所有可以由原始语法规则生成的可类型化程序仍然可以由 T-Grammar 规则生成。例如，图 4.1的中心部分显示了与原始语法规则列表对应的 T-Grammar 规则列表。通过附加类型，T-Grammar 规则也排除了一些无法类型化的程序。例如，没有语法规则 $\texttt{Exp}_{\texttt{General}} \rightarrow \texttt{Exp}_{\texttt{General}} + \texttt{Exp}_{\texttt{General}}$，因为 + 只能用于数值或字符串值，由此避免了实例 `1+true` 的生成。

通过将语法引导生成器中的语法替换为相应的 T-Grammar，使得生成器为部分 AST 产生类型，同时避免生成一些不在 T-Grammar 空间中的无法类型化的程序。

T-Graph。现有的语法引导生成器将程序视为一系列语法规则或词牌列表。Tare 通过 T-Graph 以附加了重要类型信息的图形式表示程序。T-Graph 保留了输入程序的三种类型关系：（1）子 AST 与类型之间的类型关系，（2）用户定义元素与类型之间的类型上下文关系，以及（3）类型之间的子类型关系。图4.3展示了前文所示的程序缺陷的 T-Graph。在 T-Graph 中，节点通过不同类型的边连接，并且可能包含属性。属性与节点的 ID 之间的区别在于，属性的值对神经网络可用，而 ID 仅用于区分不同的节点。由于使用邻接矩阵来表示边，本方法只允许两个变量节点之间有一种类型的边。

如图所示，AST 中的每个节点都被分配了一个类型属性，以表示类型关系：例如节点 $\texttt{Operandr}_{\texttt{Numeric}}$ 被分配了一个 Numeric 属性。这里的类型属性仍然使用抽象类型而非具体类型，因为可能需要编码部分生成的 AST，其中无法推断出具体类型。虽然抽象类型属性与附加到非终结符的类型重复，但这个属性仍然很重要，原因在于（1）终结符没有类型信息，且（2）当非终结符通过独热编码在神经网络中编码时，原始符号和附加的类型是不可区分的。这个属性有助于保留类型信息。

为了表示类型上下文关系，本方法引入了额外的节点（在图中以椭圆表示），命名为变量节点，用于上下文中的用户定义元素（如变量和参数）。例如，图中有三个变量节点，分别代表 loc0”、loc1” 和 “loc2”。变量使用和变量节点之间有一个橙色线条，每个变量也有其类型的属性。这样，本方法将每个变量的使用与其类型关联起来。此处

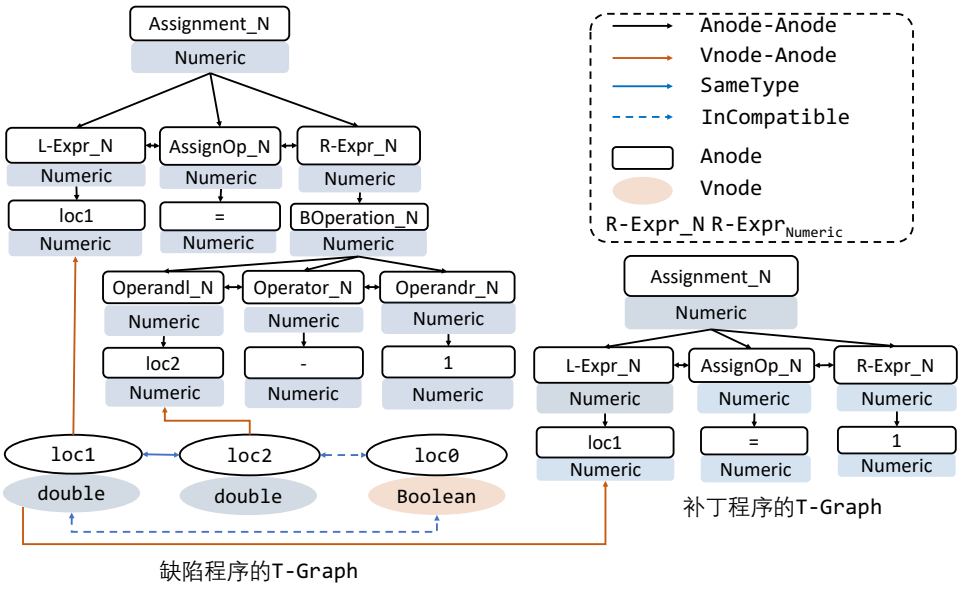


图 4.3　T-Graph 示例

使用具体类型，因为该变量的定义是从上下文中获得的，而不是生成的。

最后，本方法考虑编码类型之间的子类型关系。由于子类型关系主要用于确定类型 *A* 的值是否可以分配给类型 *B* 的变量，因此本章引入了三种类型的边，以表示是否可能进行赋值。双向的 *InCompatible*（不兼容）边表示两个变量的值都不能互相分配。单向的 *Compatible*（兼容）边表示源变量可以分配给目标变量。双向的 *SameType*（同类型）边表示两个变量是同一类型。例如，loc1" 和 loc2" 之间有 *SameType* 边，而 loc1" 和 loc0" 之间有 *InCompatible* 边。注意 *Compatible* 与子类型的关系不等价，因为 Java 中的自动装箱机制允许两种没有子类型关系的类型之间进行赋值，如从 `Integer` 到 `int`。换言之，两个变量节点之间不同方向的两个 *Compatible* 边与一个双向的 *SameType* 边是不同的。

T-Graph 编码器。T-Graph 并不是第一个用图结构来表示代码的方法。多种现有方法[65-67]已经使用图来表示代码，并使用图神经网络（GNN）来编码图。然而，T-Graph 与现有方法使用的图存在两方面不同：(1) 现有方法中使用的图是同质的，意味着边只有一种类型；而 T-Graph 是异质的，边有不同的类型。(2) 现有方法中的图不具有节点上的属性，而 T-Graph 中的节点具有属性。GNN 既不支持异质边也不支持节点的属性。因此，必须找到一种新的方式来编码这种图。

为了编码 T-Graph，本方法从之前的研究中改造了两个神经组件，用于处理异质图和具有节点属性的图，分别是一种用于编码表格[60]的技术和用于编码单词序列[61]的技术。首先，本方法改进了王等人[60]提出的关系感知注意力层，以编码异质图。与传统的 GNN 不同，该层不仅计算节点嵌入与邻居的嵌入，还计算相应的边嵌入，因此不同

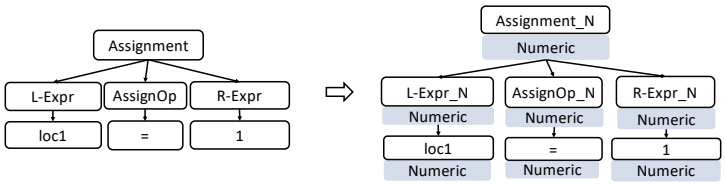


图 4.4　附加了抽象类型的 AST（T-AST ）示例

类型的边有不同的影响。其次，本方法改编了孙等人[61]提出的门控层，以结合每个节点的属性。它首先将属性转换为实值向量，然后将这些向量与相应的节点嵌入集成。结合这两个层以及一个标准的线性层，得到 T-Graph 的编码器。通过用 T-Graph 编码器替换语法引导代码生成器中现有的（部分）代码编码器，帮助语法引导生成器学习类型规则，并在推理过程中学习到类型的信息。

4.2　T-Grammar

4.2.1　抽象类型系统

如前所述，本章使用一个抽象类型系统而不是原始的具体类型系统，避免神经网络难以学习。该抽象类型系统包括一套抽象类型、抽象类型之间的子类型关系，以及一个类型推断过程，该过程给定一个原始类型系统中可类型化程序的 AST，并为每个子 AST 分配一个抽象类型。本章的第 2 节介绍的 Java 的抽象类型系统，其类型和子类型关系如图4.2所示。本节将继续讨论这个定义的具体含义。

首先，定义要求原始类型系统中所有可类型化的程序在抽象类型系统中仍然可类型化。一种方法是使抽象类型系统成为原始类型系统的一个子集，通过设计一个函数将原始类型映射到抽象类型。然而，更细致的类型实现也是可能的。例如，在当前抽象类型系统中，`Null` 类型是原始类型系统的一个细化，在 Java 中 `null` 根据上下文被类型化为其他可为空的类型。

其次，定义要求每个子 AST 都有一个类型，即使是在目标语言中没有类型的那些节点。例如，非终结符 `AssignOp` 可以生成赋值运算符，如 `=`、`+=` 和`/=`，这些在 Java 中没有类型。处理这种情况的标准方法是给所有这些子 AST 分配 `General` 类型，但在特定情况下可以分配更细化的类型。在当前抽象类型系统中，如果所有操作符都具有类型 T，本方法会为该操作符分配类型 T。例如，运算符 `+=` 在语句 `s+="a"` 中具有 `String` 类型，在语句 `a+=1` 中具有 `Numeric` 类型，但在任何语句中都不会具有 `Boolean` 类型。

最后，定义要求每个子 AST 只被分配一个类型。在经典的类型系统中，由于子类型关系的存在，一个表达式通常有多个类型，一个最小类型（如 `String`）及其所有超

类型（如 `Object`）。从经典类型系统的角度看，这个要求等同于要求所有子 AST 都有一个最小类型。本方法的 Java 抽象类型系统满足这个属性：（1）`null` 具有最小类型 `Null`。（2）除了 `null` 外，类型 `Numeric`、`Boolean` 和 `String` 之间没有交集，因此对于任何其他具有这三种类型之一的子 AST，该类型也是最小的。（3）所有其他子 AST 都具有最小类型 `General`。

附加了上述抽象类型的 AST 为 *T-AST*。图4.4展示了一个 T-AST 的例子。

4.2.2 T-Grammar 及其属性

基于抽象类型系统，本方法继续定义 T-Grammar。T-Grammar 将类型信息附加到原始语法中的非终结符，并修改语法规则以使得满足下面两个条件（1）包含所有可类型化程序，(2) 尽可能排除不可类型化的程序。形式上，设 $G = (N, \Sigma, R, S)$ 是一个上下文无关语法，其中 N 是非终结符集合，Σ 是终结符集合，R 是语法规则集合，S 是开始符号。设 T 是一个抽象类型系统。基于 G 和 T 的 T-Grammar 是一个元组 (N^T, Σ, R^T, S)，其中 N^T 和 R^T 是满足以下三个条件的最小集合。

（1）对于 N 中的任何非终结符 n 和 T 中的任何抽象类型 t，有 $n^t \in N^T$。

（2）对于 R 中的任何原始语法规则 $N \to A^1\ A^2\ \dots\ A^k$，如果在任何可类型化程序中存在该原始语法规则的使用，并且在这种情况下对应于 $N, A^1, A^2, \dots, A^k$ 的子 AST 分别具有抽象类型 $T, T_1, T_2, \dots, T_k$，那么在 R^T 中就有语法规则 $N_T \to A^1_{T_1}\ A^2_{T_2}\ \dots\ A^k_{T_k}$。这里的 A^i 可以是非终结符或终结符。当 A^i 是终结符时，定义 A^i_T 为 A^i。例如，因为 `+=` 可以被类型化为 `String` 或 `Numeric`，所以存在语法规则 $\texttt{AssignOp}_{\texttt{String}} \to \texttt{+=}$ 和 $\texttt{AssignOp}_{\texttt{Numeric}} \to \texttt{+=}$。如上所述的分析，不存在语法规则 $\texttt{AssignOp}_{\texttt{Boolean}} \to \texttt{+=}$。

（3）S 包含在 N^T 中。对于 T 中的任何抽象类型 t，$S \to S_t$ 包含在 R^T 中。

从这些条件可以看出，T-Grammar 包含了原始语法中所有可类型化的程序。这是因为（1）所有原始的可类型化程序在抽象类型系统中仍然是可类型化的，（2）第二条规则考虑了所有可类型化程序中语法规则的所有可能应用，以及（3）第三条规则确保生成任何类型的起始符号。

转换后的 T-Grammar 也排除了一些无法类型化的程序，因为 T-Grammar 在转换语法规则时只考虑最小的抽象类型。例如，因为操作符`/=` 仅适用于数值类型，所以没有语法规则 $\texttt{AssignOp}_{\texttt{General}} \to \texttt{/=}$。

4.3 T-Graph

本节将介绍 T-Graph 的详细结构。T-Graph 可以被定义为一个元组 $\mathcal{G} = \langle \mathcal{V}, \mathcal{E}, \phi \rangle$，其中 $\mathcal{V}$ 表示图中的顶点，$\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ 表示表示关系的边，而 $\phi : \mathcal{E} \to \mathcal{R}$ 表示一种边

映射函数，其中 $\mathscr{R}$ 表示预定义边类型的集合。本方法将具有缺陷的代码的 T-AST 定义为 $\mathscr{G}\text{ast} = \langle \mathscr{V}\text{ast}, \mathscr{E}\text{ast} \rangle$，并将上下文中的开发者定义的元素表示为 $\mathscr{V}\text{var}$。图4.3中的程序缺陷的 T-Graph 展示了这个过程。

4.3.1　节点的定义

本节首先介绍 $\mathscr{G}$ 中节点的组成。$\mathscr{G}$ 的节点主要由两种类型的节点组成，AST 节点和变量节点。形式上，节点集可以定义为 $\mathscr{V} = \mathscr{V}\text{ast} \cup \mathscr{V}\text{var}$。

4.3.1.1　AST 节点

节点的第一部分来自 $\mathscr{V}\text{ast}$，即 $\mathscr{G}ast$ 的节点。这种类型的节点被命名为 *Anode*，T-AST 中节点的名称作为 T-Graph 中的 ID。在当前设计中，每个 Anode 有两个属性。首先，为了表示每个重命名符号的类型信息，每个 Anode 还有一个类型属性。其次，由于几个语法引导生成器[55-56]允许复制错误方法的子树，本方法还为每个 AST 节点分配了一个布尔可复制属性，指示 AST 是否依赖于局部上下文中的变量，导致其不能复制到其他地方。如图4.3所示，表示为矩形的节点是 AST 节点，并通过几个有向边连接。

4.3.1.2　变量节点

节点的第二部分是变量节点，$\mathscr{V}_{\text{var}}$，命名为 *Vnode*。每个 Vnode 代表上下文中的用户定义元素（即变量和参数）。本方法使用变量的名称作为 T-Graph 中的 ID。每个 Vnode 有两个属性。第一个属性指示是否可以从错误位置访问此变量。第二个属性是变量的类型。如图4.3所示，以椭圆形显示的节点是变量节点，代表上下文中的局部变量。每个节点也附有相应的类型属性。

4.3.2　边

为了表示类型规则的关系，本节定义了一组预定义边类型 $\mathscr{R}$。每条边在 $\mathscr{R}$ 中有一个类型以表示关系。边有四个子集，Anode-Anode，Vnode-Anode，Anode-Vnode 和 Vnode-Vnode。形式上，边的集合可以定义为 $\mathscr{E} = \mathscr{E}_{\text{A-A}} \cup \mathscr{E}_{\text{A-V}} \cup \mathscr{E}_{\text{V-A}} \cup \mathscr{E}_{\text{V-V}}$。表4.1展示了这些边的详细信息。

4.3.2.1　Vnode-Anode

这一子集的边表示 AST 节点和上下文中变量之间的关系。假设变量 y 的声明语句的子树的根是 x，那么在 $\mathscr{G}$ 中，x 将有一个 *Var-Declaration* 边连接到 y。类似地，当 T-AST 中的终结符 x 使用变量 y 时，x 有一个 *Use-Var* 边连接到 y。形式上，这个子集

表 4.1　T-Graph 中的边

信息	**节点** x	**节点** y	**边的标签**	描述
语法	Anode	Anode	*Parent-Child*	x is the Parent node of y
			Child-Parent	x is the child node of y
			Left-Sibling	x is the left sibling node of y
			Right-Sibling	x is the right sibling node of y
上下文	Anode	Vnode	*Declaration-Var*	x is the declaration of y
			Use-Var	x uses variable y
上下文	Vnode	Anode	*Var-Declaration*	y is the declaration of x
			Var-Use	y uses variable x
类型	Vnode	Vnode	*Same-Type*	x has the same type as y
			Compatible-Type	x is unidirectional compatible with y
			InCompatible-Type	x is unidirectional incompatible with y

可以定义为 $\mathscr{E}_{\text{V-A}} = \{\langle v_a^i, v_{\text{var}}^j \rangle | v_a^i$ 声明或使用变量$v_{var}^j\}$。如图4.3所示，Vnode（“loc1”）有一条指向 Anode（“loc1”）的边，因为 Anode 使用了相应的变量。

4.3.2.2　Anode-Vnode

这一子集的边代表了 AST 节点对上下文中变量的引用关系。这种类型的边反映了代码中变量使用与声明之间的关系，对于理解变量的作用域和可访问性至关重要。例如，如果一个 AST 节点代表了一个变量的使用，则可能有一条边连接到表示该变量声明的 Vnode。这样的边帮助模型捕获变量如何在不同代码部分中传递和变化的信息。

4.3.2.3　Vnode-Vnode

Vnode 之间的边代表了变量之间的直接关系，例如赋值或类型兼容性。这些边对于理解代码逻辑和变量之间的相互作用非常重要，特别是在分析变量的数据流和控制流时。例如，一个变量可能从另一个变量获取其值，或者两个变量可能需要在类型上兼容以便于赋值或比较操作。

通过这些不同类型的边，T-Graph 能够全面地表示程序的语法结构、变量之间的关系以及类型信息，从而为神经网络提供丰富的上下文信息，以支持更准确的程序分析和修复任务。

4.3.2.4　Anode-Vnode

这一子集的边是 *Vnode-Anode* 的逆向边。因此，这个集合可以定义为 $\mathscr{E}$A-V $= \langle vvar^i, v_a^j \rangle | \langle v_a^i, v_{\text{var}}^j \rangle \in \mathscr{E}_{\text{V-A}}$。如图4.3所示，Anode（“loc1”）也有一条指向 Vnode（“loc1”）的边。

4.3.2.5　Vnode-Vnode

这种类型代表了变量之间的子类型关系。具体来说，对于所有变量，本方法定义了三种类型的关系：(1) 变量具有相同的类型，(2) 变量具有单向兼容的类型但不是相同的类型，(3) 变量是单向不兼容的。因此，每两个变量节点都通过这些边之一连接。如图4.3所示，图中有 6 条连接变量节点的边，包含了这些变量之间的类型兼容信息。形式上，这个集合可以定义为 $\mathscr{E}_{\text{V-V}} = \langle v^i_{\text{var}}, v^j_{\text{var}} \rangle | \langle v^i_{\text{var}}, v^j_{\text{var}} \rangle \in \mathscr{V}_{\text{var}} \times \mathscr{V}_{\text{var}} \wedge i \neq j$。

4.4　T-Graph 编码器

本节将介绍所提出的 T-Graph 编码器的详细结构，如图4.5所示。

T-Graph 编码器是设计用来处理 T-Graph 的神经网络结构，它能够编码图中的节点和边，以及节点的属性和边的类型。编码器的目的是将 T-Graph 转换成一个固定大小的向量表示，这个表示能够捕获程序的语法结构、类型信息和变量之间的关系。这个向量表示随后可以被用于各种下游任务，如程序修复、代码生成和代码理解等。

T-Graph 编码器通常由以下几个关键组件构成：

- 节点嵌入层：这一层负责将图中的每个节点（包括 AST 节点和变量节点）转换为嵌入向量。这些嵌入向量是节点的初始表示，可以捕获节点的局部信息和属性。
- 边编码层：在这一层中，编码器处理图中的边，特别是边的类型（如 Parent-Child、Var-Declaration 等）。这一层的目的是根据边的类型调整节点的嵌入，以反映节点之间的关系。
- 图注意力网络（GAT）层：使用图注意力网络来处理图的结构，允许信息在节点之间流动，使节点的表示能够捕获其邻居的信息。GAT 层特别适合处理 T-Graph 的异质性，因为它可以为不同类型的边分配不同的权重。
- 聚合层：聚合层将所有节点的表示聚合成图的全局表示。这一步是通过池化操作（如平均池化、最大池化）完成的，目的是生成一个固定大小的向量表示整个图。
- 输出层：最后，编码后的图表示被传递到输出层，这里可以根据具体的应用场景进行定制，如通过全连接层进一步处理向量，以用于分类、回归或其他任务。

通过这些组件，T-Graph 编码器能够有效地编码 T-Graph 中的复杂信息，为基于图的神经网络模型提供强大的表示能力。

如图所示，编码器由一个关系感知注意力层和一个门控层组成，用于处理 T-Graph。首先，在预处理过程中，T-Graph 将被转换成三个部分：节点序列、属性序列和关系矩阵，供编码器使用。然后，关系感知注意力层结合节点嵌入和关系矩阵来学习错误代码

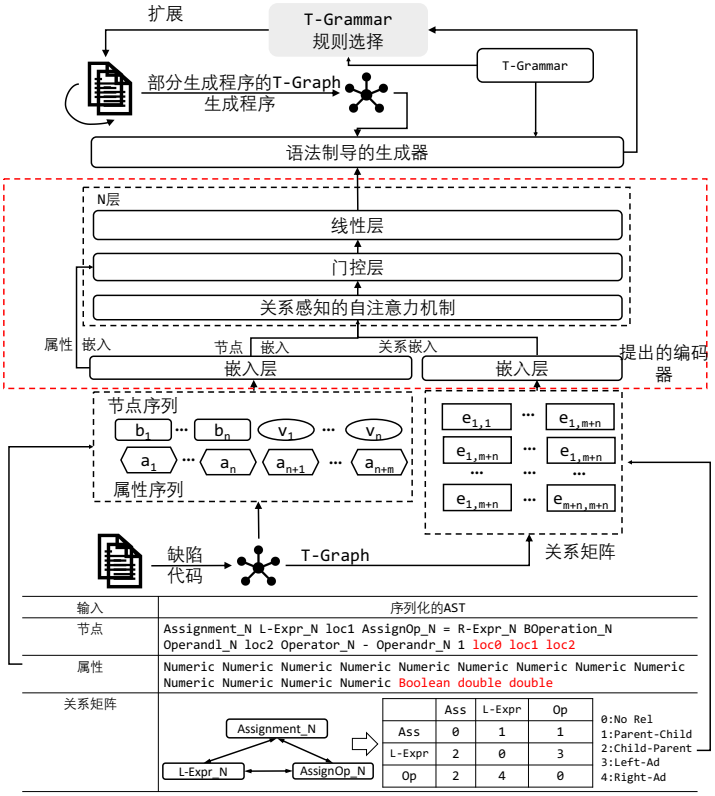


图 4.5　T-Graph 编码器的结构和输入

中的类型关系。最后，门控层将属性嵌入与第一层处理的相应节点嵌入结合起来。本节将首先描述 T-Graph 的预处理细节。

预处理。为了编码图形状的输入，首先需要通过序列化方法将其转换成序列。这里采用了前序遍历[68]的序列化方法，显著缩短了序列的长度。在这个设置下，T-Graph $\mathcal{G}$ 被表示为三个序列。

- 节点遍历序列。为了编码节点词牌，本方法首先采用节点遍历序列来表示序列信息。它由两部分组成。第一部分是 T-AST 的前序遍历，第二部分是变量的序列。假设 $\mathcal{G}$ 有 m 个 AST 节点和 n 个变量节点。因此，这个序列的长度是 $m+n$。
- 属性序列。如4.3.1节所述，T-Graph 中的每个节点都用几个属性进行了注释。为了整合这些信息，这些属性被表示为向量序列，按照节点遍历序列的顺序排列。
- 关系矩阵。本方法通过邻接矩阵 $E \in R^{(m+n)\times(m+n)}$ 来表示边，为表4.1中列出的每种类型的边在 E 中分配了一个唯一 ID。例如，$e_{i,j}$ 表示从第 i 个节点到第 j 个节点的边的 ID。特别地，如果第 i 个节点到第 j 个节点没有边，我们设置 $e_{i,j}=0$。

图4.5显示了图4.3中 T-Graph 的输入序列，这些序列通过一个关系感知的基于注意力的编码器来处理。

关系编码。受到王等人的启发[60]，本方法采用基于标准自注意力机制[69]的关系

感知注意力来编码关系矩阵。关系感知注意力层首先使用多头注意力来捕捉序列的长依赖性。为了整合节点之间的关系，这一层基于节点嵌入和边嵌入计算注意力权重。对于输入节点词牌嵌入，$\mathbf{c}_1, \mathbf{c}_2, \cdots, \mathbf{c}_n$，这一组件输出具有序列信息的一系列输出向量，$\mathbf{z}_1, \mathbf{z}_2, \cdots, \mathbf{z}_n$。第 h 个头的计算可以表示为

$$
\begin{aligned}
wi, j^{(h)} &= \frac{\mathbf{q}_i W_Q^{(h)} (\mathbf{k}_j W_K^{(h)} + \mathbf{e}_{i,j})^T}{\sqrt{d/H}} \\
\mathbf{z}_{i,j}^{(h)} &= \sum_{j=1}^{n} \sigma(w_{i,j}^{(h)})(\mathbf{v}_j W_V^{(h)} + \mathbf{e}_{i,j})
\end{aligned}
\tag{4.1}
$$

其中 $W_Q, W_K, W_V \in R^d$ 是三个全连接层的参数，d 表示嵌入大小，σ 表示评分函数（例如 softmax 或 hardmax），H 是头的数量，$\mathbf{e}_{i,j}$ 项表示相应边的嵌入。通过这一层，编码器可以通过可训练的参数学习表示类型关系。

属性编码。为了将每个节点的属性信息与节点嵌入结合起来，本方法使用了门控机制[61,70]。首先，由于 T-Graph 中的每个节点都有多个属性，通过全连接层将属性的嵌入组合成一个实数向量，对 AST 节点和变量节点使用不同的层来处理。随后和现有工作[56,61,70]类似，通过门控层将属性嵌入与节点嵌入结合起来，通过节点嵌入来决定这两种嵌入的权重。第 i 个头中门控层的计算可以表示为：

$$
\begin{aligned}
\alpha_i^o &= exp(\mathbf{q}_i^T \mathbf{k}_i^o)/\sqrt{d} \\
\alpha_i^c &= exp(\mathbf{q}_i^T \mathbf{k}_i^c)/\sqrt{d} \\
\mathbf{h}_i &= (\alpha_i^o \cdot \mathbf{v}_i^o + \alpha_i^c \cdot \mathbf{v}_i^c)/(\alpha_i^o + \alpha_i^c)
\end{aligned}
\tag{4.2}
$$

其中 $\mathbf{q}_i, \mathbf{k}_i^o, \mathbf{v}_i^o$ 都是通过节点嵌入上的一个全连接层计算出来的，$\mathbf{k}_i^c, \mathbf{v}_i^c$ 是通过属性嵌入上的另一个全连接层计算出来的。然后，$\mathbf{h}_i$ 通过一个全连接层与其他头的输出结合起来。

最后，遵循 Transformer[69]的结构，将 T-Graph 的嵌入输入到两个全连接层进行线性变换。该组件产生了机制的输出。

总而言之，编码器有 N 个这三个子层的块。对于第一个机制，它将三个序列的嵌入作为输入。对于其余的 $N-1$ 个机制，它们将前一个机制的输出作为输入。

4.5　实验设置

4.5.1　研究问题

本实验旨在回答以下研究问题：

问题 1: Tare 与现有 APR 工具相比，表现如何？为了回答这个问题，本节将 Tare 与现

有的 APR 方法在广泛使用的基准测试 *Defects4J v1.2* 上进行了比较，该测试包含 6 个项目的 393 个程序缺陷。

问题 2: Tare 在其他 APR 基准测试上的表现如何？为了展示 Tare 的通用性，本节在与其他 APR 方法相比的两个额外基准测试上评估了 Tare，分别是来自 *Defects4J v2.0* 的额外 444 个缺陷和来自 *QuixBugs* 的 40 个缺陷。

问题 3: Tare 是否提高了生成补丁的准确性？为了回答这个问题，本节比较了 Tare 与其他 APR 工具在 *Defects4J v1.2* 上正确补丁排名的表现。

问题 4: Tare 是否提高了生成补丁的可编译性？为了回答这个问题，本节计算了在 *Defects4J v1.2* 和 *QuixBugs* 上几个基于深度学习技术的 APR 工具生成的补丁的可编译率。

4.5.2 数据集

表 4.2 使用的数据集的统计信息

项目名称	版本	错误数量	描述
Chart	V1.0	26	一个为 Java 应用程序提供的 2D 图表库。
Closure	V1.0	133	一个 JavaScript 检查器和优化器。
Lang	V1.0	64	为 java.lang API 提供助手工具的一套工具。
Math	V1.0	106	各种数学相关的实用工具集。
Time	V1.0	64	Joda-Time 是广泛使用的 Java 日期和时间类库。
Mockito	V1.2	38	单元测试中最受欢迎的模拟框架。
Cli	V2.0	39	一个简单的命令行界面 API。
JackSonCore	V2.0	26	Jackson 的核心部分，定义了流 API。
JacksonDatabind	V2.0	112	Jackson 的通用数据绑定包。
JacksonXml	V2.0	6	Jackson JSON 处理器的扩展。
Compress	V2.0	47	一个用于处理压缩和归档格式的 API。
Codec	V2.0	18	各种格式的简单编码器和解码器。
Jsoup	V2.0	93	Java 的 HTML 解析器。
JxPath	V2.0	22	基于 Java 的 XPath 1.0 实现。
Gson	V2.0	18	用于将 Java 对象转换成 JSON 的 Java 库。
Csv	V2.0	16	一个简单的读写 CSV 文件的接口。
QuixBugs	-	40	基于 Quixey 挑战赛的一个基准测试集。

4.5.2.1 训练数据集

Tare 采用了一个神经模型来生成补丁，因此需要历史程序补丁来训练参数。为了公平比较，本实验使用了 Recoder[56]这篇论文收集的数据集，该数据集包含 103,585 个有效的 Java 补丁。在实验中随机将数据集分为两部分：80% 用于训练，20% 用于验证，和 Recoder[56]保持一致。

4.5.2.2　测试数据集

为了评估Tare的有效性,我们在三个基准测试上进行了实验,包括*Defects4J v1.2*[71]、*Defects4J v2.0*的额外缺陷[71]，以及*QuixBugs*[72]。*Defects4J v1.2*包含来自6个广泛使用的开源项目的393个真实的程序缺陷,是评估APR工具性能的常用基准。*Defects4J v2.0*引入了来自12个项目的额外444个缺陷。*QuixBugs*包含了从Quixey挑战赛的python程序翻译成Java的40个程序。每个程序包含一个单行的缺陷，以及通过和失败的测试用例。表4.2展示了这些基准的详细信息。

4.5.3　自变量

4.5.3.1　错误定位

本章在两种错误定位设置下评估Tare。第一种设置遵循之前的方法[56,73-75],使用了一个由GZoltar[76]实现的基于频谱的算法Ochiai；第二种设置给APR工具提供了真实的缺陷位置，这被称为完美缺陷定位，这种设置旨在在不受缺陷定位技术影响的情况下，找出工具的真实性能，在现有工作中被广泛使用[55-56,62-64]。

4.5.3.2　基线技术

本章将Tare与几种最先进的APR方法进行比较。(1) *传统APR工具*。这里选择4个基于传统技术的常用现有APR工具进行比较：CapGen[77]，TBar[73]，SimFix[74]，Hanabi[58]。(2) *基于DL的APR工具*。随着DL技术的发展，最近提出了许多基于DL的APR工具,这里选择了5个性能最好的相关模型:CoCoNuT[64],CURE[78],RewardRepair(RRepair)[79]，DLFix[55]，以及Recoder[56]。值得注意的是，前四个模型都采用基于词牌的解码器结构来生成补丁，而Recoder和DLFix使用语法引导的解码器。此外，据我们所知，Recoder在Defects4J v1.2上正确修复了最多的缺陷。遵循现有方法的通常做法[56,73-74]，所有基线的性能都是从现有论文中收集的。由于几种工具只在当前实验的一两种设置下进行了评估，对于每种设置，本实验选择了在相应设置中被评估过的最先进的APR工具（具有最佳召回率或精确度)。此外，本实验考虑了两个额外的基线。第一个是Recoder-F，它在Recoder的束搜索过程中直接使用基本类型检查过滤掉不可输入的候选项。另一个是Recoder-T，它只是将Recoder的原始语法替换为T-Grammar,以显示T-Graph的有效性。

4.5.3.3　补丁验证和正确性

在本节的实验中，Tare根据缺陷定位技术的结果生成补丁。对于每一个可疑的具有缺陷的程序语句，Tare采用大小为100的束搜索策略生成候选补丁，即基于分数为

每个可疑语句生成 100 个补丁。受运行时间的限制，本实验只为错误定位技术给出的前 500 个可疑错误语句生成补丁。生成补丁后，使用开发者编写的测试套件执行补丁，直到找到一个合理的补丁[①]。遵循之前的工作[56,73-74]，Tare 设置了 5 小时的运行时间限制。当合理的补丁与开发者编写的补丁相同或在语义上等同时，该补丁被认为是正确的。

4.5.3.4 实现

在当前的实现中，因为 Recoder 是在几个基准测试上最先进的 APR 方法之一，本实验直接采用了语法引导的生成器 Recoder[56]作为 Tare 的解码器，只将 Recoder 中用于编码部分代码的神经组件替换为 T-Graph 编码器。

4.5.3.5 超参数

对于模型的超参数，本实验设置编码器迭代次数 $N = 5$，即编码器包含 5 个堆叠的块。其他神经组件的超参数按照 Recoder 的配置进行设置。此外还在注意力层的每个块之后应用了 dropout，其中 dropout 率为 0.1。模型通过 Adam 优化，初始学习率 $lr = 0.0001$，使用了早停策略。所有实验都使用固定的随机种子进行，以避免随机性并保证可重现性。

4.6 实验结果

4.6.1 问题 1：Tare 的有效性

4.6.1.1 无完美缺陷定位的性能

表 4.3 展示了 Tare 在没有完美缺陷定位的情况下的性能。如表所示，Tare 在 *Defects4J v1.2* 上显著优于比较的其他 APR 方法。总体而言，Tare 成功修复了 62 个程序缺陷，比第二名（Recoder-T）多出 14.8%（8 个错误）。特别地，Tare 在精度方面也比 Recoder-T 提高了 7.3%。这些结果表明 Tare 成功地利用了类型关系来提高补丁的质量，而 Recoder-T 仅实现了 T-Grammar。图 4.7 展示了一个由 Tare 修复的独特缺陷：Recoder 生成了一个不兼容的变量，并将正确的补丁排除在搜索集合外；相反，Tare 在扩展 *Numeric* 类型的非终结符时不会生成 *boolean*。图 4.8 也展示了一个由 Tare 修复但 Recoder-T 未修复的错误：由于没有局部变量 “hasDecPoint” 和 “hasExp” 的类型信息，Recoder 无法生成正确的条件语句。结果表明了 Tare 的 T- Graph 的有效性。此外还可以观察到 Tare 的精度低于一些 APR 方法（Capgen, Hanabi）。本章分析原因是 Tare 专注于提高当前基于深度学习的 APR 的召回率，而其他方法专注于提高准确率。进一步

[①]一个通过所有测试用例的补丁。

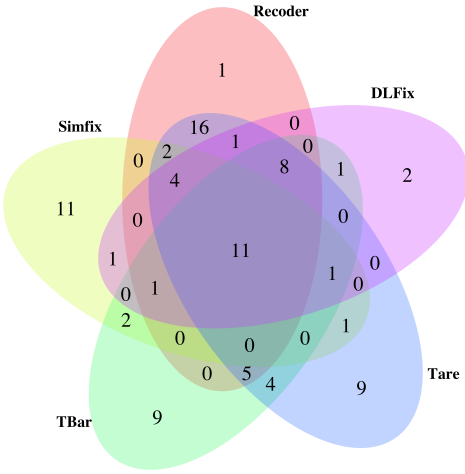


图 4.6　互补性

```
-  int j = 4 * n - 1;
+  int j = 4 * n - false;                    Recoder
+  int j = 4 * n - 4;                        Tare
```

图 4.7　Defects4J 中的 Math-80 的补丁

地，如表4.2所示，Tare 与在 Defects4J v1.2 上修复超过 30 个缺陷的其他 APR 工具具有相似的准确率。这种精度在实践中是可接受的，因为已经提出了许多方法[80-82]来处理假阳性的问题，并且能够减少至少一半的假阳性样例。

表 4.3　没有完美缺陷定位的结果比较

项目名称	缺陷数量	CapGen	SimFix	TBar	DLFix	Hanabi	Recoder	Recoder-F	Recoder-T	Tare
Chart	26	4/4	4/8	9/14	5/12	3/5	8/14	9/15	8/16	**11/16**
Closure	133	0/0	6/8	8/12	6/10	-/-	13/33	14/36	15/31	**15/29**
Lang	64	5/5	9/13	5/14	5/12	4/4	9/15	9/15	11/23	**13/22**
Math	106	12/16	14/26	18/36	12/28	19/22	15/30	16/31	16/40	**19/42**
Time	26	0/0	1/1	1/3	1/2	2/2	**2/2**	**2/2**	**2/4**	**2/4**
Mockito	38	0/0	0/0	1/2	1/1	-/-	**2/2**	**2/2**	**2/2**	**2/2**
总和	393	21/25	34/56	42/81	30/65	28/33	49/96	52/101	54/116	**62/115**
百分比 (%)	-	84.0	60.7	51.9	46.2	**84.8**	52.5	51.5	46.6	53.9

在单元格中，x/y 中的 x 表示正确补丁的数量，y 表示可以通过所有测试用例的补丁数量。

图 4.6 展示了 Tare 与其他最先进的 APR 工具，包括 Recoder、DLFix、TBar 和 Simfix 的互补性的详细分析。如图所示，Tare 在 *Defects4J v1.2* 上修复了 9 个与这些方法不同的独特错误。这表明 Tare 补充了现有的最先进 APR 方法。

4.6.1.2　完美缺陷定位下的性能

表 4.4 展示了几种 APR 工具在 *Defects4J v1.2* 上具有完美缺陷定位时的性能。如所示，在这一标准下，Tare 也实现了最佳性能，修复了 77 个缺陷。特别地，在这种设置

```
return foundDigit; }
+if((hasDecPoint || hasExp)){ +return false;}      Tare
```

图 4.8　Defects4J 中的 Lang-24 的补丁.

下，Recoder 额外处理了 3 个由 Tare 在没有完美定位的情况下修复的缺陷，说明 Tare 给出的正确补丁有更高的排名，同时能在时间限制之前得到验证，在一定程度上减轻了缺陷定位技术的影响。

表 4.4　提供完美缺陷定位的结果比较

项目名称	CoCoNuT	CURE	RRepair	Recoder	Recoder-F	Recoder-T	Tare
Chart	7	10	5	10	10	9	**11**
Closure	9	14	12	23	24	**25**	**25**
Lang	7	9	7	9	10	12	**14**
Math	16	19	18	19	20	20	**22**
Time	1	1	1	**3**	**3**	**3**	**3**
Mockito	**4**	**4**	2	2	2	2	2
总和	44	47	45	66	69	71	**77**

由于多个最先进的基于深度学习的自动程序修复工具仅在定位完美时提供正确的补丁，因此本章在此不列出相应的可能合理的补丁。

4.6.2　问题 2：Tare 的泛化能力

如 Durieux 等人[83]所发现的，“基准过拟合”是 APR 工具的一个常见现象，尤其是在 *Defects4J v1.2* 数据集上。为了展示 Tare 的泛化能力，本实验进一步在两个额外的基准数据集上评估了 Tare，这两个数据集分别是 *Defects4J v2.0* 的 444 个额外缺陷和 *Quixbugs* 的 40 个缺陷。此外还使用了 GZoltar 计算这些基准上每一行的可疑分数。

表 4.5 展示了 Tare 和其他 APR 工具的结果。本实验比较了在这些基准上评估过的几个最先进的基准模型。从表中可以观察到 Tare 在基准测试上超越了最先进的 APR 工具，在 *Defects4J v2.0* 的额外缺陷上有 33.3%（8 个缺陷）的提升，在 *QuixBugs* 上有 42.1%（8 个缺陷）的提升。这些结果突显了 Tare 的泛化能力，说明 T-Graph 学习到了由类型规则定义的类型信息，这对所有基准都普遍有用。

4.6.3　问题 3：正确补丁的排名

在这个研究问题中，为了探究改进的原因，本实验计算了在 *Defects4J v1.2* 上由 Recoder 和 Tare 修复的缺陷的正确补丁的排名。由于 Defects4J 的每个项目中都有多个缺陷，使用了不同项目的缺陷中正确补丁的最高、平均和最低排名。表 4.6展示了详细结果，其中 Tare 在所有项目的三个指标上几乎都比 Recoder 表现更好，除了 Closure 的最低排名外。对于平均排名，Tare 比 Recoder 实现了 44.7% 的改进。进一步地还可以

表 4.5　没有完美缺陷定位时在其他数据集上的结果比较

项目名称	Bugs	TBar	Recoder	Recoder-F	Recoder-T	RRepair	Tare
Cli	39	1/7	3/3	3/3	4/4	6/-	**5/13**
Clousre	43	0/5	0/7	0/8	0/6	**1/-**	0/5
JacksonDatabind	112	0/0	0/0	0/0	0/0	3/-	0/4
Codec	18	2/6	2/2	3/3	3/5	3/-	**3/7**
Collections	4	0/1	0/0	0/0	0/0	0/-	0/0
Compress	47	1/13	3/9	3/9	**4/12**	0/-	**4/13**
Csv	16	0/2	4/4	4/4	4/4	2/	**5/7**
JacksonCore	26	0/6	0/4	0/5	0/5	1/-	**2/7**
Jsoup	93	3/7	7/13	7/13	7/15	4/-	**10/16**
JxPath	22	0/0	0/4	0/4	0/6	**3/-**	2/10
Gson	18	0/0	0/0	**1/1**	**1/1**	**1/-**	**1/1**
JacksonXml	6	0/0	0	0/0	0/0	0/-	0/1
总和	444	8/50	19/46	21/50	23/58	24/-	**32/84**
Quixbugs	40	-/-	17/17	19/19	19/19	19/-	**27/27**

RRepair 的发布报告仅包括 Defects4J v2.0 中额外缺陷的正确补丁。因此，本章在此使用“-”。

表 4.6　正确补丁的排名的结果比较

项目名称	**Recoder**			**Tare**		
指标	最大值	最小值	平均值	最大值	最小值	平均值
Chart	708	1	172.5	503	1	132.1
Closure	1782	7	545.1	467	18	213.8
Lang	645	2	242.5	338	2	142.3
Math	919	1	251.2	761	1	183.7
Time	1995	846	1420.5	1248	582	915
Mockito	30	16	23	29	3	16
总和	1995	846	368.0	1248	582	203.6

表 4.7　补丁可编译率的结果比较

模型	**Top-30**	**Top-100**	**Top-200**
SequenceR[62]	33%	-	-
CoCoNuT[64]	24%	15%	6%-15%
CURE[78]	39%	28%	14%-28%
RRepair[79]	45.3%	37.5%	33.1%
Recoder[56]	43.5%	36.4%	34.2%
Tare	**54.6%**	**48.6%**	**46.7%**

观察到 Tare 在 Closure 上比 Recoder 表现出 60.8% 的提升，这是所有项目中最高的。本章分析原因是 Closure 的代码上下文比其他项目复杂得多。因此在没有类型信息的情况下，Recoder 倾向于生成更多不可类型化的补丁。

4.6.4　问题 4：补丁的可编译率

最后，为了了解 Tare 是否倾向于生成更多可编译的补丁，本实验计算了在完美缺陷定位时，前 k 个候选补丁的可编译率，其中 k 表示束大小。为了公平比较，遵循 RewardRepair[79]的做法，在评估中选择 $k = 30, 100, 200$，并使用相同的基准 *Defects4J v1.2* 和 *Quixbugs*。

表 4.7 展示了相关结果，其中直接使用了在论文[79] 中为 SequenceR、CoCoNuT、CURE 和 RewardRepair 报告的性能。对于 Recoder，本实验重新运行了作者提供的工具来计算可编译率。值得注意的是，Tare 倾向于生成比之前基于深度学习的方法更多的可编译补丁。总体而言，Tare 在 Top-30、Top-100 和 Top-200 的可编译率分别比之前的最先进工具高出 9.3%、11.1% 和 12.5%。由于 Recoder 的关键贡献是一个语法引导的解码器，它集成了语法约束，这意味着 Recoder 没有在神经网络中嵌入类型关系知识。RewardRepair 引入了一种语义训练方法，通过反向传播帮助神经模型学习相应的知识，当模型生成一个不可编译的补丁时，RewardRepair 通过在训练期间减少奖励来惩罚候选项。相反，Tare 直接将知识编码到编码器中，并使用 T-Grammar 的约束解码补丁。从表中可以观察到另一个现象：随着束大小的增加，Tare 的改进也增大了。这进一步证实了直接在编码器中编码类型规则的有效性，得益于类型规则的编码，模型为可编译的补丁估计了更高的概率。因此，随着束大小的增加，模型倾向于在束中保留更多可类型化的候选项。

4.7　本章小结

本章提出了一种针对类型系统信息的编码技术。为了整合个别类型规则的类型关系，我们将有缺陷的代码表示为一个异构图结构。此外，我们设计了一种新颖的语法

T-Grammar，将类型信息结合到标准的上下文无关语法中，使用 T-Grammar 生成的生成器直接预测部分生成程序的类型信息。最后，我们提出了一个关系感知的基于注意力的编码器 T-Graph Encoder，以嵌入 T-Graph 中包含的类型信息。我们在广泛使用的基准 Defects4J 和 QuixBugs 上进行了广泛的实验。结果表明 Tare 在所有基准上都优于现有的 APR 方法。对正确补丁排名和可编译率的进一步评估表明，Tare 能够从图中学习类型关系，并倾向于生成更多可编译的补丁。

第五章　声明-使用关系感知的学习

本章提出一种引导学习声明-使用关系的技术，通过将同一个项目的代码文件进行排列组合，来引导模型自主地学习声明-使用关系信息。这种技术的核心在于构建一个能够反映代码文件间相互依赖和关联组织的表示。这一表示不仅包括了代码文件之间的直接引用关系，比如函数调用、全局变量的引用等，也考虑了间接的上下文关系，例如模块的依赖、类的继承关系等。通过这种方式，本章可以创建一个包括引用上下文的学习单位，每个学习单位包括所有代码文件的完整的依赖上下文。

在这个基础上，为了进一步提高模型的泛化能力，本章引入了一系列的数据增强技术，包括但不限于随机删除某些引用关系、增加无关的代码文件、以及重排代码文件的顺序等。这些操作旨在模拟真实世界中代码编辑过程中可能出现的各种变化，帮助模型学习到更加鲁棒的上下文表示。

通过这种编码技术，不仅可以使模型在理解代码的结构和语义方面更加高效，还可以在一定程度上理解代码的功能意图以及实现方式。这对于一系列的代码理解和生成任务，如代码摘要、代码生成、缺陷检测等，都具有重要的意义。实验结果表明，采用本章提出的编码技术，模型在跨文件的补全任务上取得了性能的提升，充分证明了该技术的有效性和实用性。

值得注意的是，采用本技术预训练所产生的 DeepSeek Coder[84]代码预训练模型系列，能够在 16k 的上下文中学习到上下文引用的约束信息，提升代码补全和代码生成的准确性，是目前广泛采用的基座模型[14,16,85]，被应用在各种下游任务中。特别是其经过指令微调的版本，能够达到和 GPT-4 相当的水平，充分证明了本方法的有效性和先进性。

5.1　上下文依赖解析

在之前的研究中[6,8,12,86]，大型代码语言模型主要是在文件级别的源代码上进行预训练的，这忽略了项目中不同文件之间的依赖关系。然而，在实际应用中，这类模型在处理整个项目级别代码场景时面临着扩展困难。依赖信息的遗漏使得模型难以学习依赖于其他文件来推断的代码结构约束，如类型和方法调用目标。现有工作[27,87-91]表明，将相关信息组织在一起可以有效地协助模型学习代码的结构约束，从而提高其进行相关推断的能力。因此，本章将考虑如何在此步骤中利用同一存储库内文件之间的依赖关系。具体来说，本章首先解析文件之间的依赖关系，然后按照确保每个文件依赖的上下文在该文件输入序列之前的顺序来排列这些文件。通过根据文件依赖关系对

算法 1 依赖解析的拓扑排序算法

```
1:  procedure TopologicalSort(files)
2:      graphs ← {}                                  ▷ 初始化一个空的邻接矩阵
3:      inDegree ← {}                                ▷ 初始化一个空的入度哈希表
4:      for each file in files do
5:          graphs[file] ← []
6:          inDegree[file] ← 0
7:      end for
8:
9:      for each fileA in files do
10:         for each fileB in files do
11:             if HasDependency(fileA, fileB) then   ▷ 如果文件 A 依赖于文件 B
12:                 graphs[fileB].append(fileA)       ▷ 添加一条 B 到 A 的边
13:                 inDegree[fileA] ← inDegree[fileA] + 1   ▷ 增加 A 的入度
14:             end if
15:         end for
16:     end for
17:
18:     subgraphs ← getDisconnectedSubgraphs(graphs)   ▷ 识别互相不连接的子图
19:     allResults ← []
20:     for each subgraph in subgraphs do
21:         results ← []
22:         while length(results) ≠ NumberOfNodes(subgraph) do
23:             file ← argmin({inDegree[file] | file ∈ subgraph and file ∉ results})
24:             for each node in graphs[file] do
25:                 inDegree[node] ← inDegree[node] − 1
26:             end for
27:             results.append(file)
28:         end while
29:         allResults.append(results)
30:     end for
31:
32:     return allResults
33: end procedure
```

文件进行对齐，本章的数据集更准确地代表了真实的编码实践和结构。这种增强的对齐不仅使本章的数据集更加相关，还增强了模型在处理项目级代码场景时的实用性和适用性。值得注意的是，本章仅考虑文件之间的调用关系，并使用正则表达式来提取它们，例如 Python 中的 **“import”**，C# 中的 **“using”**，以及 C 中的 **“include “”**。

算法1描述了对同一项目中的文件列表进行依赖性分析的拓扑排序。最初，它设置了两个数据结构：一个用来表示文件之间依赖关系的空邻接表，命名为 **“graphs”**；一个用于存储每个文件的入度的空字典，命名为 **“inDegree”**。算法随后迭代每一对文件

以识别依赖关系，相应地更新 **“graphs”** 和 **“inDegree”**。接下来，它识别整个依赖图中的不连通子图。对于每个子图，算法采用了一种改进的拓扑排序。不同于标准方法选择入度为零的节点，此算法选择入度最小的节点，这使得它能够处理图中的循环。选定的节点被添加到一个名为 **“results”** 的列表中，并减少其连接节点的入度。这个过程持续进行，直到为每个子图生成一个拓扑排序的序列。算法运行结束后返回这些排序序列的列表，每个序列的文件被连接为单个训练样本。为了包含文件路径信息，在每个文件的开头添加注释指明文件路径，确保训练数据中保留了路径信息。

5.2　预训练数据集处理

为了验证本方法的有效性，本章收集了一个预训练数据集。该训练数据集由 87% 的源代码、10% 与代码相关的英语自然语言语料库以及 3% 与代码无关的中文自然语言语料库组成。英文语料库包含来自 GitHub 的 Markdown 和 StackExchange[①]的资料，这些被用来增强模型对代码相关概念的理解，以及提升其处理如库使用和错误修复等任务的能力。同时，中文语料库由旨在提高模型对中文语言理解能力的高质量文章组成。在本节中，本章将概述构建代码训练数据的过程。这个过程包括数据爬取、基于规则的过滤、依赖解析、仓库级去重以及质量筛选，如图5.1所示。接下来，本章将逐步描述数据创建过程。

图 5.1　数据收集的过程

本章收集了 GitHub 上 2023 年 2 月之前创建的公共仓库，并只保留了表5.1中列出的 87 种编程语言。为了减少需要处理的数据量，本章应用了类似于 StarCoder 项目[12]中使用的过滤规则，初步过滤掉了低质量的代码。通过应用这些过滤规则，本章将数据总量减少到了原始大小的 32.8%，具体包括以下规则：

首先，本章过滤掉了平均行长度超过 100 个字符或最大行长度超过 1000 个字符的文件。此外，本章还移除了字母字符少于 25% 的文件。除了 XSLT 编程语言外，本章进一步过滤掉了在前 100 个字符中出现字符串`"<?xml version="`的文件。对于 HTML 文件，本章考虑了可见文本与 HTML 代码的比例。本章保留了可见文本至少占代码 20% 并且不少于 100 个字符的文件。对于 JSON 和 YAML 文件，这些文件通常包含更多的

[①]https://stackexchange.com

表 5.1　数据集的语言分布

语言	大小 (GB)	文件数量 (k)	比例 (%)	语言	大小 (GB)	文件数量 (k)	比例. (%)
Ada	0.91	126	0.11	Literate Haskell	0.16	20	0.02
Agda	0.26	59	0.03	Lua	0.82	138	0.10
Alloy	0.07	24	0.01	Makefile	0.92	460	0.12
ANTLR	0.19	38	0.02	Maple	0.03	6	0.00
AppleScript	0.03	17	0.00	Mathematica	0.82	10	0.10
Assembly	0.91	794	0.11	MATLAB	0.01	1	0.00
Augeas	0.00	1	0.00	OCaml	0.91	139	0.11
AWK	0.09	53	0.01	Pascal	0.79	470	0.10
Batchfile	0.92	859	0.12	Perl	0.81	148	0.10
Bluespec	0.10	15	0.01	PHP	58.92	40,627	7.38
C	28.64	27,111	3.59	PowerShell	0.91	236	0.11
C#	58.56	53,739	7.34	Prolog	0.03	5	0.00
Clojure	0.90	295	0.11	Protocol Buffer	0.92	391	0.12
CMake	0.90	359	0.11	Python	120.68	75,188	15.12
CoffeeScript	0.92	361	0.12	R	0.92	158	0.11
Common Lisp	0.92	105	0.11	Racket	0.09	13	0.01
C++	90.87	36,006	11.39	RMarkdown	6.83	1,606	0.86
CSS	5.63	11,638	0.71	Ruby	15.01	18,526	1.88
CUDA	0.91	115	0.11	Rust	0.61	692	0.08
Dart	0.89	264	0.11	SAS	0.92	70	0.11
Dockerfile	0.04	48	0.00	Scala	0.81	971	0.10
Elixir	0.91	549	0.11	Scheme	0.92	216	0.12
Elm	0.92	232	0.12	Shell	13.92	10,890	1.74
Emacs Lisp	0.91	148	0.11	Smalltalk	0.92	880	0.12
Erlang	0.92	145	0.12	Solidity	0.85	83	0.11
F#	0.91	340	0.11	Sparql	0.10	88	0.01
Fortran	1.67	654	0.21	SQL	15.14	7,009	1.90
GLSL	0.92	296	0.11	Stan	0.20	41	0.03
Go	2.58	1,365	0.32	Standard ML	0.74	117	0.09
Groovy	0.89	340	0.11	Stata	0.91	122	0.11
Haskell	0.87	213	0.11	SystemVerilog	0.91	165	0.11
HTML	30.05	14,998	3.77	TCL	0.90	110	0.11
Idris	0.11	32	0.01	Tcsh	0.17	53	0.02
Isabelle	0.74	39	0.09	Tex	20.46	2,867	2.56
Java	148.66	134,367	18.63	Thrift	0.05	21	0.01
Java Server Pages	0.86	1072	0.11	TypeScript	60.62	62,432	7.60
JavaScript	53.84	71,895	6.75	Verilog	0.01	1	0.00
JSON	4.61	11956	0.58	VHDL	0.85	392	0.11
Julia	0.92	202	0.12	Visual Basic	0.75	73	0.09
Jupyter Notebook	14.38	2,555	1.80	XSLT	0.36	48	0.04
Kotlin	6.00	3,121	0.75	Yacc	0.72	67	0.09
Lean	0.52	68	0.07	YAML	0.74	890	0.09
Literate Agda	0.05	4	0.01	Zig	0.81	70	0.10
Literate CoffeeScript	0.01	3	0.00	**Total**	**797.92**	**603,173**	**100.00**

数据，本章只保留了字符数在 50 到 5000 个字符之间的文件。这有效地移除了大部分数据量较大的文件。

5.3　上下文保留的去重算法

近期的研究表明，通过对大型语言模型（LLM）的训练数据集进行去重，可以实现显著的性能提升。Lee 等人（2022）[92]展示了语言模型训练语料库常包含大量近似重复内容，通过移除长的重复子串可以增强 LLM 的性能。研究者采用了一种近似去重

(near-deduplication) 的方法处理训练数据[93]，结果显示性能有显著提升，并强调近似去重是实现代码基准任务竞争性能的一个关键预处理步骤。在本章的数据集中，本章也采用了近似去重技术。然而，本章的方法与先前的工作有所不同。本章在代码的仓库级别进行去重，而不是文件级别，因为后者可能会过滤掉仓库内的某些文件，潜在地破坏仓库的结构。具体来说，本章将仓库级别的拼接代码视为单一样本，并应用相同的近似去重算法，以确保仓库结构的完整性。

5.4　训练细节

5.4.1　训练策略

本模型的第一个训练目标被称为下一个词牌预测。在这个过程中，各种文件被连接起来形成一个固定长度的条目。然后，这些条目被用来训练模型，使其能够基于提供的上下文预测下一个词牌。

本模型的第二个训练目标称为中间填充。在代码预训练场景中，通常需要根据给定的上下文和后续文本生成相应的插入内容。由于编程语言中的特定依赖关系，仅依靠下一个词牌预测任务不足以学习这种中间填充能力。因此，已有一些工作[12,94]提出了中间填充 (Fill-in-the-Middle，FIM) 的预训练方法。该方法涉及将文本随机分为三部分，然后打乱这些部分的顺序，并用特殊字符连接它们。此方法旨在在训练过程中融入填空预训练任务。在 FIM 方法中，采用了两种不同的模式：PSM（前缀-后缀-中间）和 SPM（后缀-前缀-中间）。在 PSM 模式中，训练语料库按照（前缀, 后缀, 中间）的顺序组织，将中间段放在前缀和后缀之间。相反，SPM 模式将片段排列为（后缀, 前缀, 中间），呈现出不同的结构挑战。这些模式对于增强模型处理代码中各种结构安排的能力至关重要，为高级的代码预测任务提供了坚实的基础。为了确定 FIM 方法中各种超参数的有效性，本节进行了一系列消融实验。

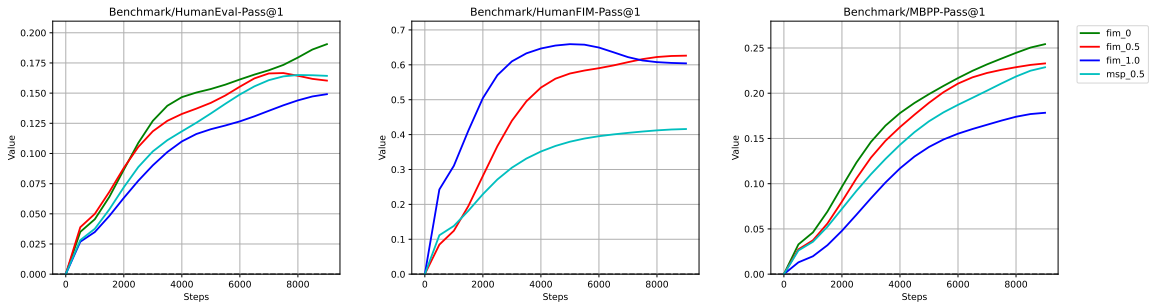


图 5.2　中间填充预训练任务的效果对比

实验设置：在此实验中采用 1.3B 的模型作为实验的基准模型架构。为了简化实

验过程，本实验聚焦于训练数据集中 Python 子集。本实验的主要目标是评估中间填充（FIM）技术的有效性，利用 HumanEval-FIM 基准测试[95]来进行评估。该基准测试专注于 Python 的单行 FIM 任务，在这个任务中，随机遮盖 HumanEval 这个数据集的正确解答中的一行代码，测试模型需要预测缺失代码行的能力。本文假设 SPM 模式相比于传统的下一个词牌预测目标可能会表现出细微差异，这主要是因为 SPM 涉及重新排列原始文本的顺序，可能会影响深度程序模型的学习。因此，本实验在四种不同配置下进行了实验：0% 的中间填充训练策略、50% 中间填充训练策略、100% 的中间填充训练策略和 50% 遮掩代码片段训练策略。遮掩片段预测（MSP）策略最初由 T5 提出[96]，该策略遮蔽多个文本片段并训练模型重建这些片段。根据 CodeGen2.5[97]，MSP 相比于 PSM 可能增强 FIM 性能。因此，在本实验的对比分析中纳入了该方法。

结果：实验结果如图 5.2 所示了，虽然模型在 HumanEval-FIM 测试中表现出 100% 的 FIM 率（即填充插入率），但这种配置也导致了最弱的代码补全能力。这表明在中间填充和代码补全能力之间存在一种权衡。此外，我们观察到，在 50% 的 PSM 模式的中间填充训练策略下，模型的表现优于片段遮掩策略。为了在中间填充代码能力和代码补全能力之间取得平衡，本模型的训练最终选择了 50% 的 PSM 模型的中间填充训练策略作为我们首选的训练策略。

在本模型的实现中，专门为此任务引入了三个哨兵词牌。对于每个训练语料，首先将其内容划分为三个部分，分别记为 f_{pre}、f_{middle} 和 f_{suf}。利用 PSM 训练策略，按照如下形式构造训练样本：

$$\texttt{<|fim\_begin|>}f_{pre}\texttt{<|fim\_hole|>}f_{suf}\texttt{<|fim\_end|>}f_{middle}\texttt{<|eos\_token|>}$$

在组织的过程中，本模型在文档级别上实现了中间填充的训练策略方法，按照 50% 的 PSM 的中间填充训练策略进行训练。

5.4.2 模型架构

为了验证本章方法的有效性，开发了一系列具有不同参数的模型，以满足多样化的应用需求，包括参数量为 1.3B、6.7B 和 33B 的模型。这些模型采用与 LLAMA[98] 中相同的架构，每个模型都是仅含解码器的 Transformer，并结合了旋转位置嵌入 (RoPE)[99]。值得注意的是，33B 的模型集成了组查询注意力 (GQA)，设置其组大小为 8，从而提高训练和推理的效率。同时本实验采用了 FlashAttention v2[100] 来加速注意力机制中的计算。本实验的模型的架构细节总结于表 5.2 中。

Hyperparameter	DeepSeekCoder 1.3B	DeepSeekCoder 6.7B	DeepSeekCoder 33B
Hidden Activation	SwiGLU	SwiGLU	SwiGLU
Hidden size	2048	4096	7168
Intermediate size	5504	11008	19200
Hidden layers number	24	32	62
Attention heads number	16	32	56
Attention	Multi-head	Multi-head	Grouped-query (8)
Batch Size	1024	2304	3840
Max Learning Rate	5.3e-4	4.2e-4	3.5e-4

表 5.2　Hyperparameters of DeepSeek Coder

5.4.3　长上下文优化

为了增强本模型处理扩展上下文的能力，特别是针对仓库级代码处理等复杂场景，本章重新配置了旋转位置编码（RoPE）参数，以扩展默认的上下文窗口。基于之前的做法[101-102]，本章采用了线性缩放策略，将缩放因子从 1 增加到 4，并将基频从 10000 改为 100000。这些调整旨在提升模型处理更长序列的能力，理论上可以处理高达 64K 词牌的上下文。

为了验证这一假设，模型在重新配置参数后，经过了额外的 1000 步训练，使用了批量大小为 512，序列长度为 16K 的设置。学习率保持在最终预训练阶段的水平，确保训练过程的稳定性和效率。通过这些调整，期望模型在处理大规模文本数据时能表现出更高的灵活性和可靠性。

尽管理论上这些修改使模型能够处理高达 64K 词牌的上下文，实证观察表明，模型在 16K 词牌范围内提供最可靠的输出。也就是说，虽然模型有能力处理更长的序列，但其最佳性能仍然在 16K 词牌范围内表现得最为稳定。未来的研究将继续探索和优化长上下文适应的方法论，旨在进一步提升模型在处理超长文本方面的效率和用户友好性。

此外，本章发现线性缩放策略在处理长序列时效果显著，但仍有进一步优化的空间。未来的工作将重点放在优化旋转位置编码参数和训练策略，以进一步提高模型的长上下文处理能力，满足更加复杂和多样化的应用需求。通过不断的实验和改进，希望能够为仓库级代码处理等高复杂度任务提供更强大和可靠的模型支持。

5.5　实验结果

本节将评估现有开源模型在跨文件代码补全任务中的性能。跨文件代码补全要求模型能够访问和理解跨越多个文件且具有众多跨文件依赖的仓库，本实验主要在两种补全模式下进行了测试，分别是中间填充方式和代码补全方式。

5.5.1 中间填充代码补全性能

本模型在预训练阶段采用了 50% 的中间填充策略进行训练。这种特殊的训练策略使模型能够根据给定代码片段的上下文（包括前缀和后缀）高效地生成代码，从而填补空白。这种能力在代码补全工具领域尤其有利。多个开源模型也具备类似的功能，其中值得注意的有 SantaCoder[103]、StarCoder[12]和 CodeLlama[8]。这些模型在代码生成和补全领域取得了不错的效果。为了评估本模型的性能，本文与上述模型进行了对比分析。此次比较的基准是由 Allal et al.[103]提出的单行填充基准测试，涵盖了三种不同的编程语言。该基准测试使用行精确匹配准确率作为评估指标。评估结果如表5.3所示。尽管

模型	大小	python	java	javascript	Mean
SantaCoder	1.1B	44.0%	62.0%	74.0%	69.0%
StarCoder	16B	62.0%	73.0%	74.0%	69.7%
CodeLlama-Base	7B	67.6%	74.3%	80.2%	69.7%
CodeLlama-Base	13B	**68.3**%	77.6%	80.7%	75.5%
DeepSeek-Coder-Base	1B	57.4%	82.2%	71.7%	70.4%
DeepSeek-Coder-Base	7B	66.6%	**88.1**%	79.7%	80.7%
DeepSeek-Coder-Base	33B	65.4%	86.6%	**82.5**%	**81.2**%

表 5.3 Performance of different approaches on the FIM-Tasks.

参数量仅为 1.3B 的最小模型，在这些基准测试中，其表现也优于更大规模的 StarCoder 和 CodeLlama 模型。这些模型都没有使用跨文件的预训练组织方式，因此这种性能上的提升主要依赖于 DeepSeek-Coder 所采用的跨文件的训练组织策略以及恰当的中间填充的训练策略。此外，观察到的一个显著趋势是模型大小与性能之间的相关性。随着模型规模的增加，性能也相应地显著提升。这一趋势强调了模型容量在实现更高代码补全准确度中的重要性。

5.5.2 自回归代码补全性能

本章使用 CrossCodeEval[104]来评估当前可用的 7B 规模开源代码模型在跨文件补全任务中的能力。这个数据集是基于四种流行编程语言（Python、Java、TypeScript 和 C#）中的真实世界、开源、许可宽松的仓库构建的，具有多样性。该数据集是专门为严格要求跨文件上下文以实现准确补全而收集的。值得注意的是，这个数据集是从 2023 年 3 月至 6 月创建的仓库中构建的，而本章的预训练数据仅包含 2023 年 2 月之前创建的代码，这确保了该数据集不在本章的预训练数据中，从而避免了数据泄露。

在进行模型性能的综合评估时，本研究特别设定了一些关键参数来确保评估的准确性和一致性。首先，本章将模型处理能力的上限设定为能够理解和生成的最大序列长度，即 2048 个词牌。这意味着模型在处理任何代码片段时，能够考虑的最大字符数

表 5.4　不同模型在补全数据集上的性能.

模型	大小	Python		Java		TypeScript		C#	
		EM	ES	EM	ES	EM	ES	EM	ES
CodeGeex2	6B	8.11%	59.55%	7.34%	59.60%	6.14%	55.50%	1.70%	51.66%
+ Retrieval		10.73%	61.76%	10.10%	59.56%	7.72%	55.17%	4.64%	52.30%
StarCoder-Base	7B	6.68%	59.55%	8.65%	62.57%	5.01%	48.83%	4.75%	59.53%
+ Retrieval		13.06%	64.24%	15.61%	64.78%	7.54%	42.06%	14.20%	65.03%
CodeLlama-Base	7B	7.32%	59.66%	9.68%	62.64%	8.19%	58.50%	4.07%	59.19%
+ Retrieval		13.02%	64.30%	16.41%	**64.64%**	12.34%	60.64%	13.19%	63.04%
DeepSeekCoder-Base	6.7B	9.53%	61.65%	10.80%	61.77%	9.59%	60.17%	5.26%	61.32%
+ Retrieval		**16.14%**	66.51%	**17.72%**	63.18%	**14.03%**	**61.77%**	**16.23%**	**63.42%**
+ Retrieval w/o Repo Pre-training		16.02%	**66.65%**	16.64%	61.88%	13.23%	60.92%	14.48%	62.38%

为 2048。此外，模型生成代码的能力也受到限制，最大输出长度被设定为 50 个词牌，确保生成的代码既精准又简洁。除此之外，本研究还引入了跨文件上下文的概念，最大限度为 512 个词牌，这允许模型在进行代码补全或修复等任务时，能够参考与当前文件相关联的其他文件内容，从而提高预测的准确性和相关性。

为了实现跨文件上下文的效果，本章采用了官方提供的 BM25 搜索结果作为参考。BM25 是一种广泛使用的信息检索算法，能够有效地评估文件与查询之间的相关性，这对于模型在处理跨文件任务时，能够准确找到与当前任务最相关的信息至关重要。

评估模型性能时，采用了精确匹配和编辑相似度这两个关键指标。精确匹配衡量模型生成的代码是否与标准答案完全一致，而编辑相似度则评估了生成代码与标准答案之间的字符级差异，从而提供了一个更细致的性能衡量指标。

评估结果显示，在处理跨文件代码完成任务时，DeepSeekCoder 模型展现出了卓越的性能，一致地超越了其他对比模型。这一结果不仅证明了 DeepSeekCoder 在理解和生成代码方面的能力，也展示了其在不同编程语言中的强大适应性和实际应用能力。

然而，当模型仅使用文件级别的代码语料库进行预训练，不包括更广泛的仓库级别数据时，本章观察到在 Java、TypeScript 和 C# 等语言中，模型的性能有所下降。这一发现强调了结合了本章提出的上下文引用约束的编码方法，能够提高模型对于跨文件上个下文引用等约束信息的学习，在程序补全和程序生成任务中的重要性，表明了包含更广泛、更丰富上下文的训练数据能够显著提升模型的表现。

5.6　本章小结

本章通过设定一系列关键技术，引入跨文件上下文的概念，使得模型能够在预训练的同时学习到上下文引用的约束信息。本章在 DeepSeekCoder 模型上进行了对照实验来对当前方法进行综合评估，展示了本方法能够提升模型学习上下文约束的能力，尤

其是在不同编程语言中的强大适应性和实际应用能力。

当模型仅限于使用文件级别的代码语料库进行预训练，未能包含更广泛的完整上下文约束级别数据时，其性能在某些语言中出现下降。这一发现强调了本章提出的上下文引用约束编码方法的重要性，表明了包含更丰富上下文的训练数据对于提升模型在程序补全和生成任务中的表现具有显著影响。

总结来说，本章的研究不仅提供了一种评估模型在代码处理任务中性能的有效方法，也指出了在模型训练中考虑更广泛上下文信息的重要性，为未来在该领域的研究提供了有价值的见解和指导。

第六章　语言定义感知的多语言深度程序预训练模型

前文所提到的三种引导神经网络学习语言定义信息的方式在单种程序语言的应用环境下适配性较强，但现实的场景中，开发人员往往处于多种程序语言交织的应用环境中，如何将这些技术结合应用于多语言的应用场景，是本章提出的技术所着重解决的问题。因此，本章首先针对不同程序语言的标识符系统做了语法适配，结合自然语言领域的字节对分词技术统一了不同语言的终结符的表现。其次，本章提出为每种编程语言的非终结符附加唯一的语言后缀来帮助模型区分不同程序语言的语言定义的信息。最后，本章针对结合程序定义的语法规则表示形式提出了对应的新颖的预训练任务，通过大量语料的训练和特定任务的学习，帮助模型学习语言定义的信息。

6.1　动机介绍

近期，人工智能领域经历了显著的进步，这主要得益于预训练模型的开发和部署。预训练模型通过在大规模数据集上执行自监督任务进行训练。在自然语言处理[51,105-107]和编程语言处理[7,9,91,108-111]的领域中，如 T5[107]和 CodeT5[7]这样的大型预训练语言模型显著超越了现有的非预训练模型。

因此，为了将前文提出的技术应用于多编程语言的场景，本章提出应用预训练技术，将三种语言定义的信息通过大量数据的预训练来和神经网络进行结合。要实现这个目的并不容易，因为语言定义信息适配到预训练模型中存在几个技术挑战。**第一个挑战是大词汇表**。在典型的编程语言语法中，一些终结符如 ⟨*identifier*⟩ 或 ⟨*constants*⟩ 代表许多可能的词法词牌，称为多值终结符。现有方法收集训练集中由多值终结符表示的词牌，并为每个收集到的词牌添加一个语法规则，例如 *identifier* → *isodd*。然而，在预训练中，训练集要大得多，添加这样的规则会导致词汇量过大。现有的预训练模型使用字节对编码（BPE）[112]来找到一组相对较小的子词牌，其连接能代表大量的词牌集。然而，如何将 BPE 与语法序列结合起来仍然不清楚。**第二个挑战是异质性语法**。现有的非预训练模型只处理一种编程语言，但本章提出的预训练模型需要同时编码多种编程语言，如何处理来自不同编程语言的语法尚未知晓。**第三个挑战是预训练任务**。预训练需要训练的自监督任务。当程序表示为语法规则序列时，应使用什么预训练任务尚未知晓。特别是，由于语法结构被显式表示，预训练任务应指导模型学习程序的语法结构。

因此，本章提出了 GrammarT5，一个集成了语言定义信息的多语言程序编解码器预训练模型，用于编程语言的程序理解和生成任务。GrammarT5 使用一种变体的语法

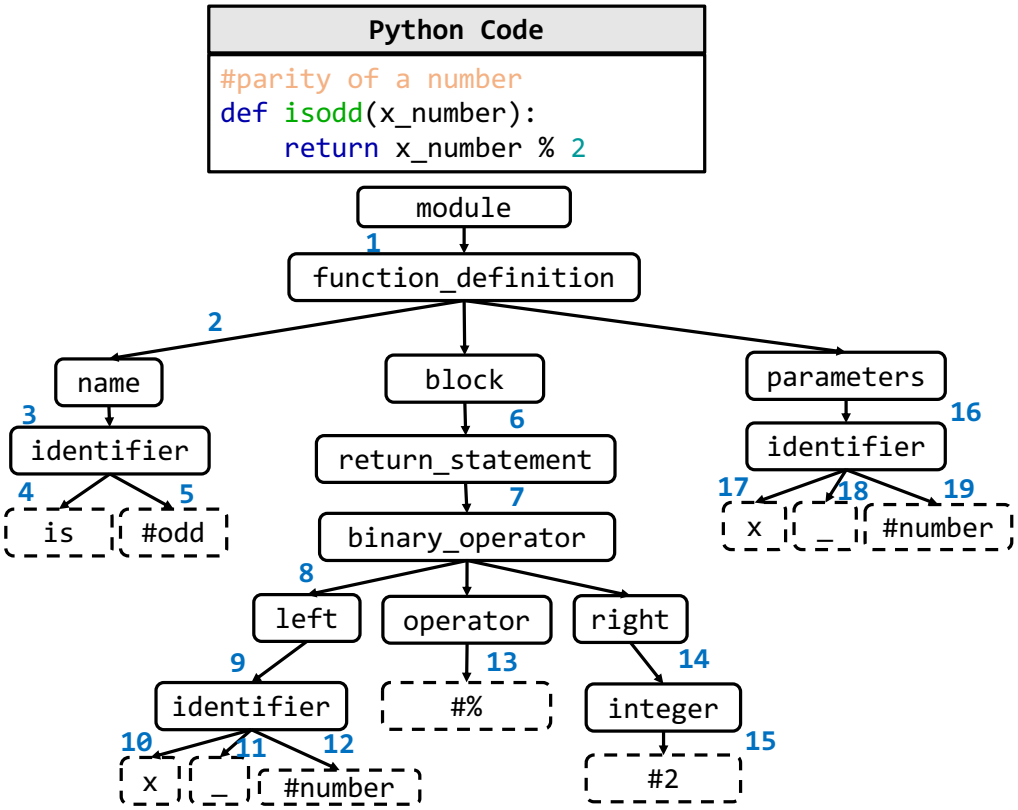


图 6.1　一段 Python 程序和对应的 tokenized AST[①]

规则序列来表示程序，称为 *Tokenized Grammar Rule Sequence (TGRS)*。为了解决第一个挑战，**本章的第一个贡献**是将 BPE 与语法规则序列的表示集成。BPE 使用一组子词牌来代表原始词牌。例如，`is` 和 `odd` 可以用来代表 `isodd`。然后可以通过以下规则扩展语法。

$$\textit{identifier} \rightarrow \texttt{is}\ \textit{identifier} \mid \texttt{odd}\ \textit{identifier} \mid \cdots$$

$$\textit{identifier} \rightarrow \varepsilon$$

然而，这种语法规则在抽象语法树（AST）中为每个具有多值终结符的叶节点引入了一个额外的节点 ε，这非平凡地增加了表示的长度。由于紧凑的表示往往会带来更好的模型性能，因此通过在最后一个子词后添加一个特殊标志 “#” 来重写第二条语法规则：

$$\textit{identifier} \rightarrow \texttt{\#odd} \mid \texttt{\#number} \cdots$$

图6.1展示了新语法中的 AST 示例，称为 *tokenized AST*。从图中可以看出，有五个节点以 “#” 开头，表明原始表示将使语法规则序列大小增加 5/19=26

为了应对第二个挑战，**本章的第二个贡献**是对不同编程语言的语法规则组合效果的实证研究。主要考虑是不同的编程语言可能有相同的语法规则，这种情况可以共享

[①]为了更好的说明，这里省略了 AST 中的一些 *identifier* 节点。

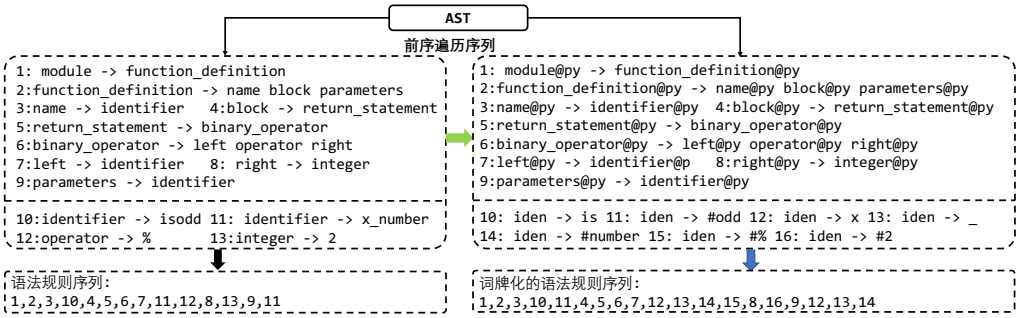


图 6.2　用语法序列和 TGRS 表示的 Python 程序

语法规则。由于上下文无关语言在并集下是封闭的，一种可能的方法是通过组合所有语法并尽可能共享语法规则来构建一个新的语法。虽然这种方法可能减少总的语法规则数量，但它可能会给神经模型学习这种混合语法中的句法信息带来挑战。另一种可能的方法是不共享任何语法规则。为了实现这一点，本方法提出为每种编程语言的非终结符附加一个特殊的语言标志，以禁止共享。例如，图6.1中的符号 *return_statement* 将被修改为 *return_statement@py*。通过实验比较了这两种方法，发现后者能带来更高的性能。

为了克服第三个挑战，**本章的第三个贡献**是两个新的预训练任务，边缘预测（*EP*）和子树预测（*STP*），以便 GrammarT5 学习 AST 的结构信息。首先，EP 要求模型在给定编码器中的 TGRS 的情况下，逐步在解码器中预测父节点。其次，STP 受到遮蔽跨度预测的启发，这是一个用于词牌序列的去噪任务。这个任务是预测输入序列中随机遮蔽的任意长度程序片段。由于其随机性，遮蔽的跨度可能会破坏输入程序的结构完整性。因此，本章提出了 STP，它随机遮蔽 AST 中的几个子树。STP 要求 GrammarT5 基于周围的上下文重新生成遮蔽的子树，以学习程序依赖性。这两个预训练任务能够指导模型有效地捕获程序的结构信息。

本章的第四个贡献是一系列全面评估 GrammarT5 在程序相关任务上性能的实验。这些实验总共花费了 50 天，并在 11 个数据集上对五项任务进行了测试，包括两项理解任务：程序搜索和注释生成，三项生成任务：程序生成、程序翻译和程序改进。为了与相同规模的最新状态预训练模型 CodeT5[7]进行比较，本章使用 CodeT5 的训练集的一个子集来训练 GrammarT5，并将所有超参数设置为与 CodeT5-base 相同。结果显示，与同等规模的模型（包括 CodeT5-base）相比，GrammarT5 在大多数任务上都达到了最新的最佳性能。此外，与规模大 3 倍的 CodeT5-large 相比，GrammarT5 也展现出了有竞争力的性能。进一步分析揭示，GrammarT5 的所有上述提出的技术都增强了该模型的性能。

6.2 GrammarT5

GrammarT5 是一个集成了语言定义信息的预训练模型，用于多模态数据（编程语言（PL）和自然语言（NL））的程序理解和生成。该模型基于 T5[107]的编码器-解码器框架，并旨在生成给定由 PL 和 NL 组成的输入的通用表示。

6.2.1 终结符的词牌化

如图 6.2所示，现有语法引导的解码器方法[27,43,88,113]直接使用唯一的语法规则代表 AST 中的标识符，例如 $identifier \rightarrow isodd$。这些方法都在小训练集上进行了实验，它们没有暴露于潜在的大词汇表的问题。然而，当转向具有大规模程序语料库的预训练场景时，由于标识符数量巨大，将暴露出大词汇表的问题。

Karampatsise 等人[114]和王等人[7]提出的方法通过在预训练中使用 BPE（字节对编码）算法[115]来缓解将程序表示为词牌序列的问题。这种算法使得单词能够被表示为子词列表。然而，这种方法的一个缺点是缺乏语法信息。

为了利用这两种方法的优势，本章对语法中的所有终结符进行了分词，并通过扩展具有这些终结符的新规则的语法获得了序列化的 AST。例如，对于图6.1中的程序，添加了以下语法规则，其中每个终端都被标记化为子词牌（即，“isodd” / “x_number” 被标记化为 “is” 和 “odd” / “x”、“_” 和 “number”）。

$$identifier \rightarrow \texttt{is}\ identifier \mid \texttt{odd}\ identifier \mid \cdots$$
$$identifier \rightarrow \texttt{\#odd} \mid \texttt{\#number} \cdots$$
$$operator \rightarrow \texttt{\#\%} \quad integer \rightarrow \texttt{\#2}$$

为了识别最后一个子词，本方法在它前面附加了一个特殊标志 “#”。

图 6.2展示了如何将 “parameters” 子树转换为四个不同的语法规则：*parameters*→*identifier*、*identifier*→x *identifier*、*identifier*→_ *identifier* 以及 *identifier* → #number。每个语法规则被分配一个唯一的 ID。例如，*parameters*→*identifier* 表示为 9。因此，词牌”x_number” 被表示为一系列数字：12、13、14。这些数字随后通过词嵌入转换为实值向量。这些向量作为神经网络的输入。这种方法有效地压缩了程序输入的长度，使其更易于模型处理，同时仍然保留了必要的语法信息。

6.2.1.1 语言标志

GrammarT5 旨在处理来自多种编程语言的程序。不同的编程语言可能共享相同的语法规则。例如，Java 和 Python 的语法都包含规则 $name \rightarrow identifier$。直接合并不同编程语言的语法可能会阻碍学习独特的语法信息。为了研究这一效应，这里通过训练

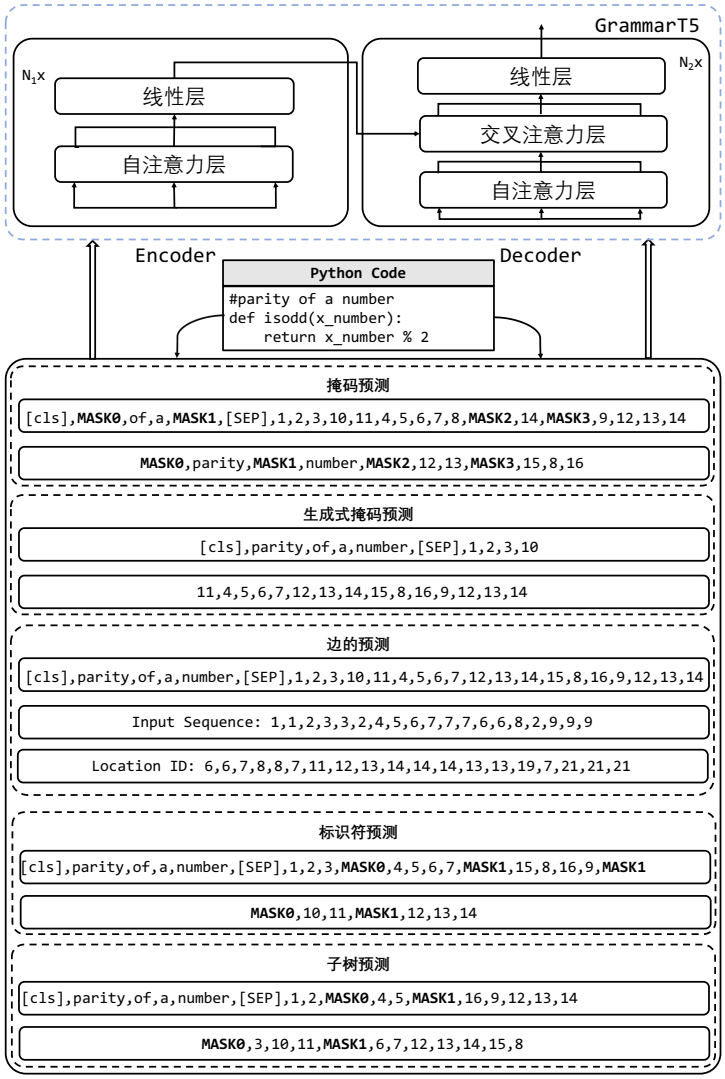


图 6.3　GrammarT5 架构总览

两个独立的模型进行了实证研究。其中一个模型使用 tree-sitter 的语法[116]，它是一个为所有主流语言设计的解析器生成工具。在 tree-sitter 定义的语法中，许多规则已经在不同的编程语言之间共享。另一个使用扩展的语法，在 tree-sitter 语法中的非终结符上附加一个特定的编程语言标志。为了防止引入过多额外的语法规则，本方法只修改原有的非终结符，而不是那些为多值终结符引入的非终结符。如图 6.2所示，语法规则中的每个非终结符后都附加了一个标志 $@py$ 以识别语言。同时，本章在几个下游任务上评估了这两个模型，并发现后者实现了更高的性能。

6.2.2　模型架构

图 6.3展示了 GrammarT5 的模型架构。GrammarT5 采用了与 T5 类似的编码器-解码器框架来处理输入。每个组件包含 N 个基于自注意力层的 Transformer 块。该层输入

三个嵌入 $\mathbf{q}, \mathbf{k}, \mathbf{v}$，并基于输入计算的注意力分数输出组合嵌入 $\mathbf{o}$。

在解码器中，每个块相比于编码器多了一个额外的编码器-解码器注意力层。其计算可以表示为 $EncAtt = Att(\mathbf{b}, \mathbf{e}, \mathbf{e})$，其中 $\mathbf{b}$ 表示解码器中前一层的输出，$\mathbf{e}$ 表示编码器的输出。

6.2.3 预训练任务

本节描述在 GrammarT5 中使用的预训练任务。如图 6.3所示，通过 5 个自监督任务对 GrammarT5 进行预训练，包括在预训练模型中使用的三个常规预训练任务和两个新提出的去噪目标。这些任务旨在使 GrammarT5 能够从仅 PL 或 NL-PL 双模态数据中学习句法和语义信息。

6.2.3.1 遮蔽程序片段预测

去噪预训练目标已被证明对于编码器-解码器模型非常有效，如 PLBART 和 CodeT5。这个目标通常首先使用某种噪声函数污染原始序列，然后要求模型恢复序列。最常用的去噪目标之一是**遮蔽跨度预测（MSP）**。这项任务随机遮蔽输入序列中的片段，这些片段长度任意。模型应该基于损坏的输入生成这些遮蔽的片段。受此现象启发，本方法在图 6.3所示的多模态数据上使用了类似的去噪目标。

具体来说，本方法使用与先前工作[7,107]中相同的遮蔽率，即 15%。此外，通过从 1 到 5 个词牌均匀采样片段来控制遮蔽长度的平均长度为 3。随后将这些遮蔽的片段通过几个特殊词牌 $MASK_i$ 连接为输出，其中 i 表示跨度的 ID，如图 6.3所示。这里将其表示为 $\mathbf{y} = y_0, y_1, \cdots, y_n$。因此，遮蔽跨度预测的损失可以计算为

$$\mathscr{L}_{MSP}(\theta) = \sum_{i=1}^{n} -log\mathscr{P}_\theta(y_i|\mathbf{x}^{mask}, \mathbf{y}_{t<i}) \tag{6.1}$$

其中 θ 是可训练参数，$\mathbf{x}mask$ 是损坏的输入序列，$\mathbf{y}t < i$ 表示到目前为止生成的序列。

6.2.3.2 生成遮蔽预测

虽然 MSP 任务有利于程序理解任务，但它与需要生成整个序列的程序生成目标有很大差异。为了解决这个问题，本方法采用了在仅解码器模型中使用的类似预训练目标[51]用于 GrammarT5。

具体来说，本方法在输入序列中随机选择一个枢轴位置。然后给定前面的序列，GrammarT5 预测后续序列，如图 6.3所示。当前的实现中，确保枢轴位置落在输入序列

的 10%到 90%之间，以控制序列长度。这个预训练任务的损失可以表示为：

$$\mathscr{L}_{GMP}(\theta) = \sum_{i=1}^{|x^{suc}|} -log\mathscr{P}_{\theta}(x_i|\mathbf{x}^{pro}, \mathbf{x}^{suc}t < i) \tag{6.2}$$

其中 $\mathbf{x}^{suc}$ 表示后续序列，$\mathbf{x}^{pro}$ 表示前序序列。

6.2.3.3 遮蔽标识符预测

在编程语言中，符号信息的重要性对于理解程序语义至关重要，特别是程序中的标识符。因此，本方法采用了一个去噪目标，遮蔽程序中的所有标识符，遵循 CodeT5[7]的方法，用唯一的哨兵词牌 $\boldsymbol{MASK}_i$ 替换输入序列中的第 i 个标识符的所有实例。然后通过连接所有唯一标识符及其哨兵词牌（如图 6.3所示）来构建目标序列。然后解码器以自回归方式从损坏的序列中预测目标序列。损失计算为：

$$\mathscr{L}_{IP}(\theta) = \sum_{i=1}^{|\mathbf{y}|} -log\mathscr{P}_{\theta}(y_i|\mathbf{x}^{MI}, \mathbf{y}t < i) \tag{6.3}$$

其中 $\mathbf{x}^{MI}$ 是遮蔽的输入序列，$\mathbf{y}_{t<i}$ 表示到目前为止生成的序列。

为了帮助 GrammarT5 学习程序特有的结构信息，本方法提出了两个额外的预训练任务：边预测和子树预测。

6.2.3.4 边预测

当将程序片段转换为 TGRS 时，一些关键的结构信息可能会丢失。现有方法提出了仅编码器模型的边缘遮蔽技术[91,108]，通过注意力得分预测遮蔽的边。受这些方法的启发，本方法提出了一个边的预测任务。解码器应该基于给定序列预测每条规则的父规则。这里使用指针网络来预测原始序列中父规则的位置，并使用父规则序列作为解码器的输入。然后，GrammarT5 应该基于父规则序列输出父规则的位置，如图 6.3所示。

给定编码器的输出 $\mathbf{o}$、解码器的输出 $\mathbf{d}$ 和父规则位置序列 $\mathbf{l}$，损失计算为：

$$\mathscr{L}EP(\theta) = \sum i = 1^{|\mathbf{x}|} -log(p_{i,l_i}), p_{i,j} \quad = \frac{exp(\mathbf{o}j\mathbf{d}i)}{\sum_{k=1}^{n} exp(\mathbf{o}k\mathbf{d}i)} \tag{6.4}$$

6.2.3.5 子树预测

由于程序具有树状结构，最普遍的去噪目标遮蔽片段预测，不加选择地在序列内遮蔽跨度，可能会损害其结构完整性。为了解决这个问题，这里提出了一个新的去噪目标，子树预测任务。

为了损坏原始序列，这个目标随机遮蔽几个子树，并使用特殊词牌 $\boldsymbol{MASK}_i$ 替换子树的 TGRS 。在当前的实现中，每个子树都有一个一致的遮蔽率 15%。为了控制输

入和输出长度，本方法限制每个遮蔽子树的长度范围为 10-60，且遮蔽子树的总长度构成输入序列长度的不到 15%。为了创建输出序列，本方法用特殊词牌连接所有遮蔽的子树。因此这个目标的损失计算可以表示为：

$$\mathscr{L}_{SP}(\theta) = \sum_{i=1}^{|\mathbf{y}|} -log\mathscr{P}_{\theta}(y_i|\mathbf{x}^{SP}, \mathbf{y}_{t<i}) \tag{6.5}$$

其中 $\mathbf{x}^{SP}$ 表示遮蔽的输入。在这个目标中，模型需要使用上下文重构遮蔽的子树，可能有助于它吸收 AST 的结构信息。

6.2.3.6 聚合操作

GrammarT5 的预训练方法包括循环执行五个不同的任务。在每一步中，从这个池中随机选择一个任务，确保每个任务有相等的被选中机会，并且对模型的学习贡献均等。预训练过程的总损失可以表示为：

$$\begin{aligned} \mathscr{L}(\theta) &= \sum_{i=1}^{5} p_i \mathscr{L}_i(\theta) \\ \mathbf{p} &= OneHot(Random(1,5)) \end{aligned} \tag{6.6}$$

其中 $\mathscr{L}_i$ 表示上述五个预训练任务，$\mathbf{p}$ 表示基于随机整数的一热向量。

6.3 实验设置

6.3.1 研究问题

本节的实验旨在回答以下研究问题：

问题 1: GrammarT5 与现有的预训练模型相比，表现如何？为了回答这个问题，本节在包含 11 个基准的 5 个程序相关任务上比较了 GrammarT5 与现有的预训练模型。

问题 2: TGRS 表示对模型性能的影响如何？为了回答这个问题，本节训练了一个对比的模型，在该模型中，用 TGRS 替换了 CodeT5 的程序表示，以比较 TGRS 与词牌序列。同时，本节比较了由 TGRS 表示的预训练数据集中程序的平均长度与词牌序列和 AST 序列的长度，这两种表示是现有预训练模型中使用的。

问题 3: 语法规则中包含语言标志对模型性能的影响如何？为了回答这个问题，本节训练了一个对照模型，使用没有语言标志的 tree-sitter 合并语法（这样不同编程语言中的通用规则是共享的），并将其与 GrammarT5 的原始版本在基准测试上进行比较。

问题 4: 提出的预训练任务对模型性能的影响如何？为了回答这个问题，本节训练了两个去掉对应预训练任务的 GrammarT5，并将它们与 GrammarT5 的原始版本在基准测试

表 6.1　使用的预训练数据集的统计信息

	CodeSearchNet		GithubCode	总和
Statistics	**Java**	**Python**	**CSharp**	-
存在自然语言描述的数量	457,380	453,750	422,457	1,333,587
不存在自然语言描述的数量	1,070,265	656,990	581,873	2,309,128
规则数	963	1105	1913	3981
总和	1,527,645	1,110,740	1,004,330	3,642,715

上进行比较。

问题 5: GrammarT5 对词牌序列的泛化能力如何？本节评估了 GrammarT5 对诸如程序补全这样的任务的适应性，这些任务涉及不可解析的部分程序。为了回答这个问题，本节在程序完成和程序翻译上评估了 GrammarT5，将不可解析的程序视为词牌序列进行处理。

6.3.2　预训练数据集

为了消除训练集对模型性能可能的影响，本节选择了同一规模下的最新状态预训练模型 CodeT5[7]的训练集，以公平地与 CodeT5 进行比较。然而，由于时间和计算资源的限制，这里只能使用训练集的一个子集来训练 GrammarT5。请注意，这种设置有利于 CodeT5，因为通常认为更多的训练数据会导致更好的模型性能。

CodeT5 是在 CodeSearchNet 数据集[117]和 GitHub 程序数据集[118]中的 C 和 CSharp 数据上训练的。这里选择 CodeSearchNet 中的 Java 和 Python 数据以及 GitHub 程序数据集中的 CSharp 数据来预训练 GrammarT5。表6.1展示了所使用的预训练数据集的详细统计信息。总共，本章使用了大约 364 万个实例来预训练 GrammarT5。

6.3.3　下游任务与评价指标

为了评估 GrammarT5 的性能，本节采用了 CodeXGLUE 基准测试[110]，这是一个用于程序编解码模型的基准数据集和开放挑战。它包含了一系列用于模型评估和比较的程序理解和生成任务。为了公平比较，这里按照先前工作[110]的方法使用相同的数据分割。此外，本节还考虑了四个额外的程序生成基准测试——MBPP[119]，Django[120]，MathQA-Python[119]，以及 Conala[121]——以进一步评估 GrammarT5 的程序生成能力。这些数据集的统计信息显示在表 3.1中。本节在这些基准测试上执行了 GrammarT5 五次，每次使用不同的随机种子，以确保结果的稳健性。这些下游任务被分为两大类：程序理解和程序生成。

6.3.3.1 程序理解

这部分实验专注于两个跨模态的下游任务：程序总结和程序搜索。

程序总结旨在给定一个函数级程序片段时生成一个自然语言(NL)描述。CodeXGLUE数据集包括 6 种编程语言。这个实验选择了 Java 和 Python 的子集来进行实验。按照现有工作[7]的方法，本节使用平滑的 BLEU-4（4-grams 的双语评估替代者）得分[122]来评估性能。BLEU 得分通过计算与真实文本相比的 n-gram 准确率得分的几何平均值来衡量生成文本的质量。

程序搜索旨在基于自然语言功能描述识别最具语义相关性的程序片段。本节在两个数据集上进行实验，即 AdvTest[110]和 CosQA[123]。AdvTest 是从 CodeSearchNet 的 Python 子集中构建的，过滤了低质量的查询。测试集规范化了 Python 函数和标识符，以更好地评估模型的泛化能力。CosQA 的程序库也来源于 CodeSearchNet 语料库，自然语言查询是从 Microsoft Bing 搜索引擎日志中收集的。在这项任务中，本节使用平均倒数排名（MRR）进行评估。MRR 是评估排名任务的指标。它计算第一个正确答案的倒数排名的平均值。MRR 越高，模型在将相关答案排名更高方面就越好。

6.3.3.2 程序生成

在程序生成中主要关注三个相关任务：基于自然语言的程序生成、程序优化和程序翻译。

基于自然语言的程序生成旨在从自然语言描述生成程序片段。本章采用三个常用的基准测试：Concode[124]、Django[120]和 CoNaLa[121]。Concode 考虑了自然语言描述和类环境上下文，Django 包括了与自然语言描述配对的 Django Web 框架的 Python 程序行，而 CoNaLa 特色是 `Stack Overflow` 上的自然语言问题和 Python 解决方案。本章使用 BLEU-4、完全匹配准确率（EM），即程序与真实文本完全相同的词牌序列的百分比；以及 CodeBLEU（C-BLEU)[125]来评估性能，后者基于数据流图考虑了语法和语义匹配。此外还评估了 GrammarT5 在 MBPP 数据集和 MathQA-Python 上的程序合成能力。第一个数据集包含 974 个用 Python 编写的编程问题，每个问题有 3 个单元测试。第二个是生成 Python 程序来解决自然语言描述的数学问题，其中程序正确性基于生成程序的执行输出来衡量。本节遵循现有工作[6,126]的做法，使用 $pass@k$ 指标来评估 GrammarT5，测量通过生成每个问题 k 个程序解决问题的百分比。

程序优化将一个有缺陷的函数转换为一个正确的函数。本节使用 Tufano 等人[127]提供的两个 Java 基准测试。这两个基准测试有不同的函数长度。Refine-small 的词牌数量较少（< 50 个词牌），而 Refine-medium 的词牌数量较多（50-100 个词牌）。这里使用与程序生成相同的指标来评估性能。

程序翻译是将一种编程语言的程序转换成另一种语言的程序。这里使用 CodeTrans 数据集[21]，其中包含了 CSharp 和 Java 的相互匹配对。本节对这项任务使用与程序生成相同的评价指标。此外，CodeXGlue 原始数据集在评估 BLEU 和 CodeBLEU 指标时没有采用特定于语言的词牌化，这可能阻碍了这些指标准确反映模型的性能。因此，这里修改了这两个指标的评估脚本。

6.3.4 比较模型

尽管一些程序预训练模型（CodeGen-6B[86]，Incoder-6B[95]）表现出了极强的性能，但它们的大小要大得多（30 倍）且由于计算资源限制难以在下游基准测试上进行微调。因此将 GrammarT5 与四个类别中大小相当的多种预训练模型进行比较：仅编码器、仅解码器、编解码器和统一编码器。对于**仅编码器**模型，这里考虑了 CodeBERT[9]，通过掩码语言建模和替换词牌检测进行训练；GraphCodeBERT[91]，使用程序中的数据流图进行训练；SynCoBERT[108]，采用 AST 来学习语法信息。对于**仅解码器**模型，这里考虑了 GPT-C[109]，在大量 Java 语料上训练，以及 CodeGPT-adapted[110]，使用 GPT-2 的参数在程序上训练。除了这两个模型，这里还考虑了两个更大的模型，CodeGen-Multi-350M 和 CodeGen-Multi-2B[86]，以确定 GrammarT5 是否仍然能够超越具有更多参数和数据的序列模型。对于**编解码器**模型，这里采用了 CodeT5[7]，使用标识符感知掩码去噪目标预训练的编解码模型；PLBART[128]，使用 BART[129]架构在程序上训练；以及 CoTexT，使用 T5[107]架构在程序上训练；CodeT5+[130]，在更多数据和训练目标上进行训练。对于**统一编码器**模型，这里选择了 Unixcoder[111]，采用统一编码器来包含上述三种模型风格的功能。此外，对于基于自然语言的程序生成，这里还将 GrammarT5-base 与领先的非预训练程序生成模型 *TreeGen+Grape*[90]进行了比较。

大多数模型都是在 CodeSearchNet 上预训练的，除了 GPT-C、PLBART、CodeT5、CodeGen 和 CodeT5+。GPT-C 使用大规模的数据集预训练，该数据集包含 12 亿行 Python、CSharp、JavaScript 和 TypeScript 的源程序。PLBART 使用更大的数据集，包括 4.7 亿个 Python 函数和 2.1 亿个 Java 函数，以及 Stack Overflow 上的 4700 万个自然语言帖子，超过了 CodeSearchNet 的规模。CodeT5 额外包括了从 GitHub 程序数据集提取的 C-Sharp 和 C 语料，以确保覆盖下游任务中使用的所有编程语言。CodeGen 和 CodeT5+ 采用了来自 BigQuery 数据集的更大训练语料，包括 GitHub 上约 115M 个程序文件，涵盖 32 种编程语言。如前所述，由于本方法使用的预训练集是 CodeT5 的子集，当与 CodeT5 比较时，最小化了预训练集的影响。

表 6.2　程序理解任务的结果比较

	程序总结		程序搜索	
子任务	**Java**	**Python**	**Adv**	**CosQA**
指标 / 模型	**BLEU**	**BLEU**	**MRR**	**MRR**
CodeBERT(110M)	17.25	19.06	27.20	65.90
GraphCodeBERT(110M)	18.93	19.39	35.20	68.55
SynCoBERT(110M)	18.89	18.74	38.30	69.19
GPT-C(110M)	17.18	17.78	24.39	50.32
CodeGPT-adapted(110M)	17.68	18.46	25.97	54.24
CodeGen-multi (350M)	19.41	18.31	35.47	69.22
CodeGen-multi (2B)	20.01	19.31	36.47	70.22
CoTexT(220M)	19.19	19.72	34.13	68.70
PLBART(220M)	18.45	19.30	34.70	65.01
CodeT5-small(60M)	19.92	20.04	30.52	66.74
CodeT5-base(220M)	20.31	20.01	39.30	67.80
CodeT5-large(770M)	**20.74**	**20.57**	42.11	71.29
CodeT5+ (220M)	20.31	20.01	43.3	72.7
Unixcoder(110M)	19.42	18.64	41.30	70.10
GrammarT5-small(60M)	19.93±0.10	19.78±0.11	37.24±0.26	70.34 ±0.12
GrammarT5-base(220M)	20.66±0.16	20.21±0.12	**44.12±0.10**	**73.48±0.05**

表 6.3　基于自然语言的程序生成任务的结果比较

数据集	**Concode**			**Conala**		**Django**		**MBPP**	**MathQA**
指标 / 模型	**BLEU**	**EM**	**C-BLEU**	**BLEU**	**EM**	**BLEU**	**EM**	**pass@80**	**pass@80**
TreeGen + Grape(35M)	26.45	17.60	30.05	20.16	2.80	75.86	77.30	2.00	26.58
GPT-C(110M)	30.85	19.85	33.10	30.32	4.80	72.56	68.91	10.40	58.94
CodeGPT-adapted(110M)	35.94	20.15	37.27	31.04	4.60	71.24	72.13	12.60	55.90
CodeGen-multi (350M)	38.23	21.25	40.57	33.14	5.70	74.45	74.23	23.50	62.10
CodeGen-multi (2B)	41.23	22.25	44.57	40.14	9.40	81.45	84.04	32.50	83.10
CoTexT(220M)	19.19	19.72	38.13	31.45	6.20	75.91	78.43	14.00	58.18
PLBART(220M)	36.69	18.75	38.52	32.44	5.10	72.81	79.12	12.00	57.25
CodeT5-small(60M)	38.13	21.55	41.39	31.23	6.00	76.91	81.77	19.20	61.58
CodeT5-base(220M)	40.73	22.30	43.2	38.91	8.40	81.40	84.04	24.00	71.52
CodeT5-large(770M)	42.66	22.65	45.08	39.96	7.40	82.11	83.16	32.40	83.14
CodeT5+(220M)	34.13	22.16	43.45	38.91	8.00	78.45	**85.21**	28	85.6
Unixcoder(110M)	38.73	22.65	40.86	36.09	10.20	78.42	75.35	22.40	70.16
GrammarT5-small(60M)	38.08±0.36	21.05±0.36	40.62±0.56	38.18±0.46	8.20±0.36	80.64±0.36	82.27±0.46	25.00±0.86	83.91±0.46
GrammarT5-base(220M)	**43.30±0.86**	**25.10±0.75**	**45.48±0.35**	**41.92±0.56**	**10.40±0.15**	**82.40±0.56**	84.17±0.16	**33.00±0.20**	**87.26±0.32**

6.4　实验结果

6.4.1　问题 1: GrammarT5 的有效性

在本节中，将 GrammarT5 与当前最先进的预训练模型在程序理解和程序生成任务上进行比较。如果模型已经在基准测试上进行了评估，则直接使用原始论文中的结果。否则，本节将使用开源的程序在相应的基准测试上运行预训练模型。

表 6.4　程序优化和程序翻译任务的结果比较

程序优化							程序翻译					
子任务	Small			Medium			Java to CSharp			CSharp to Java		
指标 / 模型	BLEU	EM	C-BLEU	BLEU	EM	C-BLEU	BLEU	EM	C-BLEU	BLEU	EM	C-BLEU
CodeBERT(110M)	78.41	16.40	78.09	86.94	5.20	83.88	85.23	62.10	86.87	85.81	61.80	85.19
GraphCodeBERT(110M)	**79.61**	17.3	**79.68**	87.63	9.10	85.33	86.35	63.10	87.6	86.50	62.10	85.18
SynCoBERT(110M)	78.81	20.32	78.56	88.37	11.17	87.05	87.04	65.10	88.26	87.80	65.20	86.81
GPT-C(110M)	70.06	13.03	71.83	85.41	8.26	82.47	78.90	61.90	81.02	84.48	60.70	83.87
CodeGPT-adapted(110M)	76.07	13.66	77.13	85.28	11.00	84.55	82.11	62.90	83.45	85.68	61.30	84.98
CodeGen-multi (350M)	78.12	20.12	77.23	88.21	13.12	86.84	90.21	66.40	90.24	89.75	65.50	88.65
CodeGen-multi (2B)	79.52	22.12	79.23	89.21	14.12	88.84	91.41	68.40	91.44	89.75	70.60	88.65
CoTexT(220M)	77.28	21.33	77.38	87.13	13.03	85.14	85.57	66.70	86.25	87.21	65.80	87.11
PLBART(220M)	77.02	19.40	77.58	88.48	8.98	86.67	87.95	67.80	88.12	87.19	67.70	87.01
CodeT5-small(60M)	76.23	19.06	76.44	89.20	10.92	87.25	88.23	65.40	88.32	87.22	69.60	87.18
CodeT5-base(220M)	77.43	21.61	77.24	87.64	13.96	87.05	88.55	66.90	88.72	87.03	68.70	86.71
CodeT5-large(770M)	77.38	21.7	77.14	89.22	14.76	87.35	88.89	67.20	88.98	87.20	68.80	87.16
CodeT5+(220M)	78.27	22.18	77.48	88.64	**15.13**	86.28	**91.66**	66.20	**91.51**	89.64	70.20	91.01
Unixcoder(110M)	79.18	19.05	79.45	87.59	13.96	86.23	90.20	67.00	90.15	90.51	70.60	90.32
GrammarT5-small(60M)	76.90±0.15	20.50±0.35	76.98±0.15	89.11±0.35	12.63±0.15	86.86±0.25	91.15±0.15	67.80±0.25	90.29±0.54	89.20±0.25	71.60±0.45	89.49±0.15
GrammarT5-base(220M)	79.39±0.05	**22.60±0.26**	78.88±0.16	**90.57±0.24**	15.32±0.40	**89.18±0.18**	91.31±0.13	**69.10±0.26**	91.11±0.28	**90.53±0.05**	**73.40±0.12**	**91.26±0.38**

6.4.1.1　程序理解

表 6.2 比较了预训练模型在程序总结和程序搜索任务上的表现。GrammarT5-base 在类似大小的模型中表现最好，在 Adv 和 CosQA 的程序搜索中分别获得了最高的 MRR 得分 43.98 和 73.58，表明其在生成程序总结和检索相关程序片段方面的准确性。GrammarT5-small 尽管参数较少，但在两项任务中均显示出强大的性能。值得注意的是，CodeGen-multi-2B（仅解码器）尽管参数量是 GrammarT5 的 10 倍，但性能仍然较低，这突出了双向和语法信息在程序理解中的重要性。

6.4.1.2　基于自然语言的程序生成

这里将 GrammarT5 与仅解码器、编解码器和统一编码器模型进行比较，因为仅编码器模型对于基于自然语言的程序生成来说效果不佳。表 6.3 显示，GrammarT5-base 在所有基准测试中均优于其他模型。对于 Conala 和 Django 基准测试，GrammarT5 超过了其他模型。Concode 基准测试要求从程序上下文生成函数片段，这是更具挑战性的，但 GrammarT5-base 模型在完全匹配准确率上实现了 2.45 点的提升，并在 CodeBLEU 上增加了 0.4 点，超过了此前领先的模型 CodeT5-large。虽然这些改进可能看起来很小，但鉴于任务的复杂性，它们是显著的。考虑到目前被视为最先进的、对现有发布方法的改进的程序生成机器学习模型，例如 CodeT5-large[131]，只在完全匹配准确率上比 CodeT5-base[7]提高了 0.35 点。

MBPP 数据集旨在用于 Python 程序合成，目的是生成全面的 Python 程序以通过指定的测试。MBPP 的训练集规模较小，仅 374 个，有效地评估了预训练模型在程序生成中的泛化能力。如表 6.3 所示，GrammarT5-base 显著优于 CodeT5-base 9.2 点，并与 CodeT5-large 持平。GrammarT5-small（60M）变体在 MBPP 数据集上表现良好，即使是四倍于其大小的 CodeT5-base，也超过了 2 点。在 MathQA-Python 任务中，GrammarT5

超越了其他预训练模型，比 CodeT5-large 有 4.32 点的优势。GrammarT5-base 模型在解决复杂问题方面表现尤为出色，对于需要超过 10 个推理步骤的问题，通过率达到了 63.27%，远高于 CodeT5-base 的 21.23%。尽管 CodeGen-multi-2B 和 CodeT5+ 有更多的训练数据，GrammarT5 的表现一致更好，突显了语法和结构信息的重要性。

为了探索以语法方式表示程序的必要性，本节分析了在 MBPP 数据集上不同大小的基线模型生成的 40,000 个程序的语法正确性。表 6.5 显示了由各种模型生成的程序中语法错误的出现情况。可以看出，大小为 220M 的模型大约有 10％或更多的程序含有语法错误，其中一些甚至包括了不正确的 Python 缩进。即使是 2B 模型，错误率仍然为 6%。此外，这些模型所犯的大多数错误涉及混合不同语言的语法。例如，在为 MBPP 数据集生成的 Python 程序中，有许多表达式如 “if (i == 0 && j == 0 or k == 0):”、“return (first_char ? (first_char - 1) : (last_char + 1))” 和 “if(num > 10){”。由于词牌序列并未明确告知模型相应的编程语言，而在语法序列中，通过语法规则明确指示了语言的语法。此外，本方法通过生成语法树，永远不会产生具有不正确缩进的程序。最后，与 CodeGen-multi-2B 相比，在本模型表现正确而 CodeGen 未能做到的问题中，有 62% 涉及 CodeGen 生成语法不正确的程序。因此，通过 GrammarT5 表示程序可以显著缩小程序的搜索空间，并增加生成正确程序的概率。

表 6.5　MBPP 数据集上的语法错误类型统计

错误类型 / 模型	**Syntax Error**	**Indentation Error**	**Tab Error**	**错误率**
CodeT5(220M)	5142	389	15	13.87%
CodeT5+(220M)	4392	294	5	11.73%
CodeT5-large(770M)	4123	229	2	10.89%
CodeGen-multi(2B)	2312	92	7	6.03%
GrammarT5	0	0	0	0%

6.4.1.3　程序到程序的生成

本节将 GrammarT5 与其他预训练模型在两个程序到程序生成任务上进行比较：程序优化和程序翻译。在程序优化任务中，源程序与目标程序之间的大量重叠可能会导致高 BLEU 分数但完全匹配数为零。因此，本节专注于这项任务的完全匹配（EM）指标。如表 6.4 所示，GrammarT5-base 在两项任务上均超过了所有基线，包括 CodeT5-large。考察 GrammarT5-small 模型，与同一大小类别的其他模型相比，它在所有任务和指标上也展现出了稳定的性能。

在程序翻译任务中，主要将 GrammarT5 与 CodeT5、CodeT5+ 和 CodeGen-multi 进行比较，这些模型都已在包含 CSharp 的语料库上进行了预训练。在这种情况下，Gram-

marT5-base 在两个不同子任务的完全匹配（EM）指标上分别超过 CodeGen-multi-2B 0.7 和 2.8 点。值得注意的是，仅有 600 万参数的 GrammarT5-small 超过了大约有 7700 万参数的 CodeT5-large 模型，后者几乎大 11 倍。

总之，与具有相同或更小大小的现有模型相比，GrammarT5 在大多数指标和任务上表现更好。此外，与更大的模型 CodeT5-large 和 CodeGen-multi-2B 相比，GrammarT5-base 实现了非常相似甚至更好的性能。这些结果表明，将程序表示为语法规则序列是有益的，且 GrammarT5 中使用的新颖技术是有效的。

表 6.6　GrammarT5 的消融实验

模型 \ 数据集	Adv MRR	Java-CSharp EM	CSharp-Java EM	MBPP pass@80
GrammarT5-small	**37.24±0.26**	**67.80±0.25**	**71.60±0.45**	**25.00±0.86**
w/o EP	35.41±0.22	66.80±0.32	71.00±0.16	23.80±0.60
w/o STP	34.03±0.36	66.60±0.22	69.60±0.28	22.40±0.44
w/o LF	35.26±0.21	67.20±0.29	69.40±0.26	23.60±0.36
CodeT5-small	30.52	65.40	68.8	19.20
CodeT5-small + TGRS	33.27±0.12	66.10±0.46	70.10±0.36	22.40±0.13

6.4.2　问题 2: TGRS 的有效性

为了强调 TGRS 的有效性，本节训练了一个删减版的 CodeT5-small 模型，唯一的改变是将表示转换为 TGRS。这个模型与 GrammarT5-small 之间的主要区别在于预训练目标，这个模型使用与原始论文相同的预训练任务。如表 6.6 所示，删减模型在所有基准测试中均优于 CodeT5-small。这些结果指出，与词牌序列相比，TGRS 能以更有组织的方式封装关键的语法信息，从而提升了在这种特定表示上预训练的模型的性能。

为了进一步理解 TGRS 的效果，本节计算了预训练数据集中不同编程语言的平均输入长度，这些编程语言由不同的表示方式表示。如表 6.7 所示，TGRS 表示在词牌序列的简洁性和 AST 的全面结构之间找到了一个平衡。TGRS 提供了比 AST 更紧凑的表示，这对于需要高效处理和减少内存使用的任务来说是有益的。

表 6.7　编程语言使用不同表示的平均输入长度

表示方法	Java	CSharp	Python
TGRS	199.20	300.67	242.32
AST	427.66	583.49	476.99
词牌序列	169.58	247.67	186.27

6.4.3 问题 3: 语言标志的有效性

如6.2.1.1节所述，结合语法规则有两种方法，即是否在不同语言之间共享相同的产生规则。为了理解它们的差异，本节训练了一个删减版本的 GrammarT5-small，使用 tree-sitter 的原始组合语法而不使用语言标志。

如表 6.6 所示，没有语言标志的 GrammarT5-small 在四个任务上的性能下降。这一结果表明，GrammarT5 难以从混合语法规则中学习语法信息。为模型提供额外的特定于语言的信息，如本方法使用的语言标志，以区分不同语言，将更为有效。

6.4.4 问题 4: 预训练任务的有效性

为了评估所提出的预训练任务的有效性，本方法通过消融实验来检验它们的贡献。由于计算资源的限制，这里只去掉两个去噪目标：边预测（EP）和子树预测（STP），在四个选定的任务上比较 GrammarT5-small。

如表 6.6 所示，当移除每个组件时，所有任务的性能都有所下降，这证明了模型中每个组件的重要性。移除 STP 在所有任务中导致性能最大的下降，表明子树预测目标对模型效果有显著影响。其他组件，如 EP 和 LF，虽然对总体性能的贡献较小，但也是不可忽视的。总之，消融实验强调了每个组件都提升了 GrammarT5 的性能。

```
public static List shunting_yard(ArrayList tokens) {
    ……
    ArrayList rpntokens = new ArrayList(100);
    ArrayDeque opstack = new ArrayDeque();
    for (Object token: tokens) {
        if (Integer.class.isInstance(token)) {
            rpntokens.add((Integer) token);
        } else {
            String operator = (String) token;
        while (!opstack.isEmpty() && precedence.get(operator)
<= precedence.get(opstack.getLast())) {
            rpntokens.add(opstack.pop());
        }
        _mask_line_;
    }
  }
  ……
```

```
GrammarT5-base:
opstack.push(operator);

Incoder-6B:
opstack.addLast(operator);


Codegen-6B:
rpntokens.add(operator);
opstack.push(operator);


ChatGPT:
opstack.push(operator);

GroundTruth:
opstack.push(operator);
```

图 6.4　QuixBugs 中的 SHUNTING_YARD 漏洞.

具体而言，这里观察到 STP 预训练任务可以应用于零样本设置下的程序填充下游任务。图 6.4 展示了 QuixBugs 基准测试中的一个真实错误。人工编写的补丁在占位符“_mask_line_”处插入了一个 push 操作。本节将 GrammarT5-base 与三个能够在指定位置生成程序的预训练模型进行比较。如示例所示，Incoder-6B 和 Codegen-6B 无法生成正确的语句，而 GrammarT5-base 产生的程序与 ChatGPT 相同，尽管其大小（220M）要小得多。这里假设这个目标可以帮助模型学习子树及其上下文之间的结构关系。这个

目标还赋予了 GrammarT5 基于上下文完成程序的能力，表明 GrammarT5 对程序补全和程序修复的潜在应用。

表 6.8　使用词牌序列的实验结果

	程序补全		程序翻译	
子任务	**Python**	**Java**	**Java-CSharp**	**CSharp-Java**
指标 / 模型	**EM**	**EM**	**EM**	**EM**
GPT-C(110M)	38.37	28.60	61.90	60.70
CodeGPT-adapted(110M)	42.37	30.60	62.90	61.30
CodeGen-multi (350M)	42.47	35.47	66.40	65.50
CodeGen-multi (2B)	**46.32**	**40.47**	68.40	70.60
PLBART(220M)	38.01	26.97	**67.80**	67.70
CodeT5-small(60M)	19.92	20.04	65.40	69.60
CodeT5-base(220M)	36.97	24.80	66.90	68.70
CodeT5-large(770M)	38.74	28.57	67.20	68.80
CodeT5+ (220M)	43.42	35.17	66.20	70.20
Unixcoder(110M)	43.12	32.90	67.00	**70.60**
GrammarT5-small(60M)	38.93±0.42	28.54±0.24	65.12±0.46	67.21 ±0.23
GrammarT5-base(220M)	43.75±0.22	34.32±0.36	67.12±0.23	70.23±0.25

6.4.5　问题 5: 词牌序列的泛化能力

程序补全这样的任务经常涉及部分的、语法不正确的程序，将其转换为 TGRS 是具有挑战性的。然而，本章发现每个程序词牌都被封装在 TGRS 序列中。本章假设在处理像程序补全这样的任务时，直接使用词牌序列进行微调是可行的。为此，本章测试了 GrammarT5 在 CodeXGlue 的行级别程序补全任务上的泛化能力。本章还在程序翻译数据集上的文本序列上进行了实验，以比较与 TGRS 相比的性能差异。

如表 6.8 所示，当程序通过词牌序列表示时，GrammarT5 显示出与相当大小的基线相似的有效性水平。使用 TGRS 时，程序翻译任务的性能有所下降。这种差异表明，虽然 GrammarT5 在处理涉及词牌序列的任务时是可靠和有效的，但其与 TGRS 的结合显著提高了其性能。此外，这一观察为未来的进展打开了有趣的可能性。词牌序列表示与 TGRS 的整合，可能通过开发额外的预训练目标，可以产生更复杂和有效的模型，本章建议这作为未来研究的一个方向。

6.5　本章小结

总结而言，本章提出了 GrammarT5，这是一个预训练的程序编解码模型，能够处理多种程序语言的混合的程序语言的定义信息。该模型采用了编码器-解码器框架，该

模型结构和 T5 相同，使用了两个独立的 Transformer 模块。本文提出了一种新颖的程序表示方法 TGRS，它在预训练中有效地将程序表示为语法规则序列。此外，本文还引入了二个预训练任务，旨在帮助模型学习 GrammarT5 中的多种编程语言的语言定义的信息。通过在十个数据集，包含五种代码相关任务上进行评估，证明了 GrammarT5 在所有任务上相较于同规模的模型都达到了最先进的性能。进一步地，消融研究表明，本章提出的技术均有效地提升了模型性能。

第七章　深度程序模型在程序搜索上的应用

本章针对程序搜索这个任务的特点，提出使用特定的重叠度矩阵来编码程序的语义相关性，以此提升程序编解码模型在该任务上的性能。

7.1　程序搜索的任务特点

程序搜索是软件工程中的一个重要问题，其目的是通过给定的自然语言描述，在一组程序片段中检索出最相关的程序片段。一个有效的程序检索器可以帮助开发人员从互联网上重用程序片段。例如，如果一个 SQL 程序员给出指令“获取表 A 中的所有数据”，程序检索器将帮助程序员在互联网上的大量程序中搜索并找到目标程序“select * from A”。

随着深度学习的发展和大规模标注数据集的收集，神经网络已被广泛应用于各个领域[54,132-133]。对于检索任务，已经提出了应用神经网络的各种方法[132-137]。这些方法主要是将问题和答案嵌入到高维向量空间中，并尝试找到问题向量与答案向量之间最相似的一个（例如，使用余弦相似度）。当应用于程序搜索时，它将自然语言描述视为问题，将目标程序视为答案[133,137]。

然而，这些检索方法未能有效处理重叠的信息，而重叠信息在程序检索中是非常重要的。一方面，不同的开发者可能会使用不同的名称来描述类似的含义，无论是在程序中还是在自然语言中，这些名称通常具有重叠的子字符串。例如，“Sort”和“QuickSort”有重叠的子字符串“Sort”。另一方面，程序中的标识符往往与自然语言描述中的单词相关，尽管它们可能不完全相等，但经常存在重叠的子字符串。例如，在图 7.1中，标

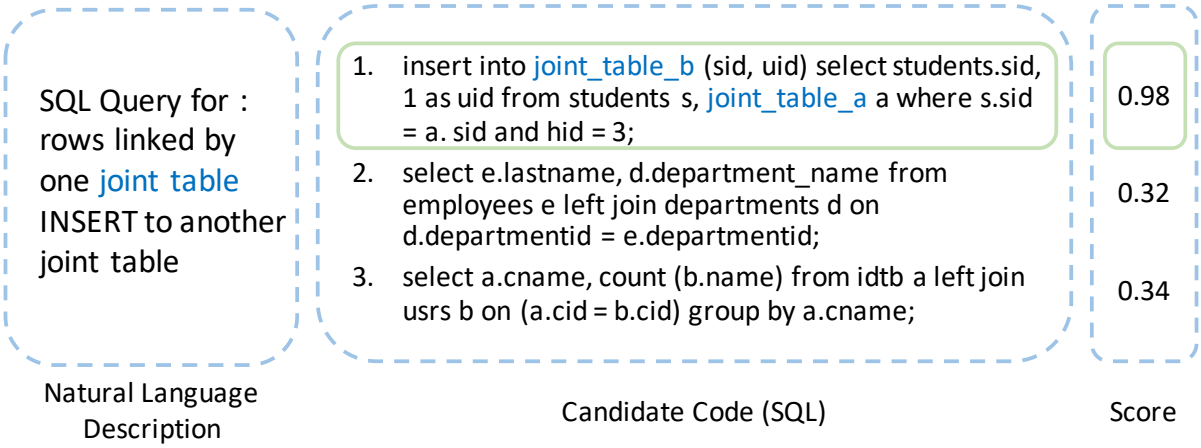
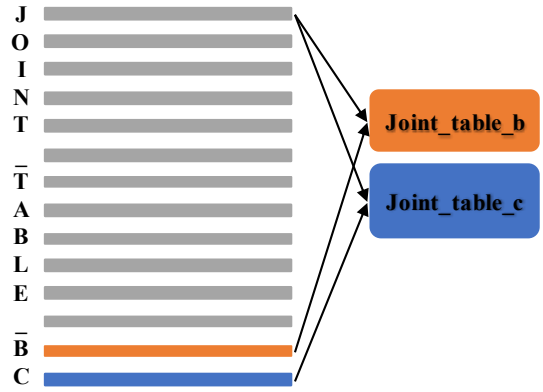


图 7.1　StaQC 数据集中的示例：根据模型输出的分数对候选的程序片段排序



(a)

	Rows	linked	by	one	joint	table
Insert	0	0.3	0	0.17	0.3	0.17
Into	0.25	0.5	0	0.25	0.75	0.25
Joint_table_b	0.08	0.16	0.08	0.16	0.4	0.4
sid	0.3	0.3	0	0	0.3	0

(b)

图 7.2　字符级嵌入和重叠矩阵的计算示例

识符 “joint_table_b” 与单词 “joint” 和 “table” 相关。到目前为止，没有现有的神经架构是专门为处理重叠而设计的。为了解决这些问题，本章提出了一种新颖的神经架构，OCoR，这是一个基于重叠特征的程序搜索结构。通过结合其中字符的表示来表示每个单词，即使用字符级嵌入来捕捉不同程序员使用的名称之间的重叠。此外，本章引入了一个新颖的重叠矩阵来表示自然语言描述中每个单词与程序中每个标识符之间重叠的程度。最后通过集成不同的程序搜索方法来增强本模型。

性能的验证实验和之前 Yao et al.,Iyer et al.[137-138]的论文一致，是在几个已建立的 SQL 和 C# 程序搜索数据集上进行的。实验结果显示，本方法提出的的模型相比已有工作提升了 13.1% 到 22.3% 的性能，并在所有数据集上达到了最佳性能。为了更好地理解本技术，本章还设计了对照实验，测试提出的技术组件的有效性，结果显示每个技术组件都对整体性能有所贡献。

总结来说，本章做出了以下贡献：

- 本章提出了一种新颖的神经架构，OCoR，用于程序搜索。OCoR 使用两种新颖的技术，即字符级嵌入和重叠矩阵，来捕捉程序中的标识符与自然语言描述中的单词之间的重叠信息。
- 本章进行了广泛的实验来评估本方法的有效性。结果显示，本方法显著优于现有方法，在测试数据集上提升了 13.1% 到 22.3%，并且本方法中的所有组件都是有效的。

7.2　方法动机

正如上文中提到的，程序任务中有两种类型的重叠信息。第一种类型的重叠信息是不同的开发者可能使用不同的名称来描述相似的含义（例如，自然语言中的单词和程序

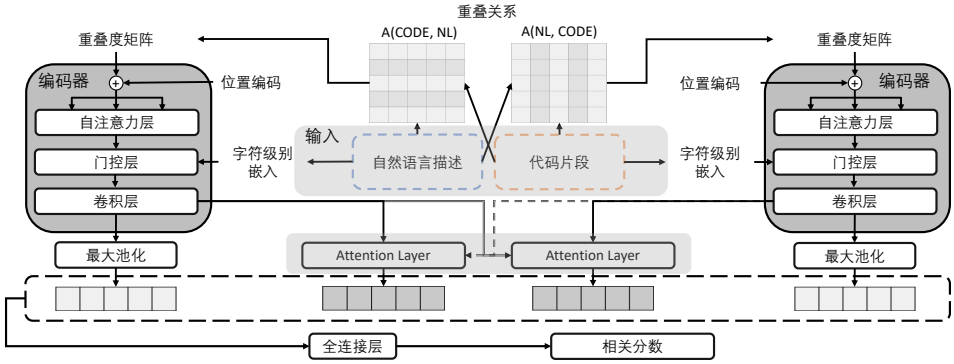


图 7.3　OCoR 的总览

中的标识符)。例如，在图 7.1中的第一个 SQL 查询中，“joint_table_a” 和 “joint_table_b” 也可以被命名为 “joint_table_1” 和 “joint_table_2”。如果在图 7.1中的程序上训练神经网络，它很难知道标识符 “joint_table_1” 和 “joint_table_2” 也与同一个查询相关。为了应对这一挑战，现有的程序搜索方法[137]通过替换变量名和原始字符串为变量类型和数字（例如，将 SQL 中的第一个表变量 “joint_table_b” 重命名为 “Table_1”)。这样，神经网络会被迫忽略标识符名称，而使用程序的结构和标识符类型。然而，标识符的名称可能携带对程序搜索有用的信息，忽略它们会降低性能。为了解决这个问题，本方法提出使用字符级嵌入来编码名称。字符级嵌入首先通过一位有效编码对每个名称中的每个字符进行编码，并通过卷积层组合这些相关向量。图7.2(a) 显示了 “joint_table_b” 和 “joint_table_c” 的字符级嵌入的计算过程。如图所示，这两个标识符的组合向量几乎是从相同的向量中计算出来的，除了最后一个字符的向量。因此，这些标识符的最终嵌入在高维空间中彼此接近。

第二种类型的重叠信息是程序中的标识符往往与自然语言描述中的某些单词相关。从一般角度来看，这种类型的重叠信息是问题和答案之间的重叠，其在现有的信息检索方法中经常被考虑。这些方法测量问题和答案之间完全匹配的词牌数量。然而，在程序搜索任务中，程序中的标识符和自然语言描述中的单词往往并不完全相等。例如，如图7.1所示，标识符 “joint_table_b” 与自然语言描述中的任何单词都不完全相等，但它与两个单词 “joint” 和 “table” 相关。为了解决这个问题，本节不仅考虑完全匹配的单词，还测量部分匹配的单词之间的重叠程度。这里设计了一种表示方法来表示自然语言描述中每个单词与程序中每个标识符之间的重叠程度，其被称为重叠矩阵。在这个矩阵中，每一行代表自然语言描述中的一个单词，而每一列代表程序中的一个标识符。每个单元格是单词和标识符之间的重叠程度。重叠程度可以通过不同的指标来测量，本方法使用最长公共子串，即在自然语言单词和程序标识符中都出现的最长连续子串 p 的比例。图7.2(b) 显示了图7.1中的问题-程序对的部分重叠矩阵。尽管标识符

“joint_table_b” 和单词 “joint” 不是完全匹配的，但它们的重叠程度仍然高于大多数其他对。最后将重叠矩阵作为输入，利用详细的重叠信息来识别最相关的程序片段。

7.3 模型架构

7.3.1 问题定义

本方法使用和现有研究[133,137]相同的定义来进行程序搜索：给定自然语言描述 Q 和一组候选程序片段 C，本任务是检索一个由 Q 指定的相关程序片段 $C^r \in C$。

如图 7.1所示，为了检索一个程序片段，首先计算每个程序片段 $c \in C$ 与输入的自然语言描述 Q 之间的相关性得分。然后对候选程序片段集 C 中的程序片段进行排名。最后，得分最高的程序 C^r 被选为本方法的输出，计算方式如下

$$C^r = \arg\max_{c \in C} R(Q, c) \tag{7.1}$$

其中 R 表示相关性得分的计算。在本方法中，相关性得分是一个介于 0 和 1 之间的实数。

7.3.2 模型总览

图 7.3展示了 OCoR 的概览。本文采用了 Jian et al.[134]在信息检索中使用的传统总体架构，分别对问题（自然语言描述）和答案（程序片段）进行编码，并通过注意力层结合输出以进一步预测目标相关性得分。基于这种架构为问题和答案设计了结构相同的两个编码器。每个编码器都将重叠矩阵和问题/答案作为输入，并将这些输入转化为一组向量。

此外，正如稍后的评估将展示的那样，本章的模型补充了现有的程序搜索方法。为了达到更好的性能，本方法使用了一个额外的集成组件，将之前的程序搜索模型与 OCoR 结合起来。

本节将逐一描述模型架构的各个组成部分。

7.3.3 模型输入

OCoR 的输入分为三个部分：1) 自然语言描述；2) 程序片段；3) 自然语言描述和候选程序片段之间的重叠部分。其中最后一部分是根据前两部分计算得出的。

预处理 对于前两部分，首先处理这些输入，使其适于被神经网络使用。对于输入的自然语言描述，首先使用 NLTK 工具包[139]中的工具对其进行分词，并将词牌内的字符转换为小写；对于输入的程序片段，保留原始的变量名和原始字符串，以保留名称中

的语义信息。然后，对程序进行分词，并将名称中的字符也转换为小写。由此得到预处理后的神经网络输入。

重叠度矩阵　如前所述，本方法使用重叠矩阵来表示自然语言单词和程序标识符之间重叠的程度。重叠矩阵是一个实数矩阵 $A(T_1, T_2) \in \mathbb{R}^{L^{(T_1)} \times L^{(T_2)}}$，包含了词牌序列 T_1 和另一个词牌序列 T_2 之间的重叠分数。矩阵中的每个单元 $A_{ij}(T_1, T_2)$ 表示 T_1 序列中的第 i 个词牌 $T_1(i)$ 和 T_2 中的第 j 个词牌 $T_2(j)$ 之间的重叠分数。在 OCoR 中，此分数的计算方式为

$$A_{ij}(T_1, T_2) = \text{len}(S(T_1(i), T_2(j)))/\text{len}(T_2(j)) \tag{7.2}$$

其中 $\text{len}(T_2(i))$ 表示单词 $T_1(i)$ 的长度，$S(T_1(i), T_2(j))$ 表示 T_1, T_2 的最长公共子字符串。特别地，重叠矩阵 A 的计算是不可交换的，即 $A(T_1, T_2) \neq A(T_2, T_1)$。本方法考虑了自然语言描述 NL 与程序 $CODE$ 之间的重叠分数以及程序与自然语言描述之间的重叠分数，即分别考虑了 $A(\mathbf{n}, \mathbf{c})$ 和 $A(\mathbf{c}, \mathbf{n})$。这两个指标进一步被输入到编码器层，以提取程序搜索的特征。

7.3.4　编码器

在 OCoR 中，有两个编码器，分别用于自然语言描述和程序。每个编码器都将重叠矩阵和自然语言描述/程序作为输入。这些输入被编码成高维空间中的向量，以用于进一步的相似度计算。

为了更好地编码输入信息，受 Vaswani et al.[54]启发，这里设计了一个由 N 个机制组成的堆栈编码器。每个机制包含三个子层：1) 一个自注意力层；2) 一个门控层；3) 一个卷积层。每个神经组件之后，使用 ResNet[140]①和层归一化[141]②。对于第一个组件，它将重叠矩阵 $A(NL, CODE)$ / $A(CODE, NL)$ 作为输入，并进一步结合自然语言描述/程序。对于其余的 $N-1$ 个机制，它们将前一个组件的输出作为输入，并且还结合了自然语言描述/程序中隐藏层的特征。

7.3.4.1　重叠矩阵

本方法中的重叠矩阵③是一个实数矩阵，其中每个单元表示自然语言单词和程序中标识符之间的重叠分数。为了将这个矩阵作为输入，首先将矩阵 $A(NL, CODE)$ 简化为一个重叠向量 $\mathbf{a}_(NL)$，其中向量中的第 i 个元素 $\mathbf{a}_i(NL)$ 表示从 $A(NL, CODE)$ 的第 i 行（代表自然语言中第 i 个词牌和程序中每个标识符的单元）计算出的新的重叠

①ResNet 是残差学习框架，用于简化网络的训练。
②层归一化将神经元的值归一化到适当的分布中，这有助于训练。
③这里以 $A(NL, CODE)$ 为例。

分数。这个过程是通过最大池化来实现的。最大池化[142]已被证明是在各个领域中将矩阵简化为向量的有效方法[137,143-144]。在进行最大池化之后，选择每列的最大值作为 $\mathbf{a}_(NL)$ 的元素，计算方式为

$$\mathbf{a}_i(\mathbf{n}) = \max_{j=1}^{\text{len}(\mathbf{c})} A_{ij}(\mathbf{n}, \mathbf{c}) \tag{7.3}$$

这个向量包含了最大重叠分数。例如，图7.2(b) 中的重叠矩阵对应的固定大小向量是 $[0.3, 0.75, 0.4, 0.3]$。为了方便神经网络处理重叠分数，进一步使用一位有效编码来嵌入这些分数。请注意，这些分数是 0 到 1 之间的实数值。本方法以 0.01 为间隔划分这些分数，并使用长度为 100 的一位有效向量来编码这些分数。

7.3.4.2 自注意力层

编码器机制中的第一个子层是自注意力层。众所周知，在自然语言描述和程序中，序列信息是非常重要的。为了有效处理这类信息，本方法使用了 Vaswani et al.[54]提出的自注意力机制，该机制已被证明是编码此类信息的有效方式[54,145]，作为编码器中的第一个子层。

自注意力层将之前的输出向量 $\mathbf{o}_1$,$\mathbf{o}_2$,$\cdots$,$\mathbf{o}_L$ 作为输入，其中 L 表示输入的自然语言描述/程序的长度。这一层包括两部分：1) 位置嵌入层；2) 多头注意力层。

7.3.4.3 位置嵌入层

位置嵌入层是变换器架构[54]中的标准层，用于提供输入矩阵中单词的索引。例如，该层应该知道 “joint” 是图7.1中自然语言描述的第 5 个词牌。如果直接对输入向量使用注意力层，位置信息将不会被考虑，这就是为什么需要位置嵌入层的原因。

在这一层中，第 i 个位置的向量表示为一个实数向量，该向量的计算方式为

$$\begin{aligned} p_{(i,2j)} &= \sin(pos/(10000^{2j/d})) \\ p_{(i,2j+1)} &= \cos(pos/(10000^{2j/d})) \end{aligned} \tag{7.4}$$

其中 $pos = i + step$，j 表示输入向量的元素，$step$ 表示嵌入大小。在获取了每个位置的向量之后，直接将这个向量加到相应的输入向量上，其中 $\mathbf{e}_i = \mathbf{o}_i + \mathbf{p}_i$。

7.3.4.4 多头注意力层

自注意力层的第二部分是多头[①]注意力层。如背景部分所介绍的，注意力机制将一个查询（query）、一个键（key）和一个值（value）映射到一个输出。在这一层中，查询、键、值和输出都是向量。

[①]多头机制由几个头组成，每个头都是一个单独的注意力层。这些头的输出通过一个全连接层进一步联合在一起。

遵循 Vaswani 等人[54]的定义，本方法将多头注意力机制的数目设为 H。每个头都是一个注意力层，它将查询 Q、键 K 和值 V 映射到一个输出，即每个头的输出 $head$。第 s 个头的计算表示为

$$head_s = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V \tag{7.5}$$

其中 $d_k = d/H$ 表示每个提取的特征向量的长度，Q、K 和 V 是通过全连接层从 Q、K、V 计算得出的。在编码器中，向量 Q、K 和 V 都是位置嵌入层 $\mathbf{e}_1, \mathbf{e}_2, \cdots, \mathbf{e}_L$ 的输出。这些头的输出通过全连接层进一步联合在一起，计算方式为

$$Att = [head_1; \cdots; head_H] \cdot W_h \tag{7.6}$$

其中 W_h 表示全连接层中的权重，输出向量 $Att = [\mathbf{a}_1, \mathbf{a}_2, \cdots, \mathbf{a}_L]$ 是高层次向量，它们结合了序列信息和原始信息。然而，至少在编码器的第一个机制中，这些向量仍未能有效地编码每个单词的语义信息。因此，这里将描述如何通过门控层来解决这个问题。

7.3.4.5　门控层

编码器中的第二个子层是门控层。这一层将上一层的输出和输入的自然语言描述/程序作为输入。现有的最先进方法[137]使用 word2vec[146]机制来利用输入的语义信息。然而，对于程序搜索而言，这可能不适用，因为不同程序员可能以不同的方式命名相似的标识符，但具有重叠的字符（例如，对于两个单词 “dataId” 和 “data_id”，它们可能具有非常相似的含义），这可能导致神经网络需要学习的词汇量很大。为了解决这个问题，本方法使用字符级语义来捕捉程序搜索中标识符之间的重叠。

7.3.4.6　字符嵌入

为了实现字符嵌入，本方法首先使用一个特殊字符将每个词牌（自然语言描述中的单词和程序中的标识符）填充到固定长度 CL。特别地，如果词牌的长度超过 CL，本方法将截断这个词牌的末尾，使其成为一个长度为 CL 的词牌。然后将词牌中的每个字符表示为一个实值向量，即 $embedding$。众所周知，一个词牌由几个字符组成。为了捕捉每个词牌的语义信息，本方法采用一组卷积层来整合词牌内字符的向量。第 i 个词牌 $\mathbf{t}_i$ 的提取的语义向量通过下式计算

$$\mathbf{t}_{(i,n)} = W_{(c,n)}[\mathbf{t}_{(1,n-1)}; \mathbf{t}_{(2,n-1)}; \cdots; t_{(CL,n-1)}] \tag{7.7}$$

其中 W_c 是卷积权重，n 表示卷积层的第 n 层。特别地，$\mathbf{t}_{(k,0)} = \mathbf{c}_k$，其中 $\mathbf{c}_k$ 表示第 i 个词牌内第 k 个字符的字符嵌入向量。在本方法中，这个字符嵌入层有三个卷积层。对于前两个卷积层，本方法使用零填充，卷积核的大小分别设置为 3 和 5。

7.3.4.7 门控机制

为了将每个词牌的语义信息与之前的输出结合起来，这里使用了一个名为门控机制[147]的神经网络结构。这个机制将输入的语义向量 $\mathbf{t}_i$ 与一个给定的控制向量[①]通过多头机制结合起来。在本文中，我们使用之前的输出向量，即 $\mathbf{a}_i$，作为控制向量。本模型中的门控层计算可以表示为

$$\alpha_i^o = exp(\mathbf{q}_i^T \mathbf{k}_i^o)/\sqrt{d} \tag{7.8}$$

$$\alpha_i^c = exp(\mathbf{q}_i^T \mathbf{k}_i^c)/\sqrt{d} \tag{7.9}$$

$$\mathbf{h}_i = (\alpha_i^o \cdot \mathbf{v}_i^o + \alpha_i^c \cdot \mathbf{v}_i^c)/(\alpha_i^o + \alpha_i^c) \tag{7.10}$$

$$head_s = [\mathbf{h}_i; \cdots; \mathbf{h}_H] \tag{7.11}$$

其中 $\mathbf{q}_i$、$\mathbf{k}_i^o$、$\mathbf{v}_i^o$ 都是通过控制向量 $\mathbf{a}_i$ 上的全连接层计算得出的；$\mathbf{k}_i^c, \mathbf{v}_i^c$ 是通过语义向量 $\mathbf{t}_i$ 上的另一个全连接层计算得出的。通过这个计算，本方法用语义信息增强了向量，提取的新特征表示为 $\mathbf{c}_1^{(com)}, \mathbf{c}_2^{(com)}, \cdots, \mathbf{c}_L^{(com)}$。

7.3.4.8 卷积层

编码器机制中的最后一个子层是一组卷积层。本方法遵循 Vaswani et al.[54]提出的编码器设计，并采用一组卷积层来提取每个令牌周围的局部特征。卷积层的计算可以表示为

$$\mathbf{y}_i^l = W_l[\mathbf{y}_{i-w}^{l-1}; \cdots; \mathbf{y}_{i+w}^{l-1}] \tag{7.12}$$

其中 l 表示集合中的第 l 个卷积层，W_l 是卷积权重，$w = (k-1)/2$，k 表示窗口大小。特别地，$\mathbf{y}_i^{l-1}$ 是前一个门控层 $\mathbf{c}_i^{(com)}$ 的输出。本方法在这个子层中使用两个卷积层，并在这些卷积子层之间添加激活函数 $GELU$[148]。特别是，本章在这些层中使用零填充。

7.3.4.9 最大池化

在编码器的这 N 个机制之后，本方法采用了一个额外的卷积层，如公式 7.12所示，该卷积层在卷积过程中用一个特殊向量进行填充。然后，本方法得到了每个词牌的最终特征，这些特征代表了输入自然语言描述/程序的高层信息。然而，这些特征与输入的维度相同。为了便于后续程序搜索，这里需要将这些特征聚合成一个固定大小的向量，这个向量的维度与输入长度无关。

最大池化在聚合特征方面展现了其强大的能力，因此，本方法对编码器的输出应用最大池化方法，提取每个编码器的固定大小向量。

[①]控制向量是本方法中给定的特殊向量。这个向量决定了不同向量的权重。

7.3.5 注意力层

编码器分别对输入的自然语言描述和输入程序的信息进行编码。然而，即使已经使用了重叠信息的先验知识，两个输入之间的关系仍然缺失。为了帮助神经网络学习两个输入之间的关系，在编码器后使用了两个注意力层。

如前一节所述，编码器的输出结合了重叠信息和语义信息（字符嵌入）。因此，本方法也将多头注意力层应用于两个编码器的输出，以提取两者之间的关系。对于自然语言描述和程序，本方法为它们设计了两个单独的注意力层。注意力机制的计算类似于“自注意力”，但输入不同。一个层将描述的编码视为查询（Q），另一个层将程序的编码视为查询（Q）。这种设计允许模型基于彼此提取两个编码器输出的加权和。注意力层之后，是两个卷积层和一个最大池化层来整合特征。

7.3.6 预测

在注意力层的所有计算之后，本方法将所有特征串联起来。它们进一步被送入一个两层的感知机，后面跟着一个 *softmax* 激活函数。这些神经元的输出是两个类别的分类概率。第一类表示输入的自然语言描述和输入程序是相关的，而第二类表示输入的自然语言描述和输入程序是不相关的。在本方法中，第一类的预测分类概率是输入自然语言描述和程序之间的相关性得分，相关性得分计算如下

$$R(Q,c) = \frac{\exp\{h_1\}}{\sum_{j=1}^{2}\exp\{h_j\}} \tag{7.13}$$

其中 h_i 是 *softmax* 的输入对数几率。

7.3.7 训练

本模型通过最小化与真实值的交叉熵损失来进行训练。具体来说，对于每个训练数据 $< Q, C, A >$，其中 Q 是描述，C 是程序，A 表示真实类别。交叉熵损失通过以下公式计算：

$$Loss(\theta) = -\sum_{i=1}^{2} g(i) * \log\theta(i) \tag{7.14}$$

其中 g 表示真实类别，θ 是神经网络预测的分类结果。

7.3.8 模型组合

在上述模型的基础上，本方法还考虑了一种通过集成不同模型来共同完成程序搜索任务的额外方法。姚等人[137]提出的 CoaCor 通过结合程序搜索与程序注释来提高性能，受此启发，本方法考虑通过整合不同模型计算出的相关性分数，并输出 OCoR 的

统计量	StaQC-train	StaQC-val	StaQC-test	DEV	EVAL	C#-train	C#-dev	C#-test
问题-程序对的数量	89,688	11,932	17,899	330	300	77,816	17,849	17,849
描述中的平均词牌数	9	9	9	10	15	12	12	12
描述中的最大词牌数	32	35	45	45	35		37	34
程序中的平均词牌数	59	62	60	47	47	38	38	38
程序中的最大词牌数	3,367	2,774	2,672	291	291	290	300	310

表 7.1　使用的数据集的统计信息

最终相关性分数来组合不同模型。分数通过线性组合计算得出：

$$R(Q,c) = \lambda * S_1 + (1-\lambda) * S_2 \tag{7.15}$$

其中 S_1 表示 OCoR 计算出的相关性分数，S_2 是组合模型计算出的分数，λ 是一个介于 0 和 1 之间的实数。

7.4　实验设置

7.4.1　研究问题

本节的评估旨在回答以下研究问题：

- **问题 1: OCoR 的性能如何？**为了回答这个问题，本节在几个已建立的数据集上进行了实验，并将 OCoR 的性能与现有的最先进方法进行了比较。
- **问题 2: OCoR 中每个组成部分的贡献是什么？**为了回答问题 2，本节从 OCoR 的完整模型开始，依次移除每个组件以了解其贡献。然后，本节还用最长公共前缀（LCP）和基于词嵌入的相似度来替换在测量重叠得分中使用的指标，以更好地理解重叠矩阵组件的贡献。
- **问题 3: 为什么模型组合有效？**实际上，问题 1 的结果将表明模型组合在整体性能中起着重要作用。为了理解为什么不同的模型可以组合在一起，本节分析了不同模型在 SQL 数据集上预测结果的分布。更具体地说，本节从这些数据集中选取了一些示例，以展示模型之间的差异。

7.4.2　数据集

本节的实验基于两个已建立的基准：StaQC 基准[149]，以及 Iyer et al.[138]使用的 C# 基准。StaQC 基准包含 119,519 个用 SQL 编写的问题-程序对，从 Stack Overflow[150]收集，是迄今为止 SQL 领域中最大的数据集。本节遵循了 StaQC 的原始训练-开发-测试划分，即 StaQC-train、StaQC-val 和 StaQC-test。为了更好的评估，本节使用了两个额外的测试数据集，SQL 的 DEV 和 EVAL，由 Iyer et al.[138]收集，分别包含 110 和 100

个用 SQL 编写的程序。对于每个片段，他们使用三个不同的人类编写的参考作为额外的测试用例。第二个基准包含 113,514 个用 C# 编写的问题-程序对，从 StackOverflow 收集。本节将数据集分为 C#-train、C#-val 和 C#-test，和 Iyer et al.[137-138]一致。这些数据集的详细统计数据列在表 7.1中。

对于 StaQC 基准，本节将 StaQC-train 的训练集作为训练集，将 DEV 集作为开发集，并将其他三个数据集，StaQC-test、StaQC-val 和 EVAL，作为测试集。对于 C# 基准，本节在训练期间也遵循了相同的实验设置，C#-val 被视为开发集。

需要注意的是，为了测试程序搜索方法的性能，不仅需要期望的问题-程序对（正面答案），还需要其他程序片段作为负面答案。因此这里将包含一个问题和一组程序片段的情况称为*检索案例*。本节使用了现有工作的相同检索案例[137-138]，这些案例的数量显示在表 7.1的 “案例数量” 行中。每个检索案例包含 1 个正面程序片段和 49 个负面程序片段。

本节遵循了 Yao et al.[137]的做法，使用了一种标准指标来衡量本方法的性能，称为平均倒数排名（MRR）[151]。MRR 指标是在整个数据集 $D = (Q_1, C_1), (Q_2, C_2), \cdots, (Q_n, C_n)$ 上计算的：

$$MRR = \frac{\sum_{i=1}^{n} 1/r_i}{|D|} \tag{7.16}$$

其中 r_i 表示 C_i 在第 i 次查询 Q_i 中的排名。在这个指标中，更高的值表示程序搜索性能更好。

本节基于 Tensorflow[152]实现了提出的方法。对于超参数的设置，这里 $N = 3$，表示实验中的每个编码器包含 3 个神经元的堆栈。本节将字符和重叠得分的嵌入大小设置为 256。除了第一层卷积层和 MLP 中的第一层使用 1024 外，所有隐藏层大小均设置为 256。在训练期间，使用 dropout[153]来避免过拟合，其中 drop 比例设置为 0.2。本模型通过 Adam 优化器[154]以 0.0001 的学习率进行优化。对于模型组合，将超参数 λ 设置为 0.1。这些超参数和参数是基于开发集（使用 DEV）选择的，遵循了现有的最先进工作[137]。对于训练语料中的每个自然语言描述的查询，在训练时期随机采样 5 个程序片段作为负面示例。在 OCoR 中，自然语言描述和程序片段共享相同的嵌入权重。

7.4.3 基准技术

本节的实验使用了现有的最先进的程序搜索方法作为比较的基准技术。

- Deep Code Search（DCS）[133]。DCS 利用基于 RNN 的神经网络共同嵌入输入程序片段和输入自然语言描述到一个高维向量空间中。通过这种方式，一个程序片段及其对应的自然语言描述有相似的向量，然后使用余弦相似度计算两个输入之间的相似性。

	模型	EVAL	StaQC-val	StaQC-test	C#
原始模型	DCS	0.555	0.534	0.529	0.441
	CODE-NN	0.514	0.526	0.522	0.531
	QN-RL^{MRR}	0.512	0.516	0.523	0.528
	OCoR	**0.601**	**0.647**	**0.643**	**0.682**
模型组合	QN-RL^{MRR} + CODE-NN	0.571	0.575	0.576	0.629
	OCoR + QN-RL^{MRR}	0.630	0.658	0.677	0.746
	OCoR + (QN-RL^{MRR} + CODE-NN)	**0.646**	**0.665**	**0.685**	**0.764**

表 7.2　程序搜索的 MRR 指标比较

- CODE-NN[138]。CODE-NN 的核心组成部分是一个基于 LSTM 的 RNN[155]，并引入了注意力机制。这个注意力机制计算给定程序片段的自然语言描述的概率。对于程序搜索，给定一个输入的自然语言描述，CODE-NN 计算每个程序的输入概率，根据概率对给定的程序片段进行排名。
- CoaCor[137]。CoaCor 使用了一个基于强化学习的框架，将程序搜索和程序注释结合在一起以增强程序搜索，还通过集成程序搜索方法以提高性能。特别地，CoaCor 的基本模型被表示为 QN-RL^{MRR}，而组合模型（有最佳性能）被表示为 QN-RL^{MRR} + CODE-NN。

7.5　结果

本节中报告了实验结果并回答了研究问题。

7.5.1　问题 1：OCoR 的性能

首先将原始的 OCoR 与原始模型进行比较。如表 7.2所示，和三个最先进模型的比较中，OCoR 在所有数据集上都实现了最佳性能，比现有的最好结果高出 9.1% 到 28.2%。

对于模型组合，首先将 OCoR 与原始模型 QN-RL^{MRR} 组合（表示为 OCoR + QN-RL^{MRR}）。本节选择 Yao et al.[137]提出的最先进模型（QN-RL^{MRR} + CODE-NN）作为基线。如表7.2所示，本方法显著优于现有的最先进模型，平均提高了 10 个百分点。特别是，本技术在所有数据集上实现了 13.1% 到 22.3% 的提高，显示了本方法的有效性。然后，本方法将 OCoR 与 Yao et al. 组合的模型（QN-RL^{MRR} + CODE-NN）结合，该模型也在所有数据集上实现最佳结果（OCoR + (QN-RL^{MRR} + CODE-NN)）。

总结：与现有的最先进方法相比，OCoR 在涵盖两种编程语言的所有数据集上表

表 7.3　OCoR 的消融实验结果

Model	EVAL	StaQC-val	StaQC-test
OCoR	**0.601**	**0.647**	**0.643**
- Overlap Score	0.420	0.545	0.538
Character-level Overlap → WordSimilarity	0.554	0.603	0.605
Overlap → LCP	0.591	0.628	0.632

现良好（13.1% **到** 22.3% **的提高）**。

7.5.2　问题 2：每个技术部件的贡献

为了回答问题 2，本节首先在 SQL 数据集上进行了消融测试，以确定每个组件的贡献。这一小节仅基于原始的 OCoR 进行实验，以更清楚地理解每个组件的贡献。

消融实验中的模型与原始的 OCoR 设置相同，但不同的是轮流移除模型中的每个组件。消融实验的结果展示在表 7.3中。本节首先从 OCoR 中移除了输入重叠分数，用自然语言描述/程序中的输入词牌作为编码器第一个机制的输入。此外，本节遵循之前的联合模型[133]，通过最小化自然语言描述和程序之间的余弦相似度来训练模型。通过应用这样的设置，性能接近于基于 word2vec 的前一种方法，这显示了重叠矩阵的效果。

为了更好地理解计算重叠分数的指标（即方程 7.2）的贡献，本节在相同的数据集上进一步进行了实验，但使用了不同的重叠分数指标。与原始指标相比的基线指标是基于词嵌入的词相似性。首先使用 GloVe[156]基于训练集和 BPE[157]预训练词嵌入向量。然后使用 $\mathbf{w}_1, \mathbf{w}_2$ 的余弦相似度作为重叠分数。对应的实验结果展示在表 7.3中。字符级指标在所有数据集上都实现了更好的性能。这个结果表明，本方法提出的指标比文档检索中使用的传统相似性矩阵更适合程序搜索。

本节还用最长公共前缀（Longest Common Prefix，LCP）替换了原始的重叠指标。一对字符串 a 和 b 的最长公共前缀是同时为这两个字符串前缀的最长字符串 p。本节分别使用 $len(p)/len(a)$，$len(p)/len(b)$ 作为两个字符串的重叠分数。本节还基于最长公共后缀计算分数，并最终选择较大的一个作为重叠分数。这个指标的性能略低。消融测试也表明，字符级信息对于程序搜索任务很重要。

总结：每个技术组件都对 OCoR 的整体性能有贡献。

7.5.3　问题 3：模型组合分析

为了回答问题 3，本节尝试找出模型组合有效的原因。首先实现了现有的方法，CoaCor（QN-RL^{MRR}）和 CODE-NN。然后，本节研究了数据集中完美排名的预测重叠。对于给定的自然语言描述和一组程序片段（其中一个为正程序片段），本文中的“完美排

名”意味着正确的程序片段被排在第一位。

结果展示在图 7.4中。如图所示，在这三个数据集上，平均有 16.2% 的完美排名案例可以被所有三种方法解决，而平均有 26.1%、3.7%、2.3%（总共 32.3%）的完美排名案例分别只能由 OCoR、CoaCor 和 CODE-NN 解决。这 32.3% 的案例显示了模型组合的潜在改进，这就是为什么模型组合在本节的方法中效果很好。

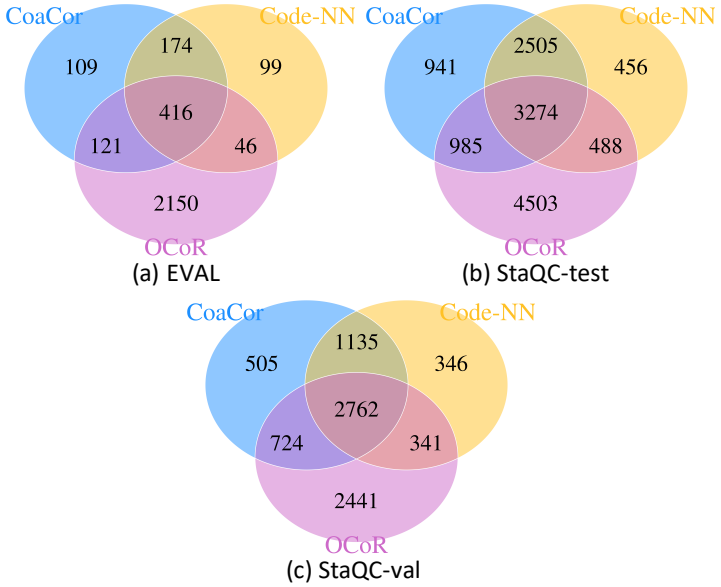


图 7.4　三个数据集上不同方法的完美排序的重叠

为了帮助理解模型组合，本节还进行了一个额外的案例研究。在这个案例研究中分析了 OCoR 与现有的最先进模型（CoaCor，QN-RL$^{\text{MRR}}$ + CODE-NN）。

案例研究　表 7.4展示了三个例子，这些例子被 OCoR 完美排名，但 CoaCor 没有做到。如所示，在这些例子中，输入的自然语言描述和 SQL 程序之间有许多重叠（例如，第一个例子中的单词 “select” 和 “data”，第 2、3 行；第二个例子中的 “table” 和 “time”）。在这些例子中，重叠分数的信息很重要，人类可以利用这些信息轻松检索到目标程序，而 OCoR 成功捕捉到了这种信息。像 CoaCor 这样的现有方法没有合理利用重叠分数的信息，其中 CoaCor 方法直接使用词牌级嵌入进行神经网络训练，并将标识符名称替换为编号占位符词牌（例如，第二个例子中的 SQL 程序在表 7.4中被转换为 “select col0 (col1) from tab1”）。因此，OCoR 在这些例子上表现良好，而 CoaCor 表现不佳，这是 OCoR 的优势。

为了了解 OCoR 的弱点，本节还进行了另一个案例研究，研究了与 CoaCor 相比 OCoR 表现不佳的例子。这些例子展示在表 7.5中，其中输入的自然语言描述和 SQL 程序之间的重叠很少。在这种情况下，OCoR 很难估计重叠分数，难以利用其中包含的

表 7.4　OCoR 可以完美排序但 CoaCor 不能的示例

类型	样例
描述	**select** a formatted **date** range from values in a table column
SQL 代码	**select** **date**_format (start**date**, '%m') + **date**_format (start**date**, '%d') + '-' + **date**_format (end**date**, '%d') + ',' + **date**_format (start**date**, '%y') from yourtable;
描述	SQL **Insert** multiple Values **where** 1 value comes **from** a **select** query
SQL 代码	**insert** into table2 (telnumber , adress) **select** '12324567890' , applicatieid **from** applicatie **where** name = 'piet' ;
描述	Quick way to **space** fill column **256** chars SQL-Server 2012
SQL 代码	select **space** (**256**) ;

关键信息。然而，CoaCor 结合了注释生成和程序搜索，将标识符名称替换为编号占位符，并提取了这些情况的高级信息，因此能在这些例子上表现良好。这些案例显示了 OCoR 的弱点，当难以测量重叠分数时，它表现不佳。结合不同方法并利用每种方法的优势可能是一个好方法。因此，本节使用模型组合来结合不同方法的优势。

类型	样例
描述	how to use max and top in sql query in oracle?
SQL 代码	select id, item, quantity, date from (select id, item, quantity, date from your_table order by quantity desc, date desc) where rownum = 1;
描述	find 1 level deep hierarchical relationship between columns of a table for one of the top level values
SQL 代码	select t2.cat_id, t2.subcat_id, t2.name from test t1 join test t2 on t1.cat_id = t2.cat_id where t1.subcat_id = 42 and t2.subcat_id <> 42 ;

表 7.5　CoaCor 可以完美排序但 OCoR 不能的示例

总结：这三种技术相互补充，使得模型组合能够产生更好的结果。

7.6　小结

本章提出了一个新颖的基于重叠感知的神经架构（OCoR）用于程序搜索。本章的方法通过使用重叠矩阵和字符嵌入，计算自然语言描述和程序之间的重叠分数。本章在几个基准数据集上评估了技术的有效性。实验结果显示，OCoR 与现有的最先进方法相比实现了显著改进。进一步的评估显示，本章方法中的每个组件都很重要。

第八章　深度程序模型在程序修复上的应用

本章针对程序自动修复这个任务的特点，提出特定的技术以提升程序模型在该任务上的性能。

8.1　程序修复的任务特点

自动程序修复（APR）旨在通过生成补丁以协助开发人员，从而减少修复缺陷的努力。由于众所周知的测试套件薄弱问题[158]，即使一个补丁通过了所有测试，这个补丁仍然有很高的概率是错误的。为了克服这个问题，现有的方法采用了不同的手段来指导补丁的生成。一个典型的方式是从现有的软件仓库中学习，例如从现有的补丁中学习模式[73-74,159-163]，并使用程序代码来指导补丁的生成[74,164-167]。

深度学习被认为是一种强大的机器学习方法。近期，一系列研究努力尝试使用深度学习技术从现有的补丁中学习，以进行程序修复[55,62,168-169]。一个典型的基于深度学习的方法是生成一个新的语句来替换通过缺陷定位方法定位的错误语句。现有的基于深度学习的方法基于编码器-解码器架构[170]：编码器将错误语句及任何必要的代码上下文编码为一个固定长度的内部表示，而解码器则从中生成一个新语句。例如，Hata等人[169]和 Tufano 等人[62]采用现有的神经机器翻译架构 NMT 来生成缺陷修复；SequenceR[168]使用带有复制机制的序列到序列神经模型；DLFix[55]进一步将缺陷语句视为一个抽象语法树（AST），而不是一系列词牌，并编码了语句的上下文。

然而，尽管存在多种尝试，基于深度学习的自动程序修复（APR）方法尚未超越传统 APR 方法。鉴于深度学习在许多领域已经超越传统方法，本章旨在进一步提升基于深度学习的 APR 的性能，以了解是否能够使用基于深度学习的方法超越传统 APR。尽管现有的基于深度学习的 APR 方法提出了不同的编码器架构用于 APR，解码器架构仍然是标准的，一次生成一个词牌序列来替换原始的错误程序片段。使用这种标准解码器显著限制了基于深度学习的 APR 的性能。这里强调三个主要现有技术的缺陷。

缺陷 1: 包含语法不正确的程序在补丁空间中。解码器的目标是从补丁空间中定位补丁。补丁空间越小，任务就越容易。然而，将补丁视为一系列词牌不必要地扩大了补丁空间，使解码任务变得困难。特别是，这种空间表示不考虑目标编程语言的语法，并包含许多语法不正确的语句，这些语句永远不能形成正确的补丁。

缺陷 2: 小编辑的低效表示。许多补丁只修改语句的一小部分，并重新生成整个语句导致不必要地增大了补丁空间。例如，图 8.1展示了 Defects4J 基准测试中的缺陷 Closure-14 的补丁，这个补丁只改变了语句中的一个词牌，但在现有表示下，它被编码

```
- cfa.createEdge(fromNode, Branch.UNCOND, finallyNode);
+ cfa.createEdge(fromNode, Branch.ON_EX, finallyNode);
```

图 8.1　Defects4J 中缺陷 Closure-14 的补丁

```
-   return cAvailableLocaleSet.contains(locale);
+   return availableLocaleSet().contains(locale);
```

图 8.2　Defects4J 中缺陷 Lang-57 的补丁

为长度为 13 的序列，包含此补丁的程序空间大约包含 n^{13} 个元素，其中 n 是词牌的总数。相反，对于仅包括单一词牌更改编辑的补丁空间，为了生成图中的补丁，只需要在错误语句中选择一个词牌并为替换选择一个新的词牌，这个补丁空间仅包含 mn 个元素，其中 m 是错误语句中的词牌数。因此，补丁空间的大小显著减小。

缺陷 3：无法生成项目特定的标识符。程序的源代码经常包含项目特定的标识符，如变量名。由于将所有可能的标识符包含在补丁空间中是不切实际的，现有基于深度学习的 APR 方法只能生成在训练集中频繁出现的标识符。然而，不同的项目有不同的项目特定标识符集，仅考虑训练集中的标识符可能会从补丁空间中排除可能的补丁。例如，图 8.2展示了 Defects4J 中缺陷 Lang-57 的补丁。为了生成这个补丁，需要生成标识符 “`availableLocaleSet`”，这是一个有错误的类的方法名，不太可能包含在训练集中。因此，现有的基于深度学习的方法无法生成这样的补丁。

本章提出了一种新颖的基于深度学习的 APR 方法，称为 Recoder。与现有方法类似，Recoder 是基于编码器-解码器架构。为了解决上述缺陷，Recoder 的解码器采用以下两种新颖技术。

创新点 1：语法引导的编辑解码与提供者/决策者架构（针对缺陷 1 和 2）。为了解决缺陷 2，Recoder 的解码器组件产生一系列编辑而不是新的语句。本章的编辑解码器基于现有神经程序生成方法中的语法引导解码器的思想[61,171-173]。对于部分抽象语法树（AST）中的一个未展开的非终端节点，解码器估计用于展开节点的每个语法规则的概率。基于这种方法，解码器选择最可能的规则序列，使用搜索算法（如束搜索）将开始符号展开成一个完整程序。

此外，编辑也可以通过语法来描述。例如，缺陷 Closure-14 的补丁可以用以下语法来描述：

$$\begin{aligned} \mathit{Edit} &\rightarrow \mathit{Insert} \mid \mathit{Modify} \mid \ldots \\ \mathit{Modify} &\rightarrow \texttt{modify}(\mathit{NodeID}, \mathit{NTS}) \end{aligned}$$

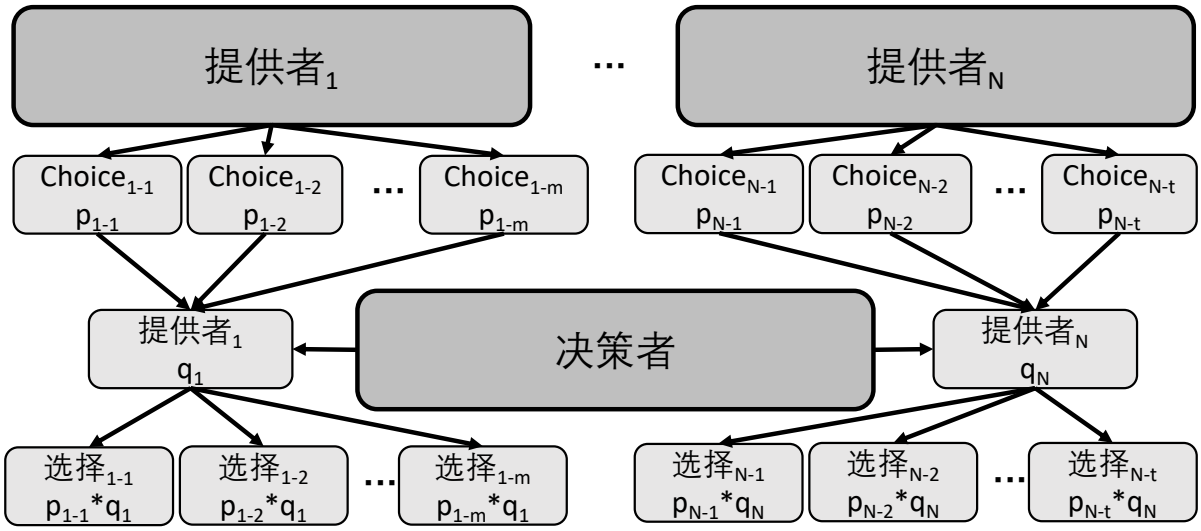


图 8.3　提供者/决策者架构

然而，直接将现有的语法引导解码器应用于上述语法，并不能形成一个有效的程序修复方法，因为展开不同非终端节点的选择可能需要同时推断不同类型的依赖性。首先，某些非终端的展开依赖于局部上下文，例如，*NodeID* 的选择依赖于有错误的声明，神经网络需要了解局部上下文才能做出合适的选择。其次，为了保证语法正确性（缺陷 1），在展开不同非终端节点的选择之间存在依赖性，例如，当 *NodeID* 展开为指向具有非终端 *JavaExpr* 的节点的 ID 时，*NTS* 也应该展开为 *JavaExpr* 以确保语法正确性。这些选择不能有效地预定义，由于现有的语法引导解码器仅在一组预定义的语法规则中进行选择，在这里不起作用。

创新点 2：占位符生成（针对缺陷 3）。为了生成项目特定的标识符，一个直接的想法是添加另一个选择器，从局部上下文中选择一个标识符。然而，要实现这样一个选择器，神经组件需要访问当前项目中的所有名称声明。这是一项困难的任务，因为神经组件很难编码整个项目中的所有源代码。

与其依赖神经网络生成项目特定的标识符，Recoder 中的神经网络生成这些标识符的占位符，当应用编辑时，这些占位符将被实例化为所有可行的标识符。一个可行的标识符是与编程语言中的约束兼容的标识符，例如类型系统。如图 8.2中所示的缺陷 Lang-57，Recoder 首先为 `availableLocaleSet`” 生成一个占位符，并将其替换为在局部上下文中可访问的、不接受任何参数且返回一个包含成员方法 `contains`” 的对象的所有方法。每次替换形成一个新的补丁。关键的见解是，当考虑到编程语言中的约束时，用一个标识符替换占位符的选择数量很小，因此用所有可能的选择实例化占位符是可行的。

为了训练神经网络生成占位符，我们用占位符替换训练集中不常见的用户定义标

识符。通过这种方式，神经网络学习为这些标识符生成占位符。本章的实验在四个基准测试上进行：(1) 用于与现有方法比较的 Defects4J v1.2 的 395 个缺陷，(2) Defects4J v2.0 的额外 420 个缺陷，(3) IntroClassJava 的 297 个缺陷，以及 (4) QuixBugs 的 40 个缺陷，以评估 Recoder 的泛化能力。结果显示，Recoder 在第一个基准测试上正确修复了 53 个缺陷，比 TBar[73]多出 26.2%（11 个缺陷）和比 SimFix[74]多出 55.9%（19 个缺陷），这两者是在 Defects4J v1.2 上表现最好的只生成单块连续补丁的 APR 方法；Recoder 在第二个基准测试上也正确修复了 19 个缺陷，比 TBar 多出 137.5%（11 个缺陷）和比 SimFix 多出 850.0%（17 个缺陷）。在 IntroClassJava 和 QuixBugs 上，Recoder 分别修复了 35 个和 17 个缺陷，也比在这两个基准测试上评估的现有 APR 工具表现得更好。结果表明，Recoder 的性能和泛化能力优于现有方法。据已有信息所知，*这是第一个在性能上超越传统 APR 方法的基于深度学习的 APR 方法*。

总结来说，本章做出了以下贡献：

- 本章提出了一种用于 APR 的语法引导编辑解码器，采用提供者/决策者架构，以准确预测编辑，并确保编辑后的程序在语法上正确，并使用占位符生成具有项目特定标识符的补丁。
- 本章设计了基于上述解码器架构的基于神经网络的程序自动修复方法 Recoder。
- 本章在 Defects4J v1.2 的 395 个缺陷和 Defects4J v2.0 的额外 420 个缺陷上评估了 Recoder。结果显示，Recoder 在单块缺陷的修复性能和泛化能力方面显著优于现有的最先进方法。

8.2 编辑操作的定义

在本节中将介绍编辑的语法和语义及其与提供者之间的关系。下一小节将讨论生成编辑和实现提供者的神经架构。

8.2.1 编辑操作的语法与语义

图8.4展示了编辑的语法。请注意，本方法并不特定于某一特定编程语言，可以应用于任何具有与语句相似概念的编程语言（称为*宿主语言*）。具体来说，当程序中存在一个语句时，同一位置也可以存在语句序列，即再任何现有语句之前插入一个语句仍然会得到一个语法正确的程序。为了确保编辑后的程序的语法正确性，编辑的语法依赖于宿主语言的语法。在图8.4中，“HL”指本方法适用的宿主编程语言。下文将依次解释图8.4中的每条规则。

根据规则 1 和规则 2 的定义，*Edits* 是由特殊符号 `end` 结束的 *Edit* 序列。一个 *Edit* 可以是 `insert` 和 `modify` 这两种编辑操作之一。

1. *Edits* → *Edit*; *Edits* | `end`
2. *Edit* → *Insert* | *Modify*
3. *Insert* → `insert(`⟨*HLStatement*⟩`)`
4. *Modify* → `modify(`
   ⟨*ID of an AST Node with a NTS*⟩,
   ⟨*the same NTS as the above NTS*⟩`)`
5. ⟨*Any NTS in HL*⟩ →
   `copy(`⟨*ID of an AST Node with the same NTS*⟩`)`
   | ⟨*The original production rules in HL*⟩
6. ⟨*HLIdentifier*⟩ → `placeholder`
   | ⟨*Identifiers in the training set*⟩

"HL" 代表 "宿主语言". "NTS" 代表 "非终结符". "⟨*HLStatement*⟩" 是宿主语言中的一个语句. "⟨*HLIdentifier*⟩" 是宿主语言中的一个标识符.

图 8.4　编辑的语法

规则 3 定义了 `insert` 操作的语法。`insert` 操作在错误语句之前插入一个新生成的语句。如规则 3 所示，`insert` 操作的参数是要插入的语句。这里的 ⟨*HLStatement*⟩ 指的是宿主语言语法中代表语句的非终结符。这个非终结符可以扩展成一个完整的语句，或者是一个从原程序复制的语句的复制操作，或者两者的混合。这种行为将在规则 5 中进一步解释。

规则 4 定义了 `modify` 操作的语法。`modify` 操作用新的 AST 子树替换错误语句中的 AST 子树。`modify` 操作有两个参数。第一个参数是要被替换的 AST 子树的根节点的 ID。节点的 ID 定义为节点在前序遍历序列中的顺序，如第六个访问的节点的 ID 为 6。第二个参数是一个 AST 子树，其根节点具有相同的符号，即根节点不能被改变。通过这种方式，替换确保了语法正确性。为确保有实际的变更，被替换的子树应该有不止一个节点，即根节点应该有一个非终结符号。

对于 `insert` 和 `modify`，本方法需要生成一个新的 AST 子树。值得注意的是，在许多补丁中，被插入或修改的 AST 子树并不完全需要从头生成。它的一些子树可能从程序的其他部分复制。利用这一属性，引入了 `copy` 操作来进一步减少补丁空间。规则 5 定义了这个操作的语法。这是一个应用于宿主语言的任何非终结符号的元规则。对于宿主语言中的任何非终结符号，本方法添加一个将其扩展为 `copy` 操作的产生规则。这个非终结符的原始产生规则也被保留，这样在生成编辑时，神经网络可以选择直接生成一个新的子树或复制一个。

`copy` 操作有一个参数，用于识别要复制的 AST 子树的根节点。AST 子树可以从具有缺陷的语句或其上下文中选择。在当前的实现中，允许从包含缺陷语句的方法中复制。此外，为了确保语法正确性，要复制的子树的根节点应该与被扩展的符号具有相

表 8.1　非终结符的提供者

组成成分	关联的非终结符
规则预测器	*Edits, Edit, Insert,* ⟨*HLIdentifier*⟩, ⟨宿主语言中的 *NTS*⟩
子树定位器	*Modify*
子树拷贝器	⟨宿主语言中的 *NTS*⟩

同的非终结符号。

最后，规则 6 将 `placeholder` 引入到语法中。通常，编程语言的语法使用终结符来表示标识符。为了使神经网络能够生成具体的标识符以及 `placeholder`，这里将标识符节点变为非终结符，它们扩展为 `placeholder` 或训练集中的常用标识符之一。在当前的实现中，如果一个标识符在训练集中出现超过 100 次，则被认为是常用的。

在应用编辑时，`placeholder` 将被替换为上下文中可行的标识符。首先通过执行词法分析来收集当前项目中的所有标识符，并收集词法类型为 ⟨`HLIdentifier`⟩ 的词牌，该符号代表宿主语言中的标识符。然后根据以下标准过滤标识符：(1) 标识符可以从本地上下文访问，并且 (2) 用标识符替换占位符不会导致类型错误。剩余的标识符是可行的标识符。

图 8.5和图 8.6展示了由编辑表示的两个示例的程序补丁。图 8.5中的补丁插入了一个 `if` 语句，其中的条件表达式包含了从错误语句中复制的方法调用。图 8.6中的补丁用另一个调用替换了方法调用的限定符，其中方法的名称是一个稍后将实例化的占位符。

定理 1. 编辑后的程序在语法上是正确的。

证明. 通过对编辑的语法进行结构归纳，可以轻易看出该定理成立。首先，宿主编程语言要求确保在另一个语句之前插入一个语句在语法上是正确的。其次，当使用 `modify` 替换一个子树时，子树的根符号保持不变。第三，`insert` 和 `modify` 中的新子树是通过使用宿主语言的语法规则生成的，或者是复制具有相同根符号的子树。最后，实例化一个 `placeholder` 确保了语法正确性，因为本方法只用词法类型为 ⟨`HLIdentifier`⟩ 的词牌替换 `placeholder`。 □

8.2.2　编辑操作的生成

由于扩展非终结符可能取决于局部上下文或先前的语法规则的选择，本方法使用提供者来提供选择并估计其概率。当前实现中有三种类型的提供者。表 8.1显示了这些提供者及其关联的非终端符号。

```
+ if((interfaceType.getImplicitPrototype() == null)){
+     return;
+ }
ObjectType implicitProto = interfaceType.getImplicitPrototype();
```

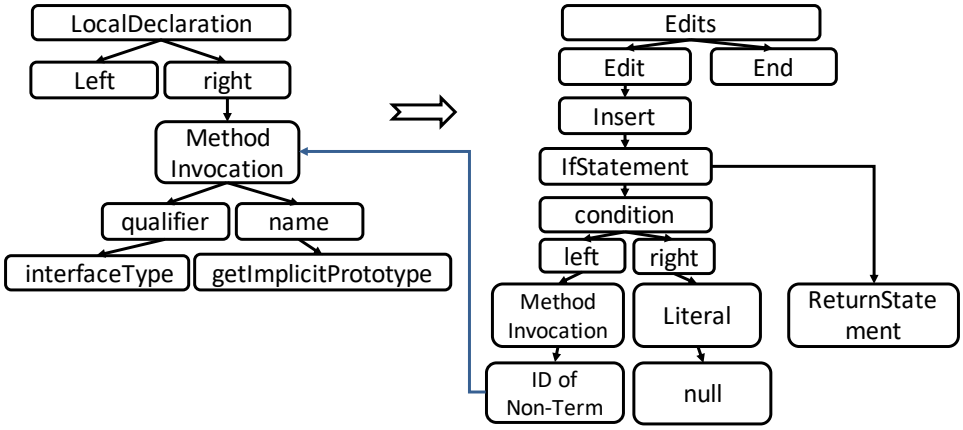


图 8.5　Insert 操作的示例 (Closure-2)

对于非终结符 *Edits, Edit, Insert* 和 ⟨*HLIdentifier*⟩，规则预测器负责提供选择并估计每个语法规则的概率。规则预测器由神经组件和逻辑组件组成。在神经组件为每个产生规则分配概率之后，逻辑组件将左侧不是相应非终端符号的规则的概率重置为零，并归一化剩余概率。对于 *Modify* 操作，子树定位器负责提供选择的子树节点。子树定位器估计错误声明中大于 1 的每个 AST 子树的概率。选择子树 *t* 意味着本方法应该将 *Modify* 展开为 `modify(`*ID*`,` *NTS*`)`，其中 *ID* 是 *t* 的根 ID，而 *NTS* 是 *t* 的根符号。对于宿主语言的语法中的任何非终结符（注意 ⟨*HLIdentifier*⟩ 是宿主语言中的终结符），规则预测器和子树复制器都负责提供选择。子树复制器需要估计具有缺陷的方法中的每

```
-   return cAvailableLocaleSet.contains(locale);
+   return availableLocaleSet().contains(locale);
```

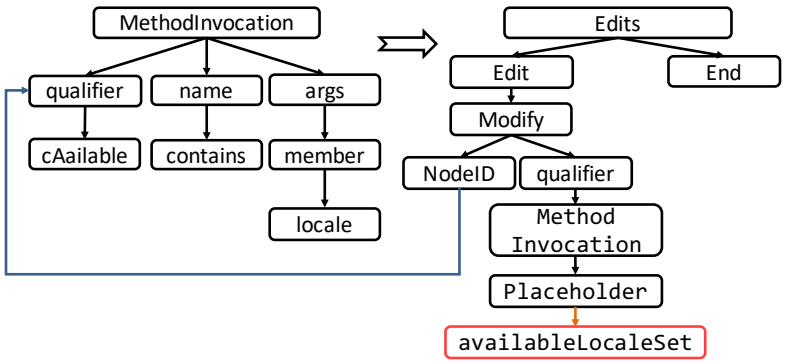


图 8.6　Modify 操作的示例 (Lang-57)

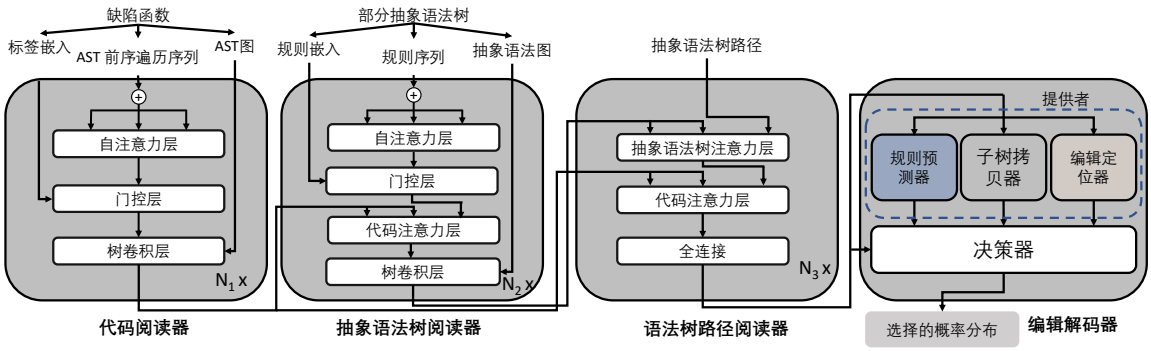


图 8.7　Recoder 的总览

个 AST 子树的概率。选择子树 t 意味着本方法应该将非终端展开为 copy(ID)，其中 ID 是 t 的根 ID。与规则预测器类似，树复制器在神经组件之后使用逻辑组件来重置根符号与被扩展的非终端符号不同的子树的概率。

8.3　模型架构

本模型的设计基于最先进的语法指导代码生成模型 TreeGen[61]。这是一个基于树的 Transformer[69]模型，可以将自然语言描述作为输入，并输出程序。由于本方法采用的是将缺陷语句及其上下文作为输入，并输出编辑操作，因此替换了 TreeGen 中用于编码自然语言描述和解码程序的组件。

图 8.7展示了模型的概览。该模型执行编辑生成过程中的预测扩展非终结节点的选择概率这一步骤，然后使用束搜索（Beam Search）来找到生成完整编辑的最佳选择序列。模型主要包含四个组成部分：

- **代码阅读器**，用于编码错误语句及其上下文。
- **AST 阅读器**，用于编码已生成编辑的部分 AST。
- **树路径阅读器**，用于编码从根节点到应当扩展的非终结节点的路径。
- **编辑解码器**，利用前三个组件的编码信息，生成扩展非终结节点每个选择的概率。

其中，AST 阅读器和树路径阅读器源自 TreeGen，而代码阅读器和编辑解码器是在本方法中新引入的，将重点描述后两个组件的细节。

8.3.1　代码阅读器

代码阅读器组件编码错误语句及其周围方法作为上下文，其中错误语句通过错误定位技术进行定位。它使用以下三种输入:**(1) AST 遍历序列**。这是一个词牌序列，遵循 AST 的前序遍历，$\mathbf{c}_1, \mathbf{c}_2, \cdots, \mathbf{c}_L$，其中 $\mathbf{c}_i$ 是第 i 个节点的词牌编码向量，通过词嵌

入[174]得到。**(2) 标签嵌入**。这是一个标签序列，遵循 AST 的相同前序遍历，其中每个标签表示相应节点属于以下哪种情况：1. 在错误语句中，2. 在错误语句之前的语句中，3. 在错误语句之后的语句中，4. 在其他语句中。每个标签表示为嵌入查找表中的元素，即将标签嵌入表示为 $\mathbf{t}_1, \mathbf{t}_2, \cdots, \mathbf{t}_L$。**(3) 基于 AST 的图**。考虑到前两个输入没有捕捉 AST 节点之间的邻接关系，为了捕获这种信息，本方法将 AST 视为一个有向图，其中节点是 AST 节点，边将一个节点连接到其每个子节点和左侧兄弟节点，如图 8.8(b) 所示。这个图被嵌入为一个邻接矩阵。

代码阅读器使用三个子层来编码上述三种输入，三个子层如下所述。

自注意力。自注意力子层编码 AST 遍历序列，遵循 Transformer[69]架构来捕获 AST 中的长依赖信息。

给定输入 AST 遍历序列的嵌入，本方法使用位置嵌入来表示 AST 令牌的位置信息。输入向量表示为 $\mathbf{c}_1, \mathbf{c}_2, \cdots, \mathbf{c}_L$，第 i 个令牌的位置嵌入计算如下：

$$p_{(i,2j)} = \sin(pos/(10000^{2j/d})) \tag{8.1}$$

$$p_{(i,2j+1)} = \cos(pos/(10000^{2j/d})) \tag{8.2}$$

其中 $pos = i + step$，j 表示输入向量的元素，$step$ 表示嵌入大小。获得每个位置的向量后，直接将其加到相应的输入向量上，其中 $\mathbf{e}_i = \mathbf{c}_i + \mathbf{p}_i$。

然后，本方法采用多头注意力层来捕获非线性特征。根据 Vaswani 等人[69]的定义将注意力机制分为 H 个头，每个头代表一个独立的注意力层，以提取独特信息。单个注意力层将查询 Q、键 K 和值 V 映射到一个加权求和的输出。第 j 个头层的计算可以表示为

$$head_j = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V \tag{8.3}$$

其中 $d_k = d/H$ 表示每个提取特征向量的长度，而 Q、K 和 V 是通过全连接层从 Q、K、V 计算得到的。在编码器中，向量 Q、K 和 V 都是位置嵌入层 $\mathbf{e}_1, \mathbf{e}_2, \cdots, \mathbf{e}_L$ 的输出。这些头的输出进一步通过一个全连接层联合起来，计算方式为

$$Out = [head_1; \cdots; head_H] \cdot W_h \tag{8.4}$$

其中 W_h 表示全连接层的权重，Out 表示自注意力子层的输出 $\mathbf{a}_1, \mathbf{a}_2, \cdots, \mathbf{a}_L$。

门控层。该子层将前一层的输出和标签嵌入作为输入，使用了 TreeGen[61]中定义的门控机制，将三个向量 $\mathbf{q}, \mathbf{c}_1, \mathbf{c}_2$ 作为输入，并基于 $\mathbf{q}$ 将 c_1 与 c_2 结合。门控机制的计

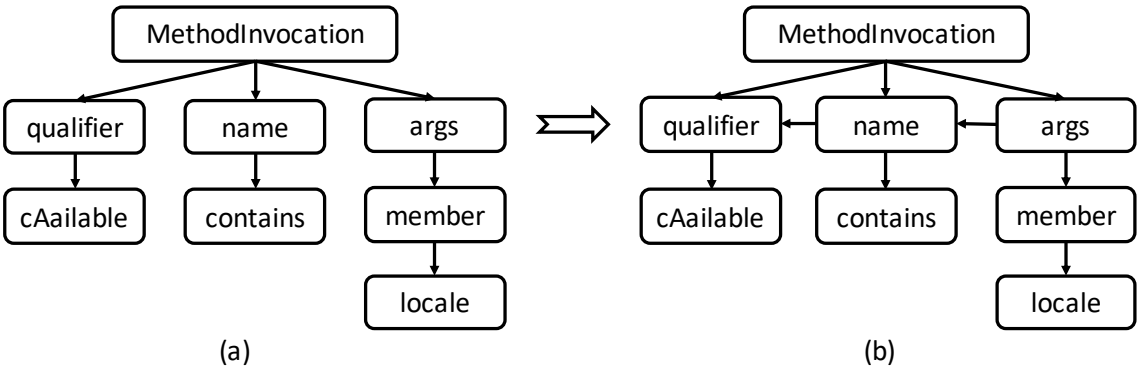


图 8.8　抽象语法图

算可以表示为

$$\alpha_i^{c_1} = \exp(\mathbf{q}_i^T \mathbf{k}_i^{c_1})/\sqrt{d_k} \tag{8.5}$$

$$\alpha_i^{c_2} = \exp(\mathbf{q}_i^T \mathbf{k}_i^{c_2})/\sqrt{d_k} \tag{8.6}$$

$$\mathbf{h}_i = (\alpha_i^{c_1}\mathbf{v}_i^{c_1} + \alpha_i^{c_2}\mathbf{v}_i^{c_2})/(\alpha_i^{c_1} + \alpha_i^{c_2}) \tag{8.7}$$

其中 $d_k = d/H$ 是一个归一化因子，H 表示头的数量，d 表示隐藏大小；$\mathbf{q}_i$ 通过控制向量 $\mathbf{q}_i$ 上的一个全连接层计算得出；$\mathbf{k}_i^{c_1}, \mathbf{v}_i^{c_1}$ 通过另一个全连接层计算得出，该层作用于向量 $\mathbf{c}_1$ 上；$\mathbf{k}_i^{c_2}$ 和 $\mathbf{v}_i^{c_2}$ 也通过相同的层但参数不同计算得出，作用于向量 $\mathbf{c}_2$ 上。

在本模型中,将自注意力子层的输出 $\mathbf{a}_1, \mathbf{a}_2, \cdots, \mathbf{a}_L$ 视为 $\mathbf{q}$ 和 $\mathbf{c}_1$,将标签嵌入 $\mathbf{t}_1, \mathbf{t}_2, \cdots, \mathbf{t}_L$ 视为 $\mathbf{c}_2$。因此，门控层的第 i 个 AST 节点的嵌入可以表示为 $\mathbf{u}_i = \text{Gating}(\mathbf{a}_i, \mathbf{a}_i, \mathbf{t}_i)$。

树卷积层。该子层将前一层的输出 $\mathbf{u}_i$ 和基于 AST 的图 G（表示为邻接矩阵）作为输入，同时采用一个 GNN[175-176]层来处理输入，邻居的编码 r_i 计算如下

$$\mathbf{g}_i = W_g \sum_{r^j \in G} A^n_{r^i r^j} \mathbf{u}^j \tag{8.8}$$

其中 W_g 是全连接层的权重，$\hat{A}$ 是 G 的归一化邻接矩阵，计算公式为 Kipf et al.[41]提出的 $\hat{A} = S_1^{-1/2} A S_2^{-1/2}$，其中 A 是 G 的邻接矩阵，S_1, S_2 是对角矩阵，其对 A 的列和行进行求和。随后邻居的编码直接加到输入向量上。

代码阅读器由 N_1 块这三个子层组成，产生输入 AST 的特征，$\mathbf{t}_1, \mathbf{t}_2, \cdots, \mathbf{t}_L$，这些将被用于 AST 阅读器和树路径阅读器。

8.3.2　AST 阅读器

AST 阅读器编码编辑的部分生成的 AST，其结构与 TreeGen[61]中的一致。该组件与代码阅读器类似，从部分生成的 AST 中派生出三个输入。规则序列被表示为实值向量后送入一个自注意力层，通过门控层将自注意力层的输出与规则编码整合，如等

式 8.7所示；同时还采用一个多头注意力层处理代码阅读器和门控层的输出，类似于 Transformer 中的解码器-编码器注意力机制。最后使用一个树卷积层，以提取结构信息。此组件的更多细节可参考 TreeGen[61]的原论文。

8.3.3 树路径阅读器

树路径阅读器与 TreeGen[61]中的相同，编码待扩展的非终结节点的信息。该组件将非终结节点表示为从根到待扩展节点的路径，并将此路径中的节点转换为实值向量。如图 8.7所示，这些向量被送入两个注意力层，类似于等式 8.4，随后是一组两个全连接层，其中第一层使用 $GELU$[177]激活函数，用于提取编辑解码器的特征。此组件的更多细节可参考 TreeGen[61]原论文，这里将树路径阅读器的输出表示为 $\mathbf{d}_1, \mathbf{d}_2, \cdots, \mathbf{d}_T$。

8.3.4 编辑解码器

编辑解码器将树路径阅读器的输出 $\mathbf{d}_1, \mathbf{d}_2, \cdots, \mathbf{d}_T$（长度为 T）作为输入。这些向量由树路径阅读器产生，包含了所有输入的编码信息：错误语句及其周围的方法、到目前为止生成的部分 AST，以及表示待扩展节点的树路径。

提供者。如前所述，目前有三种类型的提供者：规则预测器、树复制器和子树定位器。这些提供者将向量 $\mathbf{d}_1, \mathbf{d}_2, \cdots, \mathbf{d}_T$ 作为输入，并输出不同非终结符选择的概率。

规则预测器。规则预测器估计编辑语法中每个产生规则的概率。这个决策器的神经组成部分包括一个全连接层。全连接层的输出表示为 $\mathbf{s}_1, \mathbf{s}_2, \cdots, \mathbf{s}_T$。这些向量通过 softmax 归一化，计算归一化向量 $\mathbf{p}_1^r, \mathbf{p}_2^r, \cdots, \mathbf{p}_T^r$：

$$\mathbf{p}_k^r(m) = \frac{\exp \mathbf{s}_k^m}{\sum j = 1^{N_r} \exp \mathbf{s}_k^j} \tag{8.9}$$

其中 N_r 表示编辑语法中产生规则的数量，m 表示向量 $\mathbf{p}_k^r$ 的第 m 维（即 ID 为 m 的产生规则）。

本方法不允许左侧不是相应非终结符的无效规则。对于这些规则，逻辑组件将全连接层的输出重置为 $-\infty$。因此，经过 softmax 归一化后，无效规则的概率将为零。

子树复制器。这种提供者用于为编辑语法中的任何非终结符号选择局部上下文中的一个子树，基于指针网络[178]，计算可以表示为

$$\theta_i = \mathbf{v}^T \tanh(W_1 \mathbf{d}_i + W_2 \mathbf{t}) \tag{8.10}$$

其中 $\mathbf{t}$ 表示代码阅读器的输出，$\mathbf{v}, W_1, W_2$ 表示可训练参数。如果相应子树的根符号与正在扩展的符号不同，逻辑组件也将 θ 重置为 $-\infty$。然后通过 softmax 归一化这些向量，如等式 8.10所示。这里将归一化向量表示为 $\mathbf{p}_1^t, \mathbf{p}_2^t, \cdots, \mathbf{p}_T^t$。

子树定位器。此组件为非终结符号 *Modify* 在编辑语法中输出错误语句的子树的ID。此组件的计算与树复制器相同。这里将此提供者的输出向量表示为 $\mathbf{p}_1^s, \mathbf{p}_2^s, \cdots, \mathbf{p}_T^s$。

决策器。对于这三个提供者，决策器估计使用每个提供者的概率。该组件将树路径阅读器的输出 $\mathbf{d}_1, \mathbf{d}_2, \cdots, \mathbf{d}_T$ 作为输入，并输出使用每个提供者的概率，其计算过程可以表示为 $\lambda_i = W\mathbf{d}_i + \mathbf{b}$，其中 W 和 $\mathbf{b}$ 表示全连接层的参数。如果相应的提供者不负责正在扩展的符号，逻辑组件将 λ 重置为 $-\infty$，如表 8.1所示。随后通过 softmax 归一化这些向量，如等式8.9所示。这里将归一化向量表示为 $\lambda_1, \lambda_2, \cdots, \lambda_T$。每个选择的最终概率可以计算为

$$\mathbf{o}_i = [\lambda_i^r \mathbf{p}_i^r; \lambda_i^t \mathbf{p}_i^t; \lambda_i^s \mathbf{p}_i^s] \tag{8.11}$$

其中 $\mathbf{o}_i$ 是在补丁生成过程中第 i 步的下一个产生规则的概率向量。

8.3.5 训练与推理

在训练期间，本方法通过最大化最优编辑序列的负对数似然来优化模型，并且不在提供者和决策器中使用逻辑组件，因为希望 Recoder 学习逻辑组件处理的规则的分布：如果训练时存在逻辑组件，Recoder 将不会针对大部分规则进行训练；在推理时，这些未见过的规则会扭曲输出的分布，使 Recoder 无法区分它应该区分的规则部分。

在生成编辑时，推理从规则 *start* ： *start* $\longrightarrow$ *Edits* 开始，将特殊符号 `start` 扩展为 `Edits`。如果预测的 AST 中的每个叶节点都是终结符，则递归预测终止。本章使用大小为 100 的束搜索来生成多个编辑。

生成的编辑可能包含占位符。尽管单个占位符的选择数量较少，但多个占位符的组合可能很大。因此，在进行束搜索时丢弃包含多于一个 `placeholder` 符号的补丁。

8.3.6 补丁生成与验证

在本方法中，根据缺陷定位的结果，上述描述的模型被用于每一个可疑的错误语句。对于每个语句，通过束搜索生成 100 个有效的补丁候选：当束搜索生成一个有效补丁时，将其从搜索集中移除，并继续搜索下一个补丁，直到总共为该语句生成 100 个候选补丁。补丁生成后，最后一步是通过开发者编写的测试套件进行验证。验证步骤过滤掉不编译或未通过测试用例的补丁。所有生成的补丁都经过验证，直到找到一个合理的补丁（通过所有测试用例的补丁）。

8.4 实验设置

本方法主要在 Java 编程语言实现了 Recoder ，在这里报告修复 Java 缺陷的实验结果。

8.4.1　研究问题

本评估旨在回答以下研究问题：

问题 1: Recoder 的性能如何？为了回答这个问题，这里在广泛使用的 APR 基准 *Defects4J v1.2* 上评估了提出的方法，并将其与传统以及基于深度学习的 APR 工具进行了比较。

问题 2: Recoder 中每个组件的贡献是什么？为了回答这个问题，从 Recoder 的完整模型开始，依次移除每个组件，以了解每个组件对性能的贡献。

问题 3: Recoder 的泛化性如何？为了回答这个问题，首先在 *Defects4J v2.0* 的额外 420 个程序缺陷上进行了实验，同时将 Recoder 与 Defects4J v1.2 上针对单块程序缺陷的前两种具有最佳性能的 APR 方法，即 TBar[73]和 SimFix[74]进行了比较。此外，本章还通过 *RepairThemAll*[83]框架将 Recoder 应用于其他两个基准，*QuixBugs* 和 *IntroClassJava*，该框架允许在程序自动修复的基准上执行自动程序修复工具。

8.4.2　数据集

本方法中的神经网络模型需要用大量历史补丁进行训练。为了创建这个训练集，作者爬取了 2011 年 3 月至 2018 年 3 月之间在 GitHub[179]上创建的 Java 项目，并下载了 1,083,185 个项目提交，其中项目提交信息至少包含以下两组单词中的一个：(1) *fix, solve*；(2) *bug, issue, problem, error*。项目提交被过滤来筛选只包括修改单一语句或插入新语句的补丁，这对应本方法当前支持的两种编辑类型。为了避免数据泄露，本方法进一步做了数据清洗，其中（1）项目是 Defects4J 项目的克隆或使用 Defects4J 的程序修复项目，或（2）补丁修改的方法与 Defects4J v1.2 或 v2.0 中任何补丁修改的方法相同。过滤后这部分数据集剩下 103,585 个有效补丁，这些补丁进一步分为两部分：80% 用于训练，20% 用于验证。

本节使用了四个基准数据集来衡量 Recoder 的性能。第一个包含来自 *Defects4J* v1.2[71]的 395 个程序缺陷，这是自动程序修复研究中常用的基准测试集。第二个包含来自 *Defects4J v2.0*[71]的 420 个额外程序缺陷。Defects4J v2.0 比 Defects4J v1.2 新增了 438 个程序缺陷。然而，GZoltar[76]未能在项目 *Gson* 上成功运行，所以从基准测试集中去掉了 *Gson* 中的 18 个缺陷。第三个包含来自 *QuixBugs*[72]的 40 个缺陷，这是一个由测试用例指定的有缺陷的算法程序的基准测试集。最后一个基准测试集 *IntroClassJava*[180]，由来自 *IntroClass*[181] C 语言基准的 297 个有错误的 Java 程序组成。

8.4.3 缺陷定位

在本实验中，使用了两种缺陷定位设置。在第一种设置中，缺陷的位置对 APR 工具是未知的，它们依赖现有的缺陷定位方法来得到缺陷位置。Recoder 使用 Ochiai[182]（在 GZoltar[76]中实现），这是现有 APR 工具中广泛使用的方法[73-74]。在第二种设置中，APR 工具已知实际的缺陷位置。这是为了在不受特定缺陷定位工具影响的情况下，衡量补丁生成的能力，与之前的研究中的实验设置一致[62-64]。

8.4.4 对比的 APR 技术

本节选择了现有的 APR 方法作为比较的基准技术。由于 Recoder 只生成单块补丁（只更改连续代码片段的补丁），本章选择了 10 种在现有研究中经常用作基准技术的针对传统单块的 APR 方法：jGenProg[75]，HDRepair[183]，Nopol[184]，CapGen[77]，SketchFix[185]，TBar[73]，FixMiner[186]，SimFix[74]，PraPR[187]，AVATAR[188]。就本章所知，TBar 在 Defects4J v1.2 上正确修复的缺陷数量最多。本章还选择了采用编码器-解码器架构生成补丁并在 Defects4J 上进行评估的基于深度学习的 APR 方法作为对比基准。基于这个标准，选择了四种方法，分别是 SequenceR[62]，CODIT[63]，DLFix[55]和 CoCoNuT[64]。

对于 Defects4J v1.2，对比技术的性能数据从现有论文中收集[73,189]。对于 Defects4J v2.0 的额外缺陷，选择了在 Defects4J v1.2 上表现最好的两种 APR 方法，TBar 和 SimFix，进行适配和执行比较。对于 QuixBugs 和 IntroClassJava，这里选择了在 RepairThemAll[83]中使用的 APR 工具和在这两个基准数据集上实验过的基于深度学习的 APR 工具作为基线：jGenProg[75]，RSRepair[158]，Nopol[184]和 CoCoNuT[64]，使用原始论文中报告的结果[64,83]。

8.4.5 补丁的正确性

为了检查补丁的正确性，本文作者手动检查了每个补丁，看它是否与 Defects4J 提供的补丁相同或语义等价。这种做法和先前的相关工作一致[55,64,73-74,187]。

8.4.6 实现细节

本方法基于 PyTorch[190]实现，参数设置为 $N_1 = 5, N_2 = 9, N_3 = 2$，即代码阅读器包含 5 个块的堆栈，AST 阅读器包含 9 个块的堆栈，解码器包含 2 个块的堆栈。所有嵌入向量的嵌入大小设置为 256，所有隐藏层的大小都遵循 TreeGen[61]的配置。在训练过程中，使用 dropout[191]来防止过拟合，丢弃率为 0.1。模型通过 Adam[192]优化，学习率为 0.0001。这些超参数和模型参数是基于验证集上的性能选择的。遵循现有相关工

作的实验设置[55,64,74,193]，这里为 Recoder 设置了 5 小时的运行时间限制。

8.5　实验结果

8.5.1　问题 1：Recoder 的性能

没有完美缺陷定位的结果　首先，这里在没有给出缺陷位置的情况下比较了 Recoder 与基准线的性能。表 8.2所示的结果仅包括在此设置下评估过的对比技术。结果显示，Recoder 正确修复了 51 个缺陷，并且在 Defects4J v1.2 上超越了所有先前的 APR 技术。特别是，Recoder 比之前最先进的 APR 工具 TBar 多修复了 21.4%（9 个）的缺陷。和已有技术相比，Recoder 是第一个在性能上超越传统 APR 方法的基于深度学习的 APR 方法。

表 8.2　没有完美缺陷定位的结果比较

项目名称	jGenProg	HDRepair	Nopol	CapGen	SketchFix	FixMiner	SimFix	TBar	DLFix	PraPR	AVATAR	Recoder
Chart	0/7	0/2	1/6	4/4	6/8	5/8	4/8	**9/14**	5/12	4/14	5/12	8/14
Closure	0/0	0/7	0/0	0/0	3/5	5/5	6/8	8/12	6/10	12/62	8/12	**15/31**
Lang	0/0	2/6	3/7	5/5	3/4	2/3	**9/13**	5/14	5/12	3/19	5/11	**9/15**
Math	5/18	4/7	1/21	12/16	7/8	12/14	14/26	**18/36**	12/28	6/40	6/13	15/30
Time	0/2	0/1	0/1	0/0	0/1	1/1	1/1	1/3	1/2	0/7	1/3	**2/2**
Mockito	0/0	0/0	0/0	0/0	0/0	0/0	0/0	1/2	1/1	1/6	**2/2**	**2/2**
总和	5/27	6/23	5/35	21/25	19/26	25/31	34/56	42/81	30/65	26/148	27/53	**51/94**
百分比 (%)	18.5	26.1	14.3	**84.0**	73.1	80.6	60.7	51.9	46.2	17.6	50.9	54.3

本节展示了一些可能借助 Recoder 中的新技术生成的示例补丁。如图 8.9所示，Chart-8 是一个 DLFix 无法修复的缺陷。正确的补丁只改变了方法调用的一个参数，而 DLFix 需要生成整个表达式。相比之下，Recoder 生成了一个修改操作，只更改了一个参数。图 8.10展示了一个仅由 Recoder 修复的缺陷。这个补丁依赖于一个项目特定的方法，“isNoType”，因此许多现有的方法无法生成。然而，Recoder 通过生成一个占位符然后用“isNoType”实例化它来正确修复了它。

```
- this(time, RegularTimePeriod.DEFAULT_TIME_ZONE,Locale.getDefault());
+ this(time, zone, Locale.getDefault());
```

图 8.9　Chart-8 - Recoder 用 Modify 操作修复的缺陷

```
-  if (result != null) {
+ if(((result != null) && !result.isNoType())){
```

图 8.10　Closure-104 - Recoder 用占位符生成修复的缺陷

完美缺陷定位的结果表 8.3显示了提供了实际缺陷位置情况下的结果。如之前一样，仅列出了在此设置下评估过的对比技术。Recoder 仍然超越了所有现有的 APR 方

法，包括传统方法。此外，与使用 Ochiai 进行缺陷定位的 Recoder 相比，该模型实现了 35.3% 的改进。这个结果表明，Recoder 在更好的缺陷定位技术支持下可以实现更好的性能。

表 8.3　提供缺陷位置时的结果比较

项目名称	SequenceR	CODIT	DLFix	CoCoNuT	TBar	Recoder
Chart	3	4	5	7	**11**	10
Closure	3	3	11	9	17	**23**
Lang	3	3	8	7	**13**	10
Math	4	6	13	16	**22**	18
Time	0	0	2	1	2	**3**
Mockito	0	0	1	**4**	3	2
总和	13	16	40	44	**68**	66

互补程度 本节进一步研究了 Recoder 在修复单块缺陷方面与三种表现最佳的现有

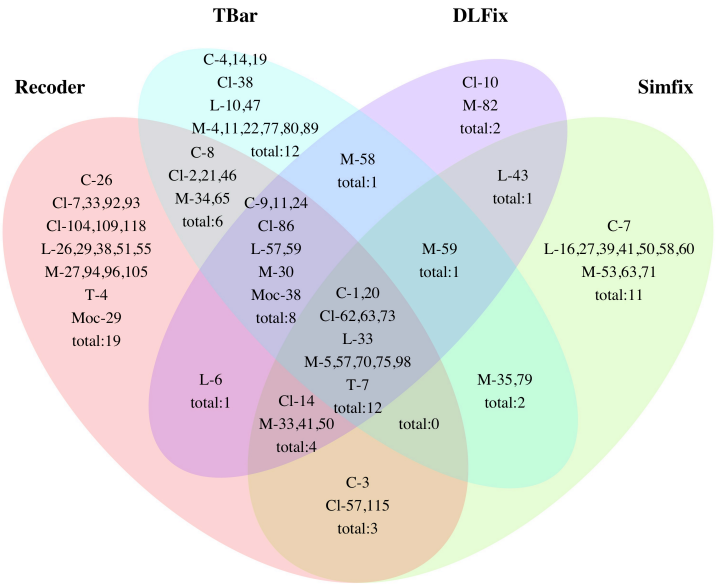


图 8.11　互补程度
项目名称：C:Chart, CL:Closure, L:Lang, M:Math, Moc:Mockito, T:Time

方法（TBar、SimFix 和 DLFix）的互补程度。图 8.11揭示了不同方法修复的缺陷之间的重叠情况。如图所示，与三个对比技术相比，Recoder 修复了 19 个独特的缺陷。此外，与 SimFix、TBar 和 DLFix 相比，Recoder 分别修复了 34 个、28 个和 27 个独特的缺陷。这个结果显示了 Recoder 与这些表现最佳的现有方法相互补充。

表 8.4　Defects4J v1.2 上的对照实验 Recoder

项目名称	-modify	-subtreecopy	-insert	-placeholder	Recoder
Chart	4	6	7	8	8
Closure	6	12	12	11	15
Lang	3	6	5	5	9
Math	7	8	9	9	15
Time	1	1	1	1	2
Mockito	2	1	1	1	2
总和	23	34	35	35	51

8.5.2　问题 2：各个组件的贡献

为了回答问题 2，本节在 Defects4J v1.2 上进行了消融实验，以确定每个组件的贡献。由于消融测试需要很多时间，本节只在 Ochiai 缺陷定位场景进行了实验。

表 8.4显示了消融测试的结果。本节分别移除了三种编辑操作，即修改、复制和插入，以及占位符的生成。如表中所示，移除任何组件都会导致性能显著下降。这个结果表明，Recoder 中提出的两种新技术是其性能的关键。

8.5.3　问题 3：Recoder 的泛化能力

表 8.5　420 个额外的缺陷上的比较

项目名称	# Used Bugs	Bug IDs	TBar	SimFix	Recoder
Cli	39	1-5,7-40	1/7	0/4	**3/3**
Clousre	43	134 - 176	0/5	**1/5**	0/7
JacksonDatabind	112	1-112	0/0	0/0	0/0
Codec	18	1-18	**2/6**	0/2	**2/2**
Collections	4	25-28	0/1	0/1	0/0
Compress	47	1-47	1/13	0/6	**3/9**
Csv	16	1-16	1/5	0/2	**4/4**
JacksonCore	26	1-26	0/6	0/0	0/4
Jsoup	93	1-93	3/7	1/5	**7/13**
JxPath	22	1-22	0/0	0/0	0/4
总和	420	-	8/50	2/25	**19/46**

在 Defects4J v2.0、QuixBugs 和 IntroClassJava 上的结果分别展示在表 8.5和表 8.6中。如结果所示，在 Defects4J v2.0 上，所有三种方法修复的缺陷比例都较小，这表明 Defects4J v2.0 上的额外缺陷可能更难以修复。尽管如此，与基线相比，Recoder 仍然修复了最多的缺陷，总共 19 个，比 TBar 多出 137.5%（11 个缺陷）的提升，比 SimFix 多出

表 8.6　在 IntroClassJava 和 QuixBugs 数据集上的结果比较

项目名称	# 使用的缺陷	jGenProg	RSRepair	Nopol	CoCoNuT	Recoder
IntroClassJava	297	1/4	4/22	3/32	-	**35/56**
QuixBugs	40	0/3	2/4	1/4	13/20	**17/17**
Total	337	1/7	6/26	4/36	13/20	**52/73**

850.0%（17 个缺陷）的提升。这里认为 TBar 和 SimFix 性能大幅下降的原因是它们的设计：TBar 是基于在 Defects4J v1.2 上验证的模式，这些模式可能无法泛化到 Defects4J v1.2 之外的项目；SimFix 依赖于同一项目中的相似代码片段，但 Defects4J v2.0 中的新项目要小得多，因此找到相似代码片段的机会也更小。另一方面，Recoder 是从不同项目收集的大量补丁中训练而来的，因此更有可能泛化到新项目。在 QuixBugs 和 IntroClassJava 上，Recoder 也分别比对比技术多修复了 775%（31 个缺陷）和 30.8%（4 个缺陷），进一步证实了 Recoder 的有效性和泛化能力。

8.6　讨论

8.6.1　数据集的充分性

为了理解本方法使用的训练数据的充分性，这里在原始训练数据集的不同大小子集上训练了 Recoder，并计算了在 Defects4J v1.2 数据集中单块缺陷上的损失。对于每个子集，本节训练 5 个具有不同随机种子的模型，并报告这些模型的平均性能。

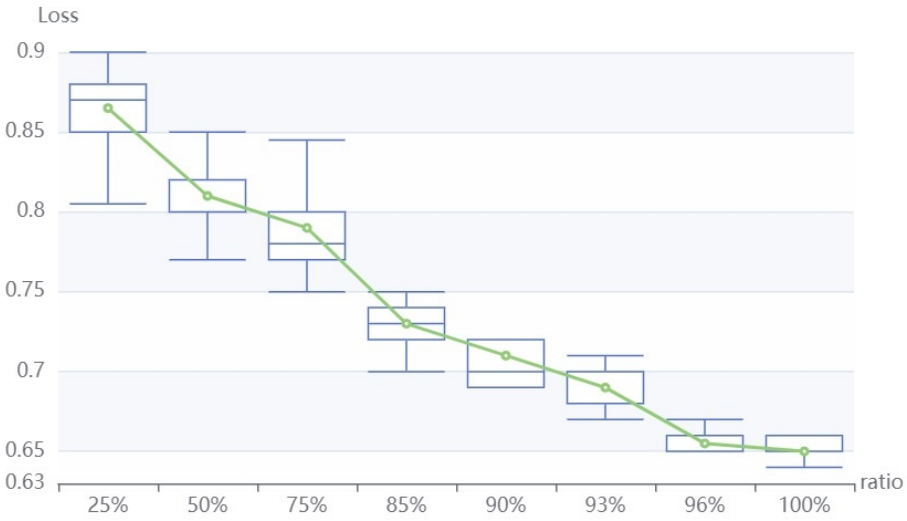


图 8.12　数据集的充分性

8.6.2　Recoder 的局限性

Recoder 存在大多数 APR 方法都有的不足：1) 要修复的缺陷应该通过失败的测试用例来重现。2) 需要有效的故障定位来识别错误语句。Recoder 也受限于深度学习方法对训练集和测试集分布相似的依赖性：如果训练集和测试集的分布不同，性能会下降。

8.7　小结

本章提出了 Recoder，一种带有占位符生成的语法引导编辑解码器，用于自动程序修复。Recoder 采用了创新的提供者/决策者架构，以确保编辑程序的准确生成和语法正确性，并为项目特定的标识符生成占位符。在实验中，Recoder 在 Defects4J v1.2 上针对单块缺陷的修复中，相比于现有的最先进的 APR 方法，取得了 21.4% 的改进（9 个缺陷）。Recoder 是第一个在该基准测试上超越传统 APR 技术的基于深度学习的 APR 方法。在其他三个基准测试上进一步的评估显示，Recoder 比一些最先进的 APR 方法具有更好的泛化能力。

第九章　总结和展望

9.1　本文工作小结

在当今的软件工程领域，随着技术的飞速发展，深度学习已经成为推动行业创新和技术突破的核心动力。特别是在程序理解、自动化编程、程序缺陷检测、程序搜索与修复等关键领域，深度学习技术不仅展示了其解决复杂问题的巨大潜力，也为软件开发和维护工作带来了革命性的改变和前所未有的价值。在这样的技术发展背景下，本文提出了一系列创新的方法和技术策略，专注于探索和利用程序语言的语言定义信息，包括上下文无关语法、类型系统的规则以及上下文引用约束，目的是通过这些高级信息来显著增强深度学习模型在处理编程语言相关任务时的准确性和通用性。

首先，针对深度程序模型，本文针对当前技术只将程序作为一种特殊的自然语言进行处理的缺陷，提出了新颖的引导学习的方法，这些方法专门针对处理程序语言的语言定义的信息。具体而言，本文的方法针对程序语言定义的三个主要内容：上下文无关语法信息、类型规则信息以及声明-使用关系信息，旨在深化深度学习模型对程序结构和语义的理解。这三种技术分别是：

- 引导上下文无关语法（CFG）的学习：在深度学习模型中直接编码程序语法信息是提升代码理解和生成质量的关键。传统的词编码方法在处理 CFG 时面临挑战，因为它们无法充分捕捉到程序的结构层次性和内容信息。本文通过引入图结构来表示 CFG，使用图神经网络（GNN）捕捉规则之间的结构关系和层次联系。GNN 通过节点和边的信息传递机制深入理解程序的结构，同时，门控层技术被用于整合从 GNN 提取的结构信息和直接从规则定义中获取的内容信息，生成一个包含结构和内容特征的综合性向量。这种方法不仅增强了模型对程序结构的理解，而且提高了生成代码的语法准确性和逻辑一致性。
- 引导类型信息的学习：程序设计中，类型系统对维护代码逻辑正确性和减少运行时错误具有重要作用。类型系统的复杂性常常超出神经网络模型的直接学习能力，因此，本文聚焦于单个类型规则的学习，这些规则通常更为简单且容易理解。通过引入一种图表示法和基于自注意力机制的网络结构，模型能够更精确地学习和处理类型信息。这种方法特别关注类型规则之间的约束关系，如赋值语句中变量和表达式的类型兼容性，通过模型学习这些简化的规则，大大提升了处理代码中类型信息的准确度和效率。
- 引导声明-使用关系信息的学习：理解代码中的上下文引用约束，如变量的作用域和生命周期，对于维护代码的逻辑结构和避免名称冲突非常关键。传统的模

型在处理这一层面的信息时存在局限，通常无法准确捕捉跨文件甚至项目级别的引用关系。本文通过在训练语料的组织上采用基于文件级别依赖的方法，利用拓扑排序将项目级别的代码组织成具有相互依赖关系的结构，这样做使得模型能够理解和处理复杂的上下文引用信息。通过这种方法，模型在自动编码、代码补全、错误检测等方面的性能得到了显著提升，填补了先前研究中的空白。

在本文的进一步探索中，为了打破引导学习语言定义的技术对单一编程语言的依赖，开创性地开发了一个语言定义感知的多语言程序编解码预训练模型。该模型综合利用了前文提到的三个核心技术创新：引导上下文无关文法、类型信息以及声明-使用关系信息的学习。通过深入学习和适配多种编程语言的语法规则、类型系统以及变量和函数的作用域等语言定义信息，该模型能够跨越语言边界，实现对不同编程语言特性的深层次理解和有效处理。

这种多语言预训练模型的开发不仅是对现有深度学习技术在程序编解码领域应用的重大拓展，也代表了在实践层面上的重要进步。该模型展现了深度学习技术在处理复杂的跨语言编程任务中的潜力，为未来的研究提供了新的方向和灵感。同时，通过跨语言的深度理解和广泛适配，该模型显著提升了在多种软件工程任务上的性能，包括代码的自动生成、自动修复程序缺陷、生成代码摘要以及改善代码搜索过程等。这些应用不仅提高了软件开发的效率和质量，也降低了软件维护的成本，对促进软件工程领域的技术进步和创新具有重要意义。

在深入探讨本文的多语言深度程序模型后，本文还专门针对程序搜索和程序修复这两项具体的下游任务，引入了一系列先进和创新性的编码技术和结构。这些专门设计的技术不仅增强了模型处理这类复杂任务的能力，而且展示了深度学习技术在软件工程应用中的广泛潜力。

具体来说，为了提升模型在处理代码微小变化和模式差异时的敏感性和精确度，本文引入了字符级别的重叠矩阵技术。这种技术通过在字符级别上细致地比较代码段，能够精确捕捉到代码之间的微小差异和潜在的模式变化，从而大大提高了程序搜索的准确性和效率。针对程序修复任务，本文通过结合编辑操作的扩展语法编码，进一步提升了模型在生成修复方案时的灵活性和准确性。通过这种扩展语法编码，模型能够基于程序的上下文信息生成具体的编辑操作，如添加、删除或替换代码片段，以实现精确的程序修复。这种方法不仅显著提高了自动程序修复的成功率，也大幅度减少了修复过程中的人工干预需求，展示了在自动化软件维护和质量保障方面的巨大应用潜力。

9.2　未来工作

首先，本文应用部分提出的技术具有广泛的通用性，适用于和多种不同的深度学习技术，例如字符级别的重叠度矩阵以及基于编辑语法的补丁表示方法可以和多种技术（Transformer 以及 LSTM）相结合。尽管本文的该部分的实验使用了早期版本的模型进行验证，但这些实验的结论已充分证明了所提方法的有效性。这些实验结果显示，即便在资源限制较大的条件下，所提方法依然能够达到预期的性能标准。由于实验对资源的巨大需求，以及考虑到实现碳中和的环保目标，本研究没有使用最新构造的模型重新进行实验。

未来的研究工作可以采用最新版本的模型，以进一步验证本方法在更新技术条件下的适用性和效果。这不仅可以验证方法的持续有效性，还可以探索其在新环境下的潜在优化空间。这将有助于本领域研究者发现新的研究方向和解决方案，为本领域带来更多创新的可能。

其次，虽然本文提出的方法和技术在多个方面取得了显著的进步，但在深度学习在软件工程领域的应用中仍然存在一些挑战和机遇，未来的研究可以从以下几个方面进行探索：

- 更广泛的编程语言支持：当前的研究主要集中在几种主流的编程语言上，未来可以扩展到更多种类的编程语言，包括但不限于功能性语言、逻辑编程语言等，这将需要对不同语言的特定语言定义进行更深入的研究和适配。
- 更复杂的程序理解任务：随着模型能力的不断提升，可以探索更多复杂的程序理解任务，例如程序的自动优化、自动并行化等，这些任务不仅需要模型理解程序的语法和语义，还需要对程序的性能和执行效率有深刻的理解。
- 程序生成质量的进一步提高：尽管本文提出的技术已经能够生成符合语法和类型约束的代码，但在逻辑正确性和执行效率方面仍有提升的空间。未来的研究可以探索如何让模型更好地理解程序的逻辑结构和算法效率，从而生成不仅正确但也高效的代码。
- 模型可解释性的提升：深度学习模型尤其是大型预训练模型的“黑箱”特性，使得模型的决策过程缺乏透明度。在软件工程的应用中，提高模型的可解释性，让开发者能够理解和信任模型的输出，是提高模型实用性的关键。

总之，本文提出的方法和技术为深度学习在软件工程领域的应用开辟了新的道路，但还有很多值得探索的空间。期待未来的研究能够在这一领域继续取得突破，为软件开发提供更多的自动化支持，进一步提高软件开发的效率和质量。

参考文献

[1] GUPTA R, PAL S, KANADE A, et al. Deepfix: Fixing common c language errors by deep learning [C]//Proceedings of the aaai conference on artificial intelligence: vol. 31: 1. 2017.

[2] CHEN Z, KOMMRUSCH S, TUFANO M, et al. Sequencer: Sequence-to-sequence learning for end-to-end program repair[J]. IEEE Transactions on Software Engineering, 2019, 47(9): 1943-1959.

[3] LEI M, LI H, LI J, et al. Deep learning application on code clone detection: A review of current knowledge[J]. Journal of Systems and Software, 2022, 184: 111141.

[4] MENDIS C, RENDA A, AMARASINGHE S, et al. Ithemal: Accurate, Portable and Fast Basic Block Throughput Estimation using Deep Neural Networks[Z]. 2019. arXiv: 1808.07412 [cs.DC].

[5] AI T. Tabnine[J/OL]., https://www.tabnine.com/.

[6] CHEN M, TWOREK J, JUN H, et al. Evaluating large language models trained on code[J]. arXiv preprint arXiv:2107.03374, 2021.

[7] WANG Y, WANG W, JOTY S, et al. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation[J]. arXiv preprint arXiv:2109.00859, 2021.

[8] ROZIERE B, GEHRING J, GLOECKLE F, et al. Code llama: Open foundation models for code[J]. arXiv preprint arXiv:2308.12950, 2023.

[9] FENG Z, GUO D, TANG D, et al. Codebert: A pre-trained model for programming and natural languages[J]. arXiv preprint arXiv:2002.08155, 2020.

[10] LIU J, XIA C S, WANG Y, et al. Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation[C/OL]//Thirty-seventh Conference on Neural Information Processing Systems. 2023. https://openreview.net/forum?id=1qvx610Cu7.

[11] XIA C S, DENG Y, ZHANG L. Top Leaderboard Ranking = Top Coding Proficiency, Always? EvoEval: Evolving Coding Benchmarks via LLM[Z]. 2024. arXiv: 2403.19114 [cs.SE].

[12] LI R, ALLAL L B, ZI Y, et al. Starcoder: may the source be with you![J]. arXiv preprint arXiv:2305.06161, 2023.

[13] LUO Z, XU C, ZHAO P, et al. Wizardcoder: Empowering code large language models with evol-instruct[J]. arXiv preprint arXiv:2306.08568, 2023.

[14] WEI Y, WANG Z, LIU J, et al. Magicoder: Source code is all you need[J]. arXiv preprint arXiv:2312.02120, 2023.

[15] YU Z, ZHANG X, SHANG N, et al. Wavecoder: Widespread and versatile enhanced instruction tuning with refined data generation[J]. arXiv preprint arXiv:2312.14187, 2023.

[16] ZHENG T, ZHANG G, SHEN T, et al. OpenCodeInterpreter: Integrating Code Generation with Execution and Refinement[J]. arXiv preprint arXiv:2402.14658, 2024.

[17] LE-CONG T, NGUYEN D, LE B, et al. Evaluating Program Repair with Semantic-Preserving Transformations: A Naturalness Assessment[J]. arXiv preprint arXiv:2402.11892, 2024.

[18] 李卿源, 钟文康, 李传艺, 等. 神经程序修复领域数据泄露问题的实证研究[J]. 软件学报, 2024,

35(7): 0–0.

[19] JERDHAF O, SANTINI M, LUNDBERG P, et al. Evaluating Pre-Trained Language Models for Focused Terminology Extraction from Swedish Medical Records[C]//Proceedings of the Workshop on Terminology in the 21st century: many faces, many places. 2022: 30-32.

[20] MOU L, LI G, ZHANG L, et al. Convolutional Neural Networks over Tree Structures for Programming Language Processing.[C]//AAAI. 2016: 1287-1293.

[21] CHEN X, LIU C, SONG D. Tree-to-tree neural networks for program translation[J]. Advances in neural information processing systems, 2018, 31.

[22] JIANG X, ZHENG Z, LYU C, et al. TreeBERT: A tree-based pre-trained model for programming language[C/OL]//de CAMPOS C, MAATHUIS M H. Proceedings of Machine Learning Research: Proceedings of the Thirty-Seventh Conference on Uncertainty in Artificial Intelligence: vol. 161. PMLR, 2021: 54-63. https://proceedings.mlr.press/v161/jiang21a.html.

[23] RABINOVICH M, STERN M, KLEIN D. Abstract Syntax Networks for Code Generation and Semantic Parsing[C]//ACL. 2017: 1139-1149.

[24] DONG L, LAPATA M. Language to Logical Form with Neural Attention[C]//ACL. 2016: 33-43.

[25] DONG L, LAPATA M. Coarse-to-fine decoding for neural semantic parsing[J]. arXiv preprint arXiv:1805.04793, 2018.

[26] SUN Z, ZHU Q, MOU L, et al. A grammar-based structural cnn decoder for code generation[C]//Proceedings of the AAAI conference on artificial intelligence: vol. 33: 01. 2019: 7055-7062.

[27] SUN Z, ZHU Q, XIONG Y, et al. Treegen: A tree-based transformer architecture for code generation [C]//Proceedings of the AAAI Conference on Artificial Intelligence: vol. 34: 05. 2020: 8984-8991.

[28] MUKHERJEE R, WEN Y, CHAUDHARI D, et al. Neural program generation modulo static analysis[J]. Advances in Neural Information Processing Systems, 2021, 34: 18984-18996.

[29] AGRAWAL L A, KANADE A, GOYAL N, et al. Monitor-Guided Decoding of Code LMs with Static Analysis of Repository Context[J]. Advances in Neural Information Processing Systems, 2024, 36.

[30] JAIN A, ADIOLE C, REPS T, et al. Coarse-Tuning Models of Code with Reinforcement Learning Feedback[J]., 2023.

[31] SHOJAEE P, JAIN A, TIPIRNENI S, et al. Execution-based Code Generation using Deep Reinforcement Learning[Z]. 2023. arXiv: 2301.13816 [cs.LG].

[32] DOU S, LIU Y, JIA H, et al. StepCoder: Improve Code Generation with Reinforcement Learning from Compiler Feedback[Z]. 2024. arXiv: 2402.01391 [cs.SE].

[33] YE H, MARTINEZ M, MONPERRUS M. Neural program repair with execution-based backpropagation[C/OL]//ICSE '22: Proceedings of the 44th International Conference on Software Engineering. ACM, 2022. http://dx.doi.org/10.1145/3510003.3510222. DOI: 10.1145/3510003.3510222.

[34] SOHN J, YOO S. Fluccs: Using code and change metrics to improve fault localization[C]//Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis. 2017: 273-283.

[35] LI Z, ZOU D, XU S, et al. Vuldeepecker: A deep learning-based system for vulnerability detection [J]. arXiv preprint arXiv:1801.01681, 2018.

[36] ALON U, ZILBERSTEIN M, LEVY O, et al. Code2Vec: Learning Distributed Representations of Code[J/OL]. POPL, 2019. http://doi.acm.org/10.1145/3290353. DOI: 10.1145/3290353.

[37] WEI H, LI M. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code.[C]//IJCAI. 2017: 3034-3040.

[38] ALLAMANIS M, BROCKSCHMIDT M, KHADEMI M. Learning to represent programs with graphs[J]. arXiv, 2017.

[39] SUN Z, ZHU Q, XIONG Y, et al. Treegen: A tree-based transformer architecture for code generation [C]//AAAI: vol. 34: 05. 2020: 8984-8991.

[40] HELLENDOORN V J, SUTTON C, SINGH R, et al. Global Relational Models of Source Code [C/OL]//ICLR. 2020. https://openreview.net/forum?id=B1lnbRNtwr.

[41] KIPF T N, WELLING M. Semi-Supervised Classification with Graph Convolutional Networks [C/OL]//ICLR. 2017. https://openreview.net/forum?id=SJU4ayYgl.

[42] LING W, BLUNSOM P, GREFENSTETTE E, et al. Latent Predictor Networks for Code Generation [C]//ACL. 2016: 599-609.

[43] YIN P, NEUBIG G. A Syntactic Neural Model for General-Purpose Code Generation[C]//ACL. 2017: 440-450.

[44] IYER S, KONSTAS I, CHEUNG A, et al. Mapping Language to Code in Programmatic Context [C/OL]//EMNLP. Brussels, Belgium, 2018: 1643-1652. https://aclanthology.org/D18-1192. DOI: 10.18653/v1/D18-1192.

[45] YE X, CHEN Q, DILLIG I, et al. Benchmarking Multimodal Regex Synthesis with Complex Structures[C]//ACL. 2020: 6081-6094.

[46] KWIATKOWSKI T, CHOI E, ARTZI Y, et al. Scaling semantic parsers with on-the-fly ontology matching[C]//EMNLP. 2013: 1545-1556.

[47] WANG A, KWIATKOWSKI T, ZETTLEMOYER L. Morpho-syntactic lexical generalization for CCG semantic parsing[C]//EMNLP. 2014: 1284-1295.

[48] YIN P, NEUBIG G. TRANX: A Transition-based Neural Abstract Syntax Parser for Semantic Parsing and Code Generation[C]//. 2018.

[49] SHAW P, MASSEY P, CHEN A, et al. Generating Logical Forms from Graph Representations of Text and Entities[C/OL]//ACL. 2019: 95-106. https://www.aclweb.org/anthology/P19-1010. DOI: 10.18653/v1/P19-1010.

[50] PARK J U, KO S K, COGNETTA M, et al. Softregex: Generating regex from natural language descriptions using softened regex equivalence[C]//EMNLP-IJCNLP. 2019: 6425-6431.

[51] RADFORD A, WU J, CHILD R, et al. Language models are unsupervised multitask learners[J]. OpenAI blog, 2019, 1(8): 9.

[52] LU S, GUO D, et al. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation[C/OL]//NeurIPS. 2021. https://datasets-benchmarks-proceedings.neurips.cc

/paper/2021/hash/c16a5320fa475530d9583c34fd356ef5-Abstract-round1.html.

[53] ALON U, BRODY S, LEVY O, et al. code2seq: Generating Sequences from Structured Representations of Code[C]//ICLR. 2018.

[54] VASWANI A, SHAZEER N, PARMAR N, et al. Attention is all you need[J]. Advances in neural information processing systems, 2017, 30.

[55] LI Y, WANG S, NGUYEN T N. DLFix: Context-Based Code Transformation Learning for Automated Program Repair[C/OL]//ICSE '20: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. Seoul, South Korea: Association for Computing Machinery, 2020: 602-614. https://doi.org/10.1145/3377811.3380345. DOI: 10.1145/3377811.3380345.

[56] ZHU Q, SUN Z, XIAO Y A, et al. A Syntax-Guided Edit Decoder for Neural Program Repair [C/OL]//ESEC/FSE 2021: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. Athens, Greece, 2021: 341-353. https://doi.org/10.1145/3468264.3468544. DOI: 10.1145/3468264.3468544.

[57] DEPT C M U C S. Speech Understanding Systems. Summary of Results of the Five-Year Research Effort at Carnegie-Mellon University[J]. cmu, 1977.

[58] XIONG Y, WANG B. L2S: A Framework for Synthesizing the Most Probable Program under a Specification[J/OL]. ACM Trans. Softw. Eng. Methodol., 2022, 31(3). https://doi.org/10.1145/3487570. DOI: 10.1145/3487570.

[59] LIU H, SHEN M, ZHU J, et al. Deep Learning Based Program Generation From Requirements Text: Are We There Yet?[J/OL]. IEEE Trans. Software Eng., 2022, 48(4): 1268-1289. https://doi.org/10.1109/TSE.2020.3018481. DOI: 10.1109/TSE.2020.3018481.

[60] WANG B, SHIN R, LIU X, et al. RAT-SQL: Relation-Aware Schema Encoding and Linking for Text-to-SQL Parsers[C/OL]//ACL 2020. 2020. https://www.microsoft.com/en-us/research/publication/rat-sql-relation-aware-schema-encoding-and-linking-for-text-to-sql-parsers/.

[61] SUN Z, ZHU Q, XIONG Y, et al. TreeGen: A Tree-Based Transformer Architecture for Code Generation[C/OL]//The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020. AAAI Press, 2020: 8984-8991. https://aaai.org/ojs/index.php/AAAI/article/view/6430.

[62] TUFANO M, WATSON C, BAVOTA G, et al. An Empirical Investigation into Learning Bug-Fixing Patches in the Wild via Neural Machine Translation[C]//2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE). 2018: 832-837. DOI: 10.1145/3238147.3240732.

[63] CHAKRABORTY S, ALLAMANIS M, RAY B. CODIT: Code Editing with Tree-Based Neural Machine Translation[J]. arXiv preprint arXiv:1810.00314, 2018.

[64] LUTELLIER T, PHAM H V, PANG L, et al. CoCoNuT: combining context-aware neural translation models using ensemble for program repair[C]//ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. 2020.

[65] BEN-NUN T, JAKOBOVITS A S, HOEFLER T. Neural Code Comprehension: A Learnable Representation of Code Semantics[C]//NIPS'18: Proceedings of the 32nd International Conference on Neural Information Processing Systems. Montréal, Canada: Curran Associates Inc., 2018: 3589-3601.

[66] ALLAMANIS M, BROCKSCHMIDT M, KHADEMI M. Learning to Represent Programs with Graphs[J/OL]. CoRR, 2017, abs/1711.00740. arXiv: 1711.00740. http://arxiv.org/abs/1711.00740.

[67] LOU Y, ZHU Q, DONG J, et al. Boosting Coverage-Based Fault Localization via Graph-Based Representation Learning[C/OL]//ESEC/FSE 2021. Athens, Greece: Association for Computing Machinery, 2021: 664-676. https://doi.org/10.1145/3468264.3468580. DOI: 10.1145/3468264.3468580.

[68] TANG Z, SHEN X, LI C, et al. AST-Trans: Code Summarization with Efficient Tree-Structured Attention[C]//2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE). 2022: 150-162. DOI: 10.1145/3510003.3510224.

[69] VASWANI A, SHAZEER N, PARMAR N, et al. Attention is All You Need[C]//NIPS'17: Proceedings of the 31st International Conference on Neural Information Processing Systems. Long Beach, California, USA: Curran Associates Inc., 2017: 6000-6010.

[70] ZHU Q, SUN Z, LIANG X, et al. OCoR: An Overlapping-Aware Code Retriever[C/OL]//35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020. IEEE, 2020: 883-894. https://doi.org/10.1145/3324884.3416530. DOI: 10.1145/3324884.3416530.

[71] JUST R, JALALI D, ERNST M D. Defects4J: A database of existing faults to enable controlled testing studies for Java programs[C]//Proceedings of the International Symposium on Software Testing and Analysis (ISSTA). San Jose, CA, USA, 2014: 437-440.

[72] LIN D, KOPPEL J, CHEN A, et al. QuixBugs: a multi-lingual program repair benchmark set based on the quixey challenge[C]//. 2017: 55-56. DOI: 10.1145/3135932.3135941.

[73] LIU K, KOYUNCU A, KIM D, et al. Tbar: Revisiting template-based automated program repair [C]//Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. 2019: 31-42.

[74] JIANG J, XIONG Y, ZHANG H, et al. Shaping Program Repair Space with Existing Patches and Similar Code[C]//International Symposium on Software Testing & Analysis. 2018: 298-309.

[75] LE GOUES C, NGUYEN T, FORREST S, et al. GenProg: A Generic Method for Automatic Software Repair[J]. IEEE Transactions on Software Engineering, 2012, 38(1): 54-72. DOI: 10.1109/TSE.2011.104.

[76] RIBOIRA A, ABREU R. The GZoltar Project: A Graphical Debugger Interface[C]//. 2010: 215-218. DOI: 10.1007/978-3-642-15585-7_25.

[77] WEN M, CHEN J, WU R, et al. Context-Aware Patch Generation for Better Automated Program Repair[C]//2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE). 2018: 1-11. DOI: 10.1145/3180155.3180233.

[78] JIANG N, LUTELLIER T, TAN L. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair[J/OL]. 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), 2021. http://dx.doi.org/10.1109/ICSE43902.2021.00107. DOI: 10.1109/icse43902.2021.00107.

[79] YE H, MARTINEZ M, MONPERRUS M. Neural Program Repair with Execution-Based Backpropagation[C/OL]//ICSE '22: Proceedings of the 44th International Conference on Software Engineering. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022: 1506-1518. https://doi.org/10.1145/3510003.3510222. DOI: 10.1145/3510003.3510222.

[80] LIU X, ZENG M, XIONG Y, et al. Identifying Patch Correctness in Test-Based Automatic Program Repair[J]., 2017.

[81] WANG S, WEN M, LIN B, et al. Automated Patch Correctness Assessment: How Far are We?[C]// 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE). 2020: 968-980.

[82] YE H, MARTINEZ M, MONPERRUS M. Automated patch assessment for program repair at scale [J/OL]. Empirical Software Engineering, 2021, 26(2). https://doi.org/10.1007%5C%2Fs10664-020-09920-w. DOI: 10.1007/s10664-020-09920-w.

[83] DURIEUX T, MADEIRAL F, MARTINEZ M, et al. Empirical Review of Java Program Repair Tools: A Large-Scale Experiment on 2,141 Bugs and 23,551 Repair Attempts[C/OL]//Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19). 2019. https://arxiv.org/abs/1905.11973.

[84] GUO D, ZHU Q, YANG D, et al. DeepSeek-Coder: When the Large Language Model Meets Programming–The Rise of Code Intelligence[J]. arXiv preprint arXiv:2401.14196, 2024.

[85] MA Y, GOU Z, HAO J, et al. SciAgent: Tool-augmented Language Models for Scientific Reasoning [J]. arXiv preprint arXiv:2402.11451, 2024.

[86] NIJKAMP E, PANG B, HAYASHI H, et al. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis[C]//The Eleventh International Conference on Learning Representations. 2022.

[87] ZHU Q, SUN Z, ZHANG W, et al. Tare: Type-aware neural program repair[C]//2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). 2023: 1443-1455.

[88] ZHU Q, SUN Z, XIAO Y A, et al. A syntax-guided edit decoder for neural program repair[C]// Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2021: 341-353.

[89] ZHANG F, CHEN B, ZHANG Y, et al. Repocoder: Repository-level code completion through iterative retrieval and generation[J]. arXiv preprint arXiv:2303.12570, 2023.

[90] ZHU Q, SUN Z, ZHANG W, et al. Grape: Grammar-Preserving Rule Embedding.[C]//IJCAI. 2022: 4545-4551.

[91] GUO D, REN S, LU S, et al. Graphcodebert: Pre-training code representations with data flow[J]. arXiv preprint arXiv:2009.08366, 2020.

[92] LEE K, IPPOLITO D, NYSTROM A, et al. Deduplicating Training Data Makes Language Models

Better[C]//Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). 2022: 8424-8445.

[93] KOCETKOV D, LI R, JIA L, et al. The Stack: 3 TB of permissively licensed source code[J]. Transactions on Machine Learning Research, 2022.

[94] BAVARIAN M, JUN H, TEZAK N, et al. Efficient training of language models to fill in the middle [J]. arXiv preprint arXiv:2207.14255, 2022.

[95] FRIED D, AGHAJANYAN A, LIN J, et al. InCoder: A Generative Model for Code Infilling and Synthesis[C]//The Eleventh International Conference on Learning Representations. 2022.

[96] RAFFEL C, SHAZEER N, ROBERTS A, et al. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer[Z]. 2023. arXiv: 1910.10683 [cs.LG].

[97] NIJKAMP E, HAYASHI H, XIONG C, et al. CodeGen2: Lessons for Training LLMs on Programming and Natural Languages[Z]. 2023. arXiv: 2305.02309 [cs.LG].

[98] DeepSeek-AI. DeepSeek LLM: Scaling Open-Source Language Models with Longtermism[J]. arXiv preprint arXiv:2401.02954, 2024.

[99] SU J, LU Y, PAN S, et al. RoFormer: Enhanced Transformer with Rotary Position Embedding[Z]. 2023. arXiv: 2104.09864 [cs.CL].

[100] DAO T. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning[Z]. 2023. arXiv: 2307.08691 [cs.LG].

[101] CHEN S, WONG S, CHEN L, et al. Extending context window of large language models via positional interpolation[J]. arXiv preprint arXiv:2306.15595, 2023.

[102] Kaiokendev. Things I'm Learning While Training SuperHOT[Z]. https://kaiokendev.github.io/til#extending-context-to-8k. 2023.

[103] ALLAL L B, LI R, KOCETKOV D, et al. SantaCoder: don't reach for the stars![J]. arXiv preprint arXiv:2301.03988, 2023.

[104] DING Y, WANG Z, AHMAD W U, et al. CrossCodeEval: A Diverse and Multilingual Benchmark for Cross-File Code Completion[C]//Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track. 2023.

[105] DEVLIN J, CHANG M W, LEE K, et al. Bert: Pre-training of deep bidirectional transformers for language understanding[J]. arXiv preprint arXiv:1810.04805, 2018.

[106] LIU Y, OTT M, GOYAL N, et al. Roberta: A robustly optimized bert pretraining approach[J]. arXiv preprint arXiv:1907.11692, 2019.

[107] RAFFEL C, SHAZEER N, ROBERTS A, et al. Exploring the limits of transfer learning with a unified text-to-text transformer[J]. The Journal of Machine Learning Research, 2020, 21(1): 5485-5551.

[108] WANG X, WANG Y, MI F, et al. Syncobert: Syntax-guided multi-modal contrastive pre-training for code representation[J]. arXiv preprint arXiv:2108.04556, 2021.

[109] SVYATKOVSKIY A, DENG S K, FU S, et al. Intellicode compose: Code generation using transformer[C]//Proceedings of the 28th ACM Joint Meeting on European Software Engineering Con-

ference and Symposium on the Foundations of Software Engineering. 2020: 1433-1443.

[110] LU S, GUO D, REN S, et al. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation[J]. CoRR, 2021, abs/2102.04664.

[111] GUO D, LU S, DUAN N, et al. Unixcoder: Unified cross-modal pre-training for code representation [J]. arXiv preprint arXiv:2203.03850, 2022.

[112] SENNRICH R, HADDOW B, BIRCH A. Neural Machine Translation of Rare Words with Subword Units[C/OL]//Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). Berlin, Germany: Association for Computational Linguistics, 2016: 1715-1725. https://aclanthology.org/P16-1162. DOI: 10.18653/v1/P16-1162.

[113] XIONG Y, WANG B. L2S: A framework for synthesizing the most probable program under a specification[J]. ACM Transactions on Software Engineering and Methodology (TOSEM), 2022, 31(3): 1-45.

[114] KARAMPATSIS R M, BABII H, ROBBES R, et al. Big code!= big vocabulary: Open-vocabulary models for source code[C]//Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. 2020: 1073-1085.

[115] PROVILKOV I, EMELIANENKO D, VOITA E. BPE-dropout: Simple and effective subword regularization[J]. arXiv preprint arXiv:1910.13267, 2019.

[116] Tree-Sitter. Tree-Sitter[EB/OL]. 2023. https://tree-sitter.github.io/tree-sitter.

[117] HUSAIN H, WU H H, GAZIT T, et al. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search[Z]. 2020. arXiv: 1909.09436 [cs.LG].

[118] Codeparrot. GitHub Code Dataset[EB/OL]. 2023. https://huggingface.co/datasets/codeparrot/github-code.

[119] AUSTIN J, ODENA A, NYE M, et al. Program synthesis with large language models[J]. arXiv preprint arXiv:2108.07732, 2021.

[120] ODA Y, FUDABA H, NEUBIG G, et al. Learning to generate pseudo-code from source code using statistical machine translation[C]//2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). 2015: 574-584.

[121] YIN P, DENG B, CHEN E, et al. Learning to mine aligned code and natural language pairs from stack overflow[C]//Proceedings of the 15th International Conference on Mining Software Repositories. 2018: 476-486.

[122] LIN C Y, OCH F J. Orange: a method for evaluating automatic evaluation metrics for machine translation[C]//COLING 2004: Proceedings of the 20th International Conference on Computational Linguistics. 2004: 501-507.

[123] HUANG J, TANG D, SHOU L, et al. Cosqa: 20,000+ web queries for code search and question answering[J]. arXiv preprint arXiv:2105.13239, 2021.

[124] IYER S, KONSTAS I, CHEUNG A, et al. Mapping language to code in programmatic context[J]. arXiv preprint arXiv:1808.09588, 2018.

[125] REN S, GUO D, LU S, et al. Codebleu: a method for automatic evaluation of code synthesis[J]. arXiv preprint arXiv:2009.10297, 2020.

[126] HENDRYCKS D, BASART S, KADAVATH S, et al. Measuring coding challenge competence with apps[J]. arXiv preprint arXiv:2105.09938, 2021.

[127] TUFANO M, WATSON C, BAVOTA G, et al. An empirical study on learning bug-fixing patches in the wild via neural machine translation[J]. ACM Transactions on Software Engineering and Methodology (TOSEM), 2019, 28(4): 1-29.

[128] AHMAD W U, CHAKRABORTY S, RAY B, et al. Unified pre-training for program understanding and generation[J]. arXiv preprint arXiv:2103.06333, 2021.

[129] LEWIS M, LIU Y, GOYAL N, et al. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension[J]. arXiv preprint arXiv:1910.13461, 2019.

[130] WANG Y, LE H, GOTMARE A D, et al. Codet5+: Open code large language models for code understanding and generation[J]. arXiv preprint arXiv:2305.07922, 2023.

[131] LE H, WANG Y, GOTMARE A D, et al. Coderl: Mastering code generation through pretrained models and deep reinforcement learning[J]. Advances in Neural Information Processing Systems, 2022, 35: 21314-21328.

[132] HU B, LU Z, LI H, et al. Convolutional Neural Network Architectures for Matching Natural Language Sentences[Z]. 2015. arXiv: 1503.03244 [cs.CL].

[133] GU X, ZHANG H, KIM S. Deep code search[C]//Proceedings of the 40th International Conference on Software Engineering. 2018: 933-944.

[134] JIAN F, QIU X, HUANG X. Convolutional Deep Neural Networks for Document-Based Question Answering[M]. 2016.

[135] HUA H, LIN J. Pairwise Word Interaction Modeling with Deep Neural Networks for Semantic Similarity Measurement[C]//Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. 2016.

[136] WAN Y, ZHAO Z, YANG M, et al. Improving automatic source code summarization via deep reinforcement learning[J]., 2018.

[137] YAO Z, PEDDAMAIL J R, SUN H. CoaCor: Code Annotation for Code Retrieval with Reinforcement Learning[J/OL]. The World Wide Web Conference on - WWW '19, 2019. http://dx.doi.org/10.1145/3308558.3313632. DOI: 10.1145/3308558.3313632.

[138] IYER S, KONSTAS I, CHEUNG A, et al. Summarizing source code using a neural attention model [C]//Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). 2016: 2073-2083.

[139] LOPER E, BIRD S. NLTK: The Natural Language Toolkit[J]., 2002.

[140] HE K, ZHANG X, REN S, et al. Deep Residual Learning for Image Recognition[Z]. 2015. arXiv: 1512.03385 [cs.CV].

[141] BA J L, KIROS J R, HINTON G E. Layer Normalization[J]., 2016.

[142] KIM Y. Convolutional Neural Networks for Sentence Classification[Z]. 2014. arXiv: 1408.5882 [cs.CL].

[143] GIUSTI A, CIREŞAN D C, MASCI J, et al. Fast image scanning with deep max-pooling con-

volutional neural networks[C]//2013 IEEE International Conference on Image Processing. 2013: 4034-4038.

[144] ZHOU P, QI Z, ZHENG S, et al. Text classification improved by integrating bidirectional LSTM with two-dimensional max pooling[J]. arXiv preprint arXiv:1611.06639, 2016.

[145] XU M, WONG D F, YANG B, et al. Leveraging Local and Global Patterns for Self-Attention Networks[C]//ACL. 2019.

[146] MIKOLOV T, CHEN K, CORRADO G, et al. Efficient estimation of word representations in vector space[J]. arXiv preprint arXiv:1301.3781, 2013.

[147] SUN Z, ZHU Q, XIONG Y, et al. TreeGen: A Tree-Based Transformer Architecture for Code Generation[J/OL]. CoRR, 2019, abs/1911.09983. arXiv: 1911.09983. http://arxiv.org/abs/1911.09983.

[148] HENDRYCKS D, GIMPEL K. Bridging Nonlinearities and Stochastic Regularizers with Gaussian Error Linear Units[J]. arXiv, 2016.

[149] YAO Z, WELD D S, CHEN W P, et al. StaQC[J/OL]. Proceedings of the 2018 World Wide Web Conference on World Wide Web - WWW '18, 2018. http://dx.doi.org/10.1145/3178876.3186081. DOI: 10.1145/3178876.3186081.

[150] OVERFLOW S. https://stackoverflow.com/[C]//. 2020.

[151] CRASWELL N. Mean reciprocal rank[J]. Encyclopedia of Database Systems, 2009: 1703-1703.

[152] ABADI M, AGARWAL A, BARHAM P, et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems[J]., 2016.

[153] HINTON G E, SRIVASTAVA N, KRIZHEVSKY A, et al. Improving neural networks by preventing co-adaptation of feature detectors[Z]. 2012. arXiv: 1207.0580 [cs.NE].

[154] KINGMA D P, BA J. Adam: A Method for Stochastic Optimization[J]. arXiv, 2014. arXiv: 1412.6980 [cs.LG].

[155] HOCHREITER S, SCHMIDHUBER J. Long short-term memory[J]. Neural computation, 1997, 9(8): 1735-1780.

[156] PENNINGTON J, SOCHER R, MANNING C D. Glove: Global vectors for word representation [C]//Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP). 2014: 1532-1543.

[157] SENNRICH R, HADDOW B, BIRCH A. Neural Machine Translation of Rare Words with Subword Units[Z]. 2015. arXiv: 1508.07909 [cs.CL].

[158] QI Z, LONG F, ACHOUR S, et al. An Analysis of Patch Plausibility and Correctness for Generate-and-validate Patch Generation Systems[C]//ISSTA. 2015: 24-36.

[159] KIM D, NAM J, SONG J, et al. Automatic patch generation learned from human-written patches [C]//2013 35th International Conference on Software Engineering (ICSE). 2013: 802-811. DOI: 10.1109/ICSE.2013.6606626.

[160] JIANG J, REN L, XIONG Y, et al. Inferring program transformations from singular examples via big code[C]//2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). 2019: 255-266.

[161] LONG F, AMIDON P, RINARD M. Automatic inference of code transforms for patch generation [C]//Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. 2017: 727-739.

[162] BADER J, SCOTT A, PRADEL M, et al. Getafix: Learning to fix bugs automatically[J]. Proceedings of the ACM on Programming Languages, 2019, 3(OOPSLA): 1-27.

[163] ROLIM R, SOARES G, D'ANTONI L, et al. Learning syntactic program transformations from examples[C/OL]//UCHITEL S, ORSO A, ROBILLARD M P. Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017. IEEE / ACM, 2017: 404-415. https://doi.org/10.1109/ICSE.2017.44. DOI: 10.1109/ICSE.2017.44.

[164] XIONG Y, WANG J, YAN R, et al. Precise condition synthesis for program repair[C]//2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). 2017: 416-426.

[165] LONG F, RINARD M. Automatic patch generation by learning correct code[C]//Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. 2016: 298-312.

[166] SAHA R K, LYU Y, YOSHIDA H, et al. ELIXIR: Effective Object Oriented Program Repair[C/OL]//ASE. Urbana-Champaign, IL, USA: IEEE Press, 2017. http://dl.acm.org/citation.cfm?id=3155562.3155643.

[167] XIONG Y, WANG B, FU G, et al. Learning to synthesize[C]//Proceedings of the 4th International Workshop on Genetic Improvement Workshop. 2018: 37-44.

[168] CHEN Z, KOMMRUSCH S J, TUFANO M, et al. SEQUENCER: Sequence-to-Sequence Learning for End-to-End Program Repair[J]. IEEE Transactions on Software Engineering, 2019: 1-1. DOI: 10.1109/TSE.2019.2940179.

[169] HATA H, SHIHAB E, NEUBIG G. Learning to Generate Corrective Patches using Neural Machine Translation[J/OL]. CoRR, 2018, abs/1812.07170. arXiv: 1812.07170. http://arxiv.org/abs/1812.07170.

[170] BAHDANAU D, CHO K, BENGIO Y. Neural Machine Translation by Jointly Learning to Align and Translate[J]. CoRR, 2015, abs/1409.0473.

[171] YIN P, NEUBIG G. A Syntactic Neural Model for General-Purpose Code Generation[C/OL]//BARZILAY R, KAN M. Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers. Association for Computational Linguistics, 2017: 440-450. https://doi.org/10.18653/v1/P17-1041. DOI: 10.18653/v1/P17-1041.

[172] RABINOVICH M, STERN M, KLEIN D. Abstract Syntax Networks for Code Generation and Semantic Parsing[C/OL]//BARZILAY R, KAN M. Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics, ACL 2017, Vancouver, Canada, July 30 - August 4, Volume 1: Long Papers. Association for Computational Linguistics, 2017: 1139-1149. https://doi.org/10.18653/v1/P17-1105. DOI: 10.18653/v1/P17-1105.

[173] SUN Z, ZHU Q, MOU L, et al. A Grammar-Based Structural CNN Decoder for Code Generation [C/OL]//The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-

First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019. AAAI Press, 2019: 7055-7062. https://doi.org/10.1609/aaai.v33i01.33017055. DOI: 10.1609/aaai.v33i01.33017055.

[174] MIKOLOV T, CHEN K, CORRADO G S, et al. Efficient Estimation of Word Representations in Vector Space[C]//ICLR. 2013.

[175] SCARSELLI F, GORI M, TSOI A C, et al. The Graph Neural Network Model[J]. TNN, 2009, 20(1): 61-80.

[176] ZHANG W, SUN Z, ZHU Q, et al. NLocalSAT: Boosting Local Search with Solution Prediction [J/OL]. Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, 2020. http://dx.doi.org/10.24963/ijcai.2020/164. DOI: 10.24963/ijcai.2020/164.

[177] HENDRYCKS D, GIMPEL K. Bridging Nonlinearities and Stochastic Regularizers with Gaussian Error Linear Units[J]., 2016.

[178] VINYALS O, FORTUNATO M, JAITLY N. Pointer Networks[J]. ArXiv, 2015, abs/1506.03134.

[179] Github. https://github.com/[C]//. 2020.

[180] DURIEUX T, MONPERRUS M. IntroClassJava: A Benchmark of 297 Small and Buggy Java Programs[R/OL]. Universite Lille 1. 2016. https://hal.archives-ouvertes.fr/hal-01272126/document.

[181] LE GOUES C, HOLTSCHULTE N, SMITH E K, et al. The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs[J]. IEEE Transactions on Software Engineering (TSE), 2015, 41(12): 1236-1256. DOI: 10.1109/TSE.2015.2454513.

[182] ABREU R, ZOETEWEIJ P, GEMUND A J C V. On the Accuracy of Spectrum-based Fault Localization[C]//Testing: Academic & Industrial Conference Practice & Research Techniques-mutation. 2007.

[183] LE X B D, LO D, LE GOUES C. History driven program repair[C]//2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER): vol. 1. 2016: 213-224.

[184] XUAN J, MARTINEZ M, DEMARCO F, et al. Nopol: Automatic repair of conditional statement bugs in java programs[J]. IEEE Transactions on Software Engineering, 2016, 43(1): 34-55.

[185] HUA J, ZHANG M, WANG K, et al. Sketchfix: A tool for automated program repair approach using lazy candidate generation[C]//Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. 2018: 888-891.

[186] KOYUNCU A, LIU K, BISSYANDÉ T F, et al. Fixminer: Mining relevant fix patterns for automated program repair[J]. Empirical Software Engineering, 2020: 1-45.

[187] GHANBARI A, ZHANG L. PraPR: Practical Program Repair via Bytecode Mutation[C]//2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). 2019: 1118-1121. DOI: 10.1109/ASE.2019.00116.

[188] LIU K, KOYUNCU A, KIM D, et al. Avatar: Fixing semantic bugs with fix patterns of static analysis violations[C]//2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER). 2019: 1-12.

[189] LIU K, WANG S, KOYUNCU A, et al. On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs[C]//Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. 2020: 615-627.

[190] Facebook. Pytorch[EB/OL]. 2023. https://pytorch.org.

[191] HINTON G E, SRIVASTAVA N, KRIZHEVSKY A, et al. Improving neural networks by preventing co-adaptation of feature detectors[J/OL]. CoRR, 2012, abs/1207.0580. arXiv: 1207.0580. http://arxiv.org/abs/1207.0580.

[192] KINGMA D P, BA J. Adam: A Method for Stochastic Optimization[J]. CoRR, 2015, abs/1412.6980.

[193] SAHA S, et al. Harnessing evolution for multi-hunk program repair[C]//2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). 2019: 13-24.

攻读博期间发表的论文及其他成果

[1] **Qihao Zhu**, Zeyu Sun, Xiran Liang, Yingfei Xiong, and Lu Zhang. "OCoR: an overlapping-aware code retriever." In Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (**ASE CCF-A**), pp. 883-894. 2020.

[2] **Qihao Zhu**, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. "A Syntax-Guided Edit Decoder for Neural Program Repair." In Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (**ESEC/FSE CCF-A**), pp. 341-353. 2021.

[3] **Qihao Zhu**, Zeyu Sun, Wenjie Zhang, Yingfei Xiong, and Lu Zhang. "Grape: Grammar-Preserving Rule Embedding." In Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence (**IJCAI CCF-A**). 2022

[4] **Qihao Zhu**, Zeyu Sun, Wenjie Zhang, Yingfei Xiong, and Lu Zhang. "Tare: Type-Aware Neural Program Repair." In Proceedings of the 45th International Conference on Software Engineering (**ICSE CCF-A**). 2023.

[5] **Qihao Zhu**, Qingyuan Liang, Zeyu Sun, Yingfei Xiong, Lu Zhang, Shengyu Cheng. "GrammarT5: Grammar-Integrated Pretrained Encoder-Decoder Neural Model for Code". In Proceedings of the 46th International Conference on Software Engineering (**ICSE CCF-A**). 2024.

[6] Jinhao Dong[#], **Qihao Zhu**[#], Zeyu Sun, Yiling Lou, Dan Hao. "Merge Conflict Resolution: Classification or Generation?" In 38th IEEE/ACM International Conference on Automated Software Engineering (**ASE CCF-A**). 2023.

[7] Daya Guo[#], **Qihao Zhu**[#], Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen et al. "DeepSeek-Coder: When the Large Language Model Meets Programming–The Rise of Code Intelligence." arXiv preprint arXiv:2401.14196 (2024).

[8] Zeyu Sun, **Qihao Zhu**, Lili Mou, Yingfei Xiong, Ge Li, and Lu Zhang, A Grammar-Based Structural CNN Decoder for Code Generation, The Thirty-Third AAAI Conference on Artificial Intelligence **(AAAI, CCF-A)**, 2019.

[9] Zeyu Sun, **Qihao Zhu**, Yingfei Xiong, Yican Sun, Lili Mou, and Lu Zhang, TreeGen: A Tree- Based Transformer Architecture for Code Generation, The Thirty-Fourth AAAI Conference on Artificial Intelligence **(AAAI, CCF-A)**, 2020.

标有"#"的作者对文章具有相同贡献

[10] Yiling Lou, **Qihao Zhu**, Jinhao Dong, Xia Li, Zeyu Sun, Dan Hao, Lu Zhang, Lingming Zhang, Boosting Coverage-Based Fault Localization via Graph-Based Representation Learning, The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering **(ESEC/FSE, CCF-A)**, 2021.

[11] Jinhao Dong, Yiling Lou, **Qihao Zhu**, Zeyu Sun, Zhilin Li, Wenjie Zhang, and Dan Hao. "FIRA: fine-grained graph-based code change representation for automated commit message generation." In Proceedings of the 44th International Conference on Software Engineering (**ICSE CCF-A**), pp. 970-981. 2022.

[12] Zhao Tian, Junjie Chen, **Qihao Zhu**, Junjie Yang, and Lingming Zhang. "Learning to Construct Better Mutation Faults." In 37th IEEE/ACM International Conference on Automated Software Engineering (**ASE CCF-A**), pp. 1-13. 2022.

[13] Wenjie Zhang, Zeyu Sun, **Qihao Zhu**, Ge Li, Shaowei Cai, Yingfei Xiong, and Lu Zhang, NLocalSAT: Boosting Local Search with Solution Prediction, the 29th International Joint Conference on Artificial Intelligence and the 17th Pacific Rim International Conference on Artificial Intelligence **(IJCAI-PRICAI, CCF-A)**, 2020.

[14] Qingyuan Liang, Zeyu Sun, **Qihao Zhu**, Wenjie Zhang, Lian Yu, Yingfei Xiong, and Lu Zhang. "Lyra: A Benchmark for Turducken-Style Code Generation." In Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence (**IJCAI CCF-A**). 2022

[15] Shao, Zhihong, Peiyi Wang, **Qihao Zhu**, Runxin Xu, Junxiao Song, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. "Deepseekmath: Pushing the limits of mathematical reasoning in open language models." arXiv preprint arXiv:2402.03300 (2024).

[16] Sun, Zeyu, Wenjie Zhang, Lili Mou, **Qihao Zhu**, Yingfei Xiong, and Lu Zhang. "Generalized Equivariance and Preferential Labeling for GNN Node Classification." In Proceedings of the AAAI Conference on Artificial Intelligence (**AAAI CCF-A**), vol. 36, no. 8, pp. 8395-8403. 2022.

[17] 梁清源，**朱琪豪**，孙泽宇，张路，张文杰，熊英飞，梁广泰. 基于深度学习的 SQL 生成研究综述. 中国科学 (**CCF-A 类中文期刊**), 52:1363-1392, 8 月 2022.

[18] Yakun Zhang, Wenjie Zhang, Dezhi Ran, **Qihao Zhu**, Chengfeng Dou, Dan Hao, Tao Xie, Lu Zhang. "Learning-based Widget Matching for Migrating GUI Test Cases." The Proceedings of the 46th IEEE/ACM International Conference on Software Engineering. 2024.

致谢

时光如梭，2015 年我怀揣着对燕园的期待和憧憬踏进了这所中国的最高学府进行学习，寒来暑往，春秋交错，9 个年头的燕园生活即将结束，我怀念未名湖边的缕缕春风，谨记理科一号楼里的谆谆教诲，回味农园食堂的美味佳肴，细数邱德拔体育馆前的怦然心动，一幕幕的回忆是我此生最珍贵的财富。细数自己的求学旅程，许许多多的人都为我做出了非常多的帮助，我也会在心里铭记。

感谢杨芙清院士和梅宏院士，他们在中国软件工程领域做出的巨大贡献令人钦佩，能够加入由两位院士领导的北京大学软件工程研究所完成我的学术探索和研究工作，我感到无比荣幸。正是因为团队整体雄厚的科研实力，我才能够和众多杰出的学术前辈和同行们进行交流与合作，共同自由地探索学术的最前沿问题。

十分感谢我的导师熊英飞副教授对我科研工作的教导和帮助。熊老师是一位不可多得的科研导师，他以循循善诱的方式帮助我从一个科研菜鸟慢慢对自己的科研方向和科研理想逐渐变得更加清晰，鼓励我向着困难的科研难题发起挑战，并在我陷入迷茫的时候给予强烈的支持和鼓励，教诲不能抱着唯论文论的想法进行科研工作，要在十年后仍然对自己当初研究的课题和难题感到自豪，而不是对自己的研究内容毫无印象。在这种中心思想支持的情况下，我选择了更加困难的程序生成问题，对我的人生发展之路产生了深厚的影响。希望未来熊老师能够和自己的学生一起，做出更多影响深远的科研工作。

感谢我们科研组的另外两位指导老师张路教授和郝丹教授。张路教授平易近人的性格和广博的学术知识时常对我起到了很大的引导作用，能够帮助我更好地润色自己的学术工作，对我的科研起到了很大的帮助。郝老师对待学术的严谨态度和为人处世的豁达态度也促进了我对于学术一丝不苟的追求和完善，感谢他们对于我在学术之路前进中的帮助。

感谢牟力立学长和孙泽宇学长在我还是个学术菜鸟时的帮助和引导，自己科研道路上的第一个工作是在两位学长的不懈努力，才能够最终完成和顺利发表，为我的学术之路打下了坚实的基础，也对神秘的神经网络产生了浓厚的兴趣，最终形成了自己的博士研究工作。

感谢我所在小组里的邹达明、娄一翎、周建祎、梁晶晶、孙泽宇、王冠成、郭翊庆、李丰、张文杰、曾慕寒、张雅坤、吴宜谦、梁清源、肖元安同学。感谢其他小组的悦茹茹、王博、姜佳君、梁晶晶、张宇霞、武健宇、杨恺、曹英魁、沈琦、王敏、陈蔚燕、刘兆鹏、包鹏等同学。谢谢各位同学在科研和生活上给予我的帮助。

感谢我的爸爸妈妈哥哥嫂子，对我生活和学业上的无私奉献，没有他们在背后的默默付出和鼎力支持，我无法完成我的科研之路。特别感谢我的女朋友龚晨，在我对人生方向做出选择的时候给予的宝贵意见和建议，也给予了我无微不至的暖心关爱和帮助，完善了我学术生活的最精彩的一部分，感谢相伴，一路携手，白头到老。

青山不改，绿水长流，我会朝着自己的科研目标继续前进，不懈奋斗！

学位论文答辩委员会名单

论文题目	语言定义感知的深度代码学习技术及应用			
作　者	朱琪豪			
专　业	计算机软件与理论			
答辩委员会成员	**姓　名**	**专业技术职称**	**从事专业**	**工作单位**
主席	魏峻	研究员	计算机软件与理论	中国科学院软件研究所
委员	李戈	教授	计算机科学与技术	北京大学
	刘辉	教授	计算机软件与理论	北京理工大学
	王迪	助理教授	计算机科学与技术	北京大学
	张路	教授	计算机科学与技术	北京大学

提交终版学位论文承诺书

本人郑重承诺，所呈交的学位论文为最终版学位论文。本人知晓，该版学位论文将用于校学位评定委员会审议学位、国家和北京市学位论文抽检。论文版本呈交错误带来的结果将由本人承担。

论文作者签名： 

日期：2024 年　5　月　30　日