学校代码：10246
学　　号：20210860082

復旦大學

硕 士 学 位 论 文

（专业学位）

基于抽象语法树的代码翻译系统的设计与实现

The Design and Implementation of Code Translation System Based on Abstract Syntax Tree

院　　系：工程与应用技术研究院
专业学位类别（领域）：　　计算机技术
姓　　名：　　杨小伟
指 导 教 师：　　殳峰　研究员
完 成 日 期：　2022 年 11 月 20 日

指导小组成员

杨晓峰　教　授
殳　峰　研究员
张志平　研究员
丁晨阳　研究员

目　录

插图目录 iii

摘　要 v

Abstract vii

第 1 章 引言 1

1.1 研究背景及意义 1

1.1.1 Matlab、C++简介 1

1.1.2 ACS SPiiPlus 系列运动控制器简介 1

1.1.3 工程实践中存在的问题 1

1.2 源代码到源代码翻译技术以及国内外研究现状 2

1.2.1 源代码到源代码翻译技术 2

1.2.2 国内外研究现状 2

1.2.3 现有系统存在的问题 7

1.3 本文主要工作 7

1.4 本文的组织结构 8

第 2 章 技术概述 9

2.1 抽象语法树 9

2.2 访问者模式 9

2.3 CMake 11

2.4 Qt 框架 12

2.5 LLVM 和 clang-format 12

2.6 本章小结 13

第 3 章 需求分析与概要设计 15

3.1 系统整体概述 15

3.2 代码翻译系统需求分析 16

3.2.1 系统的功能性需求 16

3.2.2 系统的非功能性需求 16

3.2.3 系统用例图 17

3.2.4 系统用例描述 17

3.3 代码翻译系统总体设计 19

3.3.1 系统的架构设计 19

3.3.2 系统整体工作流 20
3.4 系统各个子系统设计 21
3.4.1 预处理子系统 21
3.4.2 数据类型识别子系统 21
3.4.3 代码转换子系统 21
3.4.4 后处理子系统 25
3.5 本章小结 25

第 4 章 系统详细设计与实现 **27**
4.1 预处理子系统的实现 27
4.1.1 关键代码 27
4.2 数据类型识别子系统的实现 28
4.3 代码转换子系统的实现 29
4.3.1 词法分析器的实现 29
4.3.2 语法分析器的实现 38
4.3.3 代码生成器的实现 46
4.4 后处理子系统的实现 49
4.5 本章小结 50

第 5 章 系统测试与实验评估 **51**
5.1 系统的评估标准 51
5.1.1 系统的正确性 51
5.1.2 系统的自动化程度 51
5.1.3 目标代码的规范性 51
5.1.4 源代码的膨胀率 52
5.1.5 目标代码的性能 52
5.2 系统的测试用例 52
5.2.1 测试环境 52
5.2.2 测试用例 53
5.3 与其他系统的对比 60
5.4 本章小结 61

第 6 章 总结与展望 **63**
6.1 总结 63
6.2 展望 63

参考文献 **65**

插图目录

2-1 A = B × 2 + C × 3 的抽象语法树 9
2-2 访问者模式结构图 10
2-3 CMake 工作原理 12
2-4 LLVM 工作原理 13

3-1 基于抽象语法树的代码翻译系统应用场景图 15
3-2 代码翻译系统用例图 17
3-3 代码翻译系统架构图 19
3-4 系统整体工作流 20
3-5 预处理子系统的核心类图 21
3-6 数据类型识别子系统核心类图 22
3-7 代码转换子系统的工作流 22
3-8 词法分析器核心类图 23
3-9 语法分析器的核心类图 24
3-10 代码生成器的核心类图 25
3-11 后处理子系统核心类图 25

4-1 预处理子系统关键代码 27
4-2 数据类型识别子系统流程图 28
4-3 关系运算符的状态转换图 33
4-4 逻辑运算符的状态转换图 34
4-5 标识符的状态转移图 34
4-6 关键字 else 的状态转换图 35
4-7 空白符的状态转换图 35
4-8 数字的状态转换图 36
4-9 循环示例 36
4-10 标识符的相关信息 37
4-11 Matlab 的顶层文法 39
4-12 ClassDeclaration 节点的文法以及数据结构 39
4-13 FunctionDeclaration 节点的文法及数据结构 40
4-14 SwitchStatement 节点的文法和数据结构 40
4-15 WhileStatement 节点的文法和数据结构 41
4-16 IfStatement 节点的文法和数据结构 41
4-17 ForStatement 节点的文法和数据结构 42

4-18 TryCatchStatement 节点的文法和数据结构 42
4-19 Expression 节点的文法和数据结构 43
4-20 递归下降语法分析过程示意图 44
4-21 递归下降语法分析器顶层代码设计 45
4-22 遍历 FunctionDeclaration 类型节点 46
4-23 抽象访问者类 47
4-24 具体访问者类 47
4-25 抽象元素类 48
4-26 具体元素类 48
4-27 addOne 函数 48
4-28 中序遍历抽象语法树 49
4-29 后处理子系统的关键函数 49
4-30 添加头文件的关键代码 50

5-1 系统主界面 53
5-2 顺时针运动过程示意图 56
5-3 MatlabCoder 使用过程 60

摘　要

在工程实践中，人们对成本和效率有了更高的要求。在实验阶段，基于 Matlab 编码的高效性，人们使用 Matlab 进行编程，但是在部署阶段，渴望拥有较高的执行效率，于是使用 C++进行编程。另一方面，基于 C++的广泛适用性，可以将其部署到多种运行环境，从嵌入式到大型服务器都可以支持 C++代码的运行。但是在实验和部署阶段进行两次代码的编写会增加整个项目的成本、项目进度变慢且更容易出错，为了解决这个问题，本文设计并实现了一个代码翻译系统，自动化的将 Matlab 代码转化为 C++代码以提高效率、降低成本、加快项目进度。

本文在对现有的编译器和代码翻译系统仔细研究学习的基础上，总结实现原理，优化实现过程，设计并实现了自己的代码翻译系统。整体上，将代码翻译系统设计为前端和后端分离的模式。前端使用 Qt 框架来设计和实现与用户进行交互的界面，方便用户使用该系统。后端使用 C++语言来编写代码翻译系统的处理逻辑，方便独立测试和代码编写。后端的处理逻辑部分又可以分为四个子系统，预处理子系统对 Matlab 源代码进行预处理，数据类型识别子系统对 Matlab 源代码中的变量进行识别并记录下来，代码转换子系统对 Matlab 源代码进行词法分析、语法分析、建立抽象语法树、进行抽象语法树的转换、进行代码生成，后处理子系统进行转换后的处理。最终经过代码翻译系统的整个翻译过程，生成可正确运行和易于理解的目标代码。

本文设计和实现的代码翻译系统解决了现有系统的一些问题。首先，对和 ACS 控制器相关的 Matlab 代码和 C++代码进行了适配，使得在 ACS 控制器上进行编程时可以直接使用代码翻译系统进行代码转换。其次，相比于现有的系统，本文设计并实现的代码翻译系统自动化程度更高，几乎不需要人的参与。而且本文还实现了对常规数据类型的自动识别，不再需要人工提供配置文件或输入变量类型来协助进行代码翻译。

本文还对此系统进行了测试和评估，证明了它的易用性、正确性以及翻译以后在性能上的提升。

关键字：抽象语法树；代码翻译；Matlab；C++

Abstract

In engineering, people have higher requirements for cost and efficiency. In the experimental stage, based on the efficiency of MATLAB coding, people use matlab for programming, but in the deployment stage, they are eager to have high execution efficiency, so they use C++ for programming. On the other hand, based on the wide applicability of C++, it can be deployed to a variety of operating environments, and can support the running of C++ code from embedded to large servers. However, writing the code twice in the experiment and deployment phase will increase the cost of the whole project, slow the project progress and make mistakes more easily. In order to solve this problem, this paper designs and implements a code translation system, which automatically converts matlab code into C++ code to improve efficiency, reduce cost and accelerate the project progress.

In this paper, compiler and code translation system are studied carefully. The principle and process of their implementation are summarized. Design and implement our own code translation system. On the whole, the code translation system is designed as a model with front-end and back-end separated. The front-end uses the Qt framework to design and implement the interface to interact with users, so as to facilitate users to use the system. The back-end uses C++ language to write the processing logic of the code translation system, which is convenient for independent testing and code writing. The processing logic of the back-end can be divided into four subsystems. The Preprocessing Subsystem preprocesses the matlab source code, the data type recognition subsystem identifies and records the variables in the matlab source code, the code conversion subsystem performs lexical analysis, syntax analysis, establishment of abstract syntax trees, conversion of abstract syntax trees, code generation, and post-processing subsystem performs post-conversion processing. Finally, through the whole translation process of the code translation system, the target code that can run correctly and be easy to understand is generated.

The code translation system designed and implemented in this paper solves some problems of the existing system. First, the matlab code and C++ code related to ACS controller are adapted, so that the code translation system can be directly used for code conversion when programming on ACS controller. Secondly, compared with the existing system, the code translation system designed and implemented in this paper has a higher degree of automation, and hardly requires human participation. Moreover, this paper also realizes the automatic recognition of conventional data types, which eliminates the need to manually provide configuration files or input variable types to assist in code translation.

This paper also tests and evaluates the system, and proves its ease of use, correctness and performance improvement after translation.

Keywords: Abstract Syntax Tree; Code Translation; Matlab; C++

第 1 章　引言

1.1　研究背景及意义

计算机科学技术快速发展，软件行业迭代速度加快。将程序从一种高级语言转变为另一种高级语言可以显著提高软件开发效率，解决软件开发过程中的迫切需求。同时，降低软件的开发成本，提高软件的可靠性，对计算机行业的发展具有重要意义。

1.1.1　Matlab、C++简介

MathWorks 是全球领先的数学计算软件开发商，其产品服务于全球工程师和科学家，帮助他们加快发明、创新和开发的步伐。MathWorks 开发了众多优秀的产品，Matlab 就是其中之一。

Matlab 是一个集算法开发、数据分析、可视化和数值计算于一体的编程环境，被称为“工程师和科学家的语言”。由于编码的高效性以及 MathWorks 的全面的服务支持，使得 Matlab 在汽车、航空航天、能源、医疗设备、通信、电子、金融服务、工业自动化和机械、地球和海洋科学、生物技术和医药以及软件和互联网等行业都有广泛的应用。截至 2022 年，Matlab 在全球 190 多个国家和地区有超过 500 万的用户[1]。

C++是一种被广泛使用的计算机程序设计语言，Bjarne Stroustrup 博士在 20 世纪 80 年代于贝尔实验室工作期间发明并实现了它。C++是一种静态类型的、编译式的、广泛应用的程序语言，支持面向过程编程，支持面向对象编程，支持泛型编程，具有封装、继承和多态三大特性。C++语言在系统级编程领域一直被誉为“皇冠上的明珠”，在计算机系统、服务器开发、网络软件应用，甚至是游戏引擎、自动驾驶、VR 领域都广泛运用。并且，由于 C++语言本身的特性，使得其在工业领域得到广泛应用。首先，跨平台性非常好，几乎所有的操作系统都支持 C++。其次，运行效率高、简洁、安全。最后，C++语言对硬件的抽象能力以及对资源的掌控能力，使得在复杂场景中，具有其他语言不具有的优势。

1.1.2　ACS SPiiPlus 系列运动控制器简介

ACS Motion Control 是一家运动控制器和驱动解决方案供应商，其产品广泛应用于半导体制造、激光加工、增材制造、平板显示器制造、电子装配、生命科学等领域的高科技系统。ACS SPiiPlus 系列运动控制器是该公司旗下一款产品，提供强大的运动编程功能，并且提供强大的编程和调试工具。支持包括 C、C++、Matlab 在内的多种高级编程语言，支持所有的 Windows 环境以及部分 Linux 环境。

1.1.3　工程实践中存在的问题

工程实践中的开发过程大致可以分为实验和部署两个阶段。以 ACS SPiiPlus 系列运动控制器为例，在实验阶段，由于 Matlab 编码的高效性，工程师更倾向于使用 Matlab 编写控制算法。当算法验证完成，将其进行部署时，一方面由于 C++运行效率更高，我们希望使

用 C++语言。另一方面，在一些嵌入式系统中无法提供 Matlab 的运行环境，我们只能使用 C++语言。这样就导致在实验和部署阶段进行两次编码，使整体的开发效率大大降低，开发成本大大增加。所以本文提出把 Matlab 代码自动转化为 C++代码，解决工程实践中的痛点。

1.2　源代码到源代码翻译技术以及国内外研究现状

1.2.1　源代码到源代码翻译技术

源代码，是指一系列人类可读的使用某种编程语言编写的计算机指令，一般以文本文件的形式存储。源代码到源代码的翻译是指将某种编程语言编写的代码翻译为另一种编程语言编写的代码，且翻译前后的代码具有等效的功能。一般情况下，这两种代码处在大致相同的抽象级别，比如同为高级语言。

把一种编程语言通过人工翻译为另一种编程语言是需要耗费巨大的精力的，而且要求进行代码翻译的人员对源语言和目标语言都比较熟悉。例如，澳大利亚联邦银行花了大约 7.5 亿美元和 5 年的时间将其平台从 COBOL 转为 Java[2]。如果使用代码翻译器可能会更快且成本更低。所以，源代码到源代码的翻译技术在工程上具有重大的意义。首先，减少重复的编码工作。通过翻译器自动将一种编程语言的源代码转化为另一种编程语言的源代码，可以提高软件开发效率，优化软件开发过程。其次，生成的代码风格上更加的一致。避免了不同程序员相互协作时编码风格差异大的问题，有较好的一致性、规范性和可读性。最后，降低成本，节约开支。

1.2.2　国内外研究现状

有大量的人在进行源代码到源代码的翻译器的研究，并且，有很多已经投入使用的翻译器。总体上可以将代码翻译的方法分为三类：基于人工智能的代码翻译方法、基于抽象语法树的代码翻译方法和商业化的代码翻译方法。

基于人工智能的代码翻译方法

在自然语言方面，机器翻译已经被人们广为接受，甚至专业的翻译人员也越来越依赖于机器翻译。在理论上，这些方法也可以用来对各种编程语言进行翻译。有一些研究尝试使用机器学习的方法对编程语言进行翻译。例如，Nguyen 等人[3]在 Java-C#并行语料库上训练了一个基于短语的统计机器翻译（PBSMT）模型 Moses，他们使用两个开源项目 Lucene 和 db4o 的创建了数据集，这两个项目是用 Java 开发的，并移植到 C#。最终的结果表明，他们的模型在词法翻译上具有较高的准确率，在语法翻译上的准确率有待提高。类似地，Karaivanov 等人开发了一种从开源项目中挖掘并行数据集的工具[4]。kaggarwa 等人尝试使用统计机器翻译将 Python 2 代码转换为 Python 3 代码，对两个数据集进行了测试，获得了较好的表现。[5]。并且，他们还尝试将编程语言建模为自然语言，以在自然语言的基础上构建翻译模型，这样可进一步用于在同一编程语言的不同版本之间或者不同编程语言之间进行翻译。Chen 等人使用深度神经网络来解决代码翻译问题[6]，程序翻译是一个模块化过程，其中源语言的语法树的每一个子树都被翻译为目标子树，他们设计了一个树到树神经网络，将源树转换为目标树，同时为该网络模型开发了一种注意力机制，最终，他们比现有大翻译方法好 20 个百分点。以上这些方法都是有监督的，他们要么依赖现有的开源项目作为源代

码，要么就依赖翻译器来创建并行数据进行训练。Ahmad 等人为了让更多的人方便的使用机器学习的方法来研究代码翻译，发布了一个使用 Java 和 Python 编写的的语料库，以方便模型的训练[7]。

Facebook AI 实验室利用无监督的方法来训练一个完全无监督的神经网络翻译器[8]。他们在开源的 Github 项目的源代码上训练模型，并证明该模型可以高精度的在 C++、Java 和 Python 之间转换函数。方法完全依赖单一语言的源代码，不需要源语言或目标语言的专业知识，并且可以很容易的推广到其他编程语言。在由 852 个并行函数组成的测试集上进行测试，效果很好。

还有一些研究使用机器学习的方法生成源代码的伪代码、注释和说明文档等。Oda 等人训练 PBSMT 模型来生成伪代码[9]。为了生成训练集，他们雇佣程序员来写出已经存在的 Python 方法的伪代码。经过实验发现伪代码的准确率很高，这样就可以帮助人们来对代码进行理解。Barone 和 Sennrich 等人利用开源的 Github 中的 docstring 构建了一个 Python 函数库，他们表明，可以使用神经网络机器翻译模型将函数映射到与其相关的文档，反过来也是可行的[10]。北京大学的 Hu 等人提出了一个神经网络方法 DeepCom 来自动为 Java 方法生成注释[11]，并且在 Github 上的大量的项目中取得了不错了效果。Mahmud 等人对三种代码到注释的翻译模型进行了定量和定性比较，并且基于经验得出了误差分类法，可以推动未来的研究工作[12]。反过来，Parekh 等人利用伪代码生成程序代码[13]。他们基于神经网络创建一个翻译程序，使用神经网络技术将指定的伪代码自动转换为特定编程语言（如 C/C++/Java）的源代码。该程序允许用户使用自然语言编写代码，消除了编程语言和自然语言之间的现有差距，允许用户专注于输入输出逻辑，而不是专注于编程语言实现。从汇编语言到面向对象编程，编程已经变得容易，Rahit 等人为了进一步消除人机语言障碍，提出了一种使用递归神经网络和长短期记忆网络将人类语言转换为编程语言代码的机器学习方法[14]，外行人用自然语言编写代码，机器学习模型将其转换为目标编程语言。Koyluoglu 等人使用 transformers 把使用英语写的伪代码翻译为 C++代码[15]，并且他们专注于一行一行的翻译，使翻译出来的每一行代码都可以和上下文兼容。

机器学习的方法还可以用于代码的其他方面，比如代码比较、代码审查和代码提示等。Allamanis 等人提出了 NATURALIZE 框架[16]，该框架学习已有代码的风格，对新编写的代码的风格给出建议以保持代码风格的一致性，这可以提高代码的可读性以及程序的可维护性。Bhoopchand 等人提出了一种带有稀疏指针的神经网络语言模型[17]，旨在处理长依赖关系，与 LSTM 相比，具有更低的复杂度，准确度提高了 5 个百分点，标识符的预测精度提高了 13 倍。Li 等人试图通过神经网络来解决动态编程语言中的代码补全问题[18]，提出了一种指针混合网络，为代码补全开发了特定的注意力机制，并且在两个基准数据集上验证了指针混合网络和注意力机制的有效性。Chen 等人提出了一种新的基于序列到序列学习的端到端程序修复方法[19]，设计并实现了 SEQUENCER 技术，该技术基于源代码上的序列到序列学习来修复错误。Wang 等人提出了不同于现有的基于序列或语法的程序嵌入，他们提出了语义程序嵌入方法，该方法从程序执行轨迹中学习，结果显著优于基于序列和抽象语法树的程序嵌入，并将此方法用于修复程序[20]。Gupta 等人设计了一种称作 DeepFix 的端到端解决方案，它可以修复一些由于程序员经验不足或缺乏耐心而导致的错误，并且不需要任何的外部工具，DeepFix 的核心是包含注意力机制的多层序列到序列神经网络，最终实验表明可以对近一半的程序进行修复[21]。董文苑在深入研究代码相似性检测的相关工作基

础上，针对 Type-3 和 Type-4 类型代码克隆的识别，提出了一种多特征联合的代码相似性检测模型 JAT-BiLSTM，在 CodeOJPy 数据集的实验中准确度和 F1-score 均优于对照实验[22]。Zhang 等人为了解决代码克隆检测中的代码错位问题，提出了一种新的代码对其网络，设计了一个双向卷积神经网络来提取具有丰富结构和语义信息的代码片段的特征表示[23]。White 介绍一种基于学习的代码检测技术，并且实验结果很好[24]。Katz 等人提出了一种序列到序列模型来预测二进制程序的 C 代码[25]，标准序列到序列模型的一个常见问题是生成的函数不能保证可编译，甚至不能保证语法正确，为了解决这个问题，提出了几种对解码器使用附加约束的方法，以确保生成的函数符合目标语言的语法。Fu 等人提出了端到端的基于神经网络的代码反编译框架，与对照方法相比，实现了优异的性能[26]。Wang 等人针对现存的代码克隆检测只检测语法克隆的问题，提出要检测语义克隆，构建了流增强的抽象语法树，并且在流增强的抽象语法树上用两种不同的图神经网络来测量代码对的相似性[27]。

基于抽象语法树的代码翻译方法

抽象语法树在编译器和源代码到源代码的翻译器中都发挥着重要作用。Chaber 等人设计并实现了自动软件系统[28]，该系统将使用 Matlab 编写的模型预测控制算法转换成 C 代码并适配特定的目标微控制器。该系统包含一个翻译器用来翻译代码，一个模拟器用来验证算法，一个配置器用来配置算法参数，成功使模型控制预测算法在 STM32F746IGT6 微控制器上运行。Paulsen 等人在 EMC^2 项目中设计并实现了源到源转换器 Matlab2cpp[29]，借助于用户提供的有关变量数据类型的信息和一些特殊的转换规则，可以利用 Armadillo C++ 库将 Matlab 代码自动转换为 C++代码，使用 SeismicLab 软件包中的示例进行初步测试时，转换后的 C++代码能够进行数据处理，所以转换过程是有效的，这样就可以同时使用编码友好的 Matlab 和性能友好的 C++。作为 Matlab2cpp 的加强版，Paulsen 等人又设计和实现了 m2cpp[30]，相比初始版本，m2cpp 具有更高的自动化程度，并且通过调用 OpenMP 库和 TBB 库增加了对线程的支持，并且在进行实验时可以明显的看到性能上的改进。Bispo 等人设计并实现了 MATISSE[31] [32]，一个可以把 Matlab 代码转化为 C 代码的框架，并且转化后的代码可以适用于嵌入式系统，在转化的过程中，使用了面向切面的编程语言 LARA, 在转化的过程中提供关于 Matlab 的额外信息，比如变量的类型、变量的长度等等。Joisha 等人对 Matlab 的运行性能不满，并且认为 Mathworks 提供的 mcc（MATLAB-to-C Compiler）效率太低，所以设计并实现了 Matlab 到 C 代码的转换器，代码转换的效率以及转换后代码执行的效率都得到了明显提高[33]。中国科学技术大学的余泽霖等人设计并实现 Matlab 代码到 C 代码的转换系统[34]，将 Matlab 的高性能矢量运算和库函数对接到高性能 C 函数库 Math Kernel Library（MKL）上，并且，系统转换生成的 C 代码性能与人工编写的 C 代码性能相当，优于已有的转换方法生成的 C 代码。

除 Matlab 到 C/C++的代码翻译以外，还有许多其他语言之间代码翻译，他们也都用到了抽象语法树。Albrecht 等人设计并实现了 Ada 和 Pascal 之间的代码转换[35]，他们定义每种语言的子集，属于子集内的代码直接进行翻译。将有效的 Ada 程序和 Pascal 程序翻译为各自的子集，将一个子集转换为另一个子集，转换后的子集生成对应的代码来完成代码翻译。xu 等人设计并实现了一个 C 到 Java 的转换系统[36]，借鉴了编译器的实现，逐步的对源程序的词法、语法等方面的特性加以分析，将源程序代码解构为可直接进行翻译转换的单词符号，并对他们进行相应地转换，最终生成目标代码。Jahanzeb 等人为了提高程序运行

的性能和程序与 Modelica 应用程序集成，将 Matlab 转换为 Modelica[37]。他们设计并实现了 Matlab 到 Modelica 的转换器，词法和语法分析是在 OMCCp 工具的帮助下完成的，该工具生成 Matlab 的抽象语法树，随后由转换器生成可读和可重用的 Modelica 代码。Jurica 等人认为 Matlab 作为许多研究领域的主要计算工具，由于其是一款闭源的商业产品，阻碍了科学的进步，所以他们想要把在 Matlab 上的计算过程移植到 Python 上面，进而使用 Python 现有的开源的数据分析库，以促进科学计算的自由。所以设计并实现了 Matlab 到 Python 的编译器 OMPC[38]。de 等人描述了用于将 Matlab 程序转换为 Fortran 90 程序的 FALCON 系统的推理机制[39] [40]，FALCON 将静态与动态推理方法结合起来进行数据类型，秩和数据大小的推理，结果表明，效率得到了明显的提高。Huijsman 等人为了解决在转移到新的编程语言时需要重写大量遗留代码的问题，设计并实现了 Algol 60 到 Ada 的转换系统[41]。SmaCC 转换引擎是构建程序转换系统的工作台[42]，它被用来创建各种各样的转换，从简单的重构到更大的转换，例如在语言之间转换整个代码库，进行跨语言项目迁移等。可重用性是一直以来积极倡导的一个重要的软件工程概念，虽然使用同一编程语言实现的系统已经解决了可重用性问题，但不同编程语言之间的可重用性问题还没有解决，Trudel 等人基于从 Java 到 Eiffel 的源代码到源代码的转换，提出了一种在 Eiffel 程序中重用 Java 代码的解决方案[43]。原生的移动应用程序都具有依赖于特定平台的代码，An 等人为了将在一个平台开发的应用程序移植到其他平台，设计并实现了 j2sInferer[44]，用于将 Java 翻译为 Swift，j2sInferer 首先基于大括号和字符串相似性识别语法等价的代码，然后创建两种语言的语法树，利用语言对应关系的最小域知识迭代对齐语法树节点，并推断语法和 API 映射规则，最终实现转换。为了重用 C 语言开发的庞大代码库，以及利用现代编程语言的类型安全、面向对象契约等特征，Trudel 等人设计了 C 代码到 Eiffel 代码的源代码到源代码的翻译工具，翻译是完全自动的，支持整个 C 语言[45]。Qiu 等人尝试将 C 自动化翻译为 C++、将 Java 自动化翻译为 C++，并且阐述了翻译过程中的一些问题和挑战，同时提出，设计工具实现翻译规则并有效的融入人类互动是解决代码翻译问题的通用方法[46]。Lee 等人为了解决 GPU 编程复杂且容易出错的问题，提出了一个编译器框架，用于将标准 OpenMP 应用程序自动源到源转换为基于 CUDA 的 GPGPU 应用程序，以进一步提高可编程性，并使现有的 OpenMP 应用程序易于在 GPU 上执行[47]。

将抽象语法树和机器学习的方法结合起来也是研究的一个方向。LIANG 等人设计了一个缺陷定位系统 CAST 以帮助开发人员自动定位潜在的错误文件[48]，该系统利用深度学习和程序的自定义抽象语法树来自动有效的定位潜在的缺陷源文件，此外，该系统还使用定制的抽象语法树增强了基于树的卷积神经网络模型，区分出了到底是用户自定义的方法导致的错误还是系统方法导致的错误。将机器学习方法和抽象语法树结合以分析程序、表示程序也引起了广泛关注，Zhang 等人发现，传统的基于信息检索的方法通常将程序视为自然语言文本，这会丢失源码的重要语义信息，而基于抽象语法树的神经网络模型可以更好的表示源代码，他们将每个大型抽象语法树拆分为一系列小的代码语句对应的小型语法树，并通过捕获语句的词汇和语法信息将抽象语法树编码为向量，并最终生成代码片段的向量表示[49]。Rabinovich 等人提出了抽象语法网络，用于代码生成和语义解析，将非结构化输入映射到抽象语法树，并由解码器构造代码[50]。代码补全功能可以提高程序开发人员的效率，现有的机器学习方法进行代码补全存在忽视重要语法和语义关系信息的缺点，汤等人设计了一种基于抽象语法树的局部与全局关系的代码补全方法来解决此问题[51]。zhang 等人提

出了基于抽象语法树的软件缺陷检测技术，该技术应用半监督学习技术构建软件缺陷预测模型，使用该模型可以对工程源码进行判断[52]。cai 等人把程序代码转化为抽象语法树，结合抽象语法树的特点，把抽象语法树的节点转化成一种向量的表示形式，用以解决不可度量抽象语法树节点之间的语义距离问题[53]，并最终结合机器学习进行代码缺陷预测。

抽象语法树在其他方面也有重要的作用，比如代码比较、代码克隆检测等。Cui 等人提出了一种基于抽象语法树的剽窃检测工具 CCS，是一个代码比较系统，CCS 计算抽象语法树的哈希值，转换他们的存储形式，然后逐节点比较它们[54]。重构可以用来减少重复代码，优化软件的内部结构，Wu 等人提出了基于抽象语法树的重复代码的检测方法，提取每个源文件对应的抽象语法树，并且对所提取的抽象语法树进行方法级遍历，将抽象语法树的信息存储到哈希表中，然后对信息进行处理，得到信息之间的差异，完成重复代码的检测[55]。wang 等人为了计算短文本信息的相似度，首先对文本信息进行分词，在分词的基础上，对分词结果进行了词法分析、语法分析，构建出了基于中文文本的抽象语法树，结合向量空间模型的计算方法，进行文本相似度计算[56]。fang 等人研究了如何从抽象语法树提取信息用于静态代码检测，完成了一个基于抽象语法树的静态代码检测工具的开发[57]。fu 等人提出了两种基于抽象语法树的源代码抄袭检测方法，ASTK 和 WASTK，他们首先利用程序代码的层级结构特性将代码转换成抽象语法树，然后通过计算两棵抽象语法树的树核函数获得两份代码的相似度[58]。li 等人提出了一种基于源代码的抽象语法树的同源性对比算法，该算法针对语法树的特点，计算器 Hash 值，转换语法树的存储形式，并对语法树进行逐节点的对比，提高了算法的效率，降低了误报率[59]。he 设计实现了完备可用的基于抽象语法树的 Bug 修复模式检测系统[60]，为高效的解决 bug 修复模式检测问题提供了新的思路和系统工具。

商业化的代码翻译方法

Matlab Coder[61]从 Matlab 代码生成 C 和 C++代码，可以部署到包括桌面系统和嵌入式硬件在内的多种硬件平台，该产品支持多数 Matlab 代码，可以将生成的代码作为源代码、静态库或动态库集成到项目中。Alexander 也验证了 Matlab Coder 在某些算法开发过程中是可用的，但是在另外一些算法开发过程中不可用，并且他提出当为硬件编写优化的 C 代码时，生成的浮点 C 代码可以作为参考，然而，生成定点 C 代码对开发过程作用不大[62]。zhou 也对 Matlab Coder 自动代码转化过程的使用方法以及会遇到的问题进行了总结[63]。

除 Matlab 到 C 的转换外，也有其他的商业化的代码转换方法。为了抵消数字电路设计的高昂工程成本，硬件工程师越来越倾向于使用高级语言，如使用 C 和 C++来实现他们的设计，为此，他们采用高级合成工具，将其高级规范转化为硬件描述语言，如 Verilog，Xilinx Vivado HLS 就是这样一个工具[64]。翻译验证是一种验证代码生成器的源和目标的语义等价性的技术，Strichman 等人为实时车间代码生成器提供了一个翻译验证工具，该工具接收 Simulink 模型作为输入，并生成优化的 C 代码。Enthought 公司也在努力将 Matlab 转化为 Python[65]，以使计算更自由、成本更低。

其他的代码翻译方法

以上介绍了基于人工智能和基于抽象语法树的代码翻译方法，这里介绍一些其他的代码翻译方法。首先介绍基于 XML 的代码翻译方法，MUKHERJEE 等人设计并实现了一个翻译程序，该程序可以根据以 XML 格式提供的算法规范创建一段可执行代码，C 或者是 Java，并且很容易的扩展到不同的语言，允许用户只关注算法，而不是实现相关的问题[66]。Takizawa 等人设计了一个基于 XML 的代码翻译框架 Xevolver 以支持高性能应用程序的迁移[67]。sunitha 等人介绍了一种将模型转换为代码的方法，UML 用于建模，转换为 Java 目标语言[68]。

1.2.3 现有系统存在的问题

以上对国内外的研究现状做了一个概述，对很多代码翻译系统和代码翻译工具做了一个总结，其中和本文的 Matlab 到 C++的代码翻译系统关系最密切的是 Matlab Coder、AUTOMATIC、m2cpp、Matlab2cpp、MATISSE。但是，由于 ACS 控制器代码的特殊性，以上提到的工具或者方法都不可以直接完成 ACS 控制器的 Matlab 代码到 C++代码的翻译，他们或多或少都有一些不足，可以总结为如表 1-1 所示的几点。

表 1-1 现有系统存在的问题

系统名称	存在的问题
Matlab Coder	无法完成数据类型的自动识别，需要手动提供数据类型，且只可以翻译被封装为函数的 Matlab 代码。需要人为控制整个代码翻译的过程，翻译过程不够灵活。
AUTOMATIC	只针对特定控制算法，只适用于特定控制器，且并不适用于 ACS 控制器。
m2cpp	只可以利用利用特定的函数库，ACS 控制器的 C/C++代码库无法适用。
MATISSE	需要额外编程语言的协助，提高了代码翻译的难度，使整个的代码翻译过程变得繁琐。

1.3 本文主要工作

本文在上一节提出的代码翻译系统的理论和实践的基础上，将这些技术进行融合，设计并实现了高度自动化的完备可用的基于抽象语法树的代码翻译系统，为高效的进行代码翻译提供了新的思路和工具，加快了工程实践的进度。本文的主要工作可以概括为以下三个方面：

（1）在系统设计阶段，我仔细观察学习了多个编译器和翻译器的相关实现原理和实现过程，对他们的架构进行了认真分析，总结出各个子模块的功能以及他们之间数据的流动，对他们的数据结构的定义和实现算法进行了总结，明确每一步的输入和输出，确保对每一步的实现都有清晰的认识。

（2）在系统实现阶段，将前端界面和后端实现逻辑进行分离，系统前端使用 Qt 框架，后端的逻辑实现进行单独编写和测试，每个子模块作为独立的单元进行实现。后端的各个子系统系统化的实现论文的算法，根据设计阶段的需求分析和概要设计，用 C++语言去实现各个子系统。

（3）在用户使用阶段，用户在使用该基于抽象语法树的代码翻译系统时，操作简单明了，可以从文件管理器选择源代码文件或者直接输入源代码文件，系统对其进行翻译后，会显示翻译的结果并保存为目标代码文件。用户在使用系统时，整个翻译过程高度自动化，只需要对系统发出简单的指令就可以完成整个翻译过程。

1.4　本文的组织结构

本文的组织结构如下：

第一章引言部分。本章对代码翻译系统的源语言和目标语言、代码翻译过程中的问题、国内外相关工作人员的研究现状以及他们的研究内容、本文在该问题上的主要工作进行概述。

第二章技术概述。主要介绍项目中会使用到的理论和技术，比如抽象语法树、访问者模式、CMake、Qt 框架、LLVM 和 Clang-format。

第三章系统需求分析和概要设计。主要描述了本文的项目背景以及代码翻译系统总体设计，总结了系统的功能性需求和非功能性需求，还对系统的总体设计进行了介绍，包括系统的架构设计和系统的总体工作流。对系统的模块进行了划分，并且分别详细阐述了预处理子系统、数据类型识别子系统、代码转换子系统以及后处理子系统完成的工作和详细类图。

第四章系统详细设计和实现。在第三章概要设计的基础上，详细介绍系统的设计和实现，主要对预处理子系统、数据类型识别子系统、代码转换子系统以及后处理子系统的实现原理和实现细节进行详细介绍，展现每一个子系统的详细设计和关键代码。

第五章系统测试与实验评估。本章根据前文提出的需求和设计，对系统进行测试和评估。首先对评估的标准进行了简单介绍，对系统的使用进行了演示，对结果进行了总结。

第六章总结与展望。本章总结代码翻译系统的问题和本文的主要工作，对全文解决的问题做一个全面的总结并提出一些未来可以继续改进的地方。

第 2 章　技术概述

2.1　抽象语法树

在计算机科学中，抽象语法树（Abstract Syntax Tree，AST）是源代码语法结构的一种抽象表示。它以树状的形式表示编程语言的语法结构，树上的每个节点都表示源代码中的一种结构。之所以说是抽象的，是因为和语法分析树相比，这里并不会表示出真实语法中出现的每个细节，比如代码中嵌套的括号被隐含在树的结构中，并没有以节点的形式呈现。例如，与表达式 $A = B \times 2 + C \times 3$ 对应的抽象语法树如图 2-1 所示。

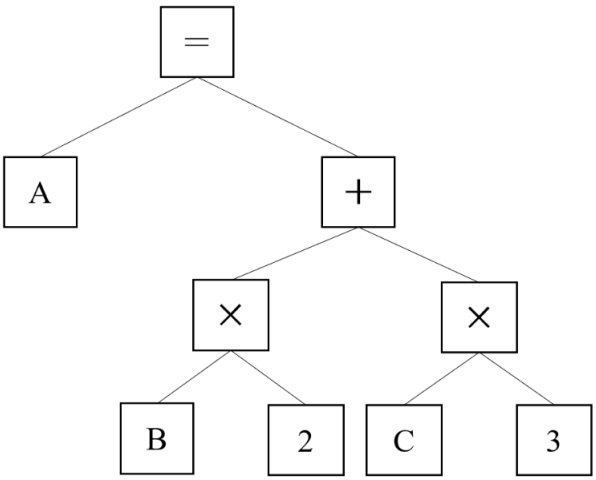

图 2-1　$A = B \times 2 + C \times 3$ 的抽象语法树

抽象语法树是一种接近源代码层次的表示，在语法分析阶段可以建立抽象语法树。抽象语法树已经用于许多实际的编译器系统，本文所做的源代码到源代码的转换系统也用到了抽象语法树，根据它可以重新生成源代码。

2.2　访问者模式

在软件开发过程中，很多人会反复遇到相同的问题，并且使用类似的方法去解决这一类问题，人们将这一类问题以及方法总结成理论，这些理论就叫做软件设计模式。使用设计模式可以提高代码的可重用性、可理解性以及可靠性[69]。设计模式一般包括模式名称、问题、目的、解决方案、效果等组成要素，这些组成要素说明了何时使用模式、如何使用模式以及使用模式会产生哪些效果。

一般将常见的设计模式分为三类，创建型模式、结构型模式和行为型模式，本文所使用的访问者模式就属于行为型模式。

访问者模式是一种较为复杂的行为型设计模式，在该模式中，将元素和对元素的操作分离开来，定义在两个类中，这样可以在添加对元素的新的操作的时候不去改变元素的类。访问者模式主要由两个部分组成，一是访问者，二是被访问元素。被访问元素属于不同的类型，定义在不同的类中，访问者对不同的被访问元素有不同的操作。访问者模式为软件系统提供了可扩展性，用户可以在不修改系统的情况下增加系统的功能，针对不同的元素提供不同的功能。

在使用访问者模式时，被访问元素通常不是单独存在的，他们存储在一个集合中，这个集合称为“对象结构”，访问者通过遍历对象结构实现对其中存储的元素的逐个操作。

在访问者模式中，被访问元素是以集合的形式出现的，不是一个一个单独存在的，一般称这个集合为“对象结构”，访问者可以对这个对象结构进行遍历而去操作对象结构中的元素。

访问者模式的结构较为复杂，其结构如图 2-2 所示。

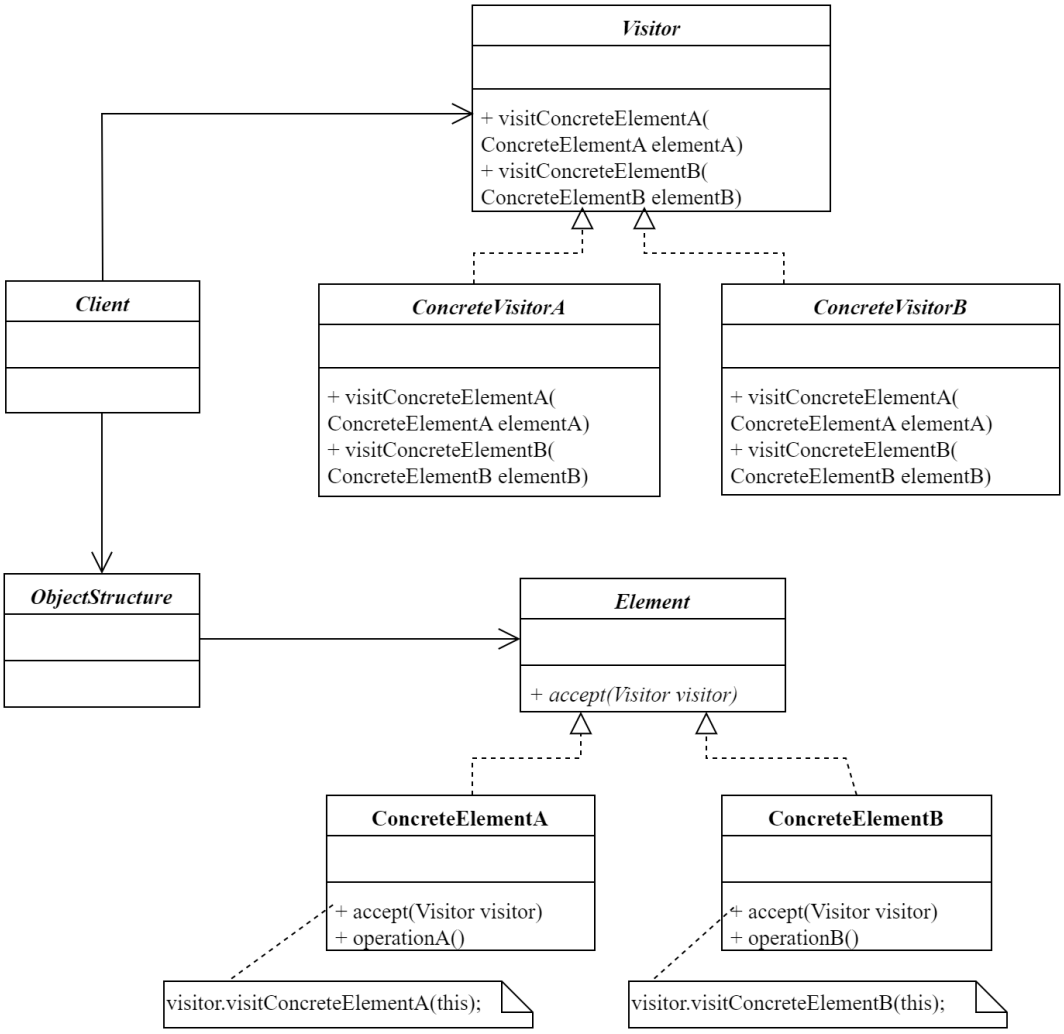

图 2-2　访问者模式结构图

从图 2-2 可以看出，访问者模式主要包含以下 5 个角色。

（1）抽象访问者（Visitor）：抽象访问者是具体访问者的父类，对于每一个具体元素类，抽象访问者都要声明一个访问操作，具体元素的类名作为访问操作的参数，并且根据操作

的名称可以知道是何种动作，这些操作在具体访问者中去实现，以访问具体元素。

（2）具体访问者（ConcreteVisitor）：具体访问者是抽象访问者的子类，对于抽象访问者中声明的操作，具体访问者来实现，用于访问对象结构某一种类型的元素

（3）抽象元素（Element）：抽象元素是具体元素的父类，一般是接口或抽象类，类中有一个 accept() 成员方法，该方法的参数是一个抽象访问者。

（4）具体元素（ConcreteElement）：具体元素是抽象元素的子类，实现了父类中定义的 accept() 方法，并且，在 accept() 方法中详细定义了对某一个元素索要执行的具体操作。

（5）对象结构（ObjectStructure）：对象结构是一个集合，有很多元素存放于该结构中，并且有方法可以遍历该结构中的元素。有很多的集合对象可以表示该结构，比如 C++ STL 中的 vector 容器。

访问者和元素是访问者模式的两个层次结构。其中，访问者层次结构由抽象访问者和具体访问者组成，访问元素对象的方法在抽象访问者中进行声明，每一种类型的元素都有对应的访问方法，并且在具体访问者中都有对应的实现。元素层次结构中包含抽象元素和具体元素，同一个访问者可以使用不同的访问方法去修改不同的元素，同一个元素也可以被不同的访问者用不同的方法修改。在该模式中，增加对元素的新的操作简单方便，无需大的修改，可扩展性非常好。

访问者模式主要适用于元素一般不会发生变化且元素较为复杂、但对元素的操作会发生变化的情况，此时，增加对元素的操作方法很方便。访问者对象来集中管理对元素对象的访问，而不是将这些方法分散在一个个的元素中。这样，用户想要添加操作的时候不会去修改元素类的层次结构，不会破坏封装性。本文所使用的抽象语法树就是一种不会轻易变动的数据结构，正适合使用此设计模式。

2.3 CMake

在软件系统开发的过程中，一个项目工程通常会包含很多的代码文件、配置文件、第三方文件、图片、样式文件等等，软件开发者需要将这些文件有效的组织起来最终形成一个可以流畅使用的应用程序，这就需要借助构建工具。如果在构建的过程中依赖人手工进行编译，工作起来会很繁琐，于是有了构建自动化工具。通过使用程序自动化的完成系列操作，将大大提升工作效率。

有很多的构建自动化工具，CMake 就是其中一种。CMake 是一个开源软件，它的扩展性很好，在很多项目中都使用它来管理构建过程，它不依赖于编译器和操作系统。与许多跨平台系统不同，CMake 与各个系统的原生构建环境结合使用。放置在每个源目录中的配置文件 CMakeLists.txt 用于生成标准构建文件，在 Linux 系统中生成 makefiles，在 windows 系统中生成.sln 文件，下一步就可以使用这些文件进行项目构建，生成可执行文件。CMake 的工作原理如图 2-3 所示。

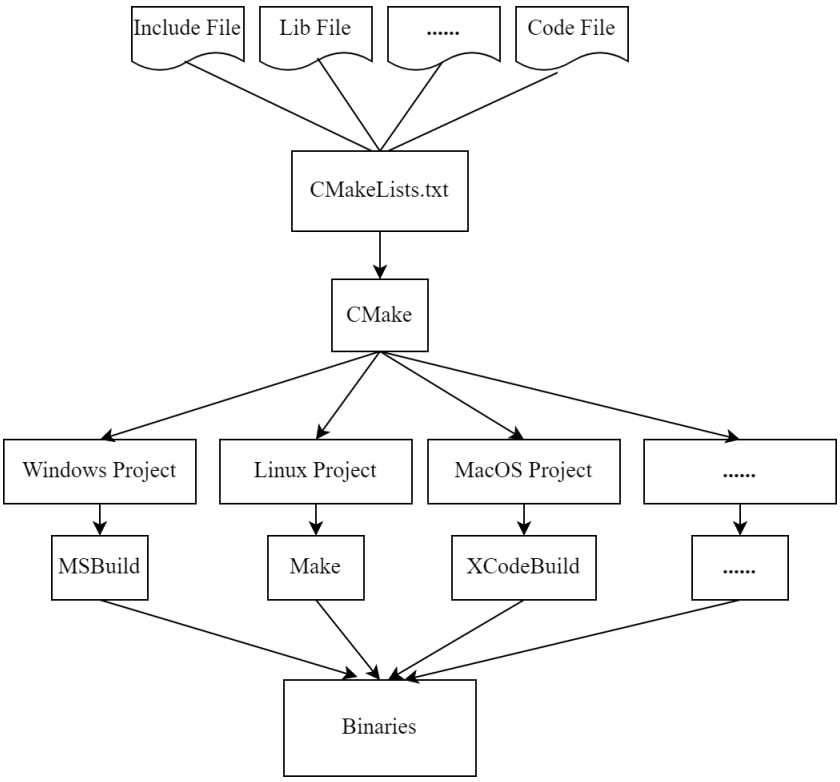

图 2-3　CMake 工作原理

2.4　Qt 框架

在软件开发过程中，经常会用到软件框架。软件框架集成了很多通用的、常用的功能，可以大大加快软件开发的进程，提高软件开发的质量，降低软件开发公司的成本。软件框架会包含很多软件开发过程中需要的工具，像编译器、常用代码库、应用程序接口和一些代码开发工具都会包含其中。它把这些工具集中在一起，减少因配置开发环境带来的时间开销，加快软件系统的开发和部署。

Qt 就是一个软件开发框架，支持在 Windos、Linux、MacOS 等操作系统上使用，一般可用于图形用户界面的开发，包含丰富的 C++类库，提供很多应用程序接口来加快程序的开发。Qt 软件框架代码在可读性、可维护性以及可重用性方面都表现很好，资源占用较少且性能较强。基于 Qt 的这些优点，选择它来开发应用程序的用户界面。

2.5　LLVM 和 clang-format

LLVM 项目是一套编译器基础设施项目，是自由软件，它由 C++编写，提供了用来开发编译器的框架，提供了一些用来开发编译器的组件和工具，方便新的编程语言的编译器的开发。LLVM 项目的核心就是 LLVM，它包含处理中间表示并将其转换为对象文件所需的所有工具、库和头文件。工具包含汇编程序、反汇编程序、位代码分析器和位代码优化器。LLVM 的工作原理如 2-4 所示。

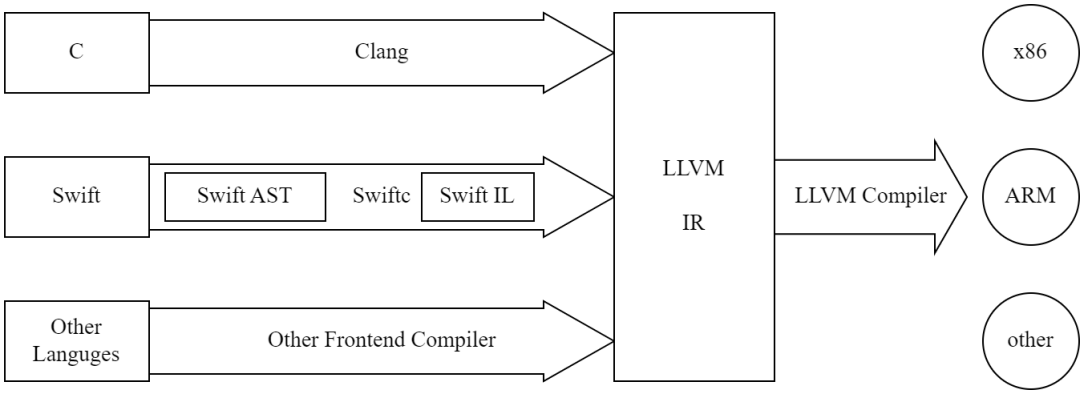

图 2-4 LLVM 工作原理

在团队进行合作的时候需要注意代码的格式，虽然很难统一每个人的编码风格，但是通过工具能够很好的管理代码格式。clang-format 就是一个进行代码格式化的工具，它基于 clang，并且能够对 C/C++/Obj-C 代码进行自动格式化，支持多种代码风格：Google，Chromium，LLVM，Mozilla，WebKit，也支持通过编写.clang-format 文件自定义风格，很方便的统一代码格式，提高代码的可读性和可维护性。

2.6 本章小结

以上阐述了在实现代码翻译系统的过程中用到的主要技术，抽象语法树用来表示代码结构，以访问者位基础来组织整个系统架构，CMake 来组织和管理具体的代码，Qt 用来实现用户界面，clang-format 对翻译得到的代码进行格式化。最终完成代码翻译系统的实现。

第 3 章　需求分析与概要设计

3.1　系统整体概述

在计算机科学中，同一个处理逻辑可以使用不同的编程语言来实现。有的编程语言注重效率，有的编程语言更关注性能，它们有各自的应用场景，但是都可以解决想要解决的问题。有时候需要兼顾效率和性能，比如在实验阶段，为了尽可能快的尝试不同的方法，需要高效率编码。如果已经得到确切结果，在部署阶段，需要性能高，功耗小。本文的代码翻译系统就是为了兼顾效率和性能，使工程师和设备都可以高效的工作。

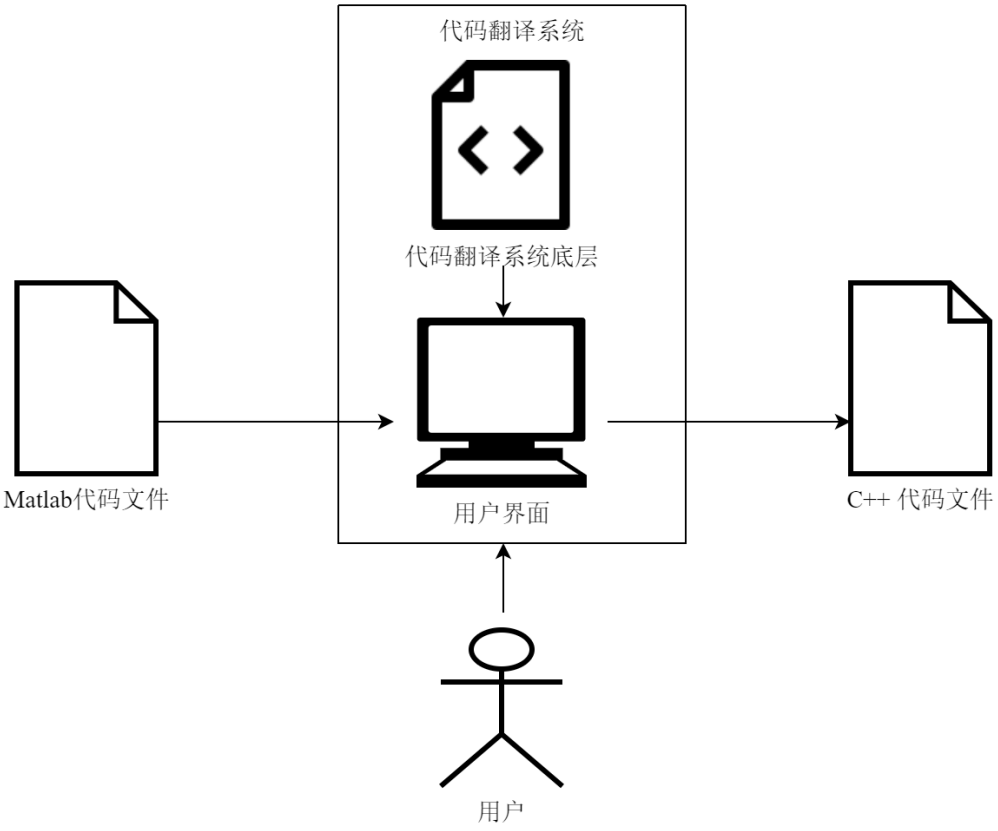

图 3-1　基于抽象语法树的代码翻译系统应用场景图

代码翻译系统的使用场景如图 3-1 所示，主要用于将 Matlab 代码翻译为 C++代码，以加速控制算法在 ACS 控制器上的实验和部署，避免了对一个算法进行两次开发，节省了大量的人力，降低了开发成本。

系统采用 C++语言开发，使用 Qt 完成用户界面的设计，代码翻译逻辑和用户界面分离，降低系统的耦合性。用户上传需要翻译的源文件，获得翻译后的 C++代码，整个过程操作简单，翻译过程自动完成。

3.2　代码翻译系统需求分析

3.2.1　系统的功能性需求

系统功能性需求描述了要开发的软件系统或组件的功能，是进行软件开发时要满足的最关键和最基本的需求。定义了系统需要做什么，实现了什么功能，提供了什么服务。表 3-1 列出了关于代码翻译系统的功能性需求，功能性需求相对比较单一，主要功能包括代码的翻译、代码的比较、代码的格式化以及代码的展示等功能。

表 3-1　代码翻译系统的功能性需求

需求 ID	需求名称	需求描述
R1	代码翻译	将用户输入或上传的 Matlab 代码翻译为 C++代码
R2	代码比较	将翻译得到的 C++代码和 Matlab 源代码进行比较，如行数比较等
R3	代码格式化	将翻译得到的 C++代码进行格式化，以增加代码的可读性
R4	代码展示	将翻译得到的 C++代码以及 Matlab 源代码展示在界面，以方便用户阅读和审查

3.2.2　系统的非功能性需求

在软件开发过程中，系统的非功能性需求是指对软件的运行状态或运行特性提出的要求，而不是对软件系统功能的要求。功能需求是系统实现的最低需求，但是为了带来更好的使用体验，我们同样需要重视非功能需求。

代码翻译系统的非功能需求如表 3-2 所示，主要包含可用性，易用性，可修改性，可扩展性，可移植性几个方面。

表 3-2　代码翻译系统的非功能性需求

需求 ID	需求名称	需求描述
R5	可用性	系统应该能稳定的执行代码翻译任务
R6	易用性	系统的各种操作简单明了，用户一看便知。系统在用户操作后给出提示信息，如状态信息，进度信息等
R7	可修改性	当用户要求修改系统功能时，系统应该是可以修改的
R8	可扩展性	当用户要求添加系统功能时，系统应该是可以进行功能添加的
R9	可移植性	系统能在 Windows、Linux 等多个操作系统下正常运行

3.2.3 系统用例图

在软件开发中，用例描述了系统为实现用户的目标而执行的功能，用例必须产生对系统用户可视的结果。用例图用来对系统行为建模并帮助捕获系统需求。用例图描述系统的高级功能和作用域，同时，用例图还标识系统与它的参与者之间的交互，用例图描述了系统执行的操作和用户使用用例的方式，但不会描述系统在内部工作的方式。

基于抽象语法树的代码翻译系统的用例图如图 3-2 所示，将 Matlab 代码翻译为 C++代码是该系统主要的功能，因此代码翻译是该系统的主要用例。另外，还有代码比较，代码的格式化等。用户同时可以在系统界面查看代码翻译系统的源代码和目标代码，对他们有一个直观的感受。

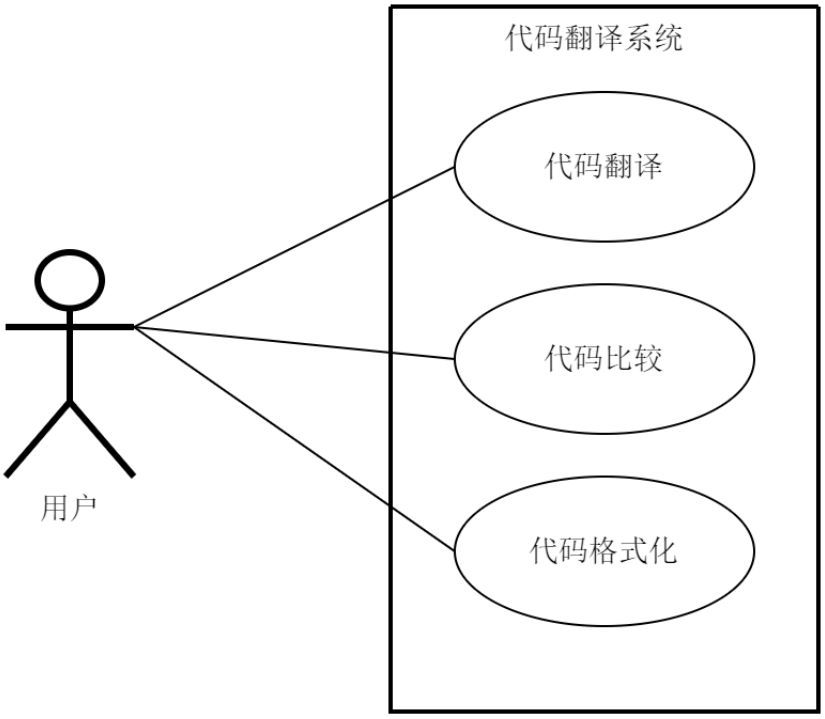

图 3-2 代码翻译系统用例图

3.2.4 系统用例描述

代码翻译是代码翻译系统的核心功能，表 3-3 是代码翻译用例的用例描述，代码翻译系统会检测用户是否上传文件，系统会读取用户上传的文件，调用底层的代码翻译逻辑对 Matlab 源代码进行翻译，并且展示翻译得到结果。

表 3-3　代码翻译用例描述

用例 ID	1
用例名称	代码翻译
参与者	用户
触发条件	用户想要把 Matlab 代码翻译为 C++
前置条件	用户已经安装代码翻译系统
后置条件	系统显示翻译得到的 C++代码
优先级	高
正常流程	1. 用户从代码翻译系统中打开文件管理器 2. 用户选择想要进行翻译的 Matlab 源代码文件 3. 系统显示选择的 Matlab 源代码文件 4. 用户点击翻译按钮进行翻译 5. 系统显示翻译结果
扩展流程	如果用户没有选择文件，给出提示信息
特殊需求	无

表 3-4 给出了代码比较的用例描述，用户在等待系统完成代码翻译以后可以进行代码比较，只需要简单的点击比较按钮就可以进行，并且比较完成后会在系统界面显示比较的结果。

表 3-4　代码比较用例描述

用例 ID	2
用例名称	代码比较
参与者	用户
触发条件	用户想要把 Matlab 源代码和 C++代码进行比较
前置条件	用户已经完成代码翻译
后置条件	系统显示代码比较的结果
优先级	高
正常流程	1. 用户点击代码比较按钮 2. 系统显示代码比较的结果
扩展流程	如果还没有完成代码翻译，给出提示信息
特殊需求	无

表 3-5 给出了代码格式化用例描述，用户在等待系统完成代码翻译以后可以进行代码格式化，只需要简单的点击格式化按钮就可以进行代码格式化，并且格式化完成后会在系统界面显示被格式化的代码。

表 3-5 代码格式化用例描述

用例 ID	3
用例名称	代码格式化
参与者	用户
触发条件	系统完成 Matlab 代码的翻译或用户想要格式化代码
前置条件	用户已经完成代码翻译
后置条件	系统显示代码格式化的结果
优先级	高
正常流程	1. 用户点击代码格式化按钮 2. 系统显示代码格式化的结果
扩展流程	如果还没有完成代码翻译，给出提示信息
特殊需求	无

3.3 代码翻译系统总体设计

3.3.1 系统的架构设计

在软件开发过程中，系统的架构设计是非常重要的一环。系统的架构描述了软件的整体结构，是对软件的一个划分。系统架构会包括软件组件、组件之间的关系、组件特性以及组件间关系的特性。系统架构可以和建筑物的架构想比拟。软件架构是构建计算机软件，开发系统以及制定开发计划的基础，可以根据软件架构制定开发团队需要完成的任务。

基于抽象语法树的代码翻译系统使用 Qt 框架进行界面开发，逻辑处理部分使用了性能较高的 C++语言，和 Qt 框架可以很好的结合，可以高效的完成代码的转换以及效果的展示。

系统总体框架如图 3-3 所示，接下来会对系统架构图中的各个模块的主要功能做具体的介绍。

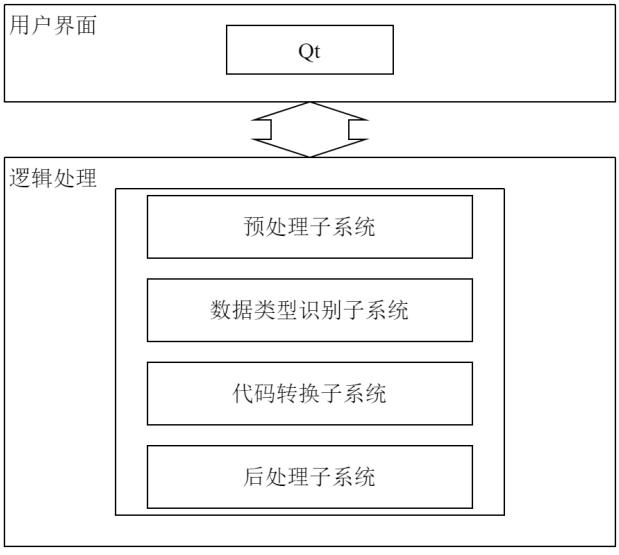

图 3-3 代码翻译系统架构图

3.3.2　系统整体工作流

系统整体工作流如图 3-4 所示，Matlab 源代码经过预处理子系统处理过后，生成预处理的 Matlab 代码，类型识别子系统读取预处理的 Matlab 代码，对其中的变量进行类型判断并记录下来，将类型记录表和预处理的 Matlab 代码一同送入代码转换子系统，得到初步的 C++代码，对得到的 C++进行进一步处理得到最终的 C++代码。

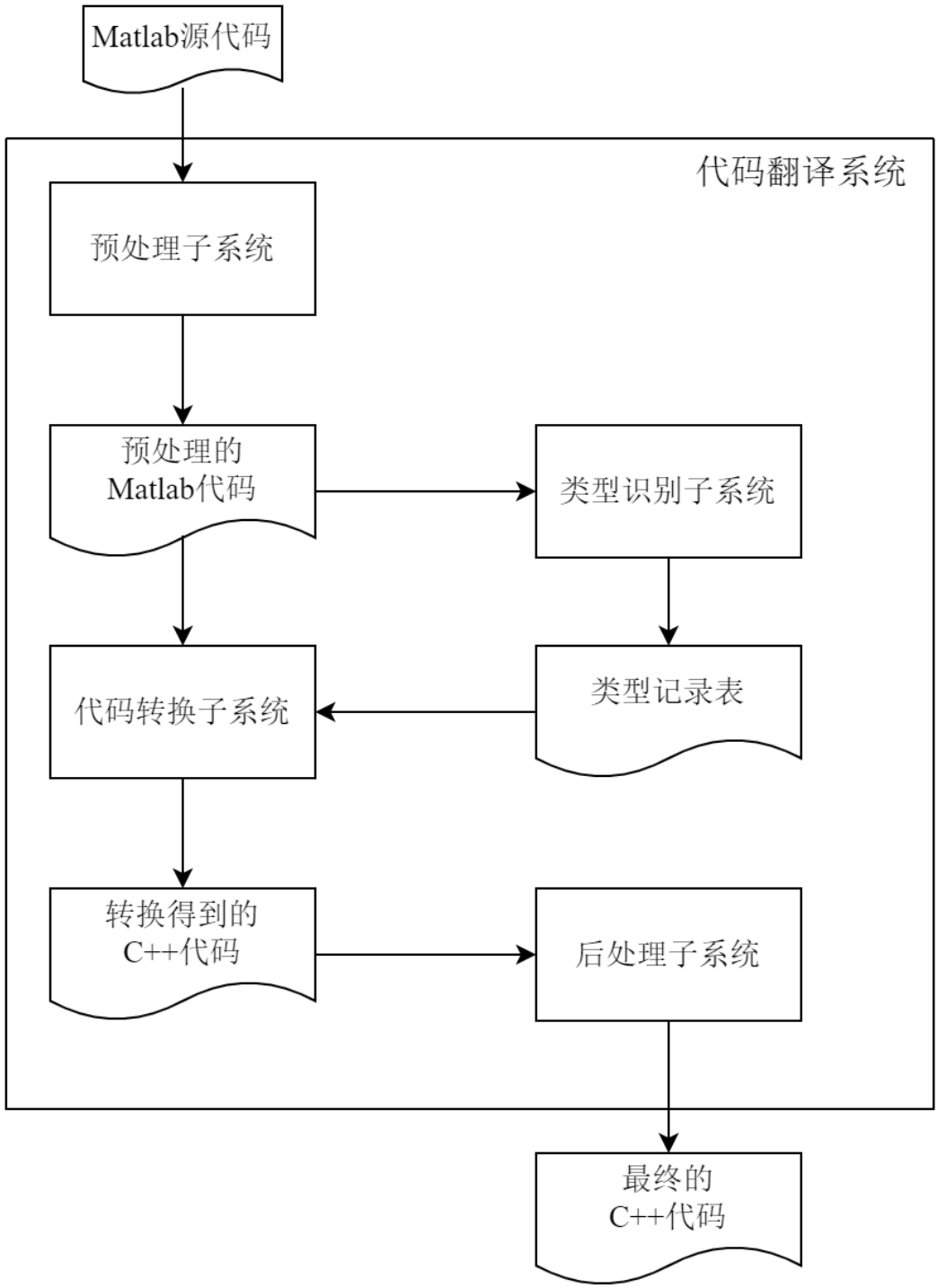

图 3-4　系统整体工作流

3.4 系统各个子系统设计

3.4.1 预处理子系统

预处理子系统主要是对 Matlab 代码进行简单的预处理。在代码翻译过程中，Matlab 代码和 C++代码的函数之间并不是严格对应的，在函数的参数个数和返回值之间可能会存在差别。代码转换子系统并不会去处理这种不对应关系，所以需要对 Matlab 代码进行预处理，消除这种不对应关系。但是，好在预处理的过程具有一定的规律性，可以进行批量的处理。经过处理以后的 Matlab 代码便可以送到代码转换子系统进行翻译。

图 3-5 是预处理子系统的核心类图。

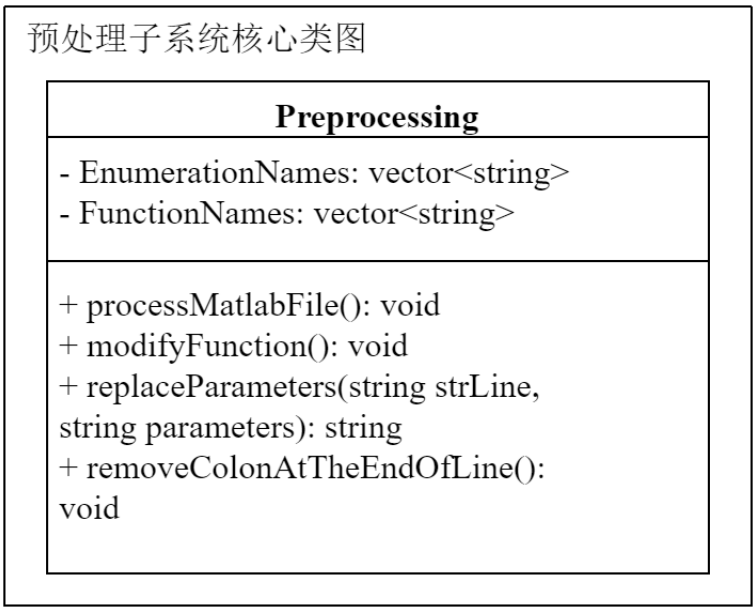

图 3-5 预处理子系统的核心类图

3.4.2 数据类型识别子系统

数据类型识别子系统主要用来对变量的数据类型进行识别。Matlab 作为一种动态类型编程语言，在编码的过程中并不需要提供变量的数据类型，程序在运行的过程中可以自动识别变量的数据类型，甚至变量的数据类型在代码运行时可以发生变化。与之相比，C++是需要提供数据类型的，并且变量一旦被初始化为某一种数据类型，一直到程序被销毁，变量的数据类型是不可以发生改变的。所以需要对变量的数据类型进行识别，将变量的数据类型记录下来，在后续的代码转换过程中会用到识别出的变量的数据类型。对变量的数据类型进行识别主要有两种思路：一是对于常规的变量，使用 C++11 提供的新特性 auto 来对变量的数据类型进行自动识别；二是对于特殊的不是内置的变量类型进行特殊处理，保证经过翻译得到的 C++代码中不会出现未定义变量。

图 3-6 是数据类型识别子系统的核心类图

3.4.3 代码转换子系统

代码转换子系统是整个代码翻译系统的核心部分，实现代码翻译的主要功能，完成从 Matlab 到 C++代码的转换。代码转换子系统的工作流如图 3-7 所示。

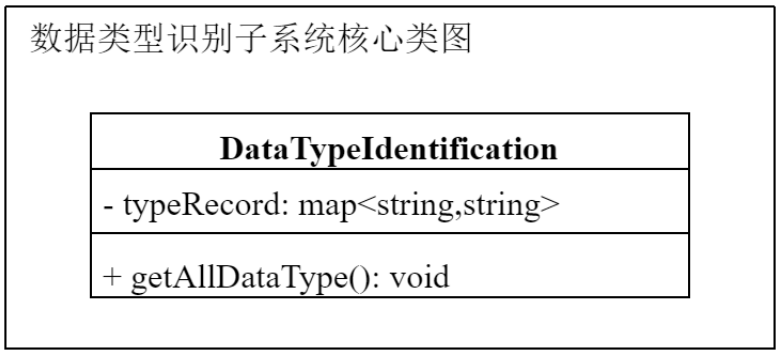

图 3-6　数据类型识别子系统核心类图

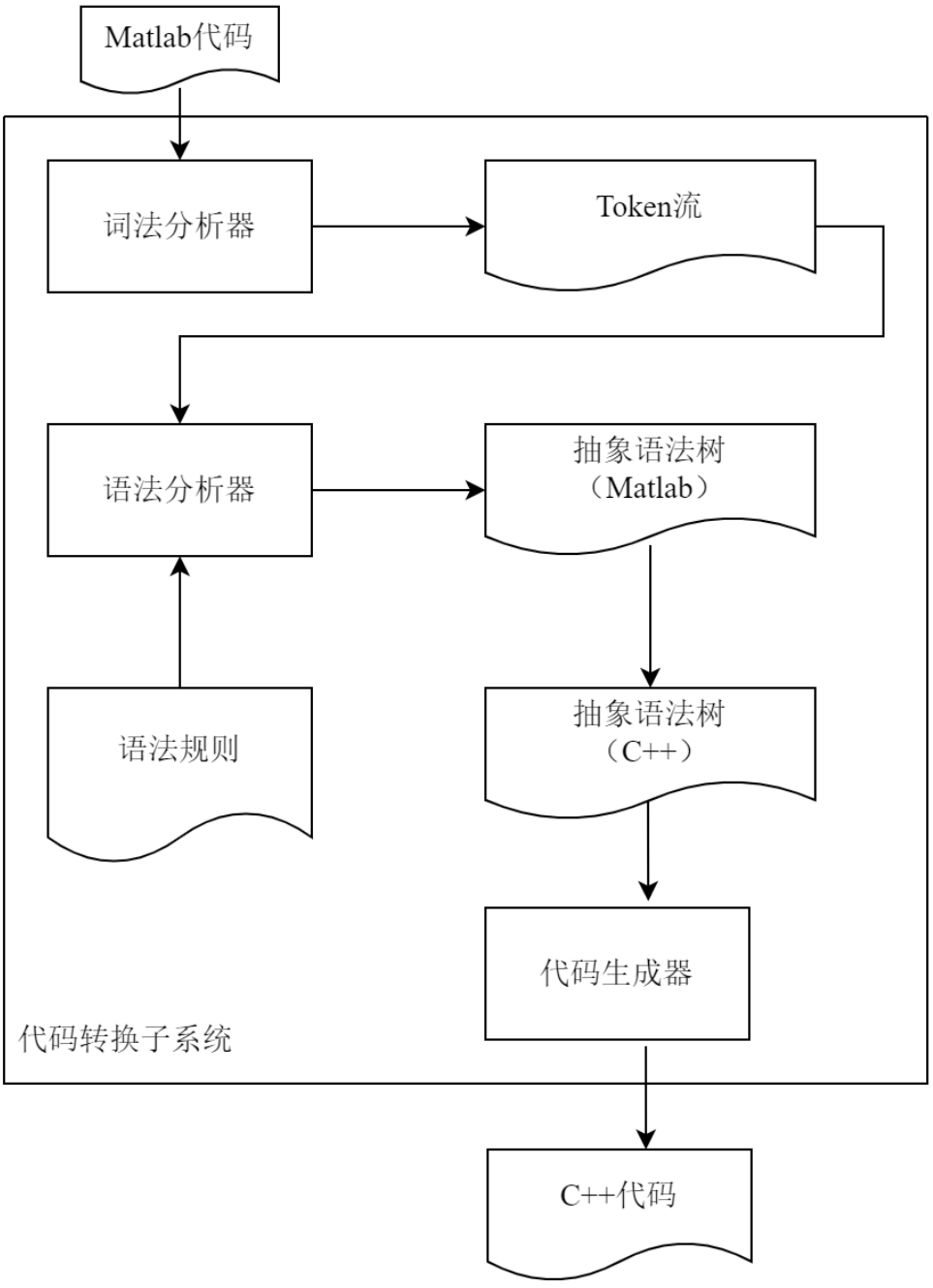

图 3-7　代码转换子系统的工作流

代码转换子系统本质上是一个小型的编译器，和编译器的区别是：编译器最终生成的是可以在机器上运行的二进制代码，而代码转换子系统最终生成的是 C++代码，需要 C++编译器进一步的编译才可以生成在计算机上运行的二进制代码。代码转换子系统由词法分析器、语法分析器和代码生成器组成，下面会分别介绍以上的各个模块。

词法分析器

词法分析是代码转换子系统的第一个步骤。将组成 Matlab 源程序的字符流送入词法分析器，词法分析器会将他们组织成为有意义的记号流。记号是一个字符串，是构成源代码的最小单位。在进行词法分析时，词法分析器还会对记号进行分类，并且标注记号的类型。在进行词法分析的过程中，如果发现无效记号，便会终止整个词法分析过程，给出错误提示；否则，完成词法分析，进行下一个步骤。

词法分析器核心类图如图 3-8 所示。

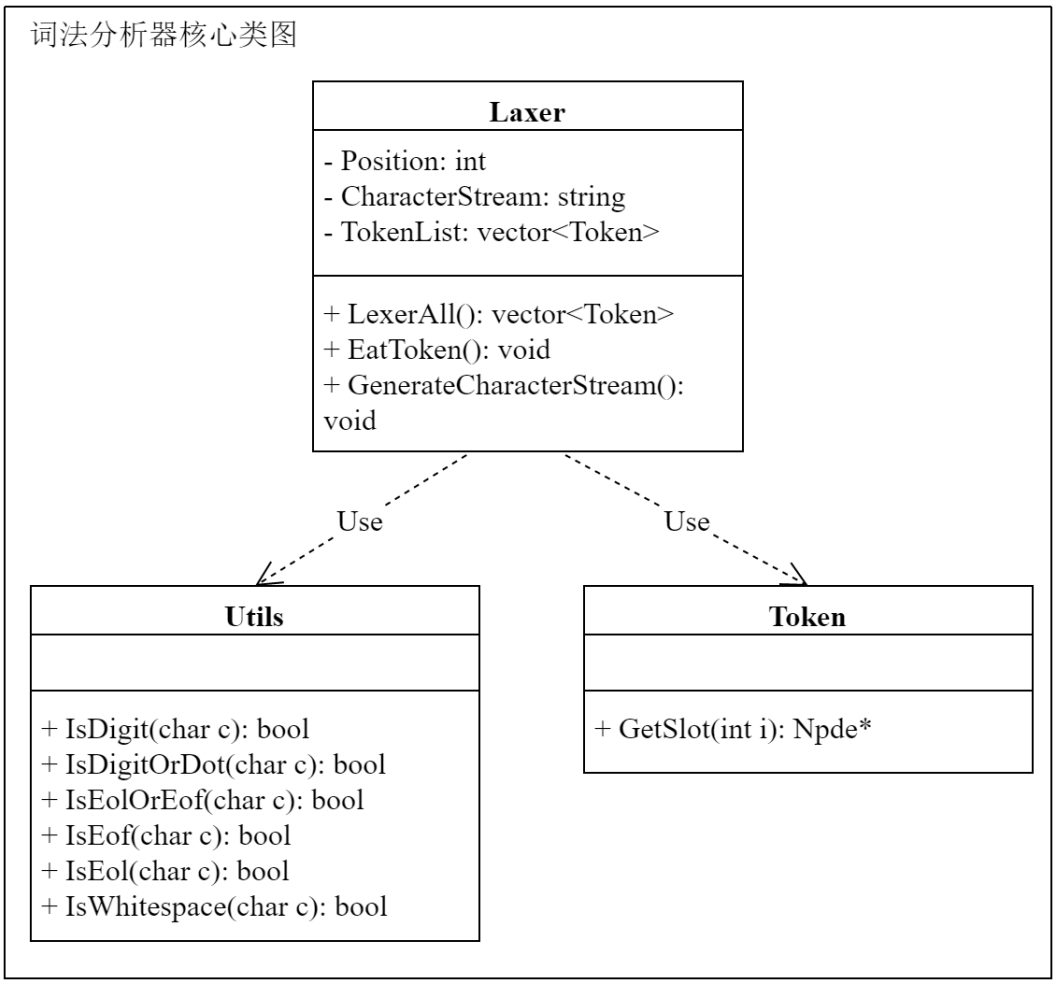

图 3-8 词法分析器核心类图

语法分析器

语法分析是代码转换子系统的第二个步骤。代码看上去是存储在文件中的字符串，但是，它本质是一个数据结构，且该数据结构具有复杂拓扑，语法分析器的作用就是表示出这

种结构。语法分析器使用由词法分析器生成的记号序列的第一个分量来创建表示 Matlab 代码结构的抽象语法树，抽象语法树在代码比较和代码转换过程中具有重要作用。Matlab 代码的各个部分被表示为抽象语法树上不同的节点，比如，表示代码块的节点，表示赋值语句的节点，表示循环语句的节点等等。最终，语法分析器会返回表示 Matlab 代码结构的抽象语法树的根节点。

语法分析器的核心类图如图 3-9 所示。

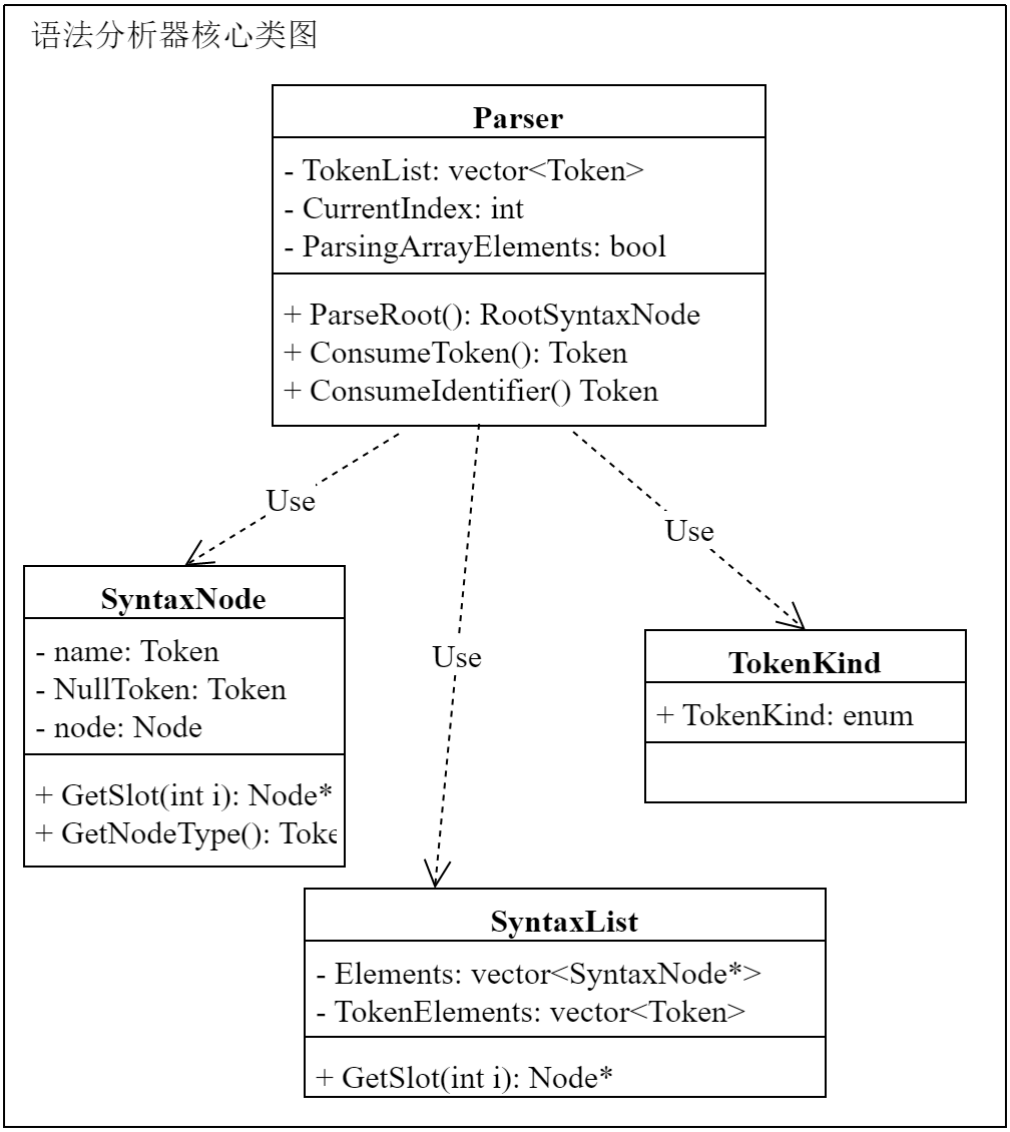

图 3-9　语法分析器的核心类图

代码生成器

最后是代码生成器。代码生成的过程和生成抽象语法树的过程是相反的，抽象语法树是它们之间的联系。在进行代码生成时，需要先把表示 Matlab 代码的抽象语法树转化为表示 C++代码的抽象语法树。转换完成以后，后续遍历表示 C++代码的抽象语法树，添加必要的信息，把抽象语法树的节点转化为与节点相对应的代码。最终完成整个代码的转换过程。

代码生成器的核心类图如图 3-10 所示。

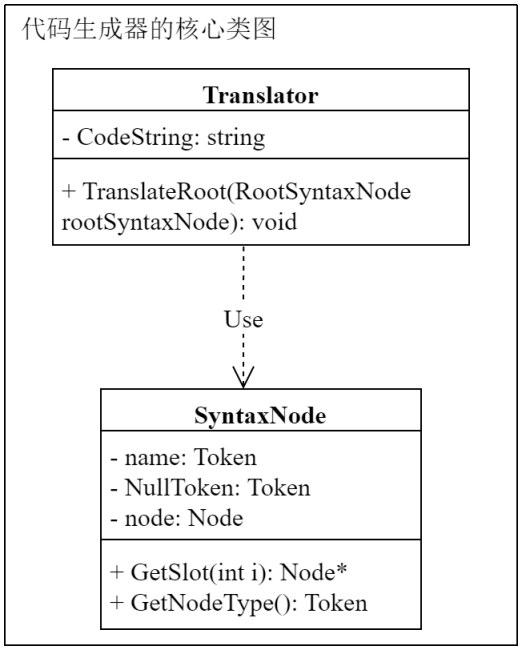

图 3-10 代码生成器的核心类图

3.4.4 后处理子系统

后处理子系统是对已经得到的代码进行小幅度的修改。转换得到的 C++代码表示出了 Matlab 代码的整体结构，但是转换得到的 C++代码缺少一些成功运行所必须的信息，这一部分信息是需要后处理子系统添加上去的。比如，处理一些特殊的函数，添加数据类型的预定义，添加 C++头文件。总之，加上一些 C++代码成功运行的必要代码，保证翻译得到的 C++代码可以直接运行而不需要在手动对代码进行调试。

后处理子系统核心类图如图 3-11 所示。

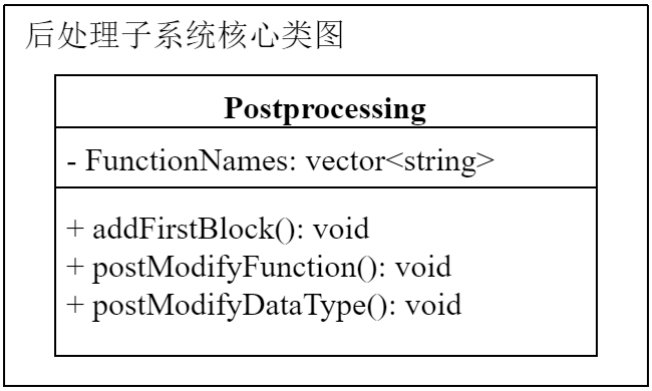

图 3-11 后处理子系统核心类图

3.5 本章小结

本章是对代码翻译系统的整体进行设计。对整体的工作流进行概述，并且分别对预处理子系统、数据类型识别子系统、代码转换子系统和后处理子系统进行分析设计，以有效的指导后续的详细设计。

第 4 章　系统详细设计与实现

4.1　预处理子系统的实现

预处理子系统主要解决 Matlab 源代码和 C++目标代码函数之间的不一致性，比如函数参数个数的不一致，函数名称的不一致，返回类型的不一致等等，方便下一步的代码转换过程。

4.1.1　关键代码

预处理子系统关键代码如图 4-1 所示。

```
void Preprocessing::modifyFunction()
{
    while (!in.eof())
    {
        getline(in,str);
        if (in.eof())
        {
            break;
        }
        for (int i = 0;i < FunctionNames.size();i++)
        {
            size_t pos = str.find(FunctionNames[i]);
            if (pos != string::npos)
            {
                str = modifyConcreteFunction(i,str);
            }
        }
        matlabText.append(str + "\n");
    }
}
```

图 4-1　预处理子系统关键代码

预处理子系统的主要处理步骤如下：

1. 打开并读取 Matlab 源代码文件。

2. 对 Matlab 源代码的每一行进行遍历，查找是否有需要处理的函数，如果有，调用处理函数进行处理。

3. 处理以后将处理的结果保存到文件中，方便下一步的处理。

4.2　数据类型识别子系统的实现

数据类型识别子系统的主要作用是对 Matlab 源代码中的变量的类型进行判断并且记录下来，方便后续步骤的使用，该子系统工作的流程图如图 5-2 所示。

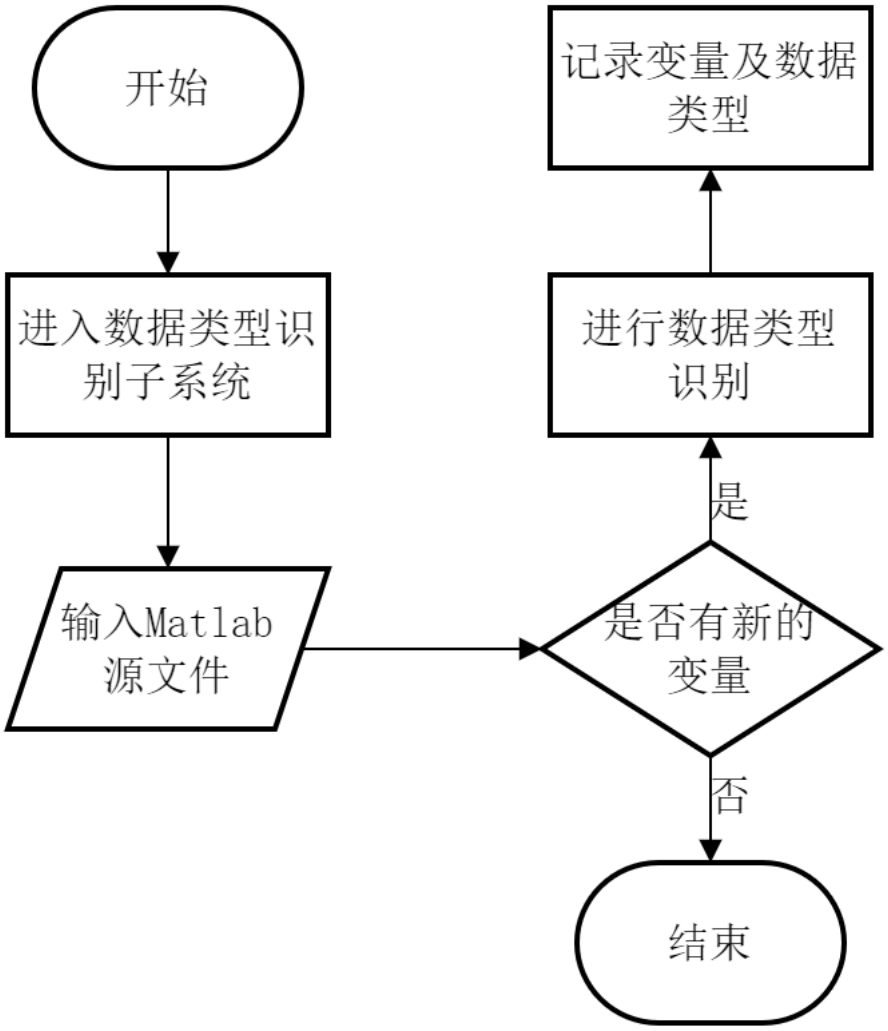

图 4-2　数据类型识别子系统流程图

主要处理步骤如下：

1. 进入数据类型识别子系统。

2. 打开并读取 Matlab 源代码文件。

3. 寻找赋值语句中新声明的变量，记录到变量表中。

4. 利用判断类对赋值语句右边的值的类型进行判断。

Matlab 中主要的数据类型如表 4-1 所示。

表 4-1 Matlab 中主要的数据类型

数据类型	描述	数据类型	描述
int8	8 位有符号整数	uint8	8 位无符号整数
int16	16 位有符号整数	uint16	16 位无符号整数
int32	32 位有符号整数	uint32	32 位无符号整数
int64	64 位有符号整数	uint64	64 位无符号整数
single	单精度数值数据	double	双精度数值数据
logical	逻辑值 1 或 0	char	字符数据
cell array	索引单元格数组，每个单元格能够存储不同维度和数据类型的数组	structure	每个结构都具有能够存储不同维度和数据类型的数组的命名字段
function handle	指向函数的指针	user classes	从用户定义的类构造的对象

4.3 代码转换子系统的实现

代码转换子系统是代码翻译系统的核心部分，本节将对如何实现代码转换子系统的各个子模块进行详细说明。

4.3.1 词法分析器的实现

在代码转换子系统中，第一个步骤就是进行词法分析。将源文件包含的字符流送入词法分析器，词法分析器发挥作用，对字符流进行分析，形成单词，输出单词流。并且会进一步判断输出的单词是否有效，如果该单词有效，会对单词进行归类。

词素、模式和词法单元

为了更清晰的介绍词法分析器的实现过程，先对其中的三个概念进行说明，分别是词素、词法单元和模式。

词素，说的简单一点，就是组成源程序代码的某个单词，可能由一个或多个字符组成，在进行词法分析时，该词素会被当成一个词法单元的实例，和词法单元的模式去匹配。

模式表明了在进行词法分析时词法单元中的词素可能具有的形式，或者说是形成词素的规则。比如，某一个词素是关键字，那么组成这个关键字的字符序列就是该词素的模式，或者说根据这个模式形成了这个关键字。其他词法单元的词素可能具有更复杂的模式。

词法单元是进行词法分析后得到的结果，包含词法单元名和词法单元的属性值。一般用一个抽象符号表示词法单元名，比如一个关键字就可以表示一个词法单元名，由字符序列组成的标识符也可以表示一个词法单元名。在后续进行语法分析时，词法单元名作为输入，并且使用词法单元名来表示词法单元。

词法分析中单词的分类

在介绍词法分析器聚合字符形成单词之前，先介绍一下聚合后形成的单词可以分为哪些种类，可以将单词分成关键字、标识符、运算符、界符和常量等。在具体实现时，采用枚举类型来对其进行表示。

在 Matlab 中可以使用 iskeyword 来获取所有关键字列表，得到的关键字如表 4-2 所示。

表 4-2　Matlab 关键字列表

”break”	”case”	”catch”
”classdef”	”continue”	”else”
”elseif”	”end”	”for”
”function”	”global”	”if”
”otherwise”	”parfor”	”persistent”
”return”	”spmd”	”switch”
”try”	”while”	

Matlab 作为数值运算的重要工具，定义了多种运算符，可以分为算术运算符、关系运算符和逻辑运算符。

Matlab 提供的算术运算符如表 4-3 所示。

表 4-3　Matlab 算术运算符

符号	描述	符号	描述
+	加法	+	一元加法
-	减法	-	一元减法
.*	按元素乘法	*	矩阵乘法
./	按元素右除	/	矩阵右除
.\	按元素左除	\	矩阵左除
.^	按元素求幂	^	矩阵幂
.’	转置	’	复共轭转置

关系运算符用来判断两个变量之间的关系是否成立，判定为真或者假。Matlab 提供的关系运算符如表 4-4 所示。

表 4-4 Matlab 关系运算符

符号	描述	符号	描述
==	等于	~=	不等于
>	大于	>=	大于等于
<	小于	<=	小于等于

逻辑运算又称布尔运算，得到的结果通常是真假。Matlab 提供的逻辑运算如表 4-5 所示。

表 4-5 Matlab 逻辑运算符

符号	描述	符号	描述
&	等于计算逻辑 AND	|	计算逻辑 OR
&&	计算逻辑 AND（具有短路功能）	||	计算逻辑 OR（具有短路功能）
~	计算逻辑 NOT		

词法分析器不仅要识别以上提到的关键字，运算符，还要识别出其他的特殊字符，比如界符等等。需要识别的特殊字符如表 4-6 所示。

表 4-6 Matlab 代码中的界符等特殊字符

符号	描述	符号	描述
@	at 符号	.	句点或点
...	省略号	,	逗号
:	冒号	;	分号
()	圆括号	[]	方括号
{}	花括号	%	百分号
%{ %}	百分号加花括号	!	感叹号
?	问号	‘ ’	单引号
“ ”	双引号	N/A	空格字符
N/A	换行符	N/A	文件结束符

词法分析器中单词的识别

上面已经介绍了源代码可以被分为哪些类别的单词，在分类之前首先就要去识别这些单词，识别以后才可以进行分类。下面会介绍词法分析器中进行单词识别的原理以及如何用代码去实现他们。

词法分析中词法单元识别的过程是这样的：将单词和模式去匹配，如果匹配成功，归类到该模式表示的类下，如果匹配失败，说明源代码有误，需要重新检查。

正则表达式是一种用来描述词素模式的重要表示方法，使得对于词法分析过程中模式类型的表达更简洁、更高效。正则表达式经常被人们拿来描述语言，对于字母表上的符号以及由这些符号进行并运算、连接运算以及闭包运算得到的新的符号，正则表达式都可以进行描述。例如，假设规定标识符只可以由字母、数字和下划线组成，用 letter_表示字母或下划线，用 digit 表示数字，并且只能以字母或下划线开头，那么可以使用如下的正则定义来描述对应于 Matlab 标识符的语言：

$$letter\_(letter\_|digit)^*$$

上式中的竖线表示并运算，括号用于把子表达式组合在一起，星号表示零个或多个括号中表达式的连接，将 letter_和表达式的其余部分并列表示连接运算。

在词法分析过程中，我们用正则表达式表示模式。有一部分单词的模式仅仅和该单词匹配，不会和其他任何单词匹配，比如关键字、操作符。但是有的模式可以和一部分单词匹配，比如描述标识符的模式，描述数值常量的模式。

下面的正则表达式是标识符对应的正则表达式：

$$(A|B|...|Z|a|b|...|z|_)((A|B|...|Z|a|b|...|z|_)|(0|1|...|9))^*$$

上面已经用正则表达式来表示一个模式，现在，我们要根据词法单元的模式去构造用以识别它们代码。这是我们去完成整个词法分析器代码的一个中间环节，将模式做进一步的转换，转换成表示构造过程的流图，即状态转换图。

状态转换图中包含很多状态，用节点或圆圈表示。在进行词法分析的过程中，对输入串进行扫描，找到和状态转换图匹配的词素，匹配过程中的每个情况都可以在状态转换图中找到。这其中的每一个状态都可以看作是对我们已经看到的位于 lexemeBegin 指针和 forward 指针之间的字符的总结，它包含了我们在进行词法分析时需要的全部信息。

状态转换图从一个状态指向另一个状态，中间用边进行连接，每条边都有标号，可以是一个或多个。假设当前的状态是 s，并且词法分析器获得的下一个字符是 a，我们就会去找标号是 a 的边，并且该边从 s 出发。如果可以找到满足条件的边，就会向前移动 forward 指针，并且将当前状态置为该边所指向的下一个状态。假定我们的状态转换图是没有二义性的，那么在当前状态和边上的标号都确定的情况下，包含该符号的边只有一条或者没有。对状态进行如下的一些约定：

（1）假设我们已经找到了一个词素，那么此时的状态应该是接受状态，即最终状态，一般我们用双层的圈来表示这样一个状态。该状态下一步的动作不是继续进行转换，而是将得到的词法单元向上返回给语法分析器以进行下一步的语法分析。

（2）假设正在被分析的词素并不能将状态转移到接受状态，那么我们需要将 forward 指针进行回退，并且，此时需要在接受状态附近添加一个*表示需要进行回退。

（3）需要有一个开始状态，该状态由一条特殊的边指出，该边没有出发节点，且该边的标号是 start 而不是其他的。在没有任何输入符号的时候，状态转换图就位于此状态。

关系运算符的状态转换图如图 4-3 所示。从初始状态 0 开始，如果第一个输入的符号是

<，那么可以匹配到的关系运算符就是<或者<=。如果第一个输入的符号是=，那么可以匹配到的关系运算符就是==。如果第一个输入的符号是>，那么可以匹配的关系运算符就是>或者>=。如果第一个输入的符号是~，那么被匹配到的关系运算符就只能是~=。

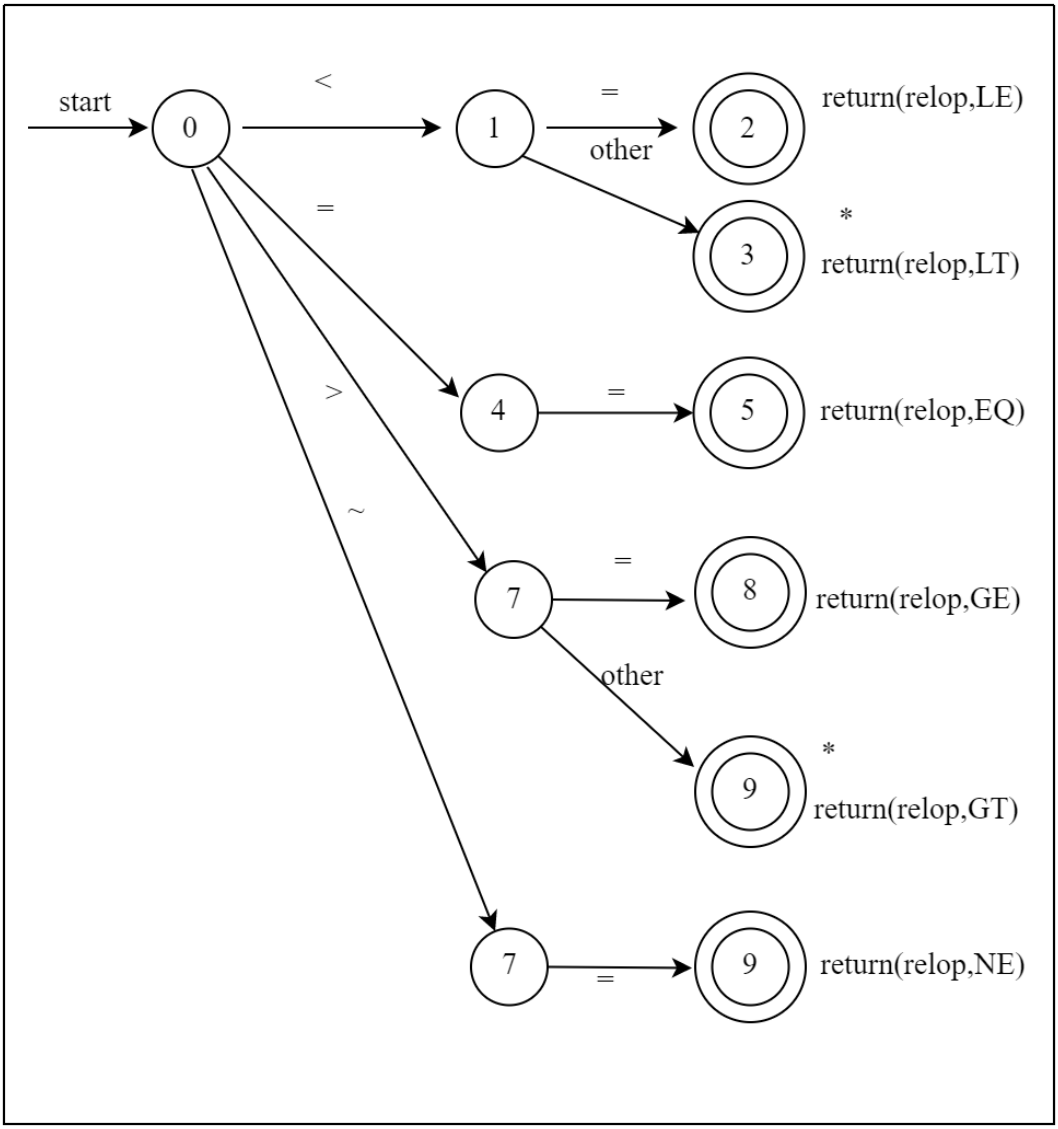

图 4-3 关系运算符的状态转换图

逻辑运算符的状态转换图如图 4-4 所示。从初始状态 0 开始，根据输入的字符进行状态的转移。如果第一个输入的符号是&，接着再来一个&符号的话，匹配到的逻辑运算符就是&&，否则就是&。如果第一个输入的符号是|，并且下一个也是符号|，那么匹配到的逻辑运算符就是||，否则就是符号|。如果只有一个~，那么就是逻辑运算符~。

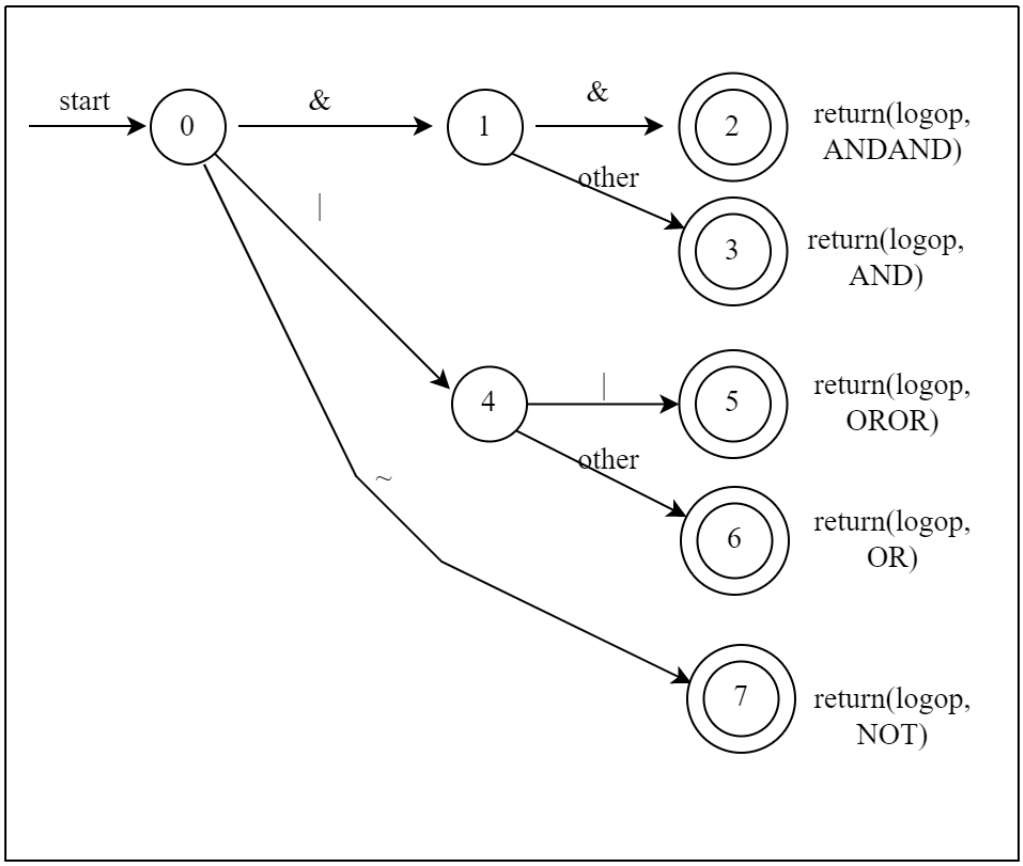

图 4-4　逻辑运算符的状态转换图

标识符的状态转移图如图 4-5 所示。标识符以字母或下划线开头，后接零个或多字母、下划线或数字。

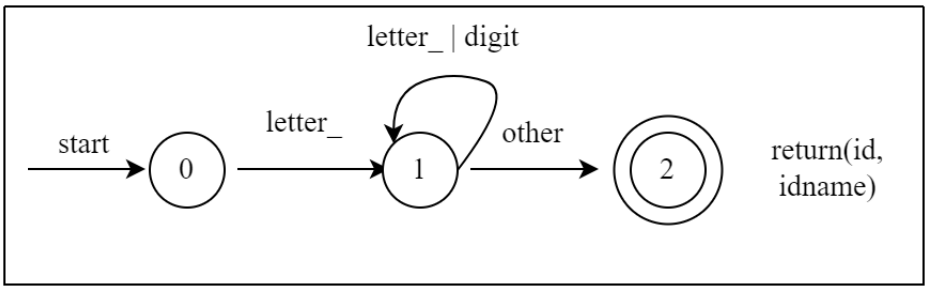

图 4-5　标识符的状态转移图

识别标识符时有一个问题要解决，通常，像 if 或 else 这样的关键字是被保留的。因此虽然他们看起来像标识符，但他们不是标识符。因此，尽管我们通常使用如图 4-5 所示的状态转换图来寻找标识符的词素，但对关键字的识别也可以使用这个图。

对于这些和标识符类似的关键字，有两种方法对他们进行处理。

（1）在初始化的符号表中填入关键字。符号表会将这些关键字标识出来以区别于普通的标识符，并指出他们所代表的词法单元。当我们找到一个标识符时，如果该标识符尚未出现在符号表中，就会将此标识符放入符号表中，并返回一个指针，指向这个刚找到的词素所

对应的符号表条目。当然，任何在词法分析时不在符号表中的标识符都不可能是一个关键字，因为如果是关键字一定在符号表中，因此它是一个表示标识符的语法单元。

（2）为每个关键字建立单独的状态转换图。图 4-6 是关键字 else 的一个例子，在 else 的状态装换图中，当出现关键字 else 的后续字母的时候，就进入一个新的状态，并且要求字母 e 之后不能再有其他的字母或者数字，如果再有其他的字母或数字的话，就会变成标识符而不是关键字。例如，假设我们遇到了 elsevier，他是以 else 为前缀的，如果我们返回 else，就会得到一个错误的结果。此时，为了避免出现这种情况，我们需要给词法单元设定优先级顺序，使得在同时遇到关键字和标识符的时候，优先识别关键字，而不是标识符。

在代码翻译系统中，我们使用的是方法 1，并没有使用方法 2。

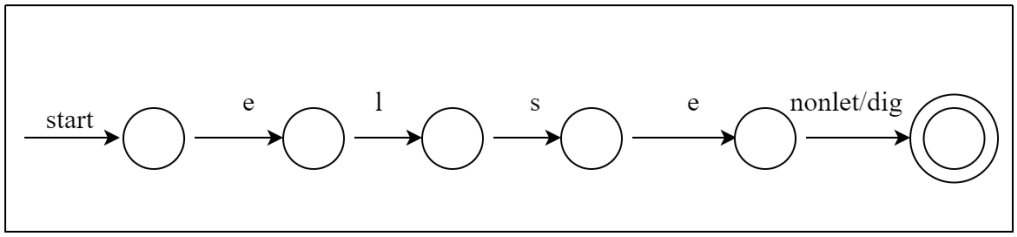

图 4-6　关键字 else 的状态转换图

图 4-7 是空白符的状态转换图，用于识别空白符。在该图中，我们寻找一个或多个空白字符，在图中用 delim 表示。典型的空白字符有空格、制表符、换行符，还有可能包括由于语言特性而导致的不可能出现在任何词法单元中的字符。假设我们在状态 2 中找到了一个连续的空白字符组成的块，且后面还跟随一个非空白字符。我们将输入回退到这个非空白符的开头，但是并不向语法分析器返回任何词法单元，只是在在这个空白符之后再次启动词法分析过程。

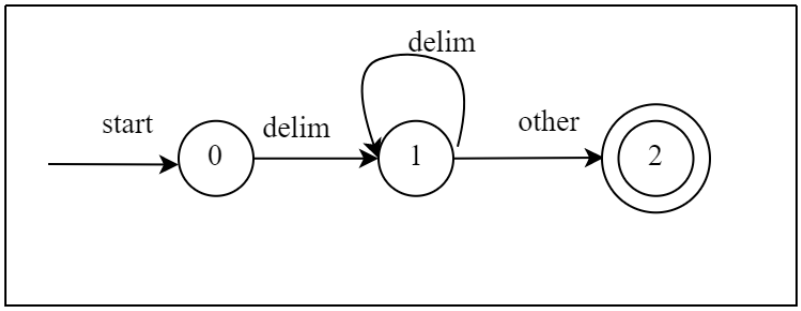

图 4-7　空白符的状态转换图

图 4-8 是数字的状态转换图。他是比较复杂的状态转换图，从状态 0 开始，如果我们看到一个数字，就转入状态 1。在该状态，我们可以读入任意数量的其他数位。接着，如果我们看到了一个不是数位，不是小数点，也不是 E 的其他字符，此时说明得到了一个整数形式的数字，比如 123。此时进入状态 8 进行处理，我们在该状态返回词法单元 number 以及一个指向常量表条目的指针，刚刚获得的词素便放在这个常量表条目中。如果我们在状态 1 看到一个小数点，那么接下来会看到可选的小数部分。于是，进入到状态 2，并寻找一个或

多个更多的数位，状态 3 就被用于此目的。如果我们看到一个 E，那么接下来就是可选的指数部分，指数部分的识别任务由状态 4 7 完成。我们在状态 3 看到的是不同于 E 和数位的其他字符，那么我们就到达了小数部分的末尾，这个数字没有指数部分，我们通过状态 9 返回刚刚找到的词素。

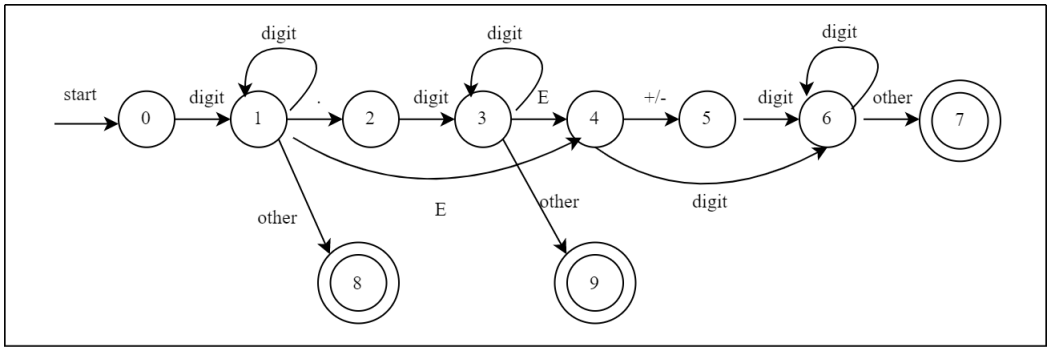

图 4-8　数字的状态转换图

词法分析器的输出

词法单元作为词法分析器的输出，一般表示成如下的二元形式：

< 单词种别，单词符号的属性值 >

在实现的过程中，单词种别一般使用整数编码。标识符一般统归为一种，常量一般按照种类进行分类，比如分为整数类别，实数类别，布尔类别，字符串类别等。运算符采用一个运算符就是一种的实现方法，界符也是采用一个符号作为一个种别的方法。对于每个单词符号，除了要说明该单词属于何种类别，该单词符号的其他属性信息也要给出，单词符号的属性包括单词符号的一些特性和一些特征。

举一个简单的例子，对于图 4-9 所示的一个简单循环，经过词法分析以后，会被转换为如表 4-7 所示的词法单元序列。

```
while(i>=j)
{
    i--;
}
```

图 4-9　循环示例

表 4-7 词法单元序列示例

< while,_ >
< (,_ >
< id, 指向 i 的符号表项的指针 >
< >=,_ >
< j, 指向 j 的符号表项的指针 >
<),_ >
< {,_ >
< i, 指向 i 的符号表项的指针 >
< –,_ >
< },_ >

变量名、函数名等都是在符号表中进行存储。符号表是一种数据结构，对源程序中变量、函数等的信息进行存储。符号表是在对源程序进行分析的过程中逐渐形成的，信息会被逐渐收集放入符号表，并且在以后的阶段中会经常用到符号表。源代码中标识符相关的信息会被存储在符号表的条目中，比如标识符的词素、标识符的类型、标识符的位置以及其他的相关信息。

词法分析器在进行词法分析的过程中对标识符进行扫描，扫描以后在符号表中进行查找，如果已经存在于符号表中，返回它的唯一标识符，否则，将标识符添加到符号表中。

可以用结构体来存储标识符的相关信息。如图 4-10 所示：

```
struct identifier
{
    int token;
    int hash;
    string name;
    int kind;
    int type;
    int value;
    int Bclass;
    int Btype;
    int Bvalue;
}
```

图 4-10 标识符的相关信息

下面对结构体中变量的含义进行解释：

1. token：表示标识符的类型，所有标识符的类型都是 Id，但是为了强调和关键字的区别，还是使用了这个变量。

2. hash：标识符经过哈希计算得到的值，可以快速地对标识符进行比较。

3. name：存放标识符的字符序列的变量。

4. kind：表示标识符类别的变量，比如局部变量、全局变量。

5. type：存储标识符的类型，比如字符串型的变量、整型的变量、浮点型的变量等。

6. value：将标识符的值存放于该变量中，例如整型的数值，字符串型的字符串。

7. BXXXX：标识符的作用域是不同的，有全局作用域和局部作用域，当两个作用域中有相同的标识符时，该变量用来保存全局标识符。

4.3.2　语法分析器的实现

在词法分析完成后，会向语法分析器传递词法单元流，语法分析器会根据这个词法单元流来判断源程序中的代码是否是符合规范的。在进行语法分析时，会根据源程序语言的语法进行推导，来判断词法分析得到的词法单元是否可以适配到语法模型中。如果源程序的代码是有效的，那么经过语法分析后就会得到表示源代码语法结构的抽象语法树，该数据结构在后续的阶段中会继续使用。如果源代码是错误的、不符合语法规范的，那么语法分析器会停止分析并报告错误。

一般来讲，计算机可以识别的程序是必须正确且不具有二义性的，这就要求该程序设计语言的语法结构必须由一组精确的规则来描述。比如，在程序设计语言中，函数如何定义、变量如何定义，这些都要遵循程序语言的语法规则。我们可以使用文法来描述这些规则，文法为设计程序语言、设计编译器以及设计代码翻译系统都提供了很大便利。

文法在语法分析阶段具有重要的作用。

（1）文法可以规范的指出在进行程序语言设计时应该遵循的语法。

（2）对于文法中的某些类型，我们可以借助自动化的工具生成高效的语法分析器，该语法分析器和手动编码的一样，都可以对源程序的语法结构进行分析。通过文法，还可以发现一些程序语言设计上可能存在的问题。

（3）代码的结构应该符合文法给出的规则，借助于文法，可以进行源程序的翻译，也可以对源代码进行错误检测。

（4）良好的文法可以增强程序设计语言的可扩展性，可以在更新迭代的过程中方便的增加新的特性，从而可以适用于更多的场景。

在编译器中，文法是用来对语言进行设计。但是在代码翻译系统中，Matlab 代码的语法结构是已知的，我们只是归纳出已经存在的文法，便于理清代码的语法结构，以进行代码翻译。

Matlab 的顶层文法如图 4-11 所示。是对 Matlab 的顶层结构进行一个设计，一个程序的源代码文件被定义为 Root，以 Root 开始，解析源代码文件，把所有的代码看作一个代码块，代码块又是由很多子块组成，子块可以分为函数定义、类定义、Switch 语句、While 语句、If 语句、For 语句、TryCatch 语句和表达式语句等。下面会对其中的每一子块进行详细的说明，说明每一部分的文法结构，以便对其进行解析。在以下的解析中，为了表示出各个部分的文法以及他们的数据数据结构，使用键值对的形式来描述他们。

```
Program -> Root
Root -> File
File -> BlockStatement

BlockStatement -> StatementList
Statement -> FunctionDeclaration | ClassDeclaration |
             SwitchStatement | WhileStatement |
             IfStatement | ForStatement |
             TryCatchStatement | ExpressionStatement
```

图 4-11 Matlab 的顶层文法

ClassDeclaration 节点的文法以及数据结构如图 4-12 所示，包括定义类的关键字、属性、类名称、基类、还有在类中定义的方法、在类中定义的属性、在类中定义的事件、在类中定义的枚举类、以及 End 关键字。

```
{
    ClassDeclarationSyntaxNode : StatementSyntaxNode
    {
        ClassdefKeyword : Token;
        Attributes : AttributeListSyntaxNode;
        ClassName : Token;
        BaseClassList : BaseClassListSyntaxNode;
        Nodes : SyntaxList;
        {
            methods : MethodsListSyntaxNode;
            properties : PropertiesListSyntaxNode;
            events : EventsListSyntaxNode;
            enumeration : EnumerationListSyntaxNode;
        }
        EndKeyword : Token;
    }
}
```

图 4-12 ClassDeclaration 节点的文法以及数据结构

FunctionDeclaration 节点的文法及数据结构如图 4-13 所示，包括 function 关键字、函数输入、可选的逗号、函数名、函数输出、函数体以及 End 关键字。

```
{
    FunctionDeclarationSyntaxNode : StatementSyntaxNode
    {
        FunctionKeyword : Token;
        OutputDescription : FunctionOutputDescriptionSyntaxNode;
        Name : Token;
        InputDescription : FunctionInputDescriptionSyntaxNode;
        Commas : SyntaxList;
        Body : BlockStatementSyntaxNode;
        EndKeyword : EndKeywordSyntaxNode;
    }
}
```

图 4-13　FunctionDeclaration 节点的文法及数据结构

SwitchStatement 节点的文法和数据结构如图 4-14 所示，主要包括 Switch 关键字、Switch 表达式、case 以及 End 关键字。case 又包括 case 关键字、case 表达式和要执行语句。

```
{
    SwitchStatementSyntaxNode : StatementSyntaxNode
    {
        SwitchKeyword : Token;
        SwitchExpression : ExpressionSyntaxNode;
        OptionalCommas : SyntaxList;
        Cases : SyntaxList;
        {
            CaseKeyword : Token;
            CaseIdentifier : ExpressionSyntaxNode;
            OptionalCommas : SyntaxList;
            Body : BlockStatementSyntaxNode;
        }
        EndKeyword : Token;
    }
}
```

图 4-14　SwitchStatement 节点的文法和数据结构

WhileStatement 节点的文法和数据结构如图 4-15 所示，主要包括 While 关键字、循环判断条件、循环体以及结束 While 语句的 End 关键字。

```
{
    WhileStatementSyntaxNode : StatementSyntaxNode
    {
        WhileKeyword : Token;
        Condition : ExpressionSyntaxNode;
        OptionalCommas : SyntaxList;
        Body : BlockStatementSyntaxNode;
        EndKeyword : Token;
    }
}
```

图 4-15 WhileStatement 节点的文法和数据结构

IfStatement 节点的文法和数据结构如图 4-16 所示，if 语句主要可以分为四部分：if 关键字、判断条件、执行代码块和 End 关键字。执行代码块又包括 elseif 代码块和 else 块，elseif 代码块包括 elseif 关键字、判断条件和要执行的代码。else 包括执行代码块。

```
{
    IfStatementSyntaxNode : StatementSyntaxNode
    {
        IfKeyword : Token;
        Condition : ExpressionSyntaxNode;
        OptionalCommas : SyntaxList;
        Body : BlockStatementSyntaxNode;
        ElseifClauses : SyntaxList;
        {
            ElseifKeyword : Token;
            Condition : ExpressionSyntaxNode;
            OptionalCommas : SyntaxList;
            Body : BlockStatementSyntaxNode;
        }
        ElseClause : ElseClauseSyntaxNode;
        {
            ElseKeyword : Token;
            Body : BlockStatementSyntaxNode;
        }
        EndKeyword : Token;
    }
}
```

图 4-16 IfStatement 节点的文法和数据结构

ForStatement 节点的文法和数据结构如图 4-17 所示，包含 for 关键字、变量初始化语句、循环体以及 end 关键字。

```
{
    IfStatementSyntaxNode : StatementSyntaxNode
    {
        ForKeyword : Token;
        Assignment : AssignmentExpressionSyntaxNode;
        OptionalCommas : SyntaxList;
        Body : BlockStatementSyntaxNode;
        EndKeyword : Token;
    }
}
```

图 4-17　ForStatement 节点的文法和数据结构

TryCatchStatement 节点的文法和数据结构如图 4-18 所示，包含 try 关键字、try 代码块、catch 关键字、catch 代码块以及 end 关键字。

```
{
    IfStatementSyntaxNode : StatementSyntaxNode
    {
        TryKeyword : Token;
        TryBody : BlockStatementSyntaxNode;
        CatchClause : CatchClauseSyntaxNode;
        {
            CatchKeyword : Token;
            CatchBody : SyntaxList;
        }
        EndKeyword : Token;
    }
}
```

图 4-18　TryCatchStatement 节点的文法和数据结构

Expression 节点的文法和数据结构比较复杂，如下所示, 可以是一元表达式，即一个可选的操作符加上一个表达式比如+e，可以是 at 运算，也可以是二元表达式，比如 e+e，不管是一元表达式还是二元表达式里的表达式可以递归的去解析成一元或二元表达式。如图 4-19 所示。

```
{
    ExpressionStatementSyntaxNode : StatementSyntaxNode
    {
        ExpressionSyntaxNode : SyntaxNode;
        {
            {
                operator : Token;
                ExpressionSyntaxNode : SyntaxNode;
            }
            | // or
            {
                atSign : Token;
                FunctionHandleExpressionSyntaxNode :
                ExpressionSyntaxNode;
            }
            | // or
            {
                ExpressionSyntaxNode : SyntaxNode;
                operator : Token;
                ExpressionSyntaxNode : SyntaxNode;
            }
        }
    }
}
```

图 4-19 Expression 节点的文法和数据结构

以上对 Matlab 语言的文法进行了归纳和总结，根据文法，我们可以进行语法分析，推导出和源代码对应的抽象语法树。我们采用自顶向下语法分析为输入串构造语法分析树，它从语法分析树的根节点开始，然后依次按照先根次序构建语法分析树的其他节点，自顶向下进行语法分析的过程也是最左推导的过程。

更具体的说，我们使用的是自顶向下的递归下降语法分析器，无回溯语法有助于实现简单高效的语法分析过程。递归下降的语法分析器，在进行语法分析时递归调用分析函数，语法中的每个非终结符号都对应于一个分析函数。假如遇到了非终结符 A，那么就调用 A 的分析函数，就可以得到 A 的一个实例。假如 A 的产生式右侧还有非终结符 B，递归调用 B 的分析函数，直到没有非终结符。

以 FunctionDeclaration 节点为例阐述递归下降语法分析器进行语法分析的过程，其他节点是类似的。如图 4-20 所示，首先语法分析器启动，设立根节点，对源代码文件进行分析，把所有的代码看作一个 BlockStatement，该节点的下一层包括图上所示的 8 个节点，假设源代码文件里有函数定义代码，对该代码块进行递归调用分析，在分析过程中遇到终结符的

话（就是图中阴影部分节点），递归调用终止，假设节点是非终结符的话，继续递归调用，直到遇到终结符。当所有代码分析完毕且没有报错，语法分析的工作就完成了。

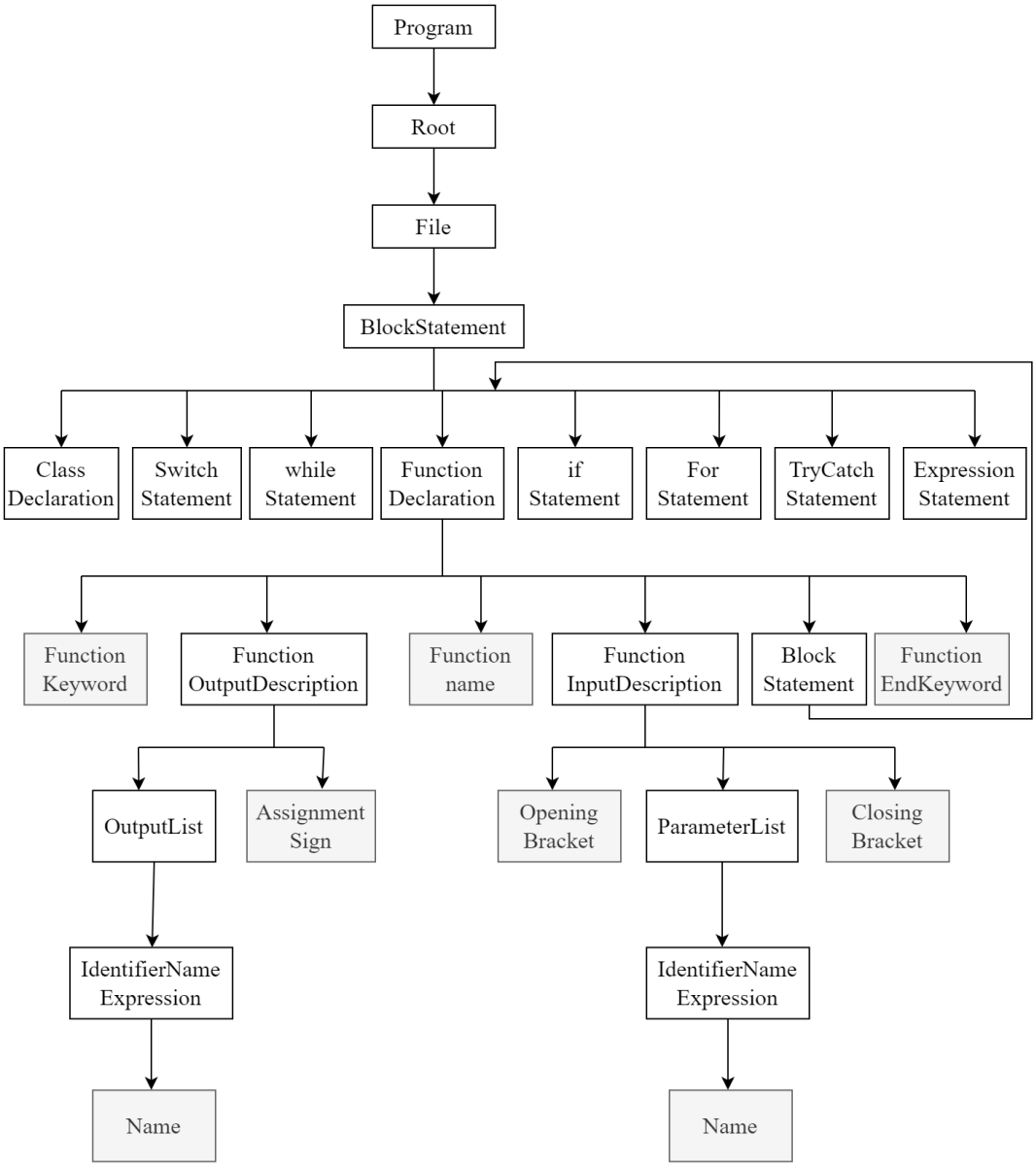

图 4-20　递归下降语法分析过程示意图

递归下降语法分析器的顶层代码的关键部分如图 4-21 所示，先判断到来的是哪一种非终结符，然后根据相应的终结符进行相应的处理，然后递归的处理推导公式右边的非终结符，直至处理完毕。

```
switch (mp[TokenList[currentIndex].Text])
{
    case 1:
        FunctionDeclarationSyntaxNode*
            functionDeclarationSyntaxNode =
            ParseFunctionDeclaration();
        return functionDeclarationSyntaxNode;
    case 2:
        ClassDeclarationSyntaxNode* classDeclarationSyntaxNode =
            ParseClassDeclaration();
        return classDeclarationSyntaxNode;
    case 3:
        SwitchStatementSyntaxNode* switchStatementSyntaxNode =
            ParseSwitchStatement();
        return switchStatementSyntaxNode;
    case 4:
        WhileStatementSyntaxNode* whileStatementSyntaxNode =
            ParseWhileStatement();
        return whileStatementSyntaxNode;
    case 5:
        IfStatementSyntaxNode* ifStatementSyntaxNode =
            ParseIfStatement();
        return ifStatementSyntaxNode;
    case 6:
        ForStatementSyntaxNode* forStatementSyntaxNode =
            ParseForStatement();
        return forStatementSyntaxNode;
    case 7:
        TryCatchStatementSyntaxNode* tryCatchStatementSyntaxNode
            = ParseTryCatchStatement();
        return tryCatchStatementSyntaxNode;
    default:
        ExpressionStatementSyntaxNode*
            expressionStatementSyntaxNode =
            ParseExpressionStatement();
        return expressionStatementSyntaxNode;
}
```

图 4-21 递归下降语法分析器顶层代码设计

4.3.3　代码生成器的实现

在语法分析阶段，输入的是词法单元流，输出的是表示代码语法结构的抽象语法树。代码生成器主要完成两部分工作，一是接收语法分析器生成的抽象语法树并对其进行遍历，在此过程中对节点进行添加、更新及移除等操作，这是代码生成器的第一个阶段。二是代码生成阶段，根据抽象语法树生成目标代码，原理相对明晰，对整个抽象语法树进行深度优先遍历，在遍历的过程中生成表示转换后代码的字符串。

想要转换抽象语法树需要先进行递归的树形遍历。以 FunctionDeclaration 节点类型为例来说明遍历的过程，如图 4-22 所示，是一个树形结构，它有几个节点，FunctionKeyword, FunctionOutputDescription, FunctionName, FunctionInputDescription, BlockStatement, FunctionEndKeyword, 每一个都有一些内嵌节点。于是我们从 FunctionDeclaration 节点开始，并且我们知道它的内部属性，所以我们依次访问每一个属性及他们的子节点。接着我们来到 FunctionKeyword，他是一个标识符，是终结符，终结符没有任何子节点属性，我们继续进行其他节点的遍历。紧接着是 FunctionOutputDescription 节点，它是非终结符节点，继续遍历它的子节点，函数返回的是一个数组，我们访问其中的每一个，都是标识符类型的单一节点，访问完以后访问 FunctionOutputDescription 的第二个子节点，以此类推，遍历完 FunctionDeclaration 的其他子节点。

在代码翻译过程中，所有节点的遍历都类似于 FunctionDeclaration 节点。

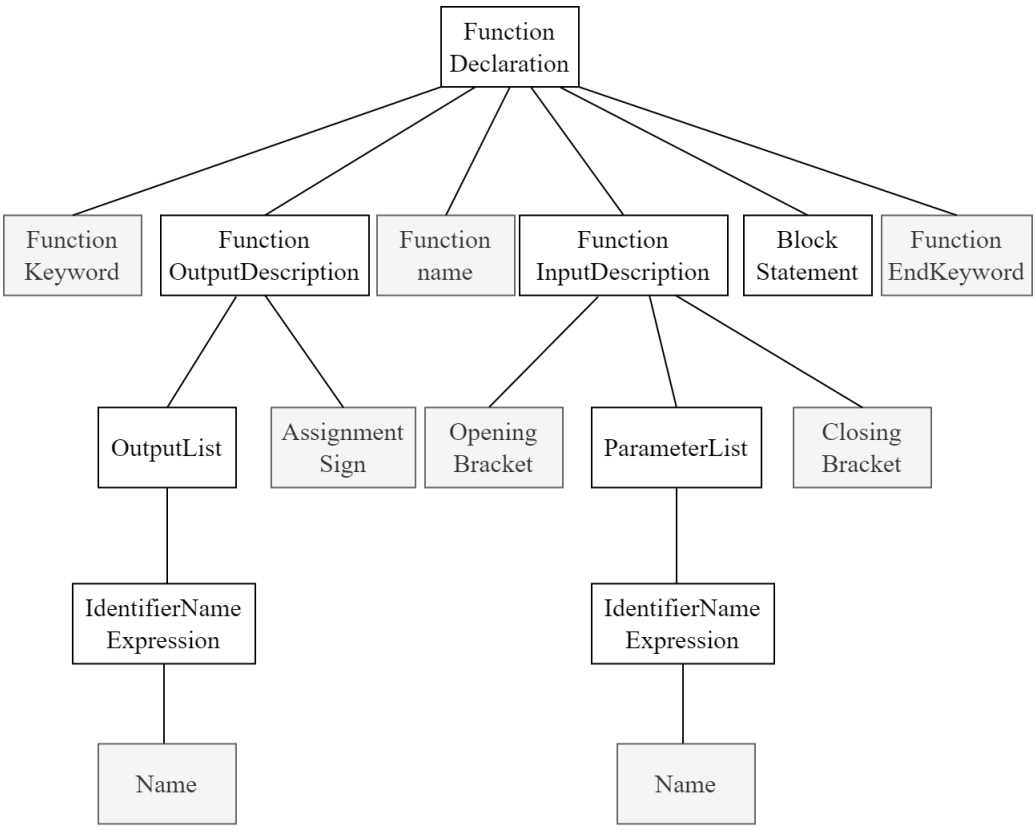

图 4-22　遍历 FunctionDeclaration 类型节点

当我们在对抽象语法树进行遍历的时候就会用到第二章提到的访问者模式，对抽象语法树进行遍历的过程就是“访问”抽象语法树的过程。访问者就是一个对象，定义了用于在

一个树状结构中获取具体节点的方法，还是以 FunctionDeclaration 节点为例来阐述访问者是如何进行抽象语法树的遍历的。

首先定义一个抽象访问者类，它定义了对每个元素访问的行为，它的参数就是被访问的元素，它的方法个数理论上与元素的个数是一样的。如图 4-23 所示。

```
class Visitor
{
    virtual void Visit(SyntaxNode node)
    {

    }
    virtual void VisitFunctionDeclaration(
    FunctionDeclarationSyntaxNode node)
    {

    }
}
```

图 4-23 抽象访问者类

接着定义一个具体访问者类，它需要给出对每一个元素类访问时所产生的具体行为。如图 4-24 所示。

```
class ConcreteVisitor
{
    void Visit(SyntaxNode node)
    {
        node.Accept(this);
    }
    void VisitFunctionDeclaration(
    FunctionDeclarationSyntaxNode node)
    {
        Visit(node.FunctionKeyword)
        Visit(node.OutputDescription);
        ...
        // dosomething
    }
}
```

图 4-24 具体访问者类

同时，要在抽象元素中定义一个 Accept 方法。如图 4-25 所示。

```
class SyntaxNode
{
    ...
    virtual void Accept(Visitor visitor)
    {

    }
    ...
}
```

图 4-25　抽象元素类

并且，在具体元素中实现这个在抽象元素中定义的 Accept 方法。如图 4-26 所示。

```
class FunctionDeclarationSyntaxNode
{
    ...
    void Accept(Visitor visitor)
    {
        visitor.VisitFunctionDeclaration(this);
    }
    ...
}
```

图 4-26　具体元素类

访问者模式的运行过程非常巧妙，是一个调用再调用的过程。比如调用 Visit(Function-Declaration) 时，会调用 FunctionDeclaration 类中的 Accept() 方法，Accept 方法会回调 Visit-FunctionDeclaration() 方法来实现具体的细节，在 VisitFunctionDeclaration() 中又会调用访问其他节点的 Visit() 方法，循环调用再调用的过程，直到遍历完毕。这样在实现继承的同时，实现了多态，使用一个 Visit() 方法来调用所有的具体实现的方法。

完成对抽象语法树的转换以后，中序遍历抽象语法树进行代码生成。比如对于如图 4-28 所示的抽象语法树，按照图中标注的遍历顺序，可以生成如图 4-27 所示的代码。

```
int addOne(int x)
{
    x = x + 1;
    return x;
}
```

图 4-27　addOne 函数

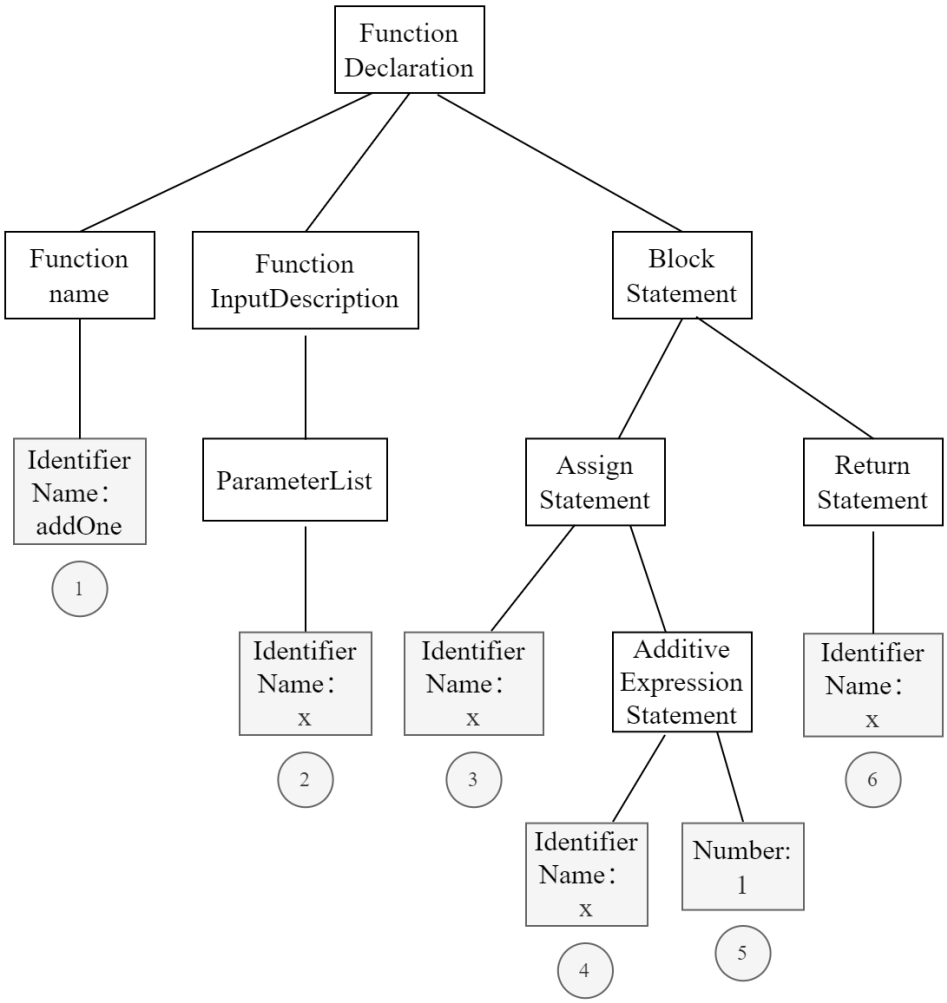

图 4-28 中序遍历抽象语法树

4.4 后处理子系统的实现

代码生成器阶段对抽象语法树进行了转换以及根据抽象语法树进行了代码生成，但是对抽象语法树的修改过程只是对已经存在的枝叶进行修改，有一部分保证 C++代码成功运行的代码并不在抽象语法树中，所以遍历抽象语法树生成代码时并不会生成这部分代码，此时就需要后处理子系统来完成这样一部分工作。

后处理子系统主要完成三部分工作，对一些函数进行修改，对一些数据类型进行检查及修改，添加 C++代码成功运行所需的头文件，对应于图 4-29 所示的三个方法。

```
1 void Postprocessing::processCppFile()
2 {
3     postModifyFunction();
4     addFirstBlock();
5     postModifyDataType();
6 }
```

图 4-29 后处理子系统的关键函数

添加头文件对应的关键代码如图 4-30 所示。

```
void Postprocessing::addFirstBlock()
{
    string cppText = "";
    ifstream in("../IntermediateCode/4_generated.cpp");
    if(!in.is_open())
    {
        cout << "error open .cpp file" << endl;
        exit(0);
    }
    string str;
    while (!in.eof())
    {
        getline(in,str);
        if (in.eof())
        {
            break;
        }
        cppText.append(str + "\n");
    }
    in.close();
    string firstBlock = "#include <ACSC.h>\n";
    firstBlock = firstBlock + "#include <iostream>\n";
    firstBlock = firstBlock + "#include <cstdio>\n";
    firstBlock = firstBlock + "using namespace std;\n";
    firstBlock = firstBlock + "int main()\n";
    cppText = firstBlock + cppText;
    ofstream out("../IntermediateCode/4_generated.cpp");
    out << cppText;
    out.close();
}
```

图 4-30　添加头文件的关键代码

4.5　本章小结

本章对第四章的详细实现过程进行了详细的叙述，对代码翻译系统的四个子系统的具体的实现原理、实现算法和实现过程进行了阐述，并且给出了关键代码，有助于更清晰的理解代码翻译系统。

第 5 章　系统测试与实验评估

5.1　系统的评估标准

从软件工程的角度来说，任何一个软件系统在设计完成以后，都要对系统进行软件测试。代码翻译系统也需要进行软件测试，我们从以下几个方面来对代码翻译系统进行测试和评价。

5.1.1　系统的正确性

系统正确性是指一个系统在使用规范下被认定为正确的判定，例如，系统对于每一个输入，都可以得到正确的输出。代码转换在编译、代码优化、程序分析、软件更新以及逆向工程方面都有着广泛的用途。保证翻译前后的代码在功能上的等价性是代码翻译系统最基本的前提，如果不能满足这种功能等价性，翻译前后的代码对于相同的输入不能得到一样的输出，在大多数情况下这不是我们想要的。如果不能保证功能上的正确性，通常意味着错误的结果，或者需要对其进行适当的调整或修改以保证正确性。因此，作为一种代码翻译技术，系统的正确性是我们追求的第一目标。

5.1.2　系统的自动化程度

系统自动化程度是指系统在无人操作的情况下完成任务的能力，具体来讲，代码翻译系统的自动化程度是指整个代码翻译过程的人的参与的程度，人的参与程度越低，代码的自动化程度越高。最理想的情况就是，人只作为一个发出开始指令的指挥官，而不需要再去做其他的任何事情就可以得到目标代码。代码翻译系统的自动化的程度越高，维护系统的人员的负担越小。与之相反，系统中不能自动转换的比例越高，意味着转换后投入的人工将会越多。

5.1.3　目标代码的规范性

做好代码规范在代码编写中是非常重要的，代码的可读性和可维护性是规范的代码中不可或缺的，代码翻译系统的目标之一就是翻译得到的代码具有较好的规范性。代码的可读性是指人们在阅读代码、理解代码和调试代码时的难易程度，可读性越高，越容易理解和调试。提高代码的可读性可以提高软件工程师的开发效率，避免在代码阅读上时浪费过多无畏的时间。代码可读性主要包括代码风格的一致性、代码表达意图的清晰性、恰到好处的注释等等。代码的可维护性是指对软件进行维护的难易程度，通常，可理解性、可测试行和可修改性是影响代码可维护性的重要因素。代码翻译系统在进行代码转换时使用的是同一转换标准，因此生成的这些代码具有完全一致的风格，比如缩进的使用、在函数的命名等方面更一致，每个翻译出来的代码块都有着一致的组织结构，维护人员只需简单了解就可以熟悉代码翻译系统中的代码风格，可以提高效率。

5.1.4　源代码的膨胀率

源代码的膨胀率指的是转换后目标程序的代码行数和转换前源程序代码行数的一个比率[70]。代码行数可以在大多数编辑器中直接获得，记录下来即可。一般来说，维护代码的成本和代码的长度成正相关，代码越长，维护成本越高。同时，代码膨胀还可能导致生成的程序文件过大、运行缓慢或者有其他浪费资源的情况。因此，代码翻译系统在进行代码翻译时要控制源代码的膨胀率，尽量减少因为代码翻译而带来的行数的增加，以使翻译得到的目标代码更轻量、更可维护，同时减少翻译得到的代码在运行时的资源占用。

5.1.5　目标代码的性能

目标代码的性能指的是转换后的代码经本地编译后运行的时间，时间越短，性能越高。代码翻译系统的目标之一就是提高程序的性能，缩短程序的运行时间，所以我们希望目标代码的性能高于源代码的性能，尽可能减少资源消耗和时间浪费，提高关键性能。在文件读写、运动控制等方面更高效、更精确。

在代码翻译系统的实现过程中，会来权衡以上五个标准，使每一个都能达到较好的状态。但是也要综合考虑，必要时牺牲某个方面的要求来满足更为迫切的其他方面的要求。

5.2　系统的测试用例

本节主要对系统的使用过程和代码翻译的结果进行介绍，并且对翻译前后的代码进行了性能测试与比较。

5.2.1　测试环境

在测试之前，首先对测试系统的配置信息进行介绍。在进行测试时，使用了比较常见的系统配置，这样能更贴近工程师工作环境中的系统配置。

系统的各个模块均采用 C++实现，在 Windows 操作系统上进行测试，测试系统的具体配置信息如表 5-1 所示。

表 5-1　系统配置信息

配置项	配置值
操作系统	Windows 10 专业版
CPU	Intel(R) Core(TM) i7-9700F CPU @ 3.00GHz
内存	16G
gcc 版本	11.2.1
Qt 版本	6.3.1
Matlab 版本	R2022a
SPiiPlus User Mode Driver	7.0.0.0

5.2.2 测试用例

首先，对系统的主界面和使用方法进行简单介绍。由于系统的功能相对单一，系统的界面和使用方法都比较简单。系统的主界面如图 5-1 所示，系统的使用方法也很简单，可以分为三步。一是上传源代码文件或者直接在左边的输入框里输入源代码。二是点击中间的翻译按钮就会进行代码翻译，翻译得到的结果会在右边的框里展示。三是对结果进行保存，点击保存按钮就可以保存到文件中。

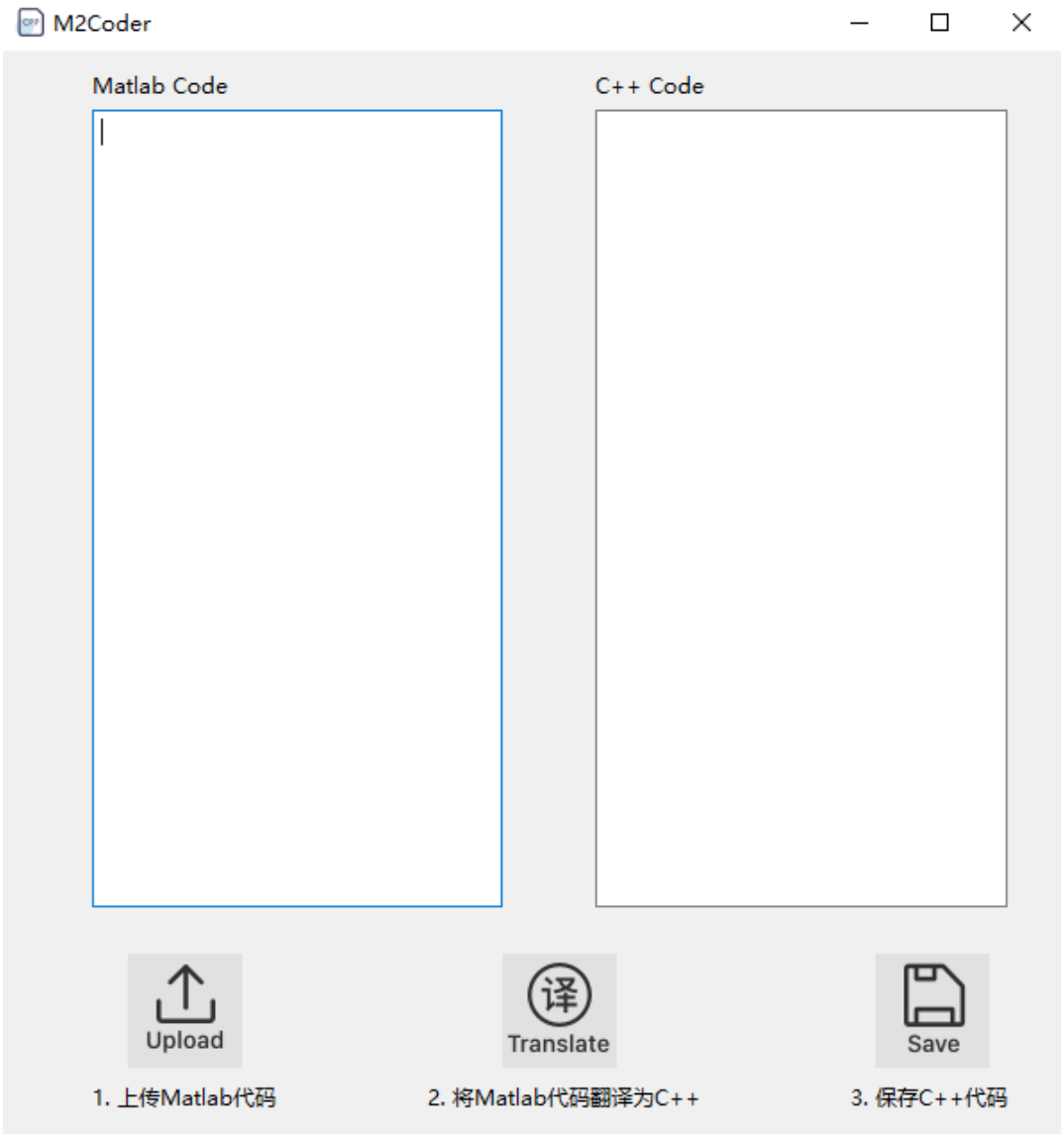

图 5-1 系统主界面

本文设计代码翻译系统的初衷就是对使用 Matlab 编码的 ACS 代码进行翻译，将其转化为 C++代码，所以我们首先对和 ACS 控制器相关的代码进行了翻译。

对 ACS 控制器进行读写的 Matlab 和 C++代码如下所示，这是一个在 ACS 控制器上进行编码的一个经典例子，首先和 ACS 模拟器建立连接，并且对 ACS 控制器数组内的数据进行读写，最终关闭和 ACS 模拟器的连接，这是一个编码控制 ACS 控制器的一般流程。代码都是可以直接运行的，同时，尝试使用 Matlab Coder 进行代码翻译，但是 Matlab Coder 只可

以翻译函数而不可以翻译一般脚本。

```
// 对ACS控制器进行读写的Matlab代码
channel = SPiiPlusMatlab;
channel.OpenCommSimulator;
AnalogOutputs = [10, 20, 30, 40];
fromIndex = 0;
toIndex = 3;
channel.WriteVariable(AnalogOutputs, 'AOUT',
ProgramBuffer.ACSC_NONE, fromIndex, toIndex);
AnalogInputs = channel.ReadVariable('AOUT',
ProgramBuffer.ACSC_NONE, fromIndex, toIndex);
channel.CloseComm

// 与Matlab代码相对应的C++代码
#include <ACSC.h>
#include <iostream>
using namespace std;
int main()
{
        auto Handle=acsc_OpenCommSimulator();
        auto AnalogOutputs=[10,20,30,40];
        auto fromIndex=0;
        auto toIndex=3;
        acsc_WriteInteger(Handle,ACSC_NONE,"AOUT",fromIndex,
        toIndex,ACSC_NONE,ACSC_NONE,AnalogOutputs,NULL);
        int AnalogInputs[toIndex-fromIndex + 1];
        acsc_ReadInteger(Handle,ACSC_NONE,"AOUT",fromIndex,
        toIndex,ACSC_NONE,ACSC_NONE,AnalogInputs,NULL);
acsc_CloseComm();
}
```

和 ACS 控制器相关的工程实践中经常需要用到对文件的读写，本文特意对文件的读写相关的代码进行翻译。和文件读写相关的函数并不是严格对应，需要先预处理。对文件读和文件写相关的代码进行翻译的例子如下。

```
// 写文件的Matlab代码
fileID = fopen('nums1.txt','w');
for i = 1 : 10000
    fprintf(fileID,'%d\n',i);
end
fclose(fileID);
```

```cpp
// 与写文件的Matlab代码对应的C++代码
#include <iostream>
#include <cstdio>
using namespace std;
int main()
{
    FILE *fileID;
    fileID = fopen("nums1acsc_txt", "w");
    for (auto i = 1; i <= 10000; i++)
    {
        fprintf(fileID, "%d\n", i);
    }
    fclose(fileID);
}
```

```matlab
// 部分读文件的Matlab代码
fileID = fopen('nums1.txt','r');
A = fscanf(fileID,'%d');
fclose(fileID);
```

```cpp
// 与读文件的MATLAB代码对应的C++代码
#include <iostream>
#include <cstdio>
using namespace std;
int main()
{
    FILE *fileID;
    fileID = fopen("nums1acsc_txt", "r");
    int *A = new int[100000000];
    int arrayIndex = 1;
    while (fscanf(fileID, "%d", &A[arrayIndex]) > 0)
    {
        arrayIndex++;
    }
    fclose(fileID);
}
```

对以上例子中的 Matlab 代码和 C++代码进行性能上的对比，在同一台机器上分别测试二者读写文件的时间。首先测试二者对文件进行写入时的性能对比，此处 Matlab 采用逐个向文件中写入整数的方式，C++采用同样的方式。以秒为单位，在 10000、100000、1000000、

10000000 四个数量级上对写入的时间进行统计，可以看到 C++的运行时间明显少于 Matlab 的运行时间。同样的，对二者的读文件的性能进行对比，MATLAB 将读到的数据直接保存在数组中，而 C/C++采取和写文件同样的策略，一个一个的读取然后保存到数组中。可以看到，C/C++的运行时间还是少于 MATLAB 的运行时间。最终结果如表 5-1，5-2 所示。

表 5-2　文件写入时间

数据大小	Matlab 的运行时间	C++的运行时间
10000	0.093618	0.001000
100000	0.965655	0.019000
1000000	9.378792	0.225000
10000000	92.049533	2.437000

表 5-3　文件读取时间

数据大小	Matlab 的运行时间	C++的运行时间
10000	0.006735	0.004000
100000	0.061480	0.048000
1000000	0.650230	0.509000
10000000	6.353722	5.350000

ACS 运动控制器经常被用在对一个运动过程进行控制，本文也对多个运动过程的代码进行了翻译。例如，模拟了一个在给定坐标的正方形上从零点开始由外向内进行顺时针运动的过程，并且在运动过程中对点的坐标进行记录以模拟对其进行的一些操作。运动过程示意图如图 5-2 所示。

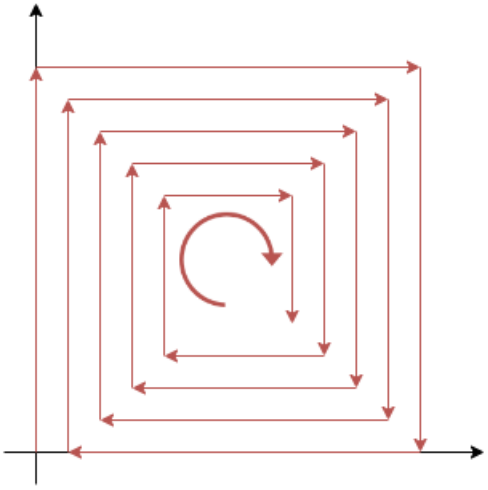

图 5-2　顺时针运动过程示意图

使用 Matlab 对上图所示的顺时针运动过程进行代码编写，从 (0,0) 开始，到遍历完正方形内的所有点结束，代码如下所示。

```
function traverseSquare()
    fileID = fopen('record.txt','w');
    length = 100;
    left = 0;
    top = length;
    right = length;
    bottom = 0;
    i = 0; % 行
    j = 0; % 列
    while left < right && bottom < top
        while i <= top
            record(fileID, i, j);
            i = i + 1;
        end
        i = i - 1;
        left = left + 1;
        while j <= right
            j = j + 1;
            record(fileID, i, j);
        end
        j = j - 1;
        top = top - 1;
        while i >= bottom
            i = i - 1;
            record(fileID, i, j);
        end
        i = i + 1;
        right = right - 1;
        while j >= left
            j = j - 1;
            record(fileID, i, j);
        end
        j = j + 1;
        bottom = bottom + 1;
    end
    fclose(fileID);
end
```

```
function record(fileID, x, y)
    fprintf(fileID,'(%f,',x);
    fprintf(fileID,'%f)',y);
end
```

将控制顺时针运动的 Matlab 代码进行翻译，得到的与之对应的控制顺时针运动的 C++ 代码如下所示。

```
#include <iostream>
#include <cstdio>
using namespace std;

void record(FILE* fileID, int x, int y)
{
    fprintf(fileID,"(%d,",x);
    fprintf(fileID,"%d)",y);
}

void traverseSquare()
{
    FILE *fileID;
    fileID = fopen("record.txt","w");
    auto length = 100;
    auto left = 0;
    auto top = length;
    auto right = length;
    auto bottom = 0;
    auto i = 0; // 行
    auto j = 0; // 列
    while (left < right && bottom < top)
    {
        while (i <= top)
        {
            record(fileID, i, j);
            i = i + 1;
        }
        i = i - 1;
        left = left + 1;
        while (j <= right)
        {
```

```
            j = j + 1;
            record(fileID, i, j);
        }
        j = j - 1;
        top = top - 1;
        while (i >= bottom)
        {
            i = i - 1;
            record(fileID, i, j);
        }
        i = i + 1;
        right = right - 1;
        while (j >= left)
        {
            j = j - 1;
            record(fileID, i, j);
        }
        j = j + 1;
        bottom = bottom + 1;
    }
    fclose(fileID);
}

int main()
{
    traverseSquare();
    return 0;
}
```

本文对顺时针运动过程中的正方形的大小进行多次改变，测得他们的遍历时间如表 5-4 所示。

表 5-4 顺时针运动过程的时间

正方形边长	Matlab 的运行时间	C++的运行时间
100	0.234854	0.003398
200	0.643439	0.013284
300	1.428270	0.029005
400	2.557394	0.052580

5.3　与其他系统的对比

现有的代码翻译系统中，把 Matlab 翻译为 C/C++的有 AUTOMATiC、m2cpp、MATISSE 和 Matlab Coder，其中 Matlab Coder 在各方面是相对更好的，我们选择将本文所介绍的代码翻译系统与 Matlab Coder 在易用性、自动化程度和代码膨胀率等方面进行对比。首先，在易用性和自动化程度方面，Matlab Coder 每次使用都要经过图 5-3 所示的若干步骤，与之相比，本文所介绍的代码翻译系统操作简单，更容易使用，并且自动化程度更高。其次，Matlab Coder 只可以对函数进行翻译，其他的 Matlab 脚本不可以进行翻译，而本文所介绍的代码翻译系统没有这个限制。

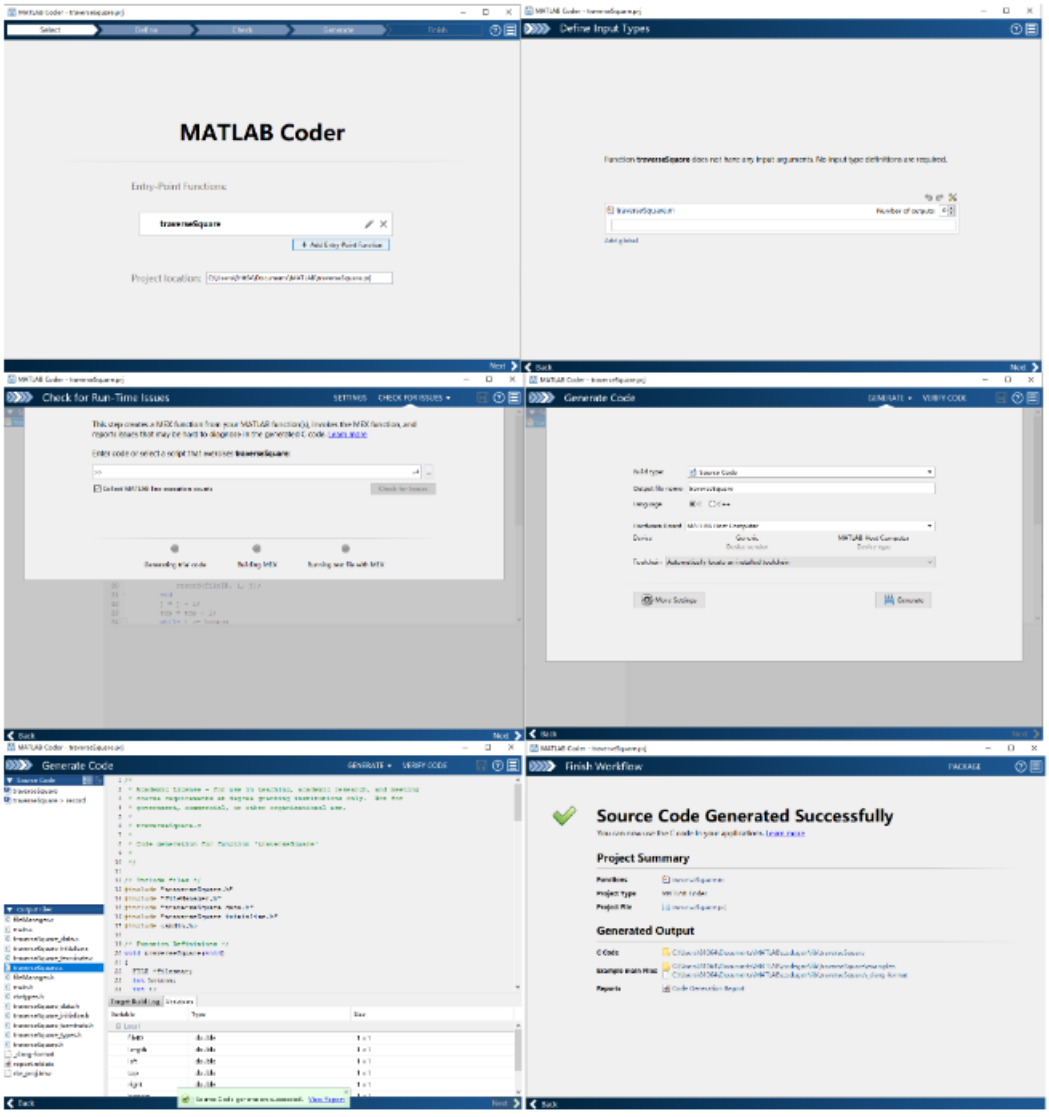

图 5-3　MatlabCoder 使用过程

本文同时对 Matlab Coder 和本文所介绍的代码翻译系统的代码行数进行了比较，如表 5-5 所示。可以看到本文所介绍的代码翻译系统的代码行数比 Matlab Coder 的代码行数更少，即代码膨胀率更低。

表 5-5 代码行数的比较

Matlab 代码行数	Matlab Coder	本系统
3	21	10
42	126	68
59	179	121

5.4 本章小结

本章通过实验说明了本文所介绍的代码翻译系统容易使用，自动化程度高，翻译后代码的运行效率高，代码膨胀率相对更低，都是优于现有的代码翻译系统的。

第 6 章　总结与展望

6.1　总结

在工程实践的过程中，每一种编程语言都有自己特殊的适用场景，比如 Matlab 适用于实验阶段，可以快速的进行算法验证，C++适用于部署阶段，可以使机器高效的运行代码，尤其是在性能较弱的机器上。我们想要在实验和部署阶段都有较高的效率，这样在整个过程中就需要进行两次编码。但是，代码的编写是一项非常艰巨而且耗时的任务，且容易出错。在需要对一个过程进行两次编码的情况下，工程进度大大降低，成本大大增加。因此，自动进行代码翻译成为软件工程中一个热门研究领域。本文设计并实现了一个基于抽象语法树的代码翻译系统，它能将源代码的结构转换为抽象语法树，再根据转化得到的抽象语法树生成目标代码，完成代码的翻译，加快工程进度、降低成本。

主要贡献可以总结为以下几个部分：

（1）设计并实现了一个完整的代码翻译系统，该系统具有清晰的模块结构，可以正确的将 Matlab 源代码翻译为 C++目标代码，翻译得到的代码是可读的且运行效率大大增加。

（2）设计并实现了一个数据类型识别子系统，对代码中的变量进行识别并记录下来为代码翻译系统的其他模块所使用，加快代码翻译过程。

（3）相比于其他的 Matlab 到 C++的代码翻译系统，该代码翻译系统的自动化程度更高。不需要使用者去填写变量的数据类型，不需要使用者去提供有关于数据类型的配置文件，也不需要使用者对翻译过程进行过多的干预，整个过程都是自动完成的。

（4）该系统对和 ACS 控制器相关的代码进行了特殊处理，解决了 ACS 控制器的 Matlab 代码库和 ACS 控制器的 C++代码库不完全一致的问题，使得翻译后的代码不需要人工修正就可以直接运行。

6.2　展望

随着信息化技术的发展，计算机得到越来越广泛的应用，人们更热衷于用编程解决大大小小的问题，代码翻译技术可以用来帮助人们减少重复编码，提高工程效率，降低工程成本，我们会从以下几个方面来继续完善代码翻译系统，使得系统更加实用和易用。

（1）在易用性方面，我们继续提高系统的易用性，使系统对用户更友好。使系统本身的功能更清晰，使工程师在使用系统进行代码翻译时操作更简单。

（2）在性能方面，我们会继续提高代码翻译系统本身的性能，使得代码翻译系统在面临大的工程项目时也能轻松的进行翻译，缩短执行代码翻译的时间，快速高效的完成代码翻译工作。

（3）对于转化得到的 C++目标代码，我们尝试对其进行优化，提高翻译得到的 C++代码的运行效率，缩短程序运行时间。

（4）在实用性方面，我们会使该代码翻译系统会去适配更多的 Matlab 代码，使得系统

本身的适用范围更广，把系统做成一个实用性更强的代码翻译系统。甚至可以尝试编写与常用 Matlab 代码对应的 C++代码库，降低代码翻译的复杂性。

参考文献

[1] MATHWORKS. 2022-company-factsheet[EB/OL]. 2022[September 4, 2022]. https://ww2.mathworks.cn/content/dam/mathworks/fact-sheet/cn-2022-company-factsheet-a4.pdf.

[2] PLAISTED D A. Source-to-source translation and software engineering[J]. Journal of Software Engineering and Applications, 2013, 06(4): 30-40.

[3] NGUYEN A T, NGUYEN T T, NGUYEN T N. Lexical statistical machine translation for language migration[C]//Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. 2013: 651-654.

[4] KARAIVANOV S, RAYCHEV V, VECHEV M. Phrase-based statistical translation of programming languages[C]//Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software. 2014: 173-184.

[5] AGGARWAL K, SALAMEH M, HINDLE A. Using machine translation for converting python 2 to python 3 code. peerj prepr. 3 (2015)[Z]. 2015.

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

[7] AHMAD W U, TUSHAR M G R, CHAKRABORTY S, et al. Avatar: A parallel corpus for java-python program translation[A]. 2021.

[8] LACHAUX M A, ROZIERE B, CHANUSSOT L, et al. Unsupervised translation of programming languages[A]. 2020.

[9] 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). IEEE, 2015: 574-584.

[10] BARONE A V M, SENNRICH R. A parallel corpus of python functions and documentation strings for automated code documentation and code generation[A]. 2017.

[11] HU X, LI G, XIA X, et al. Deep code comment generation[C]//2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC). IEEE, 2018: 200-20010.

[12] MAHMUD J, FAISAL F, ARNOB R I, et al. Code to comment translation: A comparative study on model effectiveness & errors[A]. 2021.

[13] PAREKH V, NILESH D. Pseudocode to source code translation[J]. Intl. J. Emerging Technologies and Innovative Research (JETIR), 2016, 3(11): 45-52.

[14] RAHIT K, NABIL R H, HUQ M H. Machine translation from natural language to code using long-short term memory[C]//Proceedings of the Future Technologies Conference. Springer, 2019: 56-63.

[15] KOYLUOGLU N U, ERTAS K, BROTMAN A. Pseudocode to code translation using transformers[J]. Natural Lang. Process. With Deep Learn., Stanford, CA, USA, Tech. Rep. CS224N, 2021.

[16] ALLAMANIS M, BARR E T, BIRD C, et al. Learning natural coding conventions[C]// Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. 2014: 281-293.

[17] BHOOPCHAND A, ROCKTÄSCHEL T, BARR E, et al. Learning python code suggestion with a sparse pointer network[A]. 2016.

[18] LI J, WANG Y, LYU M R, et al. Code completion with neural attention and pointer networks [A]. 2017.

[19] 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.

[20] WANG K, SINGH R, SU Z. Dynamic neural program embedding for program repair[A]. 2017.

[21] GUPTA R, PAL S, KANADE A, et al. Deepfix: Fixing common c language errors by deep learning[C]//Thirty-First AAAI conference on artificial intelligence. 2017.

[22] 董文苑. 基于代码风格分类的抄袭检测技术研究与应用 [D]. 北京邮电大学, 2021.

[23] ZHANG A, LIU K, FANG L, et al. Learn to align: A code alignment network for code clone detection[C]//2021 28th Asia-Pacific Software Engineering Conference (APSEC). IEEE, 2021: 1-11.

[24] WHITE M, TUFANO M, VENDOME C, et al. Deep learning code fragments for code clone detection[C]//2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2016: 87-98.

[25] KATZ D S, RUCHTI J, SCHULTE E. Using recurrent neural networks for decompilation[C]// 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2018: 346-356.

[26] FU C, CHEN H, LIU H, et al. Coda: An end-to-end neural program decompiler[J]. Advances in Neural Information Processing Systems, 2019, 32.

[27] WANG W, LI G, MA B, et al. Detecting code clones with graph neural network and flow-augmented abstract syntax tree[C]//2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2020: 261-271.

[28] CHABER P, ŁAWRYŃCZUK M. Automatic: Code generation of model predictive control algorithms for microcontrollers[J]. IEEE Transactions on Industrial Informatics, 2019, 16(7): 4547-4556.

[29] PAULSEN G Y, FEINBERG J, CAI X, et al. Matlab2cpp: A matlab-to-c++ code translator [C]//2016 11th System of Systems Engineering Conference (SoSE). IEEE, 2016: 1-5.

[30] PAULSEN G Y, CLARK S, NORDMOEN B, et al. Automated translation of matlab code to c++ with performance and traceability[C]//The Eleventh International Conference on Advanced Engineering Computing and Applications in Sciences, number c. 2017: 50-55.

[31] BISPO J, CARDOSO J M. A matlab subset to c compiler targeting embedded systems[J]. Software: Practice and Experience, 2017, 47(2): 249-272.

[32] BISPO J, PINTO P, NOBRE R, et al. The matisse matlab compiler[C]//2013 11th IEEE International Conference on Industrial Informatics (INDIN). IEEE, 2013: 602-608.

[33] JOISHA P G, BANERJEE P. A translator system for the matlab language[J]. Software: Practice and Experience, 2007, 37(5): 535-578.

[34] 余泽霖, 徐云. MATLAB 到高性能 C 的代码转换系统设计与实现 [J]. 信息技术与网络安全, 2022, 41(6): 8.

[35] ALBRECHT P F, GARRISON P E, GRAHAM S L, et al. Source-to-source translation: Ada to pascal and pascal to ada[J]. ACM SIGPLAN Notices, 1980, 15(11): 183-193.

[36] 徐剑峰. C-Java 自动程序转换系统原型的设计和实现 [D]. 上海师范大学, 2006.

[37] JAHANZEB M, PALANISAMY A, SJÖLUND M, et al. A matlab to modelica translator [C]//Proceedings of the 10 th International Modelica Conference; March 10-12; 2014; Lund; Sweden: number 096. Linköping University Electronic Press, 2014: 1285-1294.

[38] JURICA P, VAN LEEUWEN C. Ompc: an open-source matlab®-to-python compiler[J]. Frontiers in NEUROINFORMATICS, 2009: 5.

[39] DE ROSE L, PADUA D. A matlab to fortran 90 translator and its effectiveness[C]//Proceedings of the 10th international conference on Supercomputing. 1996: 309-316.

[40] DE ROSE L, PADUA D. Techniques for the translation of matlab programs into fortran 90 [J]. ACM Transactions on Programming Languages and Systems (TOPLAS), 1999, 21(2): 286-323.

[41] HUIJSMAN R, VAN KATWIJK J, PRONK C, et al. Translating algol 60 programs into ada [J]. ACM SIGAda Ada Letters, 1987, 7(5): 42-50.

[42] BRANT J, ROBERTS D. The smacc transformation engine: How to convert your entire code base into a different programming language[C]//Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications. 2009: 809-810.

[43] TRUDEL M, ORIOL M, FURIA C A, et al. Automated translation of java source code to eiffel [C]//International Conference on Modelling Techniques and Tools for Computer Performance Evaluation. Springer, 2011: 20-35.

[44] AN K, MENG N, TILEVICH E. Automatic inference of java-to-swift translation rules for porting mobile applications[C]//Proceedings of the 5th International Conference on Mobile Software Engineering and Systems. 2018: 180-190.

[45] TRUDEL M, FURIA C A, NORDIO M, et al. Automatic translation of c source code to eiffel [A]. 2012.

[46] QIU L. Programming language translation[R]. Cornell University, 1999.

[47] LEE S, MIN S J, EIGENMANN R. Openmp to gpgpu: a compiler framework for automatic translation and optimization[J]. ACM Sigplan Notices, 2009, 44(4): 101-110.

[48] LIANG H, SUN L, WANG M, et al. Deep learning with customized abstract syntax tree for bug localization[J]. IEEE Access, 2019, 7: 116309-116320.

[49] ZHANG J, WANG X, ZHANG H, et al. A novel neural source code representation based on abstract syntax tree[C]//2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, 2019: 783-794.

[50] RABINOVICH M, STERN M, KLEIN D. Abstract syntax networks for code generation and semantic parsing[A]. 2017.

[51] 汤闻誉. 基于抽象语法树局部与全局关系的代码补全方法 [D]. 南京大学, 2021.

[52] 张启航. 基于抽象语法树的代码缺陷检测技术设计与实现 [D]. 硕士学位论文]. 北京: 北京邮电大学, 2020.

[53] 蔡子仪. 基于抽象语法树编码的软件缺陷预测方法研究 [D]. 华南理工大学, 2020.

[54] CUI B, LI J, GUO T, et al. Code comparison system based on abstract syntax tree[C]//2010 3rd IEEE International Conference on Broadband Network and Multimedia Technology (IC-BNMT). IEEE, 2010: 668-673.

[55] 吴冲. 基于抽象语法树的重复代码检测 [D]. 上海师范大学, 2015.

[56] 王荣真. 基于抽象语法树的短文本相似度研究 [D]. 北京林业大学, 2018.

[57] 方登辉. 基于抽象语法树的代码静态缺陷检测工具开发 [D]. 北京邮电大学, 2018.

[58] 傅得强. 基于抽象语法树的源代码抄袭检测研究 [D]. 北京林业大学, 2017.

[59] 李建松. 基于抽象语法树的软件抄袭检测算法研究 [D]. 北京邮电大学, 2011.

[60] 何林洋. 基于抽象语法树的 Bug 修复模式检测系统的设计与实现 [D]. 南京大学, 2021.

[61] MATHWORKS. Matlab coder[EB/OL]. 2022. https://ww2.mathworks.cn/products/matlab-coder.html, Last accessed on 2022-08-11.

[62] VIKSTRÖM A. A study of automatic translation of matlab code to c code using software from the mathworks[Z]. 2009.

[63] 周世钦, 王波涛. MATLAB 程序转 C 代码的方法研究 [J]. 价值工程, 2018, 37(2): 182-185.

[64] LEUNG A, BOUNOV D, LERNER S. C-to-verilog translation validation[C]//2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE). IEEE, 2015: 42-47.

[65] ENTHOUGHT. Enthought-matlab-to-python-white-paper[EB/OL]. 2022. https://www.enthought.com/wp-content/uploads/2022/03/Enthought-MATLAB-to-Python-White-Paper_.pdf, Last accessed on 2022-08-11.

[66] MUKHERJEE S, CHAKRABARTI T. Automatic algorithm specification to source code translation[J]. Indian Journal of Computer Science and Engineering (IJCSE), 2011, 2(2): 146-159.

[67] TAKIZAWA H, HIRASAWA S, HAYASHI Y, et al. Xevolver: An xml-based code translation framework for supporting hpc application migration[C]//2014 21st International Conference on High Performance Computing (HiPC). IEEE, 2014: 1-11.

[68] SUNITHA E, SAMUEL P. Translation of behavioral models to source code[C]//2012 12th International Conference on Intelligent Systems Design and Applications (ISDA). IEEE, 2012: 598-603.

[69] 刘伟. 设计模式的艺术 [M]. 设计模式的艺术, 2013.

[70] 石学林. Cobol2Java 源代码翻译关键技术研究 [D]. 中国科学院研究生院 (计算技术研究所), 2005.

复旦大学
学位论文独创性声明

本人郑重声明：所呈交的学位论文，是本人在导师的指导下，独立进行研究工作所取得的成果。论文中除特别标注的内容外，不包含任何其他个人或机构已经发表或撰写过的研究成果。对本研究做出重要贡献的个人和集体，均已在论文中作了明确的声明并表示了谢意。本声明的法律结果由本人承担。

作者签名：_____ 日期：_____

复旦大学
学位论文使用授权声明

本人完全了解复旦大学有关收藏和利用博士、硕士学位论文的规定，即：学校有权收藏、使用并向国家有关部门或机构送交论文的印刷本和电子版本；允许论文被查阅和借阅；学校可以公布论文的全部或部分内容，可以采用影印、缩印或其它复制手段保存论文。涉密学位论文在解密后遵守此规定。

作者签名：_____ 导师签名：_____ 日期：_____