

东莞理工学院

本科毕业论文

毕业设计题目：面向 RISC-V 平台的 BIFANG 语
言编译系统设计与实现

学生姓名：覃思源

学 号：202041302350

院 系：学生社区知行学院

专业班级：2020 级杨振宁创新班

指导教师姓名及职称：李宽 副教授

起止时间：2023年12月--2024年5月

摘要

编译技术是计算机系统分析与设计的重要组成，作为计算机技术发展的基石，研究编译器的设计与优化目的不仅在于其本身的理论和技术，其在解决一系列问题的思路和经验上同样值得我们借鉴。

随着国内芯片产业重视程度越来越高，越来越多的厂商开始真正重视芯片的研发并且意识到芯片生态系统的重要性。RISC-V作为一种开放、精简、可扩展的芯片指令集架构，在当前计算机体系结构领域备受关注，被广泛应用于各类嵌入式系统、物联网设备以及服务器等领域。因此，针对 RISC-V 平台提供高效、实用的编译工具链，成为支持其应用场景广泛发展的关键。

本文的工作主要研究支持自定义 BIFANG 语言、面向 RISC-V 硬件平台的综合性编译系统。该编译系统的设计和实现包括词法分析、语法分析、语义分析、中间代码生成、汇编代码生成和目标代码生成等步骤，确保编译生成的可执行程序正确高效运行。其中词法分析器和语法分析器使用当前流行的 ANTLR 构建，系统最终能够将符合自定义程序设计语言 BIFANG 的测试程序编译为 RISC-V 汇编语言程序（64 位，RISC-V），并通过汇编链接后在 64 位 Fedora 操作系统的昉·星光（VisionFive）设备上运行。构成一个面向 RISC-V 平台的编译系统，为开发者编写 BIFANG 语言程序提供一个高效、适用的编译工具链。

最终经过系统测试，本文实现的 BIFANG 编译系统能够将 BIFANG 源程序编译、汇编、链接生成可执行文件，并能够在昉·星光（VisionFive）设备上正确运行，达到了预期目的。

关键词 编译、RISC-V、BIFANG、VisionFive、芯片

Abstract

Compiler technology is essential to computer systems analysis and design, and studying compilers' design and optimization offers valuable problem-solving insights. With the chip industry's growing significance in China, there is an increased focus on developing chipsets, highlighting the chip ecosystem's importance. RISC-V, an open, straightforward, and extensible instruction set architecture, is receiving wide attention and is extensively used in embedded systems, IoT devices, and servers. Providing a robust compiler toolchain for the RISC-V platform is thus vital for its broad application.

This thesis presents a compiler system supporting the BIFANG language for the RISC-V platform. It includes steps like lexical, syntax, semantic analysis, intermediate and assembly code generation, and target code production, ensuring accurate and efficient executable programs. The compiler's lexer and parser are built with the popular ANTLR, enabling compilation of BIFANG test programs into RISC-V assembly (64-bit), running on VisionFive devices with a 64-bit Fedora OS. This constitutes a RISC-V-focused compiler toolchain, offering an efficient solution for BIFANG programming.

The BIFANG compiler system devised can compile, assemble, and link source code into executables running correctly on VisionFive devices, fulfilling the research aims. This is the abstract of my undergraduate thesis in Computer Science, requiring a concise English translation and careful handling of technical terms.

Keywords Compilation, RISC-V, BIFANG, VisionFive, Chip

目 录

第 1 章 绪论	1
1.1 BIFANG 语言介绍	1
1.1.1 BIFANG 语言概述	1
1.1.2 BIFANG 语言终结符特征	2
1.2 论文研究背景及意义	3
1.3 国内外研究现状	4
1.3.1 编译器研究现状	4
1.3.2 编译器前端构建工具研究现状	5
1.3.3 指令集架构研究现状	8
1.4 主要研究内容	9
1.5 论文结构	10
第 2 章 编译系统设计基本原理	12
2.1 词法分析	12
2.1.1 正则表达式	13
2.1.2 有限自动机	14
2.1.3 词法单元识别	15
2.2 语法分析	18
2.2.1 上下文无关文法	18
2.2.2 LR 和 LL 分析法	20
2.2.3 计算 FIRST、FOLLOW 集合	20
2.3 语义分析	22
2.3.1 符号表	23
2.3.2 作用域	23
2.3.3 类型系统	24
2.4 中间代码表示	25
2.4.1 图 IR	26
2.4.2 线性 IR	28

2.4.3 混合 IR	29
2.5 代码生成	30
第 3 章 BIFANG 编译系统前端设计与实现	30
3.1 ANTLR 辅助生成工具	31
3.1.1 编写 ANTLR 词法文件	31
3.1.2 编写 ANTLR 语法文件	34
3.1.3 一键生成词法和语法分析器	36
3.1.4 ANTLR 生成文件剖析	36
3.2 BIFANG 语法分析框架	38
3.2.1 从 ANTLR 源码剖析解析树的构建过程	38
3.2.2 基于访问者模式的解析树遍历	42
第 4 章 BIFANG 编译系统后端设计与实现	45
4.1 中间代码介绍	45
4.1.1 中间代码数据结构	45
4.1.2 中间代码的组织关系	47
4.2 中间代码生成	47
4.2.1 中间代码生成依赖的工具类	48
4.2.2 中间代码生成实例分析	49
4.3 目标机器架构介绍	52
4.3.1 RISC-V 指令集架构简介	52
4.3.2 RISC-V 寄存器简介	54
4.3.3 眇 • 星光 (VisionFive) 设备简介	56
4.4 目标机器代码生成	57
4.4.1 指令选择	57
4.4.2 寄存器分配	58
4.4.3 函数调用规范与栈式存储管理	61
4.5 可执行文件生成	62
第 5 章 系统分析及测试	63
5.1 测试环境和测试方法	63
5.1.1 测试环境	63

5.1.2 测试方法	63
5.2 测试用例	63
5.3 编译系统功能测试	65
5.4 结果和分析	65
第 6 章 总结与展望	70
6.1 总结	70
6.2 展望	71
参考文献	72
致谢	73

第 1 章 绪论

随着计算机科学界的蓬勃发展，创新软件接连不断地涌现出来，特别是伴随着人工智能大模型等前沿技术的崛起，这些大模型以前所未有的速度增长，给行业带来了前所未有的活力。大模型很快成为了行内的焦点，将如同个人电脑时代的操作系统一样，有望成为人工智能基础设施的核心支柱，点燃了广泛的关注与热议。然而，尽管应用层的创新令人目不暇接，这些先进应用的实施依旧离不开编译器这类基础技术的底层支撑。编译器在应用程序与系统平台之间终扮演着应用与系统之间的桥梁角色，这种“不因乱花迷人眼”的地位，或许就是编译器的魅力。

因此，编译器在软件开发过程中占据了不可替代的地位，是保证各种新颖应用能够顺利运行的基石。编译技术的持续发展不仅为软件开发提供了强有力的支撑，也为计算机科学的未来探索打下了坚实的基础。

1.1 BIFANG 语言介绍

1.1.1 BIFANG 语言概述

BIFANG 语言是一门自己定义的程序设计语言。它是由 C 语言的子集扩展而成的，保留了函数、基本类型、语句等关键语法。每个 BIFANG 程序的源代码存储在一个扩展名为 `bf` 的文件中。该文件中有且仅有一个名为 `main` 的主函数定义，还可以包含若干全局变量声明、常量声明和其他函数定义。BIFANG 语言支持 `int/float` 类型和元素为 `int/float` 类型且按行优先存储的多维数组类型，其中 `int` 型整数为 32 位有符号数；`float` 为 32 位单精度浮点数；`const` 修饰符用于声明常量。BIFANG 支持 `int` 和 `float` 之间的隐式类型，但是无显式的强制类型转化支持。

函数：函数可以带参数也可以不带参数，参数的类型可以是 `int/float` 或者数组类型；函数可以返回 `int/float` 类型的值，或者不返回值(即声明为 `void` 类型)。当参数为 `int/float` 时，按值传递；而参数为数组类型时，实际传递的是数组的起始地址，并且形参只有第一维的长度可以空缺。函数体由若干变量声明和语句组成。

变量/常量声明：可以在一个变量/常量声明语句中声明多个变量或常量，声明时可以带初始化表达式。所有变量/常量要求先定义再使用。在函数外声明的为全局变量/常量，在函数内声明的为局部变量/常量。

语句：语句包括赋值语句、表达式语句(表达式可以为空)、语句块、`if` 语句、`while` 语句、`break` 语句、`continue` 语句、`return` 语句。语句块中可以包含若干变

量声明和语句。

表达式：支持基本的算术运算（+、-、*、/、%）、关系运算（==、!=、<、>、<=、>=）和逻辑运算（!、&&、||），非 0 表示真、0 表示假，而关系运算或逻辑运算的结果用 1 表示真、0 表示假。算符的优先级和结合性以及计算规则(含逻辑运算的“短路计算”)与 C 语言一致。

1.1.2 BIFANG 语言终结符特征

BIFANG 语言中标识符 Ident 的规范为：

字符集合：标识符由大小写字母（A-Z、a-z）、数字（0-9）和下划线（_）组成。

首字符：标识符的第一个字符必须是大小写字母（A-Z、a-z）或下划线（_）。

长度限制：标识符的长度不受限制，但是只有前 31 个字符会被编译器考虑。

区分大小写：BIFANG 语言是区分大小写的，因此大小写字母不同的标识符被视为不同的标识符。

保留字限制：标识符不能是 C 语言的保留字（也称为关键字），例如 if、else、int、return 等。

其中，关于同名标识符的约定：

全局变量和局部变量的作用域可以重叠，重叠部分局部变量优先；同名局部变量的作用域不能重叠；BIFANG 语言中变量名可以与函数名相同。

关于 BIFANG 语言中注释的规范与 C 语言一致，如下：

单行注释：以序列 ‘//’ 开始，直到换行符结束，不包括换行符。

多行注释：以序列 ‘/*’ 开始，直到第一次出现 ‘*/’ 时结束，包括结束处 ‘*/’ 。

BIFANG 语言中数值常量可以是整型数，也可以是浮点型数。

关于标识符定义的详细规范可参考 ISO/IEC 9899 第 51 页，关于注释定义的详细规范可参考 ISO/IEC 9899 第 66 页，关于整型常量定义的详细规范可参考 ISO/IEC 9899 第 54 页，关于浮点常量定义的详细规范可参考 ISO/IEC 9899 第 57 页。

一个简单的 BIFNAG 语言程序 testcase.bf 可以定义为如下形式：

```
int c = 2;
int add(int a, int b) {
    return a + b;
}
int main() {
    int sum = add(5, 3);
```

```

    return sum;
}

```

1.2 论文研究背景及意义

编译器的研究自上个世纪 50 年代以来就一直保持活跃。

从程序设计语言角度来说，其一直以惊人的速度发展进行着。从最早的 Fortran 和 Lisp，到后来的 C、C++、Java、Python 以及近些年出现的 Go、Rust 等，这些高级编程语言在各个领域有着广泛的应用。随着编程范式的多样化和编程语言的层出不穷，编译器需要针对不同的编程语言和范式提供相应的支持。例如，对于函数式编程、面向对象编程、模块化编程等不同编程风格，编译器需要具备相应的优化策略和技术。

从软件的可移植性以及跨平台兼容性的角度来说。随着云计算、物联网、边缘计算等技术的普及，应用程序需要在不同的硬件平台和操作系统上运行。编译器需要具备强大的平台适应性，以便为开发者提供一致的编程体验。

从目标机器的角度来说来说，芯片指令集架构（ISA）从复杂指令集（CISC）到精简指令集（RISC）计算机架构，从单核到双核再到多核以及当今热门的 GPU 集群，编译器需要在并行计算、能耗优化等方面发挥更大的作用，芯片架构持续发展，体系结构也随之不断发生改变。

此外，指令集作为软件和硬件之间的接口规范，是信息技术生态的起点。RISC-V，即第五代精简指令集，是一种基于 RISC 架构的 ISA，由美国加州大学伯克利分校研究团队于 2010 年设计。相对于 X86 指令集的完全封闭及 ARM 指令集高昂的授权使用费，RISC-V 指令集通过支持自由开放的指令集体系架构及架构扩展以提供软件和硬件自由。

RISC-V 的主要优点为完全开源、架构简单、易于移植、模块化设计，RISC-V 是计算机体系结构走向开放的必然产物，正在成为新一代计算机体系结构的重要代表。因此，本文设计一种面向 RISC-V 平台的编译系统，有助于满足 RISC-V 平台上软件开发的需求，推动其在各个领域的广泛应用。

因此，无论是设计新的高级程序设计语言还是让软件适配新的硬件，都离不开编译器的重要作用。编译器本身在近几年也在不断更新迭代，当前业界较为知名的编译器，例如：GCC、LLVM、Microsoft Visual C++ 等都是极为优秀的开源编译器，具有极大的学习意义。

编译原理是计算机专业必修的重要基础课程。然而，在教学过程中往往偏重于理论知识的讲解，实践环节相对较少。因为企业开发人员只需要使用集成的 IDE 开发环境，例如 Jet Brains 公司推出的 CLion，微软公司推出的 Visual Studio，苹果公司推出的 Xcode 等。这些主流的集成开发环境通常都内置了相应的编译器或编译器工具链。IDE 提供丰富的功能和工具，让开发者可以更专

注于代码编写和功能实现，而不必关注底层的编译和执行过程。然而，这种高度的抽象可能会导致开发人员对编译过程以及程序的实际执行机制缺乏深入的了解。

在可观的未来，为降低计算机编程门槛，程序设计语言等上层设施持续发展，将屏蔽更多底层细节使其变得更加抽象，甚至出现高效的自动代码生成。同时，各大语言的特性不断发展丰富，越来越多领域特定语言（DSL）也将被需要。在大模型时代，算力的重要性达到了前所未有的高度，编译器作为连接高级语言代码与底层硬件算力之间的桥梁，在提高算力利用效率、优化资源分配、支持并行计算和硬件特性利用等方面发挥着重要作用。在编译过程中编译器可以进行多种编译优化，以提高程序的执行效率。这些优化包括但不限于代码重排、循环展开、死代码消除等，可以减少程序执行时所需的计算量，从而在有限的算力下实现更高的性能。同时，编译过程中还需要考虑目标平台的资源限制，如 CPU、内存和存储。通过智能地分配和优化资源使用，编译器可以帮助程序更高效地利用算力资源，避免资源浪费。编译器还可以针对特定硬件平台的特性进行优化，如利用 CPU 的多核、GPU 的并行处理能力、FPGA 的可编程性、ASIC 加速器等。这种针对性的优化可以充分发挥硬件的算力，充分压榨硬件的性能，提升硬件的实际算力。

因此，通过设计能够将高级语言代码编译为 RISC-V 指令集的编译系统，对程序员的系统能力提高以及未来职业发展具有极大帮助，同时也为国产芯片提供更广泛的软件支持，有助于提高国产芯片的软件生态兼容性，为国产高端芯片、关键基础软件及核心系统的技术突破与产业化培养后备人才。

1.3 国内外研究现状

1.3.1 编译器研究现状

过去较长一段时间，国际上的高校和企业团队对编译器的研究有持续的关注与大量的投入。团队主要集中在国外大学和互联网企业，国内编译器的研究人员来自仅少部分高校，例如：国防科技大学、中国科学院计算技术研究所等，只有少数国内企业技术团队愿意为编译器的研究投入成本，且研究方向大多比较狭窄。近些年大模型成为热门话题，算力需求的倍增是一个显著的趋势，这导致更多的高性能芯片公司涌现，且随着国内芯片产业重视程度越来越高，越来越多的厂商开始真正重视芯片的研发并且意识到芯片生态系统的重要性，编译器作为连接高级语言代码与底层芯片算力之间的桥梁，是芯片生态系统极其重要的组成部分，扮演着越来越重要的角色。

实际上，编译技术自计算机科学诞生以来就一直是推动该领域发展的关键力量。20世纪50年代就有了早期编译器，使得如FORTRAN等高级编程语言

的出现，编译器作为将这些高级语言转换为机器语言的工具，极大地提高了程序员的生产力，使得他们能够编写更复杂、更高效的程序。在 20 世纪 70 年代初期，C 语言诞生，它最初由丹尼斯·里奇（Dennis Ritchie）在贝尔实验室开发，它的第一个版本大约在 1972 年完成，随后在 1973 年，UNIX 操作系统的第二个版本首次使用了 C 语言编写的代码，推动了操作系统领域里程碑式的进展。C 语言的出现及其编译器的迭代优化，使得系统级编程变得更加高效，为操作系统和硬件驱动程序的开发奠定了基础。至今，C 语言仍然是最广泛使用的编程语言之一。在 90 年代，Java 由 Sun Microsystems 公司正式发布，Java 的发布标志着一种新型的、面向对象的、跨平台的编程语言的诞生，它迅速成为了互联网应用开发的首选语言之一。Java 的虚拟机（JVM）和“一次编写，到处运行”的理念，为软件开发特别是 Web 应用与移动应用开发带来了革命性的变化。2010 年以来，Python 开始加速流行，它以其简洁的语法和强大的标准库，降低了编程的门槛，使得非专业程序员也能快速上手，推动了数据分析、科学计算、自动化的发展，掀起了人工智能的热潮。由此可见，计算机科学领域每一次重大变革几乎都离不开编译器的关键推动作用。

国际上，编译器优化技术的研究持续活跃，包括静态和动态优化、程序分析、并行和分布式编译等。研究者致力于提高编译器的优化能力，以适应不断增长的计算需求和多样化的硬件架构。随着多核处理器和异构计算平台（如 GPU、FPGA）的普及，编译器研究开始关注如何有效利用这些硬件资源，实现高效的并行计算。同时，为了支持跨平台开发，编译器研究也在探索如何生成能够在不同操作系统和硬件架构上运行的代码。国际上有许多知名的开源编译器项目，如 GCC、LLVM 等，这些项目吸引了全球范围内的研究者和开发者参与，推动了编译器技术的发展。

最近几年，国内在编译器领域也有所发展，例如，华为推出了自主研发的编译器“方舟编译器”，旨在提升应用的性能和安全性。此外，还有“龙芯”处理器的编译器优化研究等。在学术研究方面，国内高校和研究机构在编译技术方面进行了大量的研究工作，包括编译器优化、并行计算、程序分析等领域。在产业应用方面，国内企业在 AI、云计算、大数据等领域的产业推动了编译技术的实际应用，在大模型流行的如今，深度学习框架备受关注，其中的深度学习编译器被用于优化大模型的训练和推理过程，其对大模型的低比特量化既可以起到模型规模压缩的作用，又可以降低神经网络的计算复杂性，为资源受限的终端与嵌入式端部署深度学习模型提供了一个重要的优化手段。

1.3.2 编译器前端构建工具研究现状

编译器的前端构建工具方面，在 ANTLR 受到关注之前，词法分析程序和

语法分析程序的设计与实现一般选择 Lex、Yacc 和 Flex、Bison 工具。

Lex 和 Yacc 作为两个经典的编译器构建工具，是 1975 年美国贝尔实验室研发出来的一个编译辅助程序，其作用是根据用户定义的词法和语法规则，自动生成词法分析程序和语法分析程序，用以提高编译器的开发效率。

Lex 是一个词法分析器生成器，它根据用户定义的规则自动生成词法分析器。用户通过编写一系列描述三型语言的正规表达式，来定义输入文本到标记（tokens）的转换规则，Lex 会自动生成词法分析器，该词法分析器可以正常分析满足该正规表达式的输入串，并输出标记符号串。

Yacc 是一个语法分析器生成器，它能接受 BNF 描述的上下文无关语言的语法规则，其语法满足 LALR(1)文法的规范。用户需要定义语言的语法规则，并指定产生式规则的优先级和结合性。Yacc 生成的分析器通常用于分析和解释源代码中的语法结构，例如表达式、语句、函数定义等，但它不支持 LL 分析技术。Lex 通常与 Yacc 配合使用，以构建完整的编译器前端。

Flex 和 Bison 是 Lex 和 Yacc 的现代替代品，提供了更易于使用的特性和更好的性能。Flex 是 Lex 的改进版，提供了更高的灵活性和性能。它支持更复杂的正则表达式，并且可以生成更高效的词法分析器。Flex 的语法与 Lex 类似，但增加了一些新的特性，如支持 C++、更灵活的宏定义等。Bison 是 Yacc 的改进版，它的语法规则定义更加灵活，支持更复杂的语言结构，并且提供了更好的错误恢复机制。Bison 支持 LALR(1)和 LR(0)解析技术，可以与 Flex 配合使用，以构建更强大的编译器前端。

总之，Lex、Yacc 和 Flex、Bison 在构造词法和语法分析时都接受一个 BNF 描述文件，满足 LALR(1)的上下文无关文法。Lex、Yacc 和 Flex、Bison 实现生成的分析表体积很大，构造过程需要很大工作量，生成的方法难以阅读，同时，在 LR 分析器生成过程中，构建 LR 分析表时需要处理同心集（也称为冲突集合）。在处理同心集时，LR 分析器需要对每个同心集合中的状态进行分析，并且为每个状态计算对应的动作。这个过程可能会涉及到多次状态的合并、拆分和冲突解决，因此需要消耗大量的内存和计算资源。特别是在同心集较大或者包含大量冲突状态的情况下，由于需要在内存中维护所有状态的信息以及它们之间的关系，所以处理这些同心集会占用大量的内存空间。

ANTLR 是由旧金山大学研究开发的一款功能强大的语法分析器生成器，可以用来读取、处理、执行和转换结构化文本或二进制文件。它被广泛应用于学术界和工业界构建各种语言、工具和框架。

从符合 EBNF 范式的形式化语言描述中，ANTLR 生成该语言的语法分析器。生成的语法分析器可以自动构建语法分析树——表示文法如何匹配输入的数据结构。ANTLR 还可以自动生成树遍历器，你可以用它来访问那些树的节点，以

执行特定的代码。

ANTLR v4 的语法分析器采用自顶向下的下降递归分析方法，使用一种新的称为 Adaptive LL(*)或 ALL(*)的语法分析技术，它可以在生成的语法分析器执行前在运行时动态地而不是静态地执行文法分析。ANTLR v4 极大地简化了匹配句法结构（如算术表达式）的文法规则。对于传统的自顶向下的语法分析器生成器来说，识别表达式的最自然的文法是无效的，ANTLR v4 则不然，你可以使用像下面这样的规则来匹配表达式：

```
expr : expr '*' expr
      | expr '+' expr
      | INT
      ;
```

像 `expr` 这样的自引用规则是递归的且是左递归的，因为它的可选项中至少有一个立即引用它自身。ANTLR v4 会自动地将左递归规则（例如 `expr`）重写为非左递归等价物，唯一的约束是左递归必须是直接的，即那些规则立即引用它们自身。

目前 ANTLR 应用非常广泛，例如 Apache Maven 是一个流行的 Java 项目管理工具，它使用 ANTLR 来解析和处理 POM（Project Object Model）配置文件。IntelliJ IDEA 是一款流行的 Java 开发 IDE，它使用 ANTLR 来支持 Java、Kotlin 等编程语言的语法高亮和代码提示。Jaskell 是一个基于 Java 的 Haskell 编译器，使用 ANTLR 作为其语法解析器的工具之一。ANTLR 具有下面这些优点：

- 解析能力。ANTLR 集成了词法和语法分析，支持 LL(*)语法，比 LALRA(1) 更强大，没有移进-归约、归约-归约等语法冲突。同时支持更广泛的解析技术，而 Lex/Yacc 和 Flex/Bison 主要基于 LALR(1)和 LR(0)。ANTLR 还提供一个自动构建抽象语法树并有自由的重写规则，可以根据实际情况设计抽象语法树的节点。
- 目标语言。ANTLR 支持生成多种编程语言，而 Lex/Yacc 和 Flex/Bison 主要生成 C 语言代码。
- 语法定义。ANTLR 的语法定义更加直观和易于理解，而 Lex/Yacc 和 Flex/Bison 的语法定义较为复杂。
- 错误处理。ANTLR 提供了更灵活的错误处理机制，Bison 在这方面也有所改进。
- 社区和工具。ANTLR 拥有一个活跃的社区和丰富的工具集，这有助于解决开发过程中遇到的问题。
- 更好性能。ANTLR 通过采用最小向前看（lookhead）决策算法对 LL(K) 算法进行了改进，使得语法分析速度得到提高，也减少了分析代码的体积。其语法分析的效率接近 LL(1) 递归下降分析器。

由以上分析可知，在传统的词法分析和语法分析的设计和实现中，早期的工具如 Lex、Yacc 和 Flex、Bison 都是采用的 LR 分析方法。它们使用 BNF 的语法描述，由于向前看 1 个字符，在运行起来的时候分析表所需的内存很大，而使用采用 LL 分析方法的 ANTLR 工具可以节省很多工作量，并且 LL(*) 具有预测性，加快了预测速度，因此本文编译系统的编译器前端实现使用 ANTLR 辅助工具是一种较优异的选择。

图 1-1 展示了 ANTLR 构建的编译器前端的基本工作流。

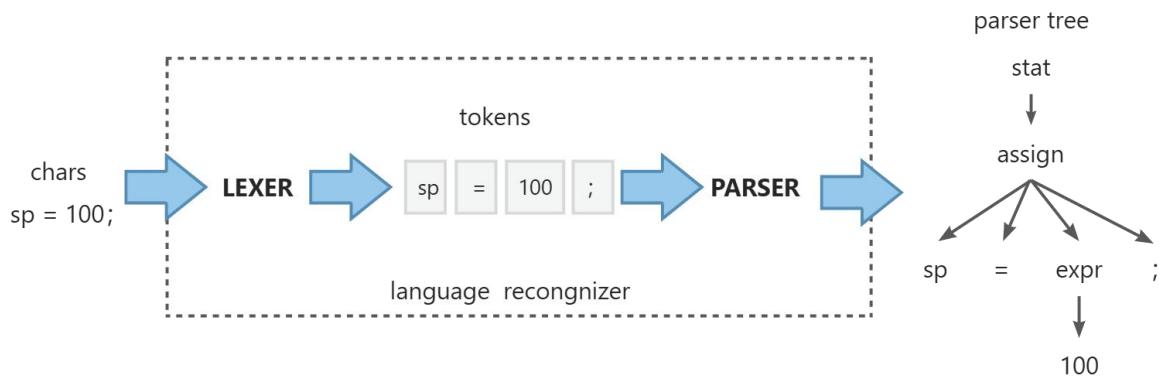


图 1-1 ANTLRv4 构建的编译器前端工作流

1.3.3 指令集架构研究现状

在目标机器架构方面，指令集架构是计算机体系结构中最重要的组成部分，它定义了处理器能够理解和执行的指令集合，也是编译器最终生成的指令需要遵循的一套规范，因此，深刻理解目标机器的指令集架构对于编译器研究人员来说极为重要。

在电子计算机的早期，指令集通常是硬编码在机器中的，每个计算机系统都有自己独特的指令集，这导致了软件的不兼容性。20世纪50年代，随着计算机技术的发展，出现了汇编语言，它提供了一种更易于理解和编写的指令集表示方法。这一时期，指令集开始标准化，如 IBM 的 System/360 架构引入了兼容的指令集，使得软件可以在不同的硬件上运行。20世纪70年代，微处理器的出现使得计算机更加普及。这一时期的指令集设计开始考虑成本和功耗，如 Intel 的 x86 架构，早期的英特尔 x86 指令集采用的是 CISC（复杂指令集计算机）架构，如今该指令集在个人电脑得到了广泛应用。20世纪80年代，RISC（精简指令集计算机）架构开始流行。RISC 架构采用简化的指令集，指令长度固定，操作数在寄存器中完成，减少了指令的复杂度。典型的 RISC 架构包括 MIPS、ARM 和 PowerPC 等。这些架构注重性能、功耗和成本之间的平衡，被广泛应用于高性能计算、嵌入式系统和移动设备等领域。与此同时，CISC 架构继续发展，试图通过复杂的指令集来提高编程效率。这两种架构的竞争推动了指令集

设计的多样化。随着摩尔定律的推动，处理器开始集成更多的核心和线程。指令集开始支持并行处理和多线程技术，如 x86 架构的超线程（Hyper-Threading）和 ARM 架构的 NEON 技术。随着计算机应用的多样化和需求的增加，CISC 和 RISC 开始融合。现代的指令集架构不完全是严格的 CISC 或 RISC，而是结合了两者的特点，以满足不同的应用需求。例如，英特尔的 x86 架构在保留 CISC 特性的同时，也引入了 RISC 风格的微操作，提高了执行效率。

在 2010 年，加州大学伯克利分校的研究人员在对比市面上所有的体系结构后发现，指令集越来越复杂和臃肿，而且授权费昂贵，于是他们打算自行设计一套全新的开源指令集。2015 年，RISC-V 基金会成立，旨在维护指令集以及体系结构规范的标准化和完整性。此外，他们还成立了 SiFive 公司，以推动 RISC-V 商业化落地。作为一个后起之秀，RISC-V 指令集在设计时充分借鉴了其他商业化指令集的优点，并且吸取了它们的经验和教训。RISC-V 指令集具有如下优点：

- 设计简洁。RISC-V 指令集和基础体系结构设计都相当简洁，特别是 RISC-V 的指令集文档和基础体系结构设计文档才几百页，而一些商用 RISC 芯片的文档多达上万页。
- 模块化。RISC-V 指令集采用模块化的设计思想。它具有一个最小的指令集，这个指令集可以完整地实现一个软件栈，然后通过模块化的方式实现其他扩展功能的指令，如浮点数乘法和除法指令、矢量指令等。模块化的 RISC-V 体系结构使得用户能够灵活选择不同的模块组合，以适应不同的应用场景，从嵌入式设备到服务器都可以使用 RISC-V 体系结构进行设计。
- 开源。RISC-V 指令集采用 BSD 开源协议授权。使用 RISC-V 进行教学、学术研究、商业化都不需要授权费。
- 具有丰富的软件生态。目前大部分开源软件支持 RISC-V 指令集，包括：Linux 内核、GCC 等。

综上所述，本文的编译系统选择具有发展潜力的 RISC-V 指令集架构作为目标机器并利用较为先进的编译器前端构建工具 ANTLR 辅助实现。

1.4 主要研究内容

本文研究的主要内容是设计并实现支持 BIFANG 语言、面向 RISC-V 硬件平台的综合性编译系统。该系统能够将符合自定义程序设计语言 BIFANG 的测试程序编译为 RISC-V 汇编语言程序（64 位，RISC-V），并通过汇编链接后在 64 位 Fedora 操作系统的昉·星光（VisionFive）设备上运行。构成一个面向 RISC-V 平台的编译系统，为开发者编写 BIFANG 语言代码提供一个简便、高效的编译工具链。该编译系统由如下两个关键模块组成：编译器前端模块和编译

器后端模块，且本文系统将每个过程封装成类，便于后续进行模块化扩展。具体研究内容如下：

(1) BIFANG 语言设计及编译器前端实现：

设计语法语义规则，支持 BIFANG 语言核心特性。使用 ANTLR 工具辅助设计并实现词法分析器，负责将 BIFANG 源代码转换为 Token 流，使用 ANTLR 工具辅助实现语法分析器，构建语法解析树以进行语义分析。

(2) 设计实现合适的中间代码 (IR)：

设计并实现简化版 LLVM IR 的中间代码，他表现为 Block、Function、Instruction 等内存中的数据结构。

(3) 实现基于语法解析树的中间代码生成器：

设计实现中间代码的生成器 builder 及其辅助工具类，在对语法解析树基于访问者模式遍历时进行中间代码的生成工作。

(4) 实现目标机器代码生成：

设计实现 RISC-V 平台的目标代码生成器 Asmbuilder 及其辅助工具类，实现基于中间代码生成 RSIC-V 汇编代码的功能。

(5) 目标机器代码汇编、链接与执行：

基于 RISC-V 汇编代码，借助成熟汇编器与链接器，汇编链接生成可执行文件，在 64 位 Fedora 操作系统上能正确运行。

(6) 测试与验证：

编写 BIFANG 语言的测试用例，覆盖各类语法和功能。在 64 位 Fedora 操作系统的昉·星光设备进行系统运行测试，验证编译系统的正确性、性能和可靠性。

1.5 论文结构

全文总共分为六大部分，其组织结构和具体内容如下：

第一章 绪论。首先，介绍了计算机科学技术领域当前热门研究方向与编译器的关系以及编译器研究的必要性。接着介绍了本文编译系统实现的 BIFANG 语言的基本语法、基本特性和终结符特征。其次，阐述了编译器的研究背景、研究意义及国内外研究现状，突出了编译器在推动计算机科学技术领域发展中发挥的关键作用。然后，介绍了编译前端设计中几款辅助设计工具的发展历程、功能作用以及对 ANTLR 的优点做了总结。接着，阐述了计算机指令集架构的发展历程、研究现状以及对 RISC-V 指令集的优点做了总结，引出本文所实现编译系统而选择 ANTLR 工具和 RISC-V 目标平台。最后，简述了本文的主要工作内容，并给出了论文的章节安排。

第二章 编译系统设计基本原理。这一章深入探讨了编译器的结构框架，包

括词法分析、语法分析、语义分析、中间代码生成和代码生成等编译过程的基本概念和设计原则。系统阐述了流行的编译理论，并解析其在 BIFANG 编译系统中的具体应用。

第三章 BIFANG 编译系统前端设计与实现。本章详述了 BIFANG 编译前端的设计过程和实现细节。内容包括词法分析器和语法分析器的构建，以及如何基于 ANTLR 来实现词法分析、语法分析、语义分析和语法解析树的生成。

第四章 BIFANG 编译系统后端设计与实现。本章聚焦于 BIFANG 编译器后端的设计与实现。介绍了中间代码的内存结构以及生成流程，然后介绍了目标代码的生成流程，包括指令选择、寄存器分配以及函数调用与栈的布局。最后在转换为可执行文件过程中，介绍了利用开源汇编器和链接器对接 BIFNAG 编译系统的具体方法。

第五章系统分析及测试。本章设计了部分测试程序以覆盖整个编译系统的功能，并通过一系列测试来验证系统的功能和稳定性。展示测试结果，并通过预期与实际结果比对，证实了 BIFANG 编译系统的实用性和有效性。

第六章总结与展望。最后一章对 BIFANG 编译系统的设计、实现及测试做了全面总结。同时指出了系统当前存在的不足与潜在改进空间，并对未来的研 究方向提出了建设性的见解和建议。

第 2 章 编译系统设计基本原理

本章主要介绍编译系统设计和实现过程中涉及的技术与原理，主要分为编译器前端设计模块和编译器后端设计模块。

这里的“前端”指的是编译器对程序代码的分析和理解过程。它通常只跟语言的语法有关，跟目标机器无关。可以看到，编译器的“前端”技术分为词法分析、语法分析和语义分析三个部分。而它主要涉及自动机和形式语言方面的基础的计算理论。而与之对应的“后端”则是生成目标代码的过程，跟目标机器有关。前端设计模块主要介绍词法分析器、语法分析器以及语义分析的设计和构建，后端设计模块主要介绍汇编生成目标文件、链接生成可执行文件以及目标机器指令集，图 2-1 展示了一个编译系统主要完成的工作。

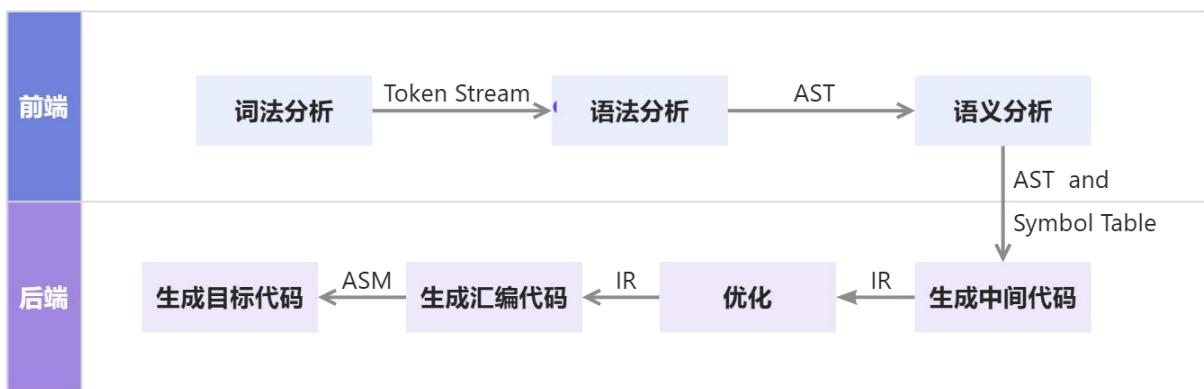


图 2-1 编译系统主要完成的工作

2.1 词法分析

编译器的第一项工作是词法分析，完成这项工作的模块叫做词法分析器，它可以从左至右读取源程序中的代码字符，并把可以识别的字符组合保存为单个词素，也叫“词法记号”（Token）。可以用下面这段代码理解词法分析器的工作：

```

#include <stdio.h>
int main(int argc, char* argv[ ]){
    int age = 45;
    if (age >= 17+8+20) {
        printf("Hello BIFANG!\n");
    }
    else{
        printf("Hello world!\n");
    }
    return 0;
}

```

词法分析器读完这段程序会识别出 if、else、int 这样的关键字，main、printf、age 这样的标识符，+、-、=这样的操作符号，还有花括号、圆括号、分号这样的符号，以及数字字面量、字符串字面量等，这些都是 Token。

那么如何写一个程序来识别 Token？可以通过指定一些规则来区分每个不同的 Token，还是以上面的代码举例，例如识别 “age” 这样的标识符，它以字母开头，后面可以是字母或数字，直到遇到第一个既不是字母又不是数字的字符时结束。识别 “>=” 这样的操作符，当扫描到一个 “>” 字符的时候，就要注意，它可能是一个 GT（Greater Than，大于）操作符，但由于 GE（Greater Equal，大于等于）也是以 “>” 开头的，所以再往下再看一位，如果是 “=” ，那么这个 Token 就是 GE，否则就是 GT。识别 “45” 这样的数字字面量，当扫描到一个数字字符的时候，就开始把它看做数字，直到遇到非数字的字符。这些规则可以通过手写程序来实现。

事实上，很多编译器的词法分析器都是手工编写实现的，例如 GNU 的 C 语言编译器。编译系统也可以选择词法分析辅助工具来生成词法分析程序，比如 ANLTR，本文采用的方法就是使用 ANTLR 辅助前端设计词法分析器。

类似 ANTLR 等生成工具是基于一些规则来工作的，这些规则用“正则文法”表达，符合正则文法的表达式称为“正则表达式”。生成工具可以读入正则表达式，生成一种叫“有限自动机”的算法，来完成具体的词法分析工作。词素的识别过程就是字符串匹配的过程，其中用到的基本原理主要就是正则表达式和有限自动机。

2.1.1 正则表达式

在前面的第一章中，本文介绍了 BIFANG 语言中标识符 Ident 的规范，本文规定了标识符由大小写字母（A-Z、a-z）、数字（0-9）和下划线（_）组成，并且标识符的第一个字符必须是大小写字母（A-Z、a-z）或下划线（_）。但这样的描述规则并不精确，需要换一种更加严谨的表达方式，这种方式就是正则表达式。

正则表达式（Regular Expression），简称为 RE，是一种用于描述字符串模式的工具。它是由一系列字符和操作符构成的表达式，用于匹配、搜索和替换文本中的字符串。正则表达式在文本处理、数据提取、验证输入等方面具有广泛的应用。一般的形式语言主要使用以下的一些基本概念和常用语法来构建正则表达式：

(1) 字符匹配

普通字符：除了元字符（后文会提到）外，正则表达式中的普通字符表示它们自身。例如，正则表达式 abc 匹配包含连续的字符 "abc" 的字符串。

字符类：用方括号 [] 表示，匹配其中任意一个字符。例如，[abc] 匹配 "a"、"b" 或 "c"。

范围类：用 '[' 和 ']' 来限定匹配的字符范围，在字符类中使用连字符 - 表示字符的范围。例如，[a-z] 匹配任意小写字母。

(2) 常用元字符

- . (句点) : 匹配除换行符 \n 外的任意字符。
- \ (反斜线) : 用于转义后续字符, 使其失去特殊意义。例如, \. 匹配句点字符 “.”。
- *: 匹配前一个字符零次或多次。
- +: 匹配前一个字符一次或多次。
- ?: 匹配前一个字符零次或一次。
- {n}: 指定前一个字符重复 n 次。

如正则表达式 a.*b 可以匹配以 a 开头且以 b 为结尾的任意长字符串, 即字符串 acb、azcdb 等均可匹配。

3. 常用特殊字符类

- \d: 匹配任意数字字符, 相当于 [0-9]。
- \D: 匹配任意非数字字符, 相当于 [^0-9]。
- \w: 匹配任意字母、数字或下划线字符, 相当于 [a-zA-Z0-9_]。
- \W: 匹配任意非字母、数字或下划线字符, 相当于 [^a-zA-Z0-9_]。
- \s: 匹配任意空白字符, 包括空格、制表符、换行符等。
- \S: 匹配任意非空白字符。

上面的例子涉及了 4 种 Token, 这 4 种 Token 用正则表达式表达为如下:

Ident : [a-zA-Z_] ([a-zA-Z_] | [0-9])*

IntLiteral: [0-9]+

GT : '>'

GE : '>='

需要注意的是, 不同的语言的标识符、字面量的规则可能是不同的。比如有些语言允许使用 Unicode 作为标识符, 这也就意味着变量的名称可以是中文的。还有一些语言规定, 十进制数字字面量的第一位不能是 0, 此时正则表达式会有不同的写法, 对应生成的有限自动机自然也是不同的。

2.1.2 有限自动机

有限自动机 (Finite Automaton, FA) 是一种抽象的计算模型, 用于描述能够接受 (或拒绝) 有限序列输入符号的计算机, 有限自动机原理广泛应用在各种各样的领域, 它在 EDA 软件设计、编译器词法分析以及自然语言处理中词性标注、句法分析、文本匹配等地方都被应用。它由一组状态、输入符号集合、转移函数和起始状态以及接受状态组成。有限自动机分为确定性有限自动机 (Deterministic Finite Automaton, DFA) 和非确定性有限自动机 (Nondeterministic Finite Automaton, NFA) 两种类型。

确定性有限自动机对于每个状态和输入符号组合, 只有一个唯一的下一个状态。

非确定性有限自动机对于某些状态和输入符号组合，可能有多个可能的下一个状态。

有限自动机可以使用状态转移图、状态转移表或者状态转移函数等方式来表示。状态转移图用图形方式表示状态和状态之间的转移关系，图的节点表示状态，图的边表示状态的转移，边的权值表示状态转移需要的条件，状态转移图常用于直观展示。

2.1.3 词法单元识别

词法分析程序将源程序作为字符流输入，其作用是把字符流进行切分，分解为多个单词（Token），每个单词都对应一个字符序列，它表示了源程序的相关信息。下面是最典型的几种单词：

关键字：比如 int 和 if 等，他们是程序语言提前规定好的且带有特殊意义的定长字符串，标识符不能与其重名。

标识符：是用户自己定义的不定长字符串，必须遵循程序语言提前规定好的命名规范，例如 BIFANG 语言规定，BIFANG 语言的标识符都是由大小写字母（A-Z、a-z）、数字（0-9）和下划线（_）组成。标识符的首字符必须是大小写字母（A-Z、a-z）或下划线（_）。标识符的长度不受限制，但是只有前 31 个字符会被编译器考虑。标识符区分大小写，因此大小写字母不同的标识符被视为不同的标识符。标识符不能是 C 语言的保留字（也称为关键字），例如 if、else、int、return 等。

常量：包括数字型常量和字符型常量等

运算符：例如：+、-、×、÷、% 等。

词法单元的识别过程，就是使用正则表达式所对应的有限自动机扫描源程序匹配单词的过程。因此在设计词法分析扫描程序时，要设计一种把正则表达式转换成计算机程序的模型，这种转换的一般流程是：

(1) 将正则表达式转换为后缀表达式。使用算法将中缀表达式（正则表达式）转换为后缀表达式。这可以通过使用栈和一组优先级规则来实现。这个步骤将确保后缀表达式在运算顺序上是明确的，减少了后续转换的复杂性。

(2) 构建非确定性有限自动机(NFA)。从后缀表达式构建非确定性有限自动机。这个过程通常使用 Thompson's construction 算法，它可以将正则表达式直接转换为 NFA。在这个过程中，需要构建一系列的状态和转移，确保它们能够正确地识别正则表达式所描述的语言。

(3) 将 NFA 转换为确定性有限自动机(DFA)。虽然 NFA 是一种理论概念，但在实际中，确定性有限自动机更易于实现和操作。使用子集构造算法，可以将 NFA 转换为等价的 DFA。这个过程会创建一个具有确定性状态和转移的自动机。

(4) 最小化 DFA。最小化 DFA 可以进一步减小自动机的规模，使其更加高效。通过消除多余的状态和转移，可以减少 DFA 的复杂性。Hopcroft's algorithm 和 Brzozowski's algorithm 是用于最小化 DFA 的两个常用算法。

(5) 编写程序实现自动机。将最小化的 DFA 实现为代码。这通常包括定义状态和转移函数，并编写状态转换逻辑，这个过程可以基于状态转移表或状态转移图。

有了 DFA 之后，就可以解析一下 “age >= 45” 这个关系表达式，图 2-2 展示了解析该表达式的一个简化过程。

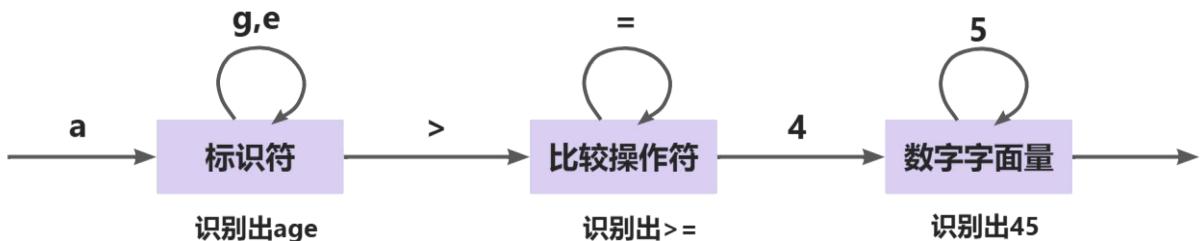


图 2-2 “age >= 45”的解析过程

词法分析程序在遇到 age、>= 和 45 时，会分别识别成标识符、比较操作符和数字字面量，但上面的图只是一个简化的示意图，严格意义上的 DFA 是如图 2-3 这种画法。

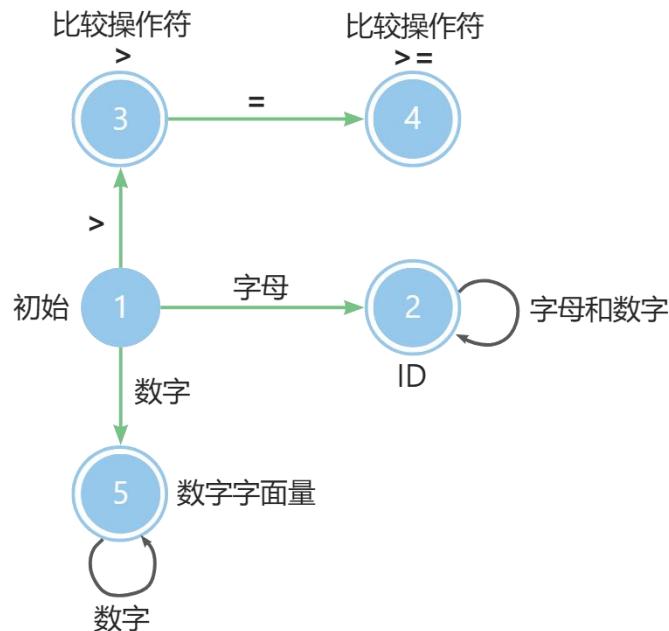


图 2-3 “age >= 45”的 DFA

初始状态：刚开始启动词法分析的时候，程序所处的状态。

标识符状态：在初始状态时，当第一个字符是字母的时候，迁移到状态 2。当后续字符是字母和数字时，保留在状态 2。如果不是，就离开状态 2，写下该 Token，回到初始状态。

大于操作符 (GT)：在初始状态时，当第一个字符是 > 时，进入这个状态。它是比较操作符的一种情况。

大于等于操作符 (GE)：如果状态 3 的下一个字符是 =，就进入状态 4，变成 >=。它也是比较操作符的一种情况。

数字字面量：在初始状态时，下一个字符是数字，进入这个状态。如果后续仍是

数字，就保持在状态 5。

另外，可以看到图中的圆圈有单线的也有双线的。双线的意思是这个状态是一个终止状态，表明此时已经是一个合法的 Token 了，单线的意思是这个状态还是临时状态。按照这 5 种状态迁移的过程，你可以很容易将有限自动机编写成计算机模型。

```

DfaState newState = DfaState::Initial;

if (isAlpha(ch)) {                                     //第一个字符是字母
    newState = DfaState::Id; // Enter Id state
    tokenType = TokenType::Identifier;
    tokenText.push_back(ch);
} else if (isDigit(ch)) {                             //第一个字符是数字
    newState = DfaState::IntLiteral;
    tokenType = TokenType::IntLiteral;
    tokenText.push_back(ch);
} else if (ch == '>') {                               //第一个字符是 >
    newState = DfaState::GT;
    tokenType = TokenType::GT;
    tokenText.push_back(ch);
}
...
switch (state) {
    case DfaState::Initial:
        state = initToken(ch, tokenType);      // 重新确定后续状态
        break;
    case DfaState::Id:
        if (isAlpha(ch) || isDigit(ch)) {
            tokenText.push_back(ch);           // 保持标识符状态
        } else {
            state = initToken(ch, tokenType);  // 退出标识符状态，并保存 Token
        }
        break;
    case DfaState::GT:
        if (ch == '=') {
            tokenType = TokenType::GE;
            state = DfaState::GE;             // 转换成 GE
            tokenText.push_back(ch);
        } else {
            state = initToken(ch, tokenType); // 退出 GT 状态，并保存 Token
        }
        break;
}

```

```

case DfaState::GE:
    state = initToken(ch, tokenType);           // 退出当前状态，并保存 Token
    break;

case DfaState::IntLiteral:
    if (isDigit(ch)) {
        tokenText.push_back(ch);               // 继续保持在数字字面量状态
    } else {
        state = initToken(ch, tokenType);     // 退出当前状态，并保存 Token
    }
    break;
}

```

运行这个程序，就可以成功解析类似“age >= 45”这样的 BIFANG 语言程序，该例子的输出如下，其中第一列是 Token 的类型，第二列是 Token 的文本值。

Identifier	age
GE	>=
IntLiteral	45

这个例子便于帮助本文阐述词法分析的原理，那就是根据构造好的有限自动机，在不同的状态中迁移，从而解析出 Token 来。要支持更多的词法识别，系统只需要扩展这个有限自动机，为它添加更多的状态节点和状态转移路线，就可以逐步实现一个完整的词法分析程序。

2.2 语法分析

每种程序设计语言都有一套明确的规则来描述该语言程序的语法结构称之为文法，而语法分析的主要功能就是检查源程序是否符合该程序设计语言的文法规则，然后基于文法规则构造一颗语法树并将其传递给编译器的后续部分进行处理，本文将介绍 LR 和 LL 两种语法分析技术。

2.2.1 上下文无关文法

程序设计语言的语法结构和规则由文法规定。根据 Noam Chomsky 对自然语言结构的研究，他把自然语言文法分为 0、1、2、3 型文法，其中 2 型文法在程序设计语言中被广泛使用，2 型文法又称为上下文无关文法，现在已成为程序设计语言结构的标准方式。

上下文无关文法描述语法规则与正则表达式描述词法规则相似，但在实际使用中存在着较大差异。在解析复杂程序结构的时候，正则文法可能无法正确解析，例如：复杂的求值顺序。因为正则表达式是线性的，而上下文无关文法是递归的，上下文无关文法包含了正则文法，比正则文法能做更多的事情。因此它是一种比正则表达式更强大的符号表示法。

在形式上，上下文无关文法 G 是一个四元组 (T, NT, S, P) ，它上下文无关文法由产生式集合 (P) 、非终结符集合 (NT) 、终结符集合 (T) 和一个开始符号 (S) 组成，其中：

(1) 产生式。上下文无关文中的每一条规则都是一个产生式。一条产生式由产生式左部，符号以及产生式右部组成，形如 $\alpha \rightarrow \beta$ ，其中 α 为产生式左部，它是一个非终结符。 \rightarrow 是符号，表示语法解析的推导过程。 β 是产生式右部，它是终结符和非终结符的混合。

(2) 非终结符。它是语法产生式中的语法变量，引入非终结符在于为产生式的定义提供抽象和结构。

(3) 终结符。它是语法产生式中的单词，单词包含一个词素及其语法范畴。是组成语言的基本符号，是产生式中最小的单位，只有终结符才可以成为 AST 的叶子节点。

(4) 开始符号。是一个非终结符，被指定为语法的起始符号，在至少一条产生式的左部出现。

上下文无关文法产生的符号表示法是巴科斯范式（Backus-Naur Form，BNF），在 BNF 表示法中，非终结符由尖括号“<>”包围，例如<expression>，终结符直接使用普通字符表示，例如“+”、“*”、“(”、number 等，用“::=” 符号来表示产生式的推导，“|”用来分隔不同的替代选项，表示“或”关系。例如，产生式可以表示为如下形式：

```

<expression> ::= <term> | <term> <add_op> <expression>
<term>      ::= <factor> | <factor> <mult_op> <term>
<factor>    ::= <number> | '(' <expression> ')'
<number>     ::= [0-9]*
<add_op>     ::= '+' | '-'
<mult_op>    ::= '*' | '/'

```

普通的巴科斯范式也会存在左递归问题，扩展巴科斯范式（Extended Backus-Naur Form，EBNF）是基于巴科斯范式的一种扩展，是一种更加强大的文法规则符号表示法，BNF 使用尖括号(<>)来标识非终结符，而 EBNF 则没有，EBNF 使用逗号(,)来表示连接，而 BNF 使用空格，在 EBNF 中，花括号({ })表示零次或多次出现，而在 BNF 中通常使用垂线(|)来表示可选的选。因此，EBNF 比 BNF 在阅读上会更加方便，在实际使用中也是使用 EBNF 来消除左递归。ANTLR 是构建词法分析和语法分析器的辅助工具，其中的规则采用的就是 EBNF 范式来描述的。下面是如上产生式的 EBNF 格式：

```

expression = term, { add_op, term } ;
term = factor, { mult_op, factor } ;
factor = number | '(', expression, ')' ;

```

```

number = digit, { digit } ;
add_op = '+' | '-' ;
mult_op = '*' | '/' ;
digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' ;

```

2.2.2 LR 和 LL 分析法

LR (Left-to-Right, Rightmost derivation) 分析法是一种自底向上的语法分析方法，通常使用有限自动机来实现。它从输入串开始向右扩展，使用最右推导，逐步进行“移进-归约”直到文法的起始符号，从语法树的角度来说，语法树的构建是从末端开始，向上“归约”，直到归约至根节点。LR 分析法的特点是采用最右推导，允许在移进和归约操作中使用向前看符号，有自动构建分析表的能力，通常使用 LR(1) 和 LALR(1) 等算法。同时 LR 分析法能够处理广泛的文法，没有左递归问题，可以处理一定程度的二义性。但 LR 实现相对复杂，且 LR 分析器通常比 LL 分析器更难以调试和理解。

LL (Left-to-Right, Leftmost derivation) 分析法是一种自顶向下的语法分析方法，它从左到右扫描输入符号串。使用最左推导来构建语法树，自顶向下就是从文法的起始符号开始，向下推导，直到叶子节点。LL 分析法通常使用递归下降的分析算法来实现。LL 分析法的特点是采用最左推导，递归下降分析器调试会比 LR 分析器的分析表简单，所需内存也小。自顶向下语法分析器的效率极其依赖于其在扩展非终结符时选择正确产生式的能力。如果语法分析器总是产生正确的选择，自顶向下语法分析就是高效的，最左匹配的自顶向下语法分析器低效的主要原因是回溯，最坏的情况是左递归导致语法分析无法终止。要消除回溯必须要对语法分析器做一个简单的修改。在语法分析器去选择下一条规则时，可以同时考虑当前关注的符号以及它的下一个符号，称为前瞻符号 (lookahead symbol)。通过向前看一个符号，语法分析器可以消除在解析右递归表达式语法时多种选择造成的不确定性，所以 LL(1) 是无回溯的，无回溯的语法也称为预测性语法。

LL 语法与 LR 相比，主要的区别是 LL 自顶向下，LR 是自底向上。显然使用自顶向下的递归下降方式的语法分析更符合逻辑，设计实现过程更加符合手工书写的习惯。同时，LL 语法没有移进-归约和归约-归约的冲突。在 ANTLR 中就是利用了 LL 分析，使其代码阅读性更强，错误处理更好，这是 LR 分析法不能做到的。

2.2.3 计算 FIRST、FOLLOW 集合

自顶向下的语法分析器的构建需要依赖于文法中相关函数的 FIRST 和 FOLLOW。FIRST 和 FOLLOW 函数获得终结符或非终结符的集合分别称为终结符或非终结符的 FIRST 集合和 FOLLOW 集合。在自上而下的语法分析过程中，可以依据下一个输入符号属于哪个符号的 FIRST 集合来选择使用哪个产生式展开。在对某条产生式的语法

分析时， FOLLOW 集合可以判断是否完成语法分析返回，还是出现语法错误。自顶向下的语法分析方法对使用的文法有严格的要求，必须是 LL(1) 文法，其中的 1 就表示在每一步中除了看当前符号还需要向前看一个输入符号，这样可以消除在解析右递归表达式语法时多种选择造成的不确定性。

因此，在定义 LL(1) 文法的时候就要避免出现左递归或者产生式右部的开始符号交集不为空而引起回溯。当编译系统选择使用递归下降法进行分析时，应该先计算出 FIRST 集合和 FOLLOW 集合，保证其文法满足 LL(1) 文法的规则。LL(1) 文法是一种特殊的上下文无关文法，具有以下特点：

- (1) 对于每个非终结符 A，每个推导式之间没有公共前缀。
- (2) 对于每个非终结符 A 和每个终结符 a，至多有一个产生式以 A 为起始符号并且以 a 为开始符号，也就是对于每个非终结符 A 和每个终结符 a，只有唯一的产生式可以被选择进行展开。
- (3) 对于每个非终结符 A，空串 ϵ 要么不在 FIRST(A) 中，要么 ϵ 在 FIRST(A) 中且 ϵ 不在 FIRST(β) 中，其中 β 是 A 的任何直接右侧符号串。

FIRST 集合是指对于文法中的每个非终结符号，其能够推导出的终结符号的集合。对于文法中的每个非终结符号 A，FIRST(A) 包含了 A 所能推导出的串的首个终结符号的集合。

算法求解 FIRST 集合的基本思路是：

- 如果 A 是终结符号，则 $\text{FIRST}(A) = \{A\}$ 。
- 如果 A 是非终结符号，且存在产生式 $A \rightarrow a$ ，则将 $\text{FIRST}(a)$ 中的所有符号加入到 $\text{FIRST}(A)$ 中。
- 如果存在产生式 $A \rightarrow \epsilon$ ，则将 ϵ 加入到 $\text{FIRST}(A)$ 中。

FOLLOW 集合是指对于文法中的每个非终结符号，其紧跟在该非终结符号后面的可能出现的终结符号的集合。对于文法中的每个非终结符号 A，FOLLOW(A) 包含了在某个句型中 A 的后面可能出现的终结符号的集合。

算法求解 FOLLOW 集合的基本思路是：

- 将 \$（文法开始符号的结束标志）加入到 $\text{FOLLOW}(S)$ 中，其中 S 是文法的开始符号。
- 如果存在产生式 $A \rightarrow a B \beta$ ，则将 $\text{FIRST}(\beta)$ 中的所有非 ϵ 符号加入到 $\text{FOLLOW}(B)$ 中。
- 如果存在产生式 $A \rightarrow a B$ 或者 $A \rightarrow a B \beta$ ，并且 β 可以推导出 ϵ ，则将 $\text{FOLLOW}(A)$ 中的所有符号加入到 $\text{FOLLOW}(B)$ 中。

下面用一个简单的文法举例说明如何计算 FIRST 和 FOLLOW 集合：

假设有以下文法：

S → AB

```
A -> a | ε
```

```
B -> b | ε
```

首先，计算 FIRST 集合：

$\text{FIRST}(A) = \{a, \epsilon\}$ ，因为 A 可以推导出 a 或者 ϵ 。

$\text{FIRST}(B) = \{b, \epsilon\}$ ，因为 B 可以推导出 b 或者 ϵ 。

$\text{FIRST}(S) = \{a, \epsilon\} \cup \{b, \epsilon\} = \{a, b, \epsilon\}$ ，因为 S 可以推导出 AB，而 A 和 B 的 FIRST 集合分别是 $\{a, \epsilon\}$ 和 $\{b, \epsilon\}$ 。

然后，计算 FOLLOW 集合：

$\text{FOLLOW}(S) = \{\$\}$ ，因为 S 是开始符号，它的结束标志是整个句子的结束标志。

$\text{FOLLOW}(A) = \{b\}$ ，因为 A 在某些情况下可以跟随 B。

$\text{FOLLOW}(B) = \{b\}$ ，因为 B 在某些情况下可以跟随 B 本身。

通过计算出的 FIRST 和 FOLLOW 集合，系统可以在构造预测分析表时用于确定产生式的选择，以及在语法分析过程中帮助确定规约动作。

建立了 FIRST 集合、FOLLOW 集合和 LL 算法计算过程的直觉认知。系统在实现 LL 算法解析语法时，可以选择两种实现方式。

第一种，还是采用递归下降算法，只不过现在的递归下降算法没有任何回溯。无论走到哪一步，系统都能准确地预测出应该采用哪个产生式。

第二种，是采用表驱动的方式。这时候需要基于计算出来的 FIRST 和 FOLLOW 集合构造一张预测分析表。根据这个表，查找再遇到什么 Token 的情况下，应该走哪个路径。

这两种方法都是等价的，在实现时可以根据实际来选择。ANTLR 则采用了第二种表驱动方式来实现 LL(*)。ANTLR 根据语法规则和计算出的 FIRST 集合、FOLLOW 集合构造预测分析表，预测分析表是在编译时构建的，它包含了对于每个非终结符在特定输入符号下应该使用哪个产生式的信息。在解析过程中能够根据当前的符号和预测分析表中的信息准确地选择正确的产生式或规则，而无需回溯。这种表驱动的方法在实际中通常能够提供更高效的语法解析。

2.3 语义分析

对于编译系统来说，要想得到正确的输出，除了要把源程序分解词素和组成为一定的结构，还必须分析源程序的内容和语义。前面通过词法和语法分析，编译器已经对源程序分析构建出了一棵语法树来描述程序的语法结构，但编译器只有这颗语法树的结构是无法得到正确执行结果的，编译器还需要得到语法树上各个节点的信息，例如一个标识符节点“`A`”，究竟是函数还是变量，如果是变量，那它是什么类型？这就是语义分析需要完成的工作，编译器需要创建一系列数据结构，用来记录这些节点的具体含义，以方便后续计算或者优化。语义分析完成后，编译器对语法树的认识更加全面，例如，一个`+` 节点可以让其子节点的两个操作数相加，想要得到相加的结

果是什么类型，这个分析过程叫类型推导。当右值计算完成后，赋值给左值时，要检查左右两边的类型是否匹配，这个过程叫类型检查。如果左值和右值的类型不匹配，但是存在缺省的转换，这个过程叫类型转换。此外，语义分析还要解决引用消解，对变量名称、常量名称、函数名称、类型名称、包名称等的消解，都属于引用消解。

BIFNAG 语言源程序都可以看成声明和语句两部分组成。声明表示对一个标识符进行了定义，该标识符可以被语句使用，而语句则是使用声明过的标识符进行计算。编译器前端语义分析的工作就是对源程序中的声明语句进行解析，并构造一系列数据结构来存储数据对象的相关属性信息，比如变量名和属性信息。

2.3.1 符号表

符号表（Symbol Table）是编译器中用于存储程序中所有符号（如变量、函数等）信息的数据结构。它是编译过程中的一个重要组成部分，用于支持语义分析、链接和其他编译阶段。它的主要作用是提供一个快速查找和访问符号属性的机制，确保编译器能够正确地处理程序中的引用和声明。

源程序中的每一个标识符的信息都会保存在符号表中，构成符号表里的一条记录，记录的字段就是变量的各个属性。符号表的逻辑组织形式主要有单表和多表两种形式。单表形式指的是将所有的符号都组织在一张符号表中。这意味着无论是全局的还是局部的标识符，都存储在同一个符号表中。在单表形式中，通常使用标识符的名字作为索引，每个条目包含了该标识符的信息，比如类型、存储位置等。单表形式的符号表通常是一个平面结构，所有的标识符都处于同一级别。多表形式指的是将类型相似的符号组织在一张符号表中。每个符号表负责存储一类符号，比如全局类的标识符、函数类的标识符、局部类标识符等。每个符号表都有自己的作用域范围，并且可能会存在嵌套关系。在多表形式中，通常会考虑综合使用符号的名字和作用域信息作为索引，以便在查找时确定符号的可见性和有效范围。

符号表通常需要具备以下功能：

- (1) 收集存储源程序中各个标识符的具体信息，例如对于变量来说，需要存储其名称、类型、value、def 集合、use 集合等。
- (2) 分配存储空间。目标代码生成阶段，编译器根据符号表记录的符号信息，为每个变量分配存储空间。
- (3) 根据上下文信息进行类型检查。查询符号信息是否复合规范，是否需要类型转换等。

2.3.2 作用域

作用域是指程序中某一部分代码中标识符的可见性和有效性范围。在一个给定的作用域内，标识符可以被访问和使用，而在其他作用域内可能无法访问。作用域可以

是全局的（全局作用域），也可以是局部的（局部作用域）。通常，一个作用域可以嵌套在另一个作用域内，形成作用域链，内层作用域可以访问外层作用域中的标识符，但反之则不成立。

在编译器中，有多种实现作用域的方式。可以维护一个作用域栈，每当进入一个新的作用域（比如函数、条件语句等）时，就会将该作用域压入栈中；当离开作用域时，则会将其从栈中弹出。也可以维护一个作用域链表，每当进入一个新的作用域时，就会用头插法插入一个新的作用域，并用当前作用域指针指向这个新的作用域，当离开作用域时，就会将当前作用域指针指向当前作用域指针的上一层作用域（链表的下一个节点）。本文实现的 BIFANG 编译系统就是采用第二种方式来实现作用域的。这样就可以在语义分析阶段通过作用域链来确定标识符的可见性和解析。

通常一个完整的编译器项目中既需要符号表来管理标识符的信息，又需要作用域来确定标识符的可见性和有效范围。符号表和作用域通常是紧密结合的，它们共同协作来支持编译器的语义分析阶段。

2.3.3 类型系统

类型系统就是一门语言所有的类型的集合，同时类型也是标识符的一个属性，它的作用是为编译器提供这个标识符的类型信息，告诉编译器可以如何使用这些数据。例如，对于整型数据，它可能占 32 或者 64 位存储，系统可以对它做加减乘除操作。而对于字符串，它可能占很多个字节，并且通过一定的编码规则来表示字符。字符串可以做连接、查找、获取子字符串等操作，但不能像整数一样做算术运算。

一门语言的类型系统是包含了与类型有关的各种规则的一个逻辑系统。类型系统包含了一系列规则，规定了如何把类型用于变量、表达式和函数等程序元素，以及如何创建自定义类型等。例如，如果定义了一个类的某些方法，那就只能通过调用这些方法来使用这个类，没有别的方法。这些强制规定减少了程序出错的可能性。

在语义分析阶段，依据静态类型检查操作语句(比如 $x + y$)的合法性时，首先要得知两个运算分量的类型信息才能进行类型兼容判断。而在汇编代码生成阶段，进行数据的定义时(比如 int a)，也要获得变量的类型信息才能知道它应该占据多大的内存空间。

此外，有了类型系统，就相当于定了规矩，可以检查施加在数据上的操作是否合法。因此类型系统最大的好处，就是可以通过类型检查降低计算出错的概率。

类型系统主要有以下三方面作用：

(1)类型推导 (Type Inference)：类型推导是指编程语言中的一种特性，即编译器能够根据上下文自动推断表达式的类型，而无需显式地指定类型。通过类型推导，编程语言可以在不失灵活性和简洁性的同时，确保类型安全性。例如，对于变量声明 int x = 10;，编译器可以根据右侧的常量值推断出 x 的类型为整数类型。再例如，对于

$a = b + 10$, 如果 b 是一个浮点型, $b+10$ 的结果也是浮点型。如果 b 是字符串型的, 有些语言也是允许执行 + 号运算的, 实际的结果是字符串的连接。类型推导通常应用于函数返回值、lambda 表达式、模板类等场景。

(2) 类型检查 (Type Checking) : 类型检查是编译器在编译程序时进行的一种检查机制, 用于确保程序中的操作符和操作数之间的类型匹配。类型检查能够捕获程序中的类型错误, 并在编译期间发出错误提示, 避免程序在运行时出现类型不匹配的问题。例如, 对于表达式 $a = b + 10;$, 在编译期间当右边值的类型推导完毕后, 编译器会检查赋值操作符两边的类型是否匹配, 如果不匹配就会发出类型错误。

(3) 类型转换 (Type Conversion) : 类型转换是指将一个数据类型转换为另一个数据类型的操作。在编程语言中, 类型转换通常分为隐式类型转换 (Implicit Type Conversion) 和显式类型转换 (Explicit Type Conversion) 两种。隐式类型转换是由编译器自动执行的类型转换, 通常发生在不同类型之间的操作或赋值时, 编译器会自动将其中一种类型转换为另一种类型, 以确保操作的正确性。显式类型转换则是由程序员显式地指定的类型转换操作, 通常使用类型转换运算符或者类型转换函数来实现。类型转换可以是安全的, 也可以是不安全的, 不安全的类型转换可能会导致数据丢失或未定义的行为。

语义分析的本质是对上下文相关情况的处理, 根据符号表、作用域等数据结构, 可以让系统使用更加形式化、更清晰的算法来完成属性计算的任务。

2.4 中间代码表示

IR, 也就是中间代码 (Intermediate Representation), 它是编译器中很重要的一种数据结构, 表示了源程序的抽象, 即在源代码和目标代码之间的一种“桥梁”或“中间件”。编译器在做完前端工作以后, 首先就是生成 IR, 并在此基础上执行各种优化算法, 最后再生成目标代码。

编译器通常组织为一连串的处理趟。随着编译器不断推导解析被编译的源程序, 它必须把前一趟收集到的信息传递到下一趟。几乎编译器的每个处理阶段都会操纵 IR 形式的程序, 在推到期间使用的中间表示代表了这个程序的权威表示形式。因此, IR 的性质, 如读写特定字段的机制、查找特定事实或注释的机制、在 IR 形式的程序中定位导航的机制, 对减轻编写编译器各个处理趟的负担, 以及对执行这些处理趟的代价, 都有直接影响。

为编译器选择一种适当的 IR, 要求对源语言、目标机器和编译器所转换的应用程序的性质都有较为深入的理解。例如源到源的转换器可以使用与源代码非常相似的 IR, 而针对微控制器输出汇编代码的编译器使用类似于汇编代码的 IR 可能会有更好的结果。

中间代码有多种形式，不同的编译器设计者可以根据编译器的目标设计和优化需求来选择不同的中间代码形式。从 IR 的结构上可以分为 3 类。

(1) 图 IR (Graphical IR)

图 IR 使用图形结构来表示程序的控制流和数据流。通常，图 IR 采用有向图的形式，其中结点表示程序中的基本块，边表示控制流转移或数据依赖关系。图 IR 通过图中的对象（节点、边、列表、树）来描述算法。它强调的是数据结构的图形表示，这使得它在某些情况下更适合于描述复杂的控制流和数据流。

(2) 线性 IR (Linear IR)

线性 IR 使用线性序列（例如列表或数组）来表示程序的控制流和数据流。每个基本块以线性的方式表示，并且在序列中按照执行顺序排列。线性 IR 通常更接近于目标代码的表示，但比目标代码更抽象。类似某些机器上的伪代码，相应的算法就是迭代遍历这个简单的线性操作序列。

(3) 混合 IR (Hybrid IR)

混合 IR 结合了图 IR 和线性 IR 的特点，旨在兼顾二者的优势。混合 IR 可以在表示控制流和数据流时使用图形结构，同时在表示单个基本块时使用线性结构。

2.4.1 图 IR

有许多编译器使用了图 IR，虽然所有图都包含了节点和边，但在抽象层次、图与底层代码之间的关系、图的结构等方面，各种图 IR 不同

1. 抽象语法树 (AST)

抽象语法树是源代码的抽象语法结构的树状表现形式。它用树状的方式表示编程语言的语法结构，树中的每个节点都表示源代码中的一种结构。抽象语法树保留了语法分析树的基本结构，但是删除了其中冗余的节点，表达式的优先级和语义仍然保持原样，但无关的节点则被删除。其有丰富的节点类型，包括：

根节点：通常代表整个源文件或代码块。

声明节点：用于表示变量、函数、类等的声明。

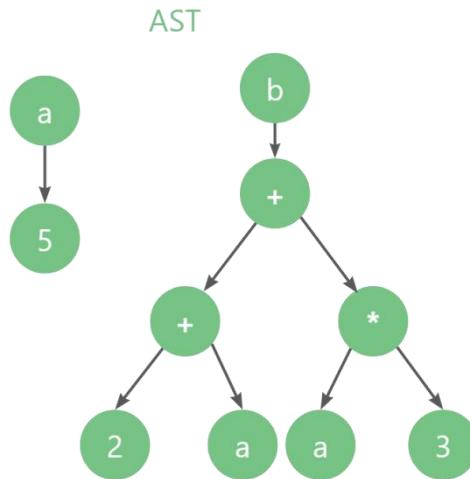
语句节点：表示控制流语句，如 if、while、for 等。

表达式节点：表示数学和逻辑表达式，如加法、减法、比较等。

叶子节点：通常表示程序中的字面量（如数字、字符串）和变量名。

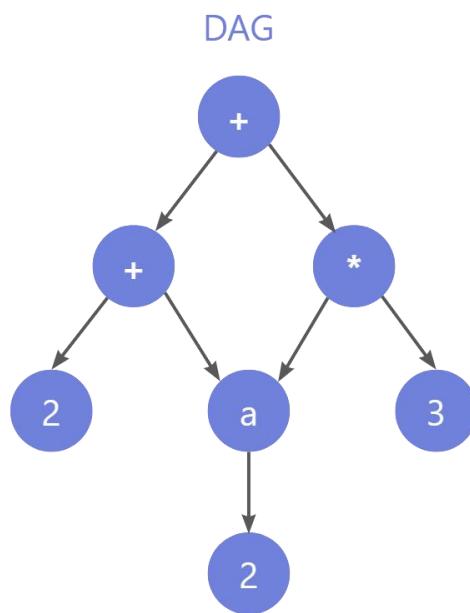
例如：语句 “`a=5; b=(2+a)+a*3;`” 的 AST 如图 2-4 所示。

AST 在编译器和解释器的设计中有多种用途。源程序转化成 AST 后，编译器可以使用 AST 来优化代码，例如消除冗余的代码、优化循环和数据结构等。对于后端CodeGen 阶段，编译器可以将 AST 转换为目标机器的机器代码或字节码，这是因为相对源程序，AST 使得后端指令映射更加便利。同时，AST 可以用于生成调试信息，帮助程序员理解代码的执行过程。

图 2-4 “ $a=5; b=(2+a)+a*3;$ ” 的 AST

2. 有向无环图 (DAG)

虽然 AST 相对于语法分析树来说已经更加简洁，但它仍然保留着源代码本来的组织结构。例如对于运算表达式 $a \times 4 + a \times 4 \times b$ ，与之对应的 AST 中包含了表达 “ $a \times 4$ ” 计算的两个不同节点，这是冗余的表达，实际中只需要保留一个 “ $a \times 4$ ” 的副本即可。有向无环图，也叫有向非循环图，比 AST 更加强大，其结构上则避免了 AST 这种冗余的副本，使子树可重用。在 DAG 中，节点可以有多个父节点，相同子树允许被重用，这种共享使得 DAG 更为紧凑。简而言之，DAG 是具有共享机制的一种 AST。相同子树只能实例化一次，他可能有多个父节点，“ $a \times 4 + a \times 4 \times b$ ” 语句的有向无环图如图 2-5 所示。

图 2-5 “ $a \times 4 + a \times 4 \times b$ ” 的 DAG

DAG 并不是单纯地共享同一个子树，如果这个子树是一个表达式求值，并且在两次使用该子树期间其内部节点没有发生改变，那么编译器在两次使用该子树期间实际上只进行了一次表达式求值（第一次使用需要子树表达式求值，第二次使用直接拿

到子树的值），这种策略可以避免求值的计算。但编译器必须证明这颗子树在两次使用期间没有发生改变，如果期间不包含子树节点的赋值操作，也不包含对其他过程的调用，则容易证明。如果在两次使用这颗共享子树之间存在对子树内部节点改变的情况，就无法重用表达式的值，此时 DAG 构造算法将子树设置成无效（即需要重新计算）。

DAG 通常更适合用于数据流分析及数据依赖关系表示，在实际系统中使用 DAG 有两个原因。第一个原因是实际系统对内存使用有限制，那么使用 DAG 将有助于减少 IR 的内存占用。第二个原因就是利用 DAG 具有的共享机制来去除冗余，达到对代码的优化效果，例如全局公共子表达式消除、复写传播、全局死代码消除等。

3. 控制流图 (CFG)

控制流图是编译器中用于表示程序控制流的一种图形结构。它用节点表示基本块 (Basic Block)，用边表示控制流转移。CFG 是编译器进行控制流分析和优化的重要数据结构。

CFG 中的每个节点代表程序中的一个基本块。基本块是程序中一串连续的指令序列，其中的各个操作总是按序全部执行，它开始于一个有标号的操作，通常以分支、跳转（如条件跳转、无条件跳转）或条件判断指令结束。编译器将程序分解为基本块，并构建控制流图以进行进一步的分析和优化。CFG 中的边表示程序执行从一个基本块到另一个基本块的控制流转移。边可以是条件边（如 if 语句中的条件满足或不满足时的跳转）或非条件边（如顺序执行或无条件跳转）。

控制流图通常有一个入口节点和一个出口节点。入口节点表示程序的起始点，出口节点表示程序的结束点。它对各种可能的运行时控制流路径提供了一种基于图的表示法，控制流图为编译器进行各种控制流分析和优化提供了基础。编译器可以在控制流图上执行各种优化技术，例如循环优化、分支优化、条件分支预测、循环展开等，从而改善程序的性能和效率。

此外，图 IR 还有依赖关系图，调用图等，它们通常用作辅助 IR，即针对特定的任务从其他 IR 构建出某类型的图 IR，使用该图 IR 完成特定的分析或优化，之后便丢弃。

2.4.2 线性 IR

典型的线性 IR 是三地址代码，三地址代码是一个由基本程序步骤组成的序列。它是一种低级的表示形式，它将源代码转换为一系列简单的指令，每条指令最多包含三个操作数。每个操作数可以是常量、变量或者是由之前指令计算得到的临时值，三地址代码和树型结构 IR 的不同之处在于它没有层次化的结构。

三地址代码指令的基本结构是 $x = y \text{ op } z$ ，其中 x 是目标操作数（可能是一个变量或者是一个临时值）， y 和 z 是源操作数（可以是变量、常量或者是之前指令产生

的临时值）， op 是操作符（例如加法、减法、乘法等）。一条三地址代码指令的右侧最多有一个运算符，也就是说不允许出现组合的算术表达式，，例如对于 $a + b \times c$ 这样的高级语言算术表达式，翻译成三地址代码之后是下面这种形式：

```
t1=b×c
t2=a + t1
```

为了表示表达式的中间结果，三地址代码经常使用临时变量。这些临时变量是在代码生成过程中创建的，用于暂时存储表达式的计算结果。除了表达式操作，三地址代码还可以表示控制流结构，例如条件语句、循环语句等。条件语句通常被转换为条件跳转指令，循环语句通常被转换为跳转指令和标签指令。

在三地址代码中函数调用通常被转换为特定的指令序列，用于将参数传递给函数、调用函数并处理返回值等操作。针对数组和指针操作，三地址代码提供了相应的指令，用于数组元素的访问、指针的解引用等操作。

汇编语言程序就是一种线性代码，它包含了一个指令序列，其中的指令按顺序执行。在编译器中使用线性 IR 的理由很简单，源程序是高级语言的线性形式代码，编译器输出的目标机器代码也是线性形式的。

三地址代码通常实现为一组四元组。一组四元组由四个字段组成：两个操作数、一个运算符、一个目标。因此，编译器为了存储一系列有序的三地址代码，需要使用便利的数据结构来存储这些四元组。例如：定义四元组的结构体，然后使用一个数组来存储这些四元组(如果把这个数组存放在 CFG 的各个节点中，那么这就是混合 IR 最常见的一种形式)；也可以使用指针数组，数组中每个元素都存储着指向一个四元组的指针，该指针数组也可以包含在 CFG 节点中形成混合 IR；还可以把各个四元组连接形成链表，这种存储方法在 CFG 中不需要消耗太多存储空间，但代价是只能顺序遍历各个四元组。

2.4.3 混合 IR

混合 IR 是线性中间表示和图中间表示的结合，这里以 LLVM IR 为例进行说明。LLVM(Low Level Virtual Machine)是 2000 年提出的开源编译器框架项目，旨在为不同的前端后端提供统一的中间表示。LLVM IR 使用线性 IR 表示基本块，使用图中间表示表示这些块之间的控制流。基本块中，每条指令以静态单赋值(Static Single Assignment, SSA) 形式呈现，这些指令构成一个指令线性列表。SSA 形式要求每个变量只赋值一次，并且每个变量在使用之前定义。控制流图中，每个节点为一个基本块，基本块之间通过边实现控制转移。

混合 IR 结合了线性 IR 和图 IR 的特性，具有线性 IR 的直观性和图 IR 的灵活性，可以同时处理控制流和数据流，并在表示形式上保持灵活性。这使得它能够同时适应不同类型的编译器任务和优化需求，例如循环优化、指针分析等。混合 IR 已经成为

了编译器界的一种标准化和通用的表示形式。值得注意的是，BIFANG 编译系统使用的就是混合 IR。

2.5 代码生成

编译器模型的最后一个步骤就是代码生成，这个过程一般由代码生成器模块来完成。代码生成器读取编译器前端所生成的 IR 和符号表等相关包含源程序信息的数据结构，输出等价语义的目标程序。

从数学的角度来说，将一个自定义的源程序转化成一个最优的目标程序是一个 NP 难的问题，因为它涉及到在组合优化的背景下解决一个复杂的问题，需要从源程序到目标程序的可能的映射中找到一个最优的映射。通常，这个映射需要满足许多约束条件，如运行速度、功耗、内存占用等。

编译器后端需要把前端生成的 IR 映射成目标机器的 ISA，这个映射过程就是指令选择。形成 IR 和目标机器 ISA 的映射后，还需要对这些映射后的指令做一个调度来决定它的执行顺序。在最终代码上的每个位置，哪些值应该保存在寄存器，哪些值应该保存在内存中，这个过程就是寄存器分配。因此，一个完整的代码生成器需要包含目标机器的汇编文件，经过汇编链接即可生成目标文件。图 2-6 详细展示了编译系统在代码生成阶段的工作。

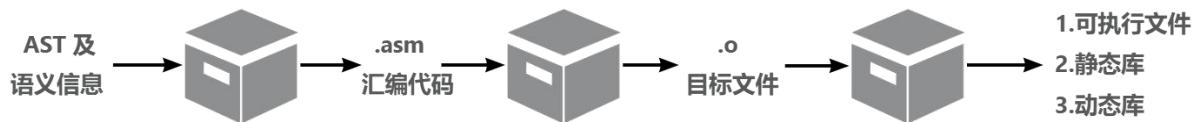


图 2-6 代码生成工作流

第 3 章 BIFANG 编译系统前端设计与实现

本章将介绍 BIFANG 语言编译系统前端部分的设计和实现工作，BIFANG 语言编译系统前端部分主要由前端构建工具 ANTLR 来辅助实现，主要完成包括词法分析、语法分析和语义分析模块的设计和实现。包含 BIFNAG 语言的词法定义、BIFNAG 语言的语法定义、Token 的解析、BIFANG 语言文法定义、符号表和作用域等用于存储运行信息的数据结构的定义和组织、基于监听者模式的 AST 可视化打印解析、基于访问者模式访问 AST 构建 LIR。

ANTLR 的语法规则文件（通常以.g4 为扩展名）用于定义词法（lexer）和语法（parser）规则。一般如果语法比较复杂则会基于 Lexer 和 Parser 写入两个不同的.g4 文件中，例如 ANTLR 仓库中的 Java8 语法文件就拆分成了两个文件。如果语法较为简单，可以把 Lexer 和 Parser 两部分写到一个.g4 文件中。本系统采用的方案是前者，这使得词法和文法结构更加清晰。当然，ANTLR 文法规则作为一种 DSL 语言，在其仓库中也有 ANTLR 的词法文件和语法文件，开发者可以根据它学习 ANTLR 自身的词法和语法结构。

3.1 ANTLR 辅助生成工具

BIFANG 编译系统前端词法分析器的设计是利用 ANTLR 辅助实现的，下文将介绍 ANTLR 实现词法分析器的方法与过程。

3.1.1 编写 ANTLR 词法文件

ANTLR 规则描述了如何将输入文本分解成一系列的标记（tokens），以及如何根据这些标记构建出一个解析树。以下是编写.g4 词法文件的基本规则，按照 ANTLR 官方给出的.g4 文件的文法规则编写词法规则，.g4 词法文件的结构主要包括：

```
lexer grammar Name;

options {...}

tokens {...}

rule1 // lexer rules
...
ruleN
```

(1) Grammar Header:

每个.g4 文件以 grammar 关键字开始，后跟文法的名称，例如：

```
grammar MyGrammar;
```

(2) Options:

可以在文件的顶部使用 options 块来设置各种选项，例如指定目标语言、导入词汇表等，例如：

```
options {
    language = Cpp;
    tokenVocab = BIFANGLexer;
    //仅在语法规则文件中使用， 指定词法分析器使用的词法单元的词汇表是 BIFANGLexer.g4
}
```

(3) Lexer Rules:

词法规则定义了如何将输入文本分解成标记。每个规则由一个名称和一个模式组成，例如：

```
ID : [a-zA-Z]+ ;
INT : [0-9]+ ;
```

(4) Token Definitions:

可以使用 tokens 关键字定义一组预定义的标记名称，例如：

```
tokens {
    PLUS;
    MINUS;
}
```

下面是 BIFANG 语言基于 ANTLR 文法编写的词法文件。

```
lexer grammar BIFANGLexer;
options {
    language = Cpp;
}
CONST: 'const';
INT: 'int';
FLOAT: 'float';
VOID: 'void';
IF: 'if';
ELSE: 'else';
WHILE: 'while';
BREAK: 'break';
CONTINUE: 'continue';
RETURN: 'return';
PLUS: '+';
MINUS: '-';
MUL: '*';
DIV: '/';
MOD: '%';
```

```

ASSIGN: '=';
EQ: '==';
NEQ: '!=';
LT: '<';
GT: '>';
LE: '<=';
GE: '>=';
NOT: '!';
AND: '&&';
OR: '||';
L_PAREN: '(';
R_PAREN: ')';
L_BRACE: '{';
R_BRACE: '}';
L_BRACKT: '[';
R_BRACKT: ']';
COMMA: ',';
SEMICOLON: ';';
IDENT: [_a-zA-Z][_a-zA-Z0-9]*;

fragment FRACTION_CONST: DIGIT* '.' DIGIT+ | DIGIT+ '.';
fragment EXPONENT_CONST: [eE] [+ -]? DIGIT+; //e+10、E-5、e3
fragment Hexadecimal_digit: [0-9a-fA-F]+;
fragment Binary_exponent: [pP] [+ -] DIGIT+;
WS: [ \r\n\t]+ -> skip;
LINE_COMMENT: '//' .*? '\n' -> skip;
MULTILINE_COMMENT: '/*' .*? '*/' -> skip;
fragment DIGIT: '0' .. '9';

FLOAT_CONST:
    FRACTION_CONST (EXPONENT_CONST)?
    | [0-9]+ EXPONENT_CONST
    | '0' [xX] (Hexadecimal_digit)? '.' Hexadecimal_digit Binary_exponent
    | '0' [xX] Hexadecimal_digit '.' Binary_exponent
    | '0' [xX] Hexadecimal_digit Binary_exponent;

INTEGER_CONST:
    ('0' | [1-9] DIGIT*)
    | '0' [0-7]*
    | '0' [xX][0-9a-fA-F]*;

```

至此，构建 BIFNAG 语言词法分析器所需要的前需准备已经完成。需要解释的是

在 ANTLR 的.g4 文件中的 `fragment` 关键字用于定义所谓的"碎片"规则，这些规则是词法规则的一部分，但它们本身不会被任何语法规则（parser rules）直接引用。`fragment` 规则通常用于构建更复杂的词法规则，它们类似于其他编程语言中的私有或内部辅助函数。被 `fragment` 修饰的 `lexer rule` 可以被其他词法规则组合使用，但不能独立产生标记（tokens）。这有助于组织和模块化词法规则，使得词法规则文件更加清晰和易于维护。

3.1.2 编写 ANTLR 语法文件

以下是编写.g4 语法文件的基本规则，.g4 语法文件的结构主要包括：

```
parser grammar Name;
options {...
rule1 // parser rules
...
ruleN
```

其中的语法规则（Parser Rules）定义了标记如何组合成更大的结构。规则由名称、可选的参数、备选分支和子规则组成，例如：

```
statement : ID | INT ;
expression : expression ('+' | '-') expression ;
```

其他文法规则同 Lexer 中介绍的规则相同。

下面是 BIFANG 语言基于 ANTLR 文法编写的语法文件。

```
parser grammar BIFANGParser;
options {
    tokenVocab = BIFANGLexer;
    language = Cpp;
}
program: compUnit;
compUnit: (funcDef | decl)+ EOF;
decl: constDecl | varDecl;
constDecl: CONST bType constDef (COMMA constDef)* SEMICOLON;
bType: INT | FLOAT;
constDef:
    IDENT (L_BRACKT constExp R_BRACKT)* ASSIGN constInitVal;
constInitVal:
    constExp
    | L_BRACE (constInitVal (COMMA constInitVal)*)? R_BRACE;
varDecl: bType varDef (COMMA varDef)* SEMICOLON;
varDef: IDENT (L_BRACKT constExp R_BRACKT)* (ASSIGN initVal)?;
initVal: exp | L_BRACE (initVal (COMMA initVal)*)? R_BRACE;
```

```

funcDef: funcType IDENT L_PAREN funcFParams? R_PAREN block;
funcType: VOID | INT | FLOAT;
funcFParams: funcFParam (COMMA funcFParam)*;
funcFParam: bType IDENT (L_BRACKT R_BRACKT (L_BRACKT exp R_BRACKT)*)?;
block: L_BRACE blockItem* R_BRACE;
blockItem: decl | stmt;
stmt:
    lVal ASSIGN exp SEMICOLON           # assignStmt
    | exp? SEMICOLON                   # expStmt
    | block                           # blockStmt
    | IF L_PAREN cond R_PAREN stmt      # ifStmt
    | IF L_PAREN cond R_PAREN stmt ELSE stmt   # ifElseStmt
    | WHILE L_PAREN cond R_PAREN stmt     # whileStmt
    | BREAK SEMICOLON                  # breakStmt
    | CONTINUE SEMICOLON                # continueStmt
    | RETURN (exp)? SEMICOLON          # returnStmt;

exp:
    L_PAREN exp R_PAREN            # expParenthesis
    | lVal                         # lValExp
    | number                        # numberExp
    | IDENT L_PAREN funcRParams? R_PAREN  # callFuncExp
    | unaryOp exp                  # unaryOpExp
    | exp (MUL | DIV | MOD) exp    # mulExp
    | exp (PLUS | MINUS) exp       # plusExp;

cond:
    exp                         # expCond
    | cond (LT | GT | LE | GE) cond # ltCond
    | cond (EQ | NEQ) cond        # eqCond
    | cond AND cond              # andCond
    | cond OR cond               # orCond;

lVal: IDENT (L_BRACKT exp R_BRACKT)*;
number: FLOAT_CONST | INTEGER_CONST ;
unaryOp: PLUS | MINUS | NOT;
funcRParams: param (COMMA param)*;
param: exp;
constExp: exp;

```

可以看出 stmt 可以有非常多的类型，例如变量定义、if、while 等，这些如果没有进一步区分，解析出来的语法树会很不清晰，需要结合很多的标记才能完成具体语

句的识别，这种情况下，可以使用替代标签完成区分。在.g4 文件的 rule 后如果带有 '#' 号，则 '#' 后面的标签就是替代标签，它用于定义 ANTLR 解析器生成的解析树（AST）的节点类型。每个标签对应规则的一个备选分支，当解析器匹配到该分支时，它会创建一个与替代标签同名的节点，并将其添加到解析树中。

在 stmt 语法规则中有多个备选分支，每个分支后面都有一个 '#' 号和替代标签，这意味着当解析器匹配到 IVal ASSIGN exp SEMICOLON 模式时，它会创建一个 assignStmt 类型的节点；当解析器匹配到 exp? SEMICOLON 模式时，它会创建一个 expStmt 类型的节点，其他情况以此类推。这种机制允许 ANTLR 在生成解析树时提供更多的信息，因为它可以根据匹配的规则和备选分支创建具有特定类型的节点。这对于后续的树遍历和处理非常有用，因为它允许访问者模式（后文将详细介绍）的实现根据节点类型执行不同的操作。

3.1.3 一键生成词法和语法分析器

编写完 BIFANG 语言对应的 Lexer 和 Parser 的.g4 文件后，即可调用 ANTLR Recognizer。在 Windows 平台可以选择在 Clion 中右键.g4 文件，点击“Generate ANTLR Recognizer”生成前端程序，Linux 平台可以在命令行运行：

```
antlr4 -Dlanguage=Cpp BIFANGLexer.g4
antlr4 -Dlanguage=Cpp BIFANGParser.g4
```

生成以 C++ 为目标语言的词法分析器与语法分析器代码。

3.1.4 ANTLR 生成文件剖析

使用 ANTLR 生成代码后，主要存在以下这些文件：

①词法分析器

```
BIFANGLexer
|---- BIFANGLexer.cpp
|---- BIFANGLexer.h
|---- BIFANGLexer.interp
|---- BIFANGLexer.tokens
```

②语法分析器

```
BIFANGParser
|---- BIFANGParser.cpp
|---- BIFANGParser.h
|---- BIFANGParser.interp
|---- BIFANGParser.tokens
|---- BIFANGParserBaseVisitor.cpp
|---- BIFANGParserBaseVisitor.h
|---- BIFANGParserBaseListener.cpp
```

```

|---- BIFANGParserBaseListener.h
|---- BIFANGParserVisitor.cpp
|---- BIFANGParserVisitor.h

```

本文主要关注.cpp 和.h 文件，下面将逐个文件解释：

- BIFANGLexer 类属于词法分析器类，通常以 NameLexer.cpp 和 NameLexer.h 的形式生成，它继承自 antlr4::Lexer 类，Lexer 类提供了词法分析的基本功能。这个类可以重写 Lexer 类的方法也可以包含 Lexer 类的基本方法，例如 nextToken 方法——词法分析器的核心方法，它负责匹配并返回下一个 token；除此以外，BIFANGLexer 类还包含所有 Token 类型的枚举定义、词法分析的逻辑，能够将输入文本分解成一系列的 Tokens。
- BIFANGParser 类属于语法分析器类，通常以 NameParser.cpp 和 NamePaser.h 的形式生成，它继承自 antlr4::Parser 类，Parser 类提供了语法分析的基本功能。这个类可以重写 Parser 类的方法也可以包含 Parser 类的基本方法，例如 match 方法——尝试匹配当前输入符号与指定的 token 类型，BIFANGParser 类包含许多方法，每个方法对应于 BIFANGParser.g4 文件中定义的一个语法规则，例如：program()方法——作为整个解析过程的入口点，它开始解析程序的顶层结构;stmt()方法——语句是构成程序逻辑的基本单元，这个规则负责解析各种类型的语句。
- BIFANGParserVisitor 类和 BIFANGParserBaseVisitor 类都属于访问者类的父类，通常以 NameParserVisitor 和 NameParserBaseVisitor 的形式生成，其中 BIFANGParserBaseVisitor 类继承自 BIFANGParserVisitor 类，重写了父类中的 visitxxx 方法（后文将介绍访问者模式）。BIFANGParserVisitor 类又继承自 antlr4::tree::AbstractParseTreeVisitor 类，AbstractParseTreeVisitor 类中重写了其父类（ParseTreeVisitor 类）的方法，其中存在两个帮助理解访问者模式较为重要的方法：visit 方法和 visitChildren 方法

```

virtual std::any visit(ParseTree *tree) override {
    return tree->accept(this);
}

virtual std::any visitChildren(ParseTree *node) override {
    std::any result = defaultResult();
    size_t n = node->children.size();
    for (size_t i = 0; i < n; i++) {
        if (!shouldVisitNextChild(node, result)) {
            break;
        }
        std::any childResult = node->children[i]->accept(this);
        result = aggregateResult(std::move(result), std::move(childResult));
    }
}

```

```

    }
    return result;
}

```

visit 方法默认实现调用传入树的 accept 方法。这是访问者模式中的一个核心方法，用于开始对解析树的访问过程。visitChildren 方法实现了访问当前节点的孩子节点，并使用 aggregateResult 方法将对解析树中每个子节点的访问结果进行合并。

- BIFANGParserListener 类属于监听器类，通常以 NameParserListener 的形式生成。BIFANGParserListener 类继承自 antlr4::tree::ParseTreeListener 类，定义了对于每个非终结符的 enter 方法和 exit 的虚函数，在其父类中，定义了 enterEveryRule、exitEveryRule、visitTerminal 和 visitErrorNode 虚函数。如果不需要对不同的非终结符采用不同的监听方法，例如对解析树构建过程进行 Debug，那么可以定义自己的监听类直接继承 ParseTreeListener 类，然后重写它的四个虚函数方法即可，例如：重写 enterEveryRule 方法，让访问者在访问任何节点时都打印该节点信息；重写 visitTerminal 方法，让访问者每次访问到终结符都把它打印出来等。

3.2 BIFANG 语法分析框架

3.2.1 从 ANTLR 源码剖析解析树的构建过程

要理解并熟练掌握 ANTLR 功能的强大之处，本文主要关注 BIFANGLexer 类和 BIFANGPaser 类在编译器里建立前端语法分析框架的工作流。

```

if(argc == 1){
    test_file = testcase_path + "/testcase1.bf";
}
else if(argc == 2) {
    test_file = testcase_path + "/" + argv[1];
}
std::ifstream stream(test_file);
assert(stream);
antlr4::ANTLRInputStream input(stream);
BIFANGLexer lexer(&input);
antlr4::CommonTokenStream tokens(&lexer);
BIFANGParser parser(&tokens);
antlr4::tree::ParseTree *tree = parser.program();

```

在 main 函数里面，test_file 是一个字符串变量，它包含了要解析的 BIFANG 源文件的路径。这个路径是通过连接 testcase_path（测试用例的路径）和指定测试文件名来构建的。然后使用 std::ifstream 尝试打开 test_file 指定的文件，并创建一个 ifstream 对象 stream。如果文件成功打开，stream 将能够读取文件内容。断言语句

assert 检查 stream 是否成功打开文件。如果 stream 为 nullptr 或者文件没有成功打开，程序将抛出异常。然后创建了一个 ANTLRInputStream 对象 input，它使用 stream 作为输入源。这个对象将用于提供字符流给 ANTLR 的词法分析器。然后创建了一个 BIFANGLexer 对象 lexer，并将 input 作为参数传递给它。BIFANGLexer 负责将输入的字符流分解成一系列的词法单元，这个 Token 流作为参数传给 BIFANGParser 根据语法规则解析 Token。调用 program() 方法后解析正式开始，解析完成后最终生成一个代表整个程序的 Tree 结构。

解析树包含多种节点，并以枚举类型列出：

```
enum {
    CONST = 1, INT = 2, FLOAT = 3, VOID = 4, IF = 5, ELSE = 6, WHILE = 7,
    BREAK = 8, CONTINUE = 9, RETURN = 10, PLUS = 11, MINUS = 12, MUL = 13,
    DIV = 14, MOD = 15, ASSIGN = 16, EQ = 17, NEQ = 18, LT = 19, GT = 20,
    LE = 21, GE = 22, NOT = 23, AND = 24, OR = 25, L_PAREN = 26, R_PAREN = 27,
    L_BRACE = 28, R_BRACE = 29, L_BRACKT = 30, R_BRACKT = 31, COMMA = 32,
    SEMICOLON = 33, IDENT = 34, WS = 35, LINE_COMMENT = 36, MULTILINE_COMMENT = 37,
    FLOAT_CONST = 38, INTEGER_CONST = 39
};

enum {
    RuleProgram = 0, RuleCompUnit = 1, RuleDecl = 2, RuleConstDecl = 3,
    RuleBType = 4, RuleConstDef = 5, RuleConstInitVal = 6, RuleVarDecl = 7,
    RuleVarDef = 8, RuleInitVal = 9, RuleFuncDef = 10, RuleFuncType = 11,
    RuleFuncFParams = 12, RuleFuncFParam = 13, RuleBlock = 14, RuleBlockItem = 15,
    RuleStmt = 16, RuleExp = 17, RuleCond = 18, RuleLVal = 19, RuleNumber = 20,
    RuleUnaryOp = 21, RuleFuncRParams = 22, RuleParam = 23, RuleConstExp = 24
};
```

这两个枚举为程序提供了一种方便的方式来表示和处理源代码的结构和语义。通过使用这些枚举，程序可以更容易地对源代码进行分析、解释和转换。

```
BIFANGParser parser(&tokens);
antlr4::tree::ParseTree *tree = parser.program();
```

下面本文通过 ANTLR4 的源码，解析上方两行代码构建解析树的执行过程：

```
BIFANGParser::ProgramContext* BIFANGParser::program() {
    ProgramContext *_localctx =
        _tracker.createInstance<ProgramContext>(_ctx, getState());
    enterRule(_localctx, 0, BIFANGParser::RuleProgram);
#if __cplusplus > 201703L
    auto onExit = finally([=, this] {
```

```

#else
    auto onExit = finally([=] {
#endif
    exitRule();
});

try {
    enterOuterAlt(_localctx, 1);
    setState(50);
    compUnit();
}

catch (RecognitionException &e) {
    _errHandler->reportError(this, e);
    _localctx->exception = std::current_exception();
    _errHandler->recover(this, _localctx->exception);
}

return _localctx;
}

```

本文的重点将放在部分函数上：

- BIFANGParser::ProgramContext* BIFANGParser::program() 这是一个成员函数的定义，函数名为 program，它属于 BIFANGParser 类。函数返回一个指向 ProgramContext 对象的指针。ProgramContext 是一个在语法分析过程中用来保存程序相关信息的上下文类。
- ProgramContext * _localctx = _tracker.createInstance<ProgramContext>(_ctx, getState()); 在函数体内，首先创建了一个名为 _localctx 的 ProgramContext 类型的指针变量。_tracker 是一个用来管理语法分析器中所有上下文对象的跟踪器。createInstance 是 _tracker 的一个方法，用来创建一个新的 ProgramContext 实例，它接收当前的上下文 (_ctx) 和当前的状态（通过 getState() 方法获取），该函数的作用就是创建当前规则的上下文。
- enterRule(_localctx, 0, BIFANGParser::RuleProgram); 这行代码告诉语法分析器，系统现在正在进入一个新的规则解析阶段。_localctx 是当前规则的上下文。第二个参数 0 是这个规则在语法分析器中的索引。BIFANGParser::RuleProgram 是这个规则的具体名称。

下面本文详细解释该 enterRule 函数：

```

void Parser::enterRule(ParserRuleContext *localctx, size_t state, size_t
/*ruleIndex*/) {
    setState(state);
    // 将 ctx 设置成当前的解析上下文
    _ctx = localctx;
}

```

```

    _ctx->start = _input->LT(1);
    if (_buildParseTrees) {
        addContextToParseTree();
    }
    //如果有监听器注册
    if (_parseListeners.size() > 0) {
        triggerEnterRuleEvent();
    }
}

```

`if(_buildParseTrees)`:如果成员变量`_buildParseTrees`为真(表示解析器被设置为构建解析树),则执行`addContextToParseTree()`;调用`addContextToParseTree`函数,将当前上下文添加到解析树的孩子节点中(`_ctx`是`ParserRuleContext`类,继承自`RuleContext`类,该类又继承自`tree::ParseTree`类);`if(_parseListeners.size() > 0)`如果有解析事件监听器注册,则执行`triggerEnterRuleEvent()`,触发进入规则的事件,通常系统使用监听者模式是为了Debug,打印构建 AST 的流程。

- 退出`enterRule`函数,系统回到`program`函数

```

        exitRule();
    });
    try {
        enterOuterAlt(_localctx, 1);
        setState(50);
        compUnit();
    }
    catch (RecognitionException &e) {
        _errHandler->reportError(this, e);
        _localctx->exception = std::current_exception();
        _errHandler->recover(this, _localctx->exception);
    }
    return _localctx;
}

```

重点关注`compUnit()`函数,该函数之所以会出现在`program()`函数里面,是因为在本文的`BIFANGParser.g4`文件中定义了“`program: compUnit;`”这样的语法规则,这是因为ANTLR读取了系统的语法规则,因此Parser过程是根据本文自定义的语法规则进行遍历的。

- 在`compUnit()`函数里,按照`program()`函数同样的流程,首先是创建新的上下文环境并使用`localctx`指针指向,然后调用`enterRule`,将`_ctx`设置成当前的解析上下文,然后调用`addContextToParseTree()`函数将当前上下文添加到解析树的孩子节点中。

- 接下来是一个 switch 语句，原因是 BIFNAG 语法规则中 compUnit 的向下推导式是 “compUnit: (funcDef | decl)+ EOF;” ，因此会有 funcDef 和 decl 两种备选分支，而这个选择功能是由 adaptivePredict 函数实现的，它的作用是根据当前的输入流和解析器状态，来预测下一个标记可能属于哪个备选分支，这个 adaptivePredict 函数是解析器在语法分析过程中进行决策的核心部分，它使用 DFA（确定性有限自动机）和 ATN（增强的转换网络）来预测哪个备选分支可能是下一个正确的选择，这种预测机制可以提高解析的效率，尤其是在解析复杂或模糊语法时。

```

do {
    ...
    switch (getInterpreter<atn::ParserATNSimulator>() ->adaptivePredict(_input,
    0, _ctx)) {
        case 1: {
            setState(52);
            funcDef();
            break;
        }
        case 2: {
            setState(53);
            decl();
            break;
        }
        default:
            break;
    }
    ...
    match(BIFANGParser::EOF);
}while(...);

```

- 如果进入 funcDef () 紧接着就会进入 “funcDef: funcType IDENT L_PAREN funcFParams? R_PAREN block;” 这个分支的规则，如果进入 decl () 紧接着就会进入 “decldecl: constDecl | varDecl;” 这个分支的规则，以此类推。语法解析树就顺着语法规则逐步构建起来。

3.2.2 基于访问者模式的解析树遍历

至此，语法解析树已经建立，它在本系统中保存在命名为 “tree” 的根节点中。如果系统要基于这棵解析树做前端优化、IR 生成、代码生成等工作，就需要一种简单且高效的遍历方法，好在 ANTLR 在构建解析树的时候就考虑到了这个问题，因为 ANTLR 提供了一种基于访问者模式遍历解析树的方法。

访问者模式（Visitor Pattern）是一种行为设计模式，它允许你在不修改已有类的

代码的前提下，定义新的操作。这种模式特别适用于操作一个复杂对象结构，如解析树或者对象集合。

在访问者模式中，有两个核心概念：元素（Element）和访问者（Visitor）。元素是构成对象结构的组件，在本系统中就是指解析树上的每个节点(所有继承自 tree::ParseTree 的类)，它们都重写了 tree::ParseTree 中的 accept 方法，该方法接受一个访问者对象作为参数；对象结构中的每个元素都可以被访问者访问。访问者是一个接口或抽象类，它定义了一系列访问元素的方法，每个方法对应一种元素类型。具体的访问者类实现了这些方法，以定义对各种元素的具体操作。

访问者模式的工作原理是，当访问者被传递给元素的 accept 方法时，元素会调用访问者中对应的访问方法，并将自己作为参数传递。这样，访问者就可以访问元素的内部状态，并执行特定的操作。

在生成解析树后，本系统中定义了一个名为“BIFANGVisitor”的类，它是本系统中解析树的访问者，它继承自“BIFANGParserBaseVisitor”类，“BIFANGParserBaseVisitor”是由 ANTLR 生成的，因为 BaseVisitor 是 ANTLR 按照词法和语法规则生成的，所以在其头文件中定义有许多访问每一个语法规则的虚函数，例如：

```
virtual std::any visitProgram(BIFANGParser::ProgramContext *ctx) override
{
    return visitChildren(ctx);
}

virtual std::any visitCompUnit(BIFANGParser::CompUnitContext *ctx) override
{
    return visitChildren(ctx);
}

virtual std::any visitDecl(BIFANGParser::DeclContext *ctx) override
{
    return visitChildren(ctx);
}

.....
```

只要是在语法规则中的一个非终结符，ANTLR 都会默认为其生成这样的 visit 方法来访问它。并且生成的这些方法它们都是虚函数，默认是不做任何操作，直接返回访问其孩子节点的结果，也就是说“BIFANGVisitor”可以在类中重写这些方法，来自定义访问每个节点的操作（例如构建 IR）。

需要注意的是，本文在“3.1.2 编写 ANTLR 文法文件”中提到——在语法规则文件中，如果某条 parser rule 后面带有‘#’，那么‘#’后面的标签就是替代标签，它用于对应规则的一个备选分支，当解析器匹配到该分支时，它会创建一个与替代标签同名的节点，并将其添加到解析树中。因此，ANTLR 在解析这些规则时会因不同备选分支的替代标签名称不同而生成不同的 visitxxx 方法。例如对于如下这个语法规则：

```
cond:
    exp                                # expCond
    | cond (LT | GT | LE | GE) cond      # ltCond
    | cond (EQ | NEQ) cond               # eqCond
```

```

| cond AND cond          # andCond
| cond OR cond           # orCond;

```

该语法规则对应五个备选分支，因此会生成 visitExpCond、visitLtCond、visitEqCond、visitAndCond、visitOrCond 总共 5 种方法，其他语法规则（stmt、exp）以此类推。

BIFANGVisitor 类是本系统中的访问者，在其内部重写了其父类“BIFANGParserBaseVisitor”中的虚函数方法，下方两行代码表示创建 BIFANGVisitor 实例，然后调用其 visit 方法访问解析树的根节点。

```

BIFANGVisitor* visitor = new BIFANGVisitor();
visitor->visit(tree);

```

需要解释的是，BIFANGVisitor 本身没有重写 visit 方法，而是调用其基类的 visit 方法。

```

virtual std::any visit(ParseTree *tree) override {
    return tree->accept(this);
}

```

该方法接受一个 ParseTree 的对象，this 指针是一个特殊的指针，它指向当前对象的地址，在成员函数中，this 用于指代调用该成员函数的对象实例，在这里 this 指向的是 BIFANGVisitor 类的实例 visitor，由于 tree 是 ProgramContext 类，因此 visitor->visit(tree) 实际调用了 ProgramContext 类（antlr4::tree::ParseTree 是其基类）的 accept 方法，该方法是 ProgramContext 类对 antlr4::tree::ParseTree 类中 accept 方法的重写。同样的道理，每种节点都对这个基类中的 accept 方法进行了重写，从而自定义了访问者访问自身要进行的操作。

本文以 CompUnitContext 的 accept 方法为例子，解释 visitor 是如何基于访问者模式仅一次 visitor->visit (tree) 调用就能访问整棵解析树的：

1. visitor->visit (tree) tree 是被 ParseTree 指针指向的其子类的实例，因此实际是调用了其子类的 accept 方法，本例是调用 ProgramContext 的 accept。
2. ProgramContext 的 accept 方法内部调用了 BIFANGVisitor 的 visitProgram 方法。
3. BIFANGVisitor 的 visitProgram 方法内调用了 BIFANGParserBaseVisitor 的 visitProgram 方法，该方法调用了 AbstractParseTreeVisitor 的 visitChildren 方法。需要说明的是，visitProgram 通过虚函数实现了多态，所以内部实现不同。
4. AbstractParseTreeVisitor 的 visitChildren 方法实现了遍历当前节点的所有孩子节点，并对每个孩子节点都会调用 accept 方法。由于语法规则规定 “program: compUnit;”，所以对于 program 节点，只有 compUnit 一个孩子节点。
5. 因为当前节点的每个孩子都调用了 accept 方法，因此回到步骤 1，以此类推，直至访问者访问完这颗解析树的所有节点。

第 4 章 BIFANG 编译系统后端设计与实现

本章主要介绍 BIFNAG 编译系统后端完成的工作，包括 BIFANG 编译系统中端使用的中间代码数据结构，中间代码生成器模块的设计与实现，目标机器架构介绍以及目标代码生成。首先通过对前端生成的解析树进行二次遍历，根据相应的规则产生相应的中间代码，根据中间代码输出生成汇编代码。主要解决了四个问题：第一是 BIFANG 编译系统采取什么组织形式的中间代码，第二是如何基于访问者模式遍历前端建立的解析树去生成中间代，第三个是目标机器是什么指令集架构，第四是如何生成目标机器代码。

4.1 中间代码介绍

中间代码通常被称为 IR，它是高级语言在编译过程中用于表示源代码的中间形式。其抽象层次通常位于源代码和目标代码之间，在编译过程中可以用来做程序分析、优化等。中间代码虽然更加抽象和简洁，但是它仍保留了源程序中的结构和语义信息。

中间代码的实现形式多种多样，包括语法树、三地址代码等。本系统使用一种类似 LLVM IR 的混合代码，它相对于 LLVM IR 更加简洁，保留了函数、程序基本块、指令等关键数据结构，其在结构上非常灵活，表示形式与汇编代码有共同之处，在源代码和目标代码之间可以进行有效转换，对于编译器后端实现目标代码的生成有重要的作用。

4.1.1 中间代码数据结构

BIFNAG 语言编译系统的中间代码形式参考较为灵活的 LLVM IR 形式，本小节主要介绍最为关键的几种数据结构。

- Type 类。Type 类是 BIFANG 编译系统的中间代码表示里最为基础的类，在其内部定义了枚举类型 “Type_Enum” 包括：VOIDTYPE、INT32TYPE 等，用于表示变量类型等。Type 类包含两个成员变量：一个 Type_Enum 类型的 “type”，一个整型 size 表示内存占用大小。另外，存在 FuncType、ArrayType、PointerType 三个类都继承自 Type 类，这三个类在其内部都定义了各自特有的成员变量，例如 FuncType 定义了 vector<Type*> 类型的成员变量用于存储函数的参数类型；Type* 类型的成员变量用于表示函数的返回值类型。
- Value 类。值也是 BIFANG 编译系统的中间代码表示里最为基础的类，它是一种抽象概念，表示数据或者计算结果。在其内部定义了值的类型（例如：IntConst、IntVar 等）以及值的名称。存在多种值的衍生类，它们各自内部定义了特殊的数据结构以及重写了 Value 的纯函数，例如 Bool_Const 中含有 bool 类型的成员变量、Float_Const 中含有 double 类的 value 等。另外，常量、变量、参数、基本块等都

继承自 Value 类，因此值可以是指令、常量、全局变量、参数等。

- symbol 类。symbol 类表示一个符号，可以把它理解成一个标识符 Ident，因此他可以表示函数、变量、常量、数组等。它继承自 Value 类，因此存在名称 (name) 和类型(type)成员变量，同时还包含 is_const、is_global 等特有的成员变量。
- Instruction 类。指令是 BIFANG 中间代码的原子操作，如加法(ADD)、乘法(MUL)、比较(EQ)等。每个指令都有一个操作码 (opcode) 和零个或多个操作数(Operands)，操作数中又分为目的操作数和源操作数等。在 Instruction.h 中，定义了枚举类型 InstType_Enum，包含 “LOAD”、“STORE” 等基本操作码类型。因此 Instruction 类拥有一个此类型的成员变量用来表示指令类型，另外 Instruction 类中有 outPut、codegen 这样的虚函数，便于子类重写。
- IRInstruction 类。该类继承自 Instruction 类，其内部包含了所属基本块、寄存器的计数器、浮点数常量表。定义了一些生成 IR 的函数，例如函数返回用于恢复栈帧和栈顶的 IR 生成被包装在 retFunc 函数中，类似的例子有很多。IRInstruction 类有许多衍生类，例如 StoreInstruction 类，LoadInstruction 类，这些类之间的区别是指令类型不同以及重写的 codegen 和 output 函数定义内容不同。
- BasicBlock 类。BasicBlock 也叫基本块，该类继承自 Value 类，其内部含有丰富的成员变量，包括：装载 Instruction 类的 vector 表示该基本块内包含的指令，该基本块所属的函数、基本块对应的标签等。基本块是一系列有序的指令，它们在控制流图中形成一个节点。基本块没有内部跳转，只有一个入口点和一个退出点，通常以终结指令（如 ret、br 等）结束。
- Function 类。函数是模块中的基本执行单元，它包含了一系列的基本块。每个函数都有一个唯一的名称，并且可以有参数列表、返回类型和一系列局部变量，局部变量保存在 Function 的符号表中。
- GlobalUnit 类。GlobalUnit 类是本编译系统中最高等级的表达结构，也可以理解为表示范围最广的结构，通常一个源程序就是一个 GlobalUnit。其内部包含了许多的函数和全局变量，并以函数名为 Key，函数块为 Value 的表存储函数，以全局符号名称为 Key，符号为 Value 的表存储全局变量。
- Scope 类。作用域定义了符号的可见性和生命周期。在一个作用域内定义的符号，在该作用域内是可见的。本系统考虑到作用域的嵌套关系，为方便管理作用域嵌套，实现了作用域类，每个作用域类都有一个符号表，一个指向上一级作用域的指针，作用域的嵌套关系是使用链表来实现的，从链表尾节点开始是最外层作用域（全局符号表在 GlobalUnit 中单独实现），越靠近头节点，作用域嵌套越深。
- xxxContext 类。这个类代表了 BIFANG 编译系统在编译工作时的一个上下文，它是 ANTLR 基于语法规则文件生成的，该类继承自 ParseTree，伴随着整个编译过程。在语法规则文件中，每种非终结符都对应着一种 xxxContext 类，其内部包含

了返回该语法推导式右侧所有成员的函数以及编译工作中的部分数据，例：对于“funcDef: funcType IDENT L_PAREN funcFParams? R_PAREN block;”这个语法推导式，存在一个由 ANTLR 生成的 FuncDefContext 类，其类中包含了一个返回 FuncTypeContext* 类型的函数 funcType()；返回 Node* 类型的 IDENT()、L_PAREN()、R_PAREN()；一个返回 BlockContext*类型的函数 block(); 一个返回 FuncFParamsContext*的函数 funcFParams()。同理，其他 xxxContext 类的成员函数以此类推。

4.1.2 中间代码的组织关系



在这个层次结构中，编译单元是最顶层的模块，它包含一组函数，一张全局符号表。每个函数又包含一组基本块，每个基本块包含一组指令。每条指令有一个或多个类型和值，每个值都有一个符号表示。作用域是在编译单元级别定义的，它决定了符号的可见性和生命周期。Context 是编译系统在编译时的一个上下文，包含了编译工作中的一些数据，比如各个语法用到的变量及非终结符等。

当我们把语法解析树读入，就能在内存层面建立起 IR 的对象模型，可以简化为如图 4-1 的形式，对象模型是 BIFANG 编译系统运行时的核心。

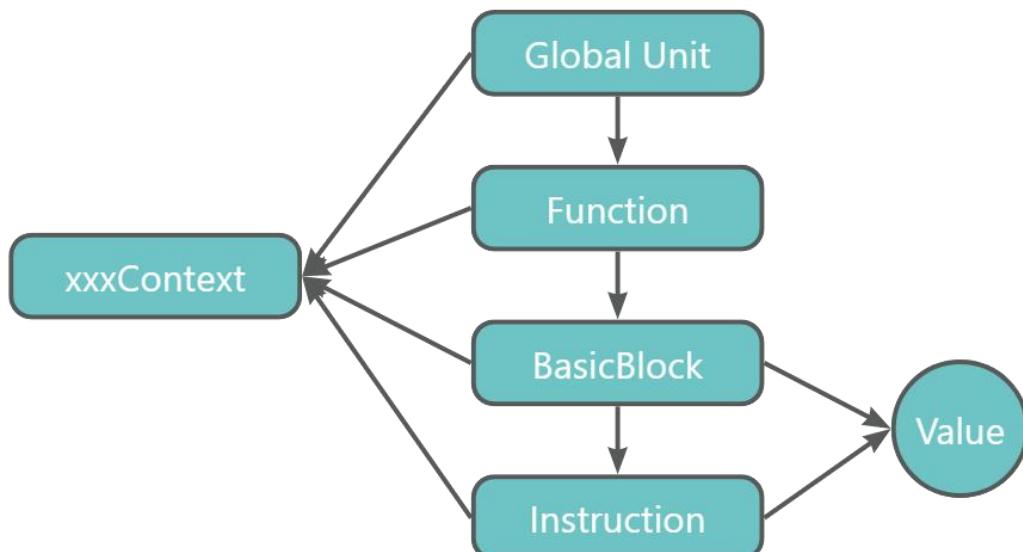


图 4-1 中间代码的对象模型

4.2 中间代码生成

中间代码数据结构看似错综复杂，实际各结构之间存在着密不可分的关系，中间代码的生成工作主要是根据前端生成的解析树，基于访问者模式二次扫描生成的，图 4-2 展示了实现访问者模式的类图。

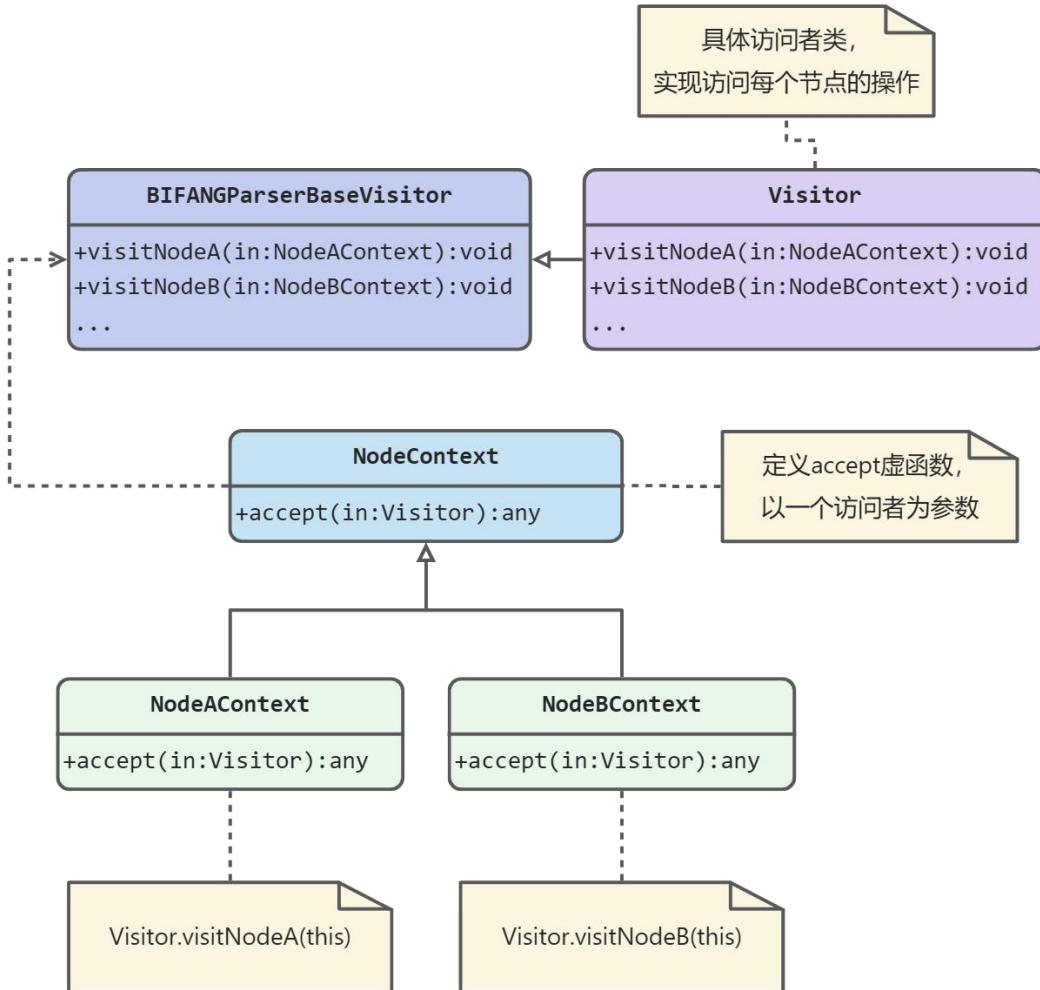


图 4-2 实现访问者模式的类图

基于访问者模式，访问者类即可根据访问到的解析树节点，调用对应的访问方法，进而构造中间代码。

4.2.1 中间代码生成依赖的工具类

中间代码生成的工作除了中间代码本身需要使用上节提到的数据结构，还需要有构造中间代码的构建器，命名为 **Builder** 类。**Builder** 类主要负责基本块的构建和基本块的切换功能。同时，它可以记录下当前作用的基本块，在生成 IR 指令的时候，通过 `builder` 向对应基本块中添加指令。另外，`builder` 可自动将基本块追加到对应函数的 `block_list` 中，便于构建 `function` 包含 `block` 的嵌套结构。**Builder** 类包含三个成员变量，一个指向当前基本块的指针，一个指向当前函数的指针，一个指向全局单元 (`globalUnit`)的指针。**Builder** 类提供 5 个函数，其中需要着重解释的是 `AppendCode` 方

法，该方法会接受一个指向 IRInstruction 类的指针，如果当前位置没有在任何基本块中，即在全局，则把指令添加到全局单元的全局指令容器中，否则就添加到相应的函数中，如果是 ALLOCA 类型的指令则把该指令添加到该函数的 EntryBlock 中，否则添加到当前块中。

Builder 负责在访问解析树过程中直接生成内存层面的指令，那么在构造各种类型的指令时需要一个工具类来集成构造指令的函数，把它命名为 LLVM 类。该类中定义了构造各种指令的函数，包括：

- 二元操作符指令构造的函数。这些函数用于构建一些基本的二元操作指令，如加法、减法、乘法、除法和取模等。
- 一元操作符指令构造的函数。这些函数用于构建一些基本的一元操作指令，如取反、逻辑非、零扩展、整数转浮点数、浮点数转整数等。
- 变量声明与存取指令构造的函数。这些函数用于构建变量声明和存取相关操作的指令，包括分配局部变量、添加全局变量、加载和存储操作等。
- 与或操作指令构造的函数。这些函数用于构建一些基本的按位与以及按位或操作指令。
- 比较操作指令构造的函数。这些函数用于构建一些基本的比较操作指令，包括等于、不等于、小于、大于、大于等于等，同时还有一个用于统一调用比较操作的函数 BuildCmp。
- 函数调用和返回操作指令构造的函数。这些函数用于构建函数调用和返回操作的指令。
- 跳转操作指令构造的函数。这些函数用于构建分支控制相关操作的指令，包括无条件分支和条件分支。
- 数组操作指令构造的函数。这些函数用于构建与数组操作相关操作的指令，包括获取数组元素指针、获取数组元素索引等。

4.2.2 中间代码生成实例分析

下面本文通过一个 func() 函数来解释如何生成中间代码。

```
int fun(int a, int b){
    return a+b;
}

int main(){
    .....
}
```

当编译器调用 visitor->visit(tree) 时，根据访问者模式，tree 会调用其 accept 方法，即调用了 CompUnitContext::accept(visitor)，该函数在其内部调用了 Visitor::visitCompUnit(CompUnitContext *ctx)，该函数在内部调用 visitChildren 方法，

根据“compUnit: (funcDef | decl)+ EOF;”可知，compUnit 的孩子节点都调用了一遍自己的 accept 方法。同理，对于 funcDef 调用 accept 是调用了 visitFuncDef(FuncDefContext *ctx)，本文重点分析该函数。

第一步，先处理函数头和函数返回值。通过 ctx->IDENT()->getText()拿到函数名“fun”，通过 ctx->funcType()->getText()拿到函数返回值类型“int”。通过 ctx->funcFParams()->funcFParam(i)->bType()->getText()拿到每个参数类型，即“int”和“int”，至此，即可构建一个函数模块：

```
Function* function = globalUnit->addFunc(funcName, retType, paramsTypes);
```

第二步为该 Function 内部创建两个基本块（Start 块和 Entry 块）并依次添加到 Function 模块的 Block 容器中：

```
string startName = funcName + "Start";
BasicBlock* startBlock = builder->AppendBasicBlock(curFunc, startName);
curFunc->entry = startBlock;
string entryName = funcName + "Entry";
BasicBlock* entryBlock = builder->AppendBasicBlock(curFunc, entryName);
```

第三步根据 paramType 把函数参数创建出来并添加到作用域和符号表中：

```
//拿到函数参数 "%1" 和 "%2"
argVal = new Int_Var()
argVal->get_Ref()
//向 curScope 创建并添加一个变量
Symbol* symbol = curScope->addVar(builder, paramType, paramName, false,
argVal);
//向 curFunc 或全局中创建并添加一个变量
globalUnit->addSymbol(curFunc, symbol->name, symbol);
//生成函数头的中间代码表示
std::string funcDef1 = "define " + retType->getTypeName() + "@" + funcName +
 "(" + args + ")" + " {\n";
```

第四步新建并进入作用域，访问函数体。根据“funcDef: funcType IDENT L_PAREN funcFParams? R_PAREN block;” visitFuncDef 函数在此处调用了 visitBlock(ctx->block()) 来访问函数体 block，根据“block: L_BRACE blockItem* R_BRACE;” visitBlock 里会调用 visitBlockItem(BlockItemContext *ctx)。同理，根据“blockItem: decl | stmt;”和“stmt:| RETURN (exp)? SEMICOLON”该函数会调用 visitReturnStmt(ReturnStmtContext *ctx)。又根据语法“exp: | exp (PLUS | MINUS) exp”，该函数会调用 visitPlusExp，而 visitPlusExp 函数的主要功能是分别计算加号两边对应的两个操作数：

```
auto* lhs = any_cast<ValueRef*>(BIFANGVisitor::visit(ctx->exp(0)));
auto* rhs = any_cast<ValueRef*>(BIFANGVisitor::visit(ctx->exp(1)));
```

```
| res = BuildAdd(builder, lhs, rhs);
```

进一步根据“exp: | IVal”和“ IVal: IDENT (L_BRACKT exp R_BRACKT)*;”，visitPlusExp 函数会调用 visitLVal(LValContext *ctx)，其通过 ctx->IDENT()->getText();拿到符号的名称“a”和“b”，然后在当前作用域的符号表通过符号名称查找符号“a”和“b”并返回该标识符赋值给 lhs 和 rhs。拿到两个操作数后回到 visitPlusExp 函数，通过“BuildAdd(builder, lhs, rhs)”构建了一个操作数为 ADD 的指令并返回 Value。继续返回到 visitReturnStmt 函数中，现在该函数已经拿到了返回值，还需要构建 Store 指令，将返回值存储到 retVal 这个符号中，最后构建跳转指令 BuildBr(builder,retBlock);跳转到函数的返回块。值得注意的是，如果 res 类型是浮点型而函数返回值类型是整型，则需要对该浮点数进行截断再返回；如果 res 类型是整型而函数返回值类型是浮点型，则需要进行类型转换。

第五步，回到 visitFuncDef 函数中，系统已经通过 visitBlock(ctx->block())访问了函数体并创建了相应的指令，现在把返回值的变量也创建出来，并把他加进作用域和符号表，同时在 startBlock 的末尾构建跳转到 entryBlock 的无条件分支指令，然后构建 Load 指令把函数的返回值加载出来，紧接着在返回基本块的末尾构建 Ret 指令。

第六步恢复环境。至此，该函数的中间代码生成完毕，还需将生成的 RetBlock 插入函数的 block 列表，然后恢复上一级作用域，重置 curFunc 和 nowBlock 指针。

生成的中间代码的嵌套结构如图 4-3 所示：

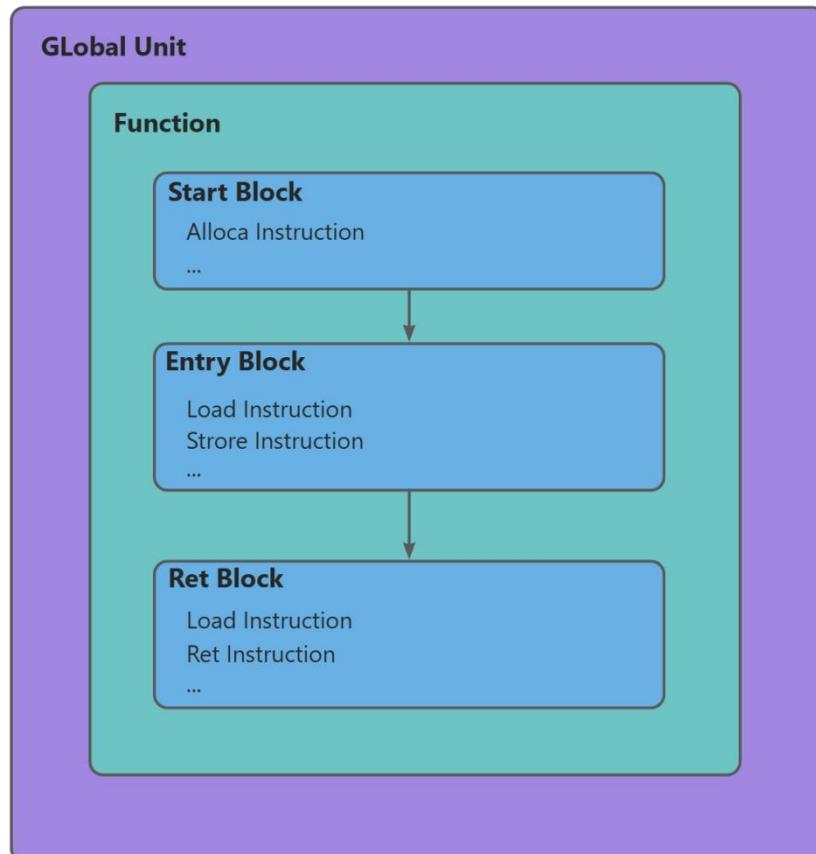


图 4-3 中间代码的嵌套结构

生成的中间代码详细描述为：

```
define i32 @main(i32 %0, i32 %1) {
    mainStart1:
        %RetVal3 = alloca i32, align 4
        %a4 = alloca i32, align 4
        %b5 = alloca i32, align 4
        br label %mainEntry2

    mainEntry2:
        store i32 0, i32* %RetVal3, align 4
        store i32 %0, i32* %a4, align 4
        store i32 %1, i32* %b5, align 4
        %2 = load i32, i32* %a4, align 4
        %3 = load i32, i32* %b5, align 4
        %4 = add i32 %2, %3
        store i32 %4, i32* %RetVal3, align 4
        br label %mainRet0

    mainRet0:
        %5 = load i32, i32* %RetVal3, align 4
        ret i32 %5
}
```

至此 BIFANG 编译系统就能在计算机中建立起源程序对应的中间代码内存模型了。

4.3 目标机器架构介绍

本节主要完成 BIFANG 编译系统的后端部分目标机器架构以及目标代码生成工作的介绍，首先介绍了目标机器的 RISC-V 指令集和目标机器的寄存器，然后介绍了 BIFANG 编译系统实际目标机器的环境与参数。

4.3.1 RISC-V 指令集架构简介

指令集是处理器体系结构设计的重点部分之一，作为软硬件的接口，它紧密连接软件和硬件。RISC-V 指令集采用模块化的设计，即设计一个最小集合和最基础的指令集，这个最小的指令集可以完整地实现一个软件栈，其他具备特别功能的指令集就可以在这个最小的指令集的基础上通过模块间组合来实现。

在不考虑压缩扩展指令的情况下，RISC-V 的指令每条宽度为 32 位，包括 RV32 指令集和 RV64 指令集。其指令格式大概分为 6 类：

1. R 类型：寄存器与寄存器算术指令
2. I 类型：寄存器与立即数算术指令或加在指令
3. S 类型：存储指令

4. B 类型：条件跳转指令
5. U 类型：长立即数操作指令
6. J 类型：无条件跳转指令

六种指令类型具有不同的指令集编码格式，这里不展开阐述。

代码指令一般通过各种寻址方式来获取常量、变量的内存地址，进而访问它们所存储的数据。如表 4-1 展示了 RISC-V 常用的寻址方式。

表 4-1 RISC-V 常用的寻址方式

寻址模式	解释	例子
立即数寻址	操作数是指令本身的一部分	addi x5, x6, 20
寄存器寻址	操作数存放在寄存器中，指令中指定访问的寄存器从而获取该操作数	add x5, x6, x7
基址寻址	操作数在内存中，指令中通过指定寄存器（基址 base）和立即数（偏移量 offset），通过 base + offset 的方式获得操作数在内存中的地址从而获取该操作数	sw x5, 40(x6)
PC 相对寻址	在指令中通过 PC 和指令中的立即数相加获得目标地址的值	beq x5, x6, 100

下面本文将介绍 RV64I 的几种基础指令的格式以及一种扩展指令集。在下方的指令中，rd 表示目标寄存器，rs1 表示源寄存器 1，rs2 表示源寄存器 2，(rs1) 表示以 rs1 寄存器的值为基地址进行寻址，offset 表示以源寄存器的值作为基地址的偏移量，值得注意的是，因为 I 类型和 S 类型指令其指令编码中规定立即数 imm 在指令编码中占用 12 位，且立即数大部分是符号扩展的，所以 offset 也是 12 位的有符号数，其取值范围是[-2048,2047]，如果超出这个范围，则需要把 offset 装入寄存器再进行后续运算。这也是本系统在加载和存储指令中增加了 offset 大小判断操作的原因。

(1) 加载指令

与其他的 RISC 架构指令集一样，RISC-V 体系结构也基于加载和存储的体系结构设计理念。在这种指令集架构下，所有的数据处理与计算都需要在通用寄存器中去完成，而不能在内存中直接完成。

```
| l { d | w | h | b } { u } rd, offset( rs1 )
```

其中{d|w|h|b}表示加载的数据宽度，{u}是可选项，表示加载的数据位无符号数，即采用零扩展的方式。如果没有这个选项，则表示加载的数据为有符号数，即采用有符号扩展方式。

(2) 存储指令

```
| s { d | w | h | b } rd2, offset( rs1 )
```

(3) 位操作指令

```
and rd, rs1, rs2
or rd, rs1, rs2
....
```

(4) 算术指令

```
add rd, rs1, rs2
sub rd, rs1, rs2
....
```

(5) 比较指令

```
slt rd, rs1, rs2
....
```

(6) 无条件跳转指令

```
jal rd, offset
....
```

(7) 条件跳转指令

```
beq rs1, rs2, label
blt rs1, rs2, label
....
```

本系统涉及到的扩展指令集——单精度浮点数扩展指令集。RISC-V 的单精度浮点数扩展指令集，也被称为 F 扩展或 FPU（浮点处理单元），是 RISC-V 指令集架构的一部分，专门用于处理单精度浮点数运算。这个扩展提供了一组丰富的指令，用于执行各种浮点算术和逻辑操作。以下是一些核心的单精度浮点数扩展指令及其功能：

(1) 浮点加法：

```
fadd.s rd, rs1, rs2
```

(2) 浮点减法

```
fsub.s rd, rs1, rs2
```

(3) 浮点除法

```
fdiv.s rd, rs1, rs2
```

(4) 浮点最小值/最大值

```
fmin.s rd, rs1, rs2
fmax.s rd, rs1, rs2
....
```

4.3.2 RISC-V 寄存器简介

通用寄存器是一组用于存储数据、地址或其他信息的寄存器，它们可以被多种指令直接访问和操作，因其提供了临时存储和数据交换的机制，所以对于程序执行非

常重要。64 位的 RISC-V 体系结构提供 32 个 64 位的整型通用寄存器，称为 x 寄存器集，分别是 x0-x31 寄存器，它们是 32 位 RISC-V 架构中的寄存器集的扩展。在 64 位模式下，每个寄存器都是 64 位宽。对于浮点数运算，64 位的 RISC-V 体系结构也提供 32 个浮点数通用寄存器，分别是 f0-f31 寄存器，图 4-4 展示了 RV64 的 32 个整型通用寄存器。

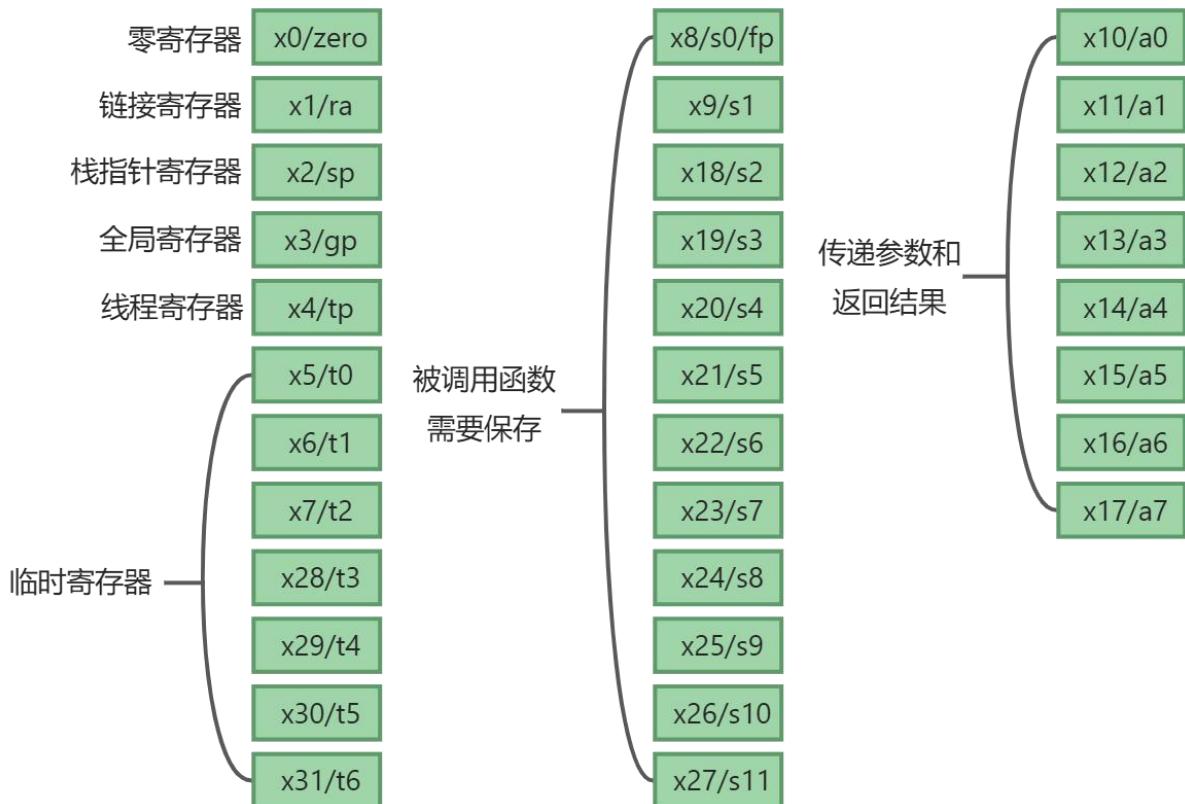


图 4-4 RISC-V 整型通用寄存器

RISC-V 的通用寄存器通常具有别名和特殊用途，在汇编指令中可以直接使用它的别名：

x0 寄存器的别名是 zero。寄存器的内容全是 0，既可以用作源寄存器，也可以用作目标寄存器。

x1 寄存器的别名是 ra——链接寄存器，用于保存函数的返回地址。

x2 寄存器的别名是 sp——栈指针寄存器，指向栈的地址。

x3 寄存器的别名是 gp——全局寄存器，用于链接器松弛优化。

x4 寄存器的别名是 tp——一线程寄存器，通常在操作系统中保存指向进程控制块 task_struct 数据结构的指针。

x5-x7 以及 x28-x31 寄存器是临时变量寄存器，它们的别名分别是 t0-t6。

x8-x9 以及 x18-x27 寄存器的别名分别是 s0-s11。如果在函数调用过程中要使用这些寄存器，则需要主动先把其中的内容保存到栈里。另外 s0 寄存器可以用作帧指针 (FP)。

x10-x17 寄存器的别名分别为 a0-a7，在函数调用时用来传递参数和返回值。

除了上面提到的通用寄存器以外，RISC-V 还定义了许多系统控制和状态寄存器（CSR），可以通过访问和设置这些系统寄存器来完成对处理器不同的功能配置，例如 MSTATUS、MISA、MTVEC 等系统寄存器，CSR 允许软件控制处理器的各种特性和行为，例如中断使能、定时器设置、电源管理等，这里不展开说明，具体内容可以参考 [RISC-V Specifications](#)。

4.3.3 眇·星光（VisionFive）设备简介

眇·星光是一款基于 RISC-V 架构的处理器，由上海赛昉科技有限公司开发。它旨在为各种应用提供高性能和高能效的解决方案，包括但不限于边缘计算、智能物联网、人工智能和自动驾驶等领域。其支持 Linux 操作系统 Fedora，且眇·星光完全开源，拥有开源软件、开源硬件设计和 RISC-V 开源架构。

眇·星光搭载 RISC-V SiFive U74 双核 64 位 RV64GC ISA 的芯片平台（SoC）及 8 GB LPDDR4 RAM，具有丰富的外设 I/O 接口，包括 USB 3.0、40-Pin GPIO Header、千兆以太网连接器、Micro SD 卡插槽等，图 4-5 展示了其外观视图。

眇·星光具有神经网络引擎和 NVDLA 引擎，提供丰富的 AI 功能；眇·星光不仅具有板载音频和视频处理功能，还具有用于视频硬件的 MIPI-CSI 和 MIPI-DSI 接口。眇·星光支持 Wi-Fi 和蓝牙无线功能，兼容大量软件，包括提供对 Fedora 的支持，表 4-2 展示了眇·星光外设规格的详细信息。

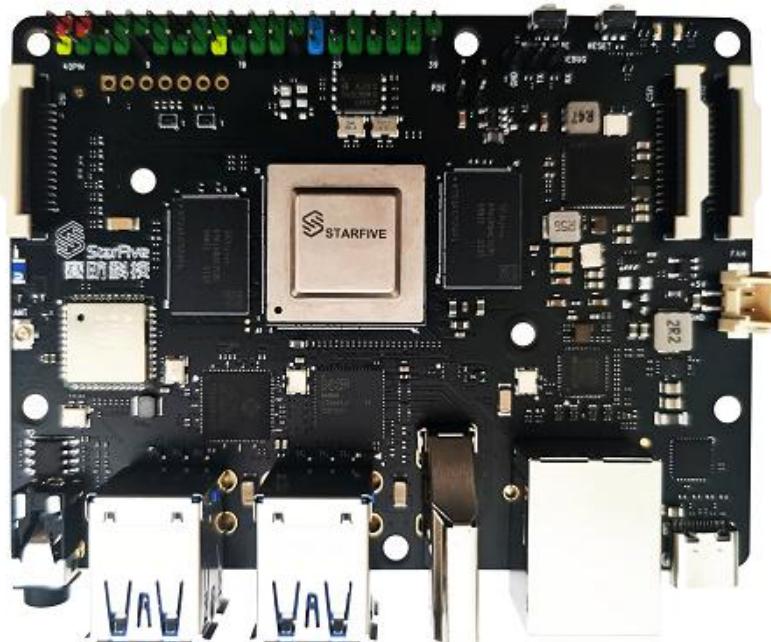


图 4-5 眇·星光外观顶部视图

表 4-2 眺·星光规格说明

规格	详细规格
处理器	<ul style="list-style-type: none"> RISC-V SiFive U74 双核 64 位 RV64GC ISA 芯片平台 (SoC) 搭载 2MB L2 缓存 @ 1.0GHz 用于计算机视觉的 Vision DSP Tensilica-VP6, 600MHz NVDLA 引擎 (配置 2048 Mac, 800MHz) 神经网络引擎 (1024MACs, 500MHz)
内存	8 GB LPDDR4
无线连接	<ul style="list-style-type: none"> 2.4 GHz Wi-Fi (IEEE 802.11b/g/n) 蓝牙 4.2 (BLE)
外设	<ul style="list-style-type: none"> 4 x USB 3.0 千兆以太网连接器 适用于操作系统和数据存储的 Micro SD 卡槽 支持 DMAC、QSPI 及其它外设 1 x Reset 按钮及 1 x BOOT 按钮
操作系统	64 位 Fedora 33
汇编和链接器	gcc version 12.2.0 (Debian 12.2.0-14)
电源	<ul style="list-style-type: none"> 最低: 5 V / 1.5 A 推荐: 5 V / 3 A
电源接口	USB Type-C 接口或 40-Pin GPIO Header
尺寸	100 mm x 72 mm

眺·星光设备搭载 RV64GC(RV64IMAFDC) ISA，即其支持整数指令 I、整数乘除 M、单精度浮点 F、双精度浮点 D、原子指令 A、压缩指令 C。具体扩展指令可以查看 [RISC-V Specifications](#)。

4.4 目标机器代码生成

编译器的代码生成阶段是将优化后的高级中间表示转换成目标机器代码的过程。目标代码生成是编译器最终目标，也是最后的任务，它的目的是生成尽可能高效和优化的机器代码，以便在目标硬件上运行。

4.4.1 指令选择

指令选择是编译器后端的一个关键阶段，它负责将高级中间表示中的操作转换为目标机器的机器指令。BIFANG 系统的指令选择采用简单的基于静态规则的指令选择，这种策略下，指令选择的静态规则可以被看作是一组预定义的映射，每个 IR 操作都有一个明确对应的目标机器指令来实现它。这种方法简化了指令选择过程。该选择过

程实际就是程序上对应 IR 的规则匹配过程，该静态的匹配规则在 IRInstruction 的 codegen 中定义。

例如，表 4-3 展示了加法操作符运算的指令选择规则。

表 4-3 加法运算的指令映射规则

指令的操作码	源操作数 1	源操作数 2
ADDW	整型变量	整型变量
ADDI	整型变量	整型常量
ADDI	整型常量	整型变量
FADD	浮点型变量	浮点型变量
LA、FLW 以及 FADD	浮点型变量	浮点型常量
LA、FLW 以及 FADD	浮点型常量	浮点型变量

需注意的是，若整型常量超出范围，则应该先把它加载进寄存器，再使用 ADD 操作码的指令对两寄存器进行相加操作。另外，两个常量类型操作数相加不需要生成额外汇编代码，因为该情况在编译时就可以计算这两个常量的和，并将结果作为一个常量存储在程序的数据段中，在代码生成阶段直接使用这个计算好的常量值即可。

4.4.2 寄存器分配

在中间代码向目标代码转换的过程中，编译器后端需要对程序中生成的各种数据值进行妥善的存储管理，即决策各个值应该被存储在什么位置。寄存器作为计算机处理器中速度最快的计算单元，对提升程序性能具有显著作用。因此，编译系统应该尽可能充分的利用寄存器来操作值，但目标机器的寄存器数量通常是有限的，所以编译系统需要对寄存器资源做最佳的分配。

BIFANG 编译系统采用线性扫描算法作为寄存器分配算法。这种算法的目标是在有限的寄存器资源中，为程序中的变量分配寄存器，以减少内存访问次数，从而提高程序的执行效率。线性扫描算法比基于图形着色的算法要快得多，易于实现，并且生成的代码几乎与使用基于图形着色的更复杂且耗时的寄存器分配算法获得的代码一样高效。

线性扫描算法基于一种简单的思想：按顺序扫描中间代码中的每个虚拟寄存器，并为其分配物理寄存器。当遇到新的虚拟寄存器时，检查是否有可用的物理寄存器分配给它；如果没有，则根据一定的策略选择一个需要被替换的变量，并将其分配到新的物理寄存器。该算法是一种贪心算法，通过按顺序处理变量，尽可能地将其分配到物理寄存器中。线性扫描算法的主要步骤：

- (1) 构建活跃区间：首先，通过数据流分析计算每个虚拟寄存器的活跃区间，即虚拟寄存器在程序中被使用的区间，通常是虚拟寄存器在程序中的生命周期。
- (2) 排序活跃区间：将所有活跃区间按照其开始位置从小到大进行排序。

- (3) 遍历活跃区间：从排好序的活跃虚拟寄存器列表中，按顺序遍历每个区间。在遍历过程中，为每个区间分配一个可用的寄存器，并在需要时将之前分配的寄存器释放。
- (4) 确定溢出：如果某个区间没有足够的可用寄存器，即产生溢出，算法会尝试将某个寄存器的值移出到内存中，以腾出空间来分配给新的区间。

下面详细介绍本系统采用的活跃区间分析方法。为了理清每个基本块有哪些虚拟寄存器被定义，哪些虚拟寄存器被使用，本系统在每个基本块中都维护了 `def` 集合和 `use` 集合，分别表示当前基本块中定义的虚拟寄存器集合和被使用到的虚拟寄存器集合。同样的，为了理清每个基本块中有哪些虚拟寄存器作为输入进入基本块，有哪些虚拟寄存器被输出到后继基本块，本系统在每个基本块中都维护了输入活跃虚拟寄存器集合 `in` 和输出活跃虚拟寄存器集合 `out`，分别表示当前基本块中输入的虚拟寄存器集合和输出到后继基本块的虚拟寄存器集合。

活跃虚拟寄存器分析过程主要由两个函数构成，`calcLiveDefUse` 函数和 `calcLiveInOut` 函数。

`calcLiveDefUse` 函数遍历整个程序单元中的每个函数的每个基本块中的每个指令，并计算每个指令的活跃定义(`def`)和活跃使用(`use`)虚拟寄存器。对于每个指令的使用(`use`)操作数，如果该操作数对应的定义(`def`)不在当前基本块的活跃定义集合中，将该操作数添加到当前基本块的活跃使用集合中。同时，对于每个指令的定义(`def`)操作数，将该操作数添加到当前基本块的活跃定义集合中。当遍历完所有指令后，每个基本块中虚拟寄存器的 `def-use` 集合就构建完成了。

`calcLiveInOut` 函数使用迭代的数据流分析方法，首先根据函数的逆后序遍历顺序对基本块进行排序，然后进行迭代计算。对于每个基本块，首先计算出该基本块的输出活跃虚拟寄存器集合 `out`: 遍历该基本块的所有后继基本块，将每个后继基本块的输入活跃虚拟寄存器集合合并起来作为该基本块的输出活跃虚拟寄存器集合。对于每个基本块，计算出该基本块的输入活跃虚拟寄存器集合 `in`: 将该基本块的输出活跃虚拟寄存器集合去除该基本块的活跃定义集合 `def`，得到的结果即为该基本块的输入活跃虚拟寄存器集合 `in`。不断迭代直到稳定，即每个基本块的输入和输出活跃虚拟寄存器集合不再发生变化，每个基本块中虚拟寄存器的 `in-out` 集合就构建完成了。

至此，完成了活跃区间分析准备阶段的工作，为线性扫描分配寄存器算法计算每个虚拟寄存器生命周期提供了数据支持。BIFANG 系统基于活跃区间分析后进行使用线性扫描算法进行寄存器分配的几个步骤：

对定义的两组寄存器列表，一组整数寄存器和另一组浮点寄存器进行初始化，这些寄存器在算法开始时被设置为空闲的。

其中 `pass` 函数是线性扫描算法的主要入口点，它遍历机器单元中的所有函数，并为每个函数计算活跃区间，然后执行寄存器分配和代码重写。

(一) 计算活跃区间

`computeLiveIntervals` 函数计算每个虚拟寄存器的活跃区间，即虚拟寄存器在程序中被使用的时间范围。这包括处理基本块的后序遍历和更新活跃区间的开始和结束位置。

首先，函数通过 `IR_func->getReversePostOrder()` 获取基本块的逆后序遍历顺序。然后对于每个基本块 `bb`，在遍历顺序中，计算基本块的输出活跃虚拟寄存器集合。对于每个输出活跃虚拟寄存器，将其活跃区间插入到 `live_ranges` 映射中，活跃区间由基本块的开始位置和结束位置定义。然后对于基本块中的每条指令 `inst`，逆向遍历指令列表，对于每条指令的活跃定义虚拟寄存器，更新其活跃区间。活跃区间的开始位置是指令的位置，结束位置是指令位置之前的位置。同样的道理，对于每条指令的活跃使用虚拟寄存器，更新其活跃区间。活跃区间的开始位置是指令的位置，结束位置是基本块的开始位置。接下来，需要合并虚拟寄存器重叠的活跃区间，对于每个虚拟寄存器的活跃区间集合(`live_range`)执行合并重叠区间的操作。例如：如果当前区间与下一个区间重叠，则合并这两个区间，不断迭代比较它们开始和结束位置直至最终完成合并。在这之后，计算每个虚拟寄存器其活跃区间的最小开始位置和最大结束位置，并将这些信息存储在 `live_intervals` 哈希表（虚拟寄存器名为 `key`，其活跃最小和最大位置的 `pair` 为 `value`）中。以此类推，浮点数类型的虚拟寄存器运用同样的方法，构造出这些信息存储在 `f_live_intervals` 哈希表中，后续将用于寄存器分配决策。

(二) 执行寄存器分配

函数首先通过遍历 `mFunc->live_intervals` 和 `mFunc->f_live_intervals`，将每个虚拟寄存器的活跃区间转换为 `Interval` 对象，并存储在 `intervals` 和 `f_intervals` 向量中。表示虚拟寄存器的生命周期。然后，使用标准库中的 `std::sort` 函数，根据活跃区间的开始位置对整数和浮点活跃区间进行排序，确保了在分配寄存器时，先考虑那些最早需要使用的区间。

接下来，函数会遍历排序后的活跃区间列表。对于每个整数活跃区间，调用 `expireOldIntervals` 函数，移除那些已经结束的活跃区间，为新的活跃区间腾出空间。同时，检查剩余的空闲寄存器是否足够，如果足够将当前活跃区间分配给一个空闲的寄存器，并将其从空闲列表中移除。如果没有足够的空闲寄存器，调用 `spillInterval` 函数，将当前活跃区间 `spill` 到内存中。以此类推，浮点数虚拟寄存器也遵循相同的操作分配寄存器。在为虚拟寄存器分配了物理寄存器或进行了 `spill` 操作之后，函数将结果存储在 `reg_allocation` 哈希表中，以变量名为 `key` 其 `interval` 为 `value`，以便后续的代码重写阶段可以使用这些信息。

如果在分配过程中发生了寄存器溢出（即没有足够的物理寄存器来容纳所有活跃寄存器），则需要进行 `spill` 操作。在 `spillInterval` 和 `spillInterval_f` 函数中，如果当前活跃区间可以被 `spill`，则将其存储到内存中，并更新 `spilledReg` 计数器。在分配物理

寄存器后，函数会更新 `used_physical_regs` 集合，记录下在当前函数中使用的物理寄存器。

(三) 代码重写

遍历 `MachineFunction` 中的所有机器指令，对于每条指令的定义(`def`)和使用(`use`)活跃虚拟寄存器，检查它们是否为虚拟寄存器(VREG 或 FVREG)。若是，则从 `reg_allocation` 哈希表中获取该虚拟寄存器的生命周期等信息，如果该虚拟寄存器未被 `spill` (即未溢出到内存)，则更新一个维护函数与物理寄存器集合的哈希表，记录已使用的物理寄存器。然后删除原始的虚拟寄存器变量，并用新的物理寄存器变量替换它们。

在函数调用指令 (JAL) 之前，保存所有调用者保存的寄存器(s1-s11)。即将这些寄存器的值存储到栈上，以便在函数返回后能够恢复它们。如果有寄存器被 `spill`，会为这些寄存器生成相应的内存访问指令，即在 `spill` 区域的开始处分配内存地址，并在每个指令中插入加载和存储指令。根据 `spill` 的数量和大小，调整栈指针的位置确保所有的 `spill` 值在需要时可以被访问。

一个区间进行了 `spill` 之后，就不再占有任何一个寄存器了，但是在使用点依旧需要对寄存器进行使用，所以在使用点可以使用任意一个寄存器，临时寄存器的选取只要保证不和指令内其他寄存器重复即可。此时，需要为该变量在栈上分配一个空间，在对变量进行赋值之后要将值回存到栈上的空间。由于需要占用其他寄存器，那么可能会破坏寄存器中的值，所以还需要另外一个栈上的空间能够存放被临时占用的寄存器的值。

4.4.3 函数调用规范与栈式存储管理

函数调用规范用来描述父、子函数是如何编译和链接的，如栈的布局、参数的传递等。下面重点介绍 RISC-V 的函数调用规范规则：

- 函数的前 8 个参数使用 a0-a7 寄存器来传递
- 如果函数参数多于 8 个，除前 8 个参数使用寄存器来传递之外，后面的参数使用栈来传递
- 函数的返回参数保存到 a0 和 a1 寄存器中
- 函数的返回地址保存在 ra 寄存器中
- 如果子函数使用了 s0-s11 寄存器，那么子函数在使用前需要把这些寄存器的内容保存到栈中，使用完之后再从栈中恢复内容到这些寄存器里
-

通常栈是一种从高地址往低地址扩展的数据存储结构，栈的起始地址称为栈底，栈从高地址往低地址延伸过程中的某个点称为栈顶。栈需要一个指针来指向栈最新分配的地址，即指向栈顶，这个指针叫栈指针。数据入栈时，`sp` 指向的地址减少，栈空

间增大；数据出栈时，`sp` 指向的地址增加，栈空间减少。另外，还有一个指针叫栈帧指针（`fp`），用来指向栈底，在 RISC-V 中使用 `s0` 寄存器作为 `fp`。

BIFANG 系统的栈式存储管理采取动态分配的栈式结构。当一个函数被调用时，新的栈帧被动态创建（其实质就是 `sp` 和 `fp` 向低地址移动），栈帧中需要包含函数的返回地址、局部变量和保存的寄存器状态。函数返回时，相应的栈帧被销毁，栈指针回退到调用者的栈帧。动态栈的内存分配是连续的，大小可以根据需要增长和缩小，这种管理方式适用于递归调用、大量局部变量或不确定数量的函数调用等情况。

另外，有不少处理器的指令集中都存在入栈和出栈专用的指令，例如 ARM 的指令集提供了 `POP` 和 `PUSH` 指令用来实现出栈和入栈，但在 RSIC-V 中没有入栈和出栈指令，只能通过使用基础指令集操作 `sp` 和 `fp` 寄存器实现出栈和入栈。

4.5 可执行文件生成

BIFANG 系统的汇编与链接部分借助了较为成熟的 GCC 工具链，其汇编器与链接器分别负责将编译器生成的汇编文件进行汇编、链接，最后生成可执行文件。可执行文件的生成主要有两种平台的实现方式，一种是在 x86 架构中使用 BFIANG 编译系统对 `tesecase.bf` 文件进行交叉编译，然后将其传输到 RISC-V 平台运行，另一种是直接在 RSIC-V 平台使用 BIFANG 编译系统对 `testcase.bf` 文件进行编译，本地运行。

下面介绍在 x86 架构的 `ubuntu22.04` 操作系统中使用 BIFNAG 编译系统向 RISC-V 平台交叉编译 `testcase.bf` 文件的具体流程：

1. 进入 BIFNAG 项目的根目录，创建 `build` 目录，进入 `build` 目录，执行 `cmake ..` 命令生成 `makefile` 文件。
2. 在 `build` 目录下执行 `make`，编译整个项目，得到 BIFANG 的可执行文件。
3. 使用命令 `./BIFANG ..//testcase/testcase.bf` 将测试代码编译生成 RISC-V 汇编文件 `out.s`。
4. 使用提前安装好的交叉编译工具链 `riscv64-unknown-linux-gnu-gcc`，执行命令 `riscv64-unknown-linux-gnu-gcc -c out.s` 生成 Obj 文件 `out.o`。
5. 执行命令 `riscv64-unknown-linux-gnu-gcc -o out out.o` 生成可执行文件 `out` 将 `out` 传输到 RISC-V 平台执行 `./out` 运行。

第 5 章 系统分析及测试

5.1 测试环境和测试方法

5.1.1 测试环境

编译平台: Ubuntu 22.04 2 核(vCPU) 4 GiB

编译工具链: gcc (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0 CMake version 3.27.9

程序运行平台: 眺·星光 VisionFive (RISC-V 64 位 Fedora 操作系统)

5.1.2 测试方法

本系统采用黑盒测试法, 设置覆盖全部功能的测试用例, 分别对编译系统前端和编译系统后端进行测试, 测试者只需并运行测试脚本, 即可对 BIFNAG 编译系统进行测试, 检验系统是否符预期需求。

5.2 测试用例

编号: T0001

用例名: 基本变量声明和赋值

用例内容:

```
const int a = 10, b = 20;
const float c = 3.14, d = 2.718;
int main() {
    int x = a;
    float y = c;
    return x;
}
```

编号: T0002

用例名: 函数调用与参数传递

用例内容:

```
int add(int a, int b) {
    return a + b;
}

int main() {
    int sum = add(5, 3);
    return sum;
}
```

编号：T0003

用例名：数组声明和用法

用例内容：

```
int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    int i = 0;
    while (i < 3) {
        i = i+1;
        arr[3] = arr[3] + 1;
    }
    return arr[3];
}
```

编号：T0004

用例名：控制流语句和逻辑操作

用例内容：

```
int main() {
    int x = 5, y = 10;
    if (x > y) {
        return x;
    } else {
        return y;
    }
    return 0;
}
```

编号：T0005

用例名：循环语句和函数调用

用例内容：

```
int doWork() {
    int count = 0;
    while (count < 5) {
        count = count + 1;
    }
    return count;
}

int main() {
    int sum = doWork();
    return sum;
}
```

```

}

```

编号：T0006

用例名：嵌套函数调用

用例内容：

```

int square(int n) {
    return n * n;
}

int cube(int n) {
    return n * square(n);
}

int main() {
    int result = cube(3);
    return result;
}

```

5.3 编译系统功能测试

表 5-1 展示了系统的功能测试结果。

表 5-1 功能测试的测试结果

编号	用例名	预期结果	实际结果
T0001	基本变量声明和赋值	赋值成功，返回 10	赋值成功，返回 10
T0002	函数调用与参数传递	函数调用和参数传递成功,返回 8	函数调用和参数传递成功,返回 8
T0003	数组声明和用法	数组声明成功，函数调用成功， 返回 7	数组声明成功，函数调用成功，返 回 7
T0004	控制流语句和逻辑操作	控制流分支判断成功和逻辑操作判 断成功，返回 10	控制流分支判断成功和逻辑操作判 断成功，返回 10
T0005	循环语句和函数调用	循环语句分支判断成功，函数调 用成功，返回 5	控制流分支判断成功，函数调用成 功，返回 5
T0006	嵌套函数调用	函数嵌套调用成功，返回 27	函数嵌套调用成功，返回 27

5.4 结果和分析

以编号 T0002 测试用例测试过程举例：

```

root@ALiCloud:/home/qsy/workspace/BIFANG/testcases# ./build/BIFANG

```

```

testcase2.bf

Entering rule:
intadd(inta,intb){returna+b;}intmain(){intsum=add(5,3);returnsum;}<EOF>

Entering rule:
intadd(inta,intb){returna+b;}intmain(){intsum=add(5,3);returnsum;}<EOF>

Entering rule: intadd(inta,intb){returna+b;}

Entering rule: int

Visiting terminal: int

Exiting rule: int

Visiting terminal: add

Visiting terminal: (

Entering rule: inta,intb

Entering rule: inta

Entering rule: int

Visiting terminal: int

Exiting rule: int

Visiting terminal: a

Exiting rule: inta

Visiting terminal: ,

Entering rule: intb

Entering rule: int

Visiting terminal: int

Exiting rule: int

Visiting terminal: b

Exiting rule: intb

Exiting rule: inta,intb

Visiting terminal: )

Entering rule: {returna+b;}

Visiting terminal: {

...

```

查看生成的中间代码文件(.ll):

```

root@ALiCloud:/home/qsy/workspace/BIFANG/testcases# cat testcase2.bf_ir.ll
define i32 @add(i32 %0, i32 %1) {
addStart1:           ;No predecessor!!    succs:addEntry2,
  %RetVal3 = alloca i32, align 4
  %a4 = alloca i32, align 4
  %b5 = alloca i32, align 4
  br label %addEntry2
addEntry2:           ;preds: addStart1,    succs:addRet0,
  store i32 0, i32* %RetVal3, align 4

```

```

    store i32 %0, i32* %a4, align 4
    store i32 %1, i32* %b5, align 4
    %2 = load i32, i32* %a4, align 4
    %3 = load i32, i32* %b5, align 4
    %4 = add i32 %2, %3
    store i32 %4, i32* %RetVal3, align 4
    br label %addRet0

addRet0:           ;preds: addEntry2,      succs:
    %5 = load i32, i32* %RetVal3, align 4
    ret i32 %5
}

define i32 @main() {
mainStart7:          ;No predecessor!!      succs:mainEntry8,
    %RetVal9 = alloca i32, align 4
    %sum10 = alloca i32, align 4
    br label %mainEntry8

mainEntry8:           ;preds: mainStart7,      succs:mainRet6,
    store i32 0, i32* %RetVal9, align 4
    %0 = call i32 @add(i32 5,i32 3)
    store i32 %0, i32* %sum10, align 4
    %1 = load i32, i32* %sum10, align 4
    store i32 %1, i32* %RetVal9, align 4
    br label %mainRet6

mainRet6:             ;preds: mainEntry8,      succs:
    %2 = load i32, i32* %RetVal9, align 4
    ret i32 %2
}

```

测试程序编译后汇编代码输出：

```

root@ALiCloud:/home/qsy/workspace/BIFANG/testcases# cat testcase2.bf_out.s
.globl  main
add:
    sd ra, -8(sp)          # 0
    sd fp, -16(sp)         # 1
    add fp, sp, zero       # 2
    addi sp, sp, -64        # 3
    sd s1, 0(sp)
    addi t0, a0, 0          # 4
    addi t1, a1, 0          # 5
    j  addEntry2            # 6

addEntry2:

```

```

    li t2, 0          # 7
    sw t2, -28(fp)      # 8
    sw t0, -20(fp)      # 9
    sw t1, -24(fp)      # 10
    lw t1, -20(fp)      # 11
    lw t0, -24(fp)      # 12
    addw t2, t1, t0      # 13
    sw t2, -28(fp)      # 14
    j  addRet0          # 15

addRet0:
    lw t2, -28(fp)      # 16
    addi a0, t2, 0        # 17
    ld s1, 0(sp)
    addi sp, sp, 64      # 18
    ld ra, -8(sp)        # 19
    ld fp, -16(sp)       # 20
    jr ra                # 21

main:
    sd ra, -8(sp)        # 22
    sd fp, -16(sp)       # 23
    add fp, sp, zero      # 24
    addi sp, sp, -64       # 25
    sd s1, 0(sp)
    j  mainEntry8         # 26

mainEntry8:
    li t0, 0              # 27
    sw t0, -20(fp)        # 28
    addi a0, zero, 5        # 29
    addi a1, zero, 3        # 30
    addi sp, sp, -16
    sd t0, 0(sp)
    jal ra, add            # 31
    ld t0, 0(sp)
    addi sp, sp, 16
    addi t0, a0, 0          # 32
    sw t0, -24(fp)        # 33
    lw t0, -24(fp)        # 34
    sw t0, -20(fp)        # 35
    j  mainRet6            # 36

mainRet6:
    lw t0, -20(fp)        # 37

```

```
addi a0, t0, 0          # 38
ld s1, 0(sp)
addi sp, sp, 64         # 39
ld ra, -8(sp)           # 40
ld fp, -16(sp)          # 41
jr ra                  # 42
```

对 BIFANG 编译系统的测试结果表明，所有的测试用例基本覆盖了整个系统功能，测试用例均通过了测试，整个系统测试效果理想。

第 6 章 总结与展望

6.1 总结

编译技术自计算机科学诞生以来就一直是推动该领域发展的关键力量。无论是在编程语言或是软件性能优化等领域中都占据着重要的作用，是整个计算机科学发展的基石。

虽然当前工业界内有例如 GCC、LLVM 等非常成熟的编译系统以及 IDE 中丰富的编译和调试功能供人们使用，为软件开发者带来了高效便捷的编码体验。然而却使学习者忽视了程序底层编译和执行，这种高度的抽象可能会导致学习者对编译过程以及程序的实际执行机制缺乏深入的了解。

本文详细介绍了面向 RISC-V 平台的 BIFANG 语言编译系统设计与实现的全过程，主要包括：

- (1) 基于 ANTLR 实现的一个词法分析器。在词法分析阶段，本文借助当前较为流行的前端构建工具 ANTLR，基于编写词法规则文件实现生成词法分析器，将源代码转换为 Token 流。
- (2) 基于 ANTLR 实现的一个语法分析器。本文同样使用 ANTLR 工具，基于扩展巴科斯范式的文法编写语法规则文件，实现了自动生成语法分析器，并构建了语法解析树。
- (3) 本文设计实现了符号表、作用域以及类型系统等数据结构。语义分析与语法分析同时进行，便于记录构建过程的语义信息。
- (4) 设计实现了中间代码及其生成器。本文设计并实现了简化版本的 LLVM IR 表示，包括 GlobalUnit、Function 等中间代码的内存结构，实现了基于访问者模式遍历解析树的 IR 生成器，根据中间代码生成的主要算法框架，将语法解析树翻译为合适的中间代码表示。
- (5) 设计实现了汇编代码生成器。针对 RISC-V 指令集架构实现了指令选择和寄存器分配，包括设计并实现了指令选择的静态映射规则，设计并实现了线性扫描寄存器分配算法，基于中间代码进行汇编代码生成的算法框架，实现了将源程序翻译为汇编代码。
- (6) 汇编链接生成可执行文件。使用开源汇编器与链接器将编译器生成的汇编代码经过汇编、链接生成可执行文件。
- (7) 最后针对本文实现的 BIFANG 编译系统进行系统测试，验证了编译系统的有效性和可靠性。所有测试用例均成功编译并运行，表明系统能够满足基本的编译需求，并具有良好的扩展性和适应性。

通过设计一个能够将高级语言代码编译为 RISC-V 指令集的编译系统，有利于提

高学习者的系统能力，为学习者在编译系统方向提供一定思路。同时，随着国内芯片产业重视程度越来越高，越来越多的研究所和厂商开始重视芯片的研发并且意识到芯片生态系统的重要性，本文的研究对国产芯片、关键基础软件及核心系统的技术突破与产业化后备人才培养具有一定的促进作用。

6.2 展望

尽管本文设计的编译系统实现了大部分基本的编译功能，但研究人员不能仅仅满足于掌握基本的编译器设计和实现原理，在优化程序性能、增强扩展性、支持跨平台等诸多方面值得进一步探索和完善，例如：

- (1) 编译优化：目前 BIFANG 编译系统虽然能够生成正确的目标代码，但在编译优化方面还有很大的提升空间。未来的工作可以集中在支持常见的前端优化、优化寄存器分配算法、增强指令选择的智能化以及实现更高效的指令调度策略。
- (2) 扩展性增强：随着编程语言和计算机体系结构的不断发展，编译系统需要不断适应新的编程范式和硬件特性。未来的研究可以致力于提高编译器的模块化和可配置性，以便更容易地添加新的语言特性和支持新的硬件平台。
- (3) 跨平台支持：虽然本系统主要面向 RISC-V 平台，但跨平台的编译支持也是一个重要的研究方向。未来的工作可以探索如何让编译系统支持多种目标架构。

编译系统的设计和实现是一个复杂和持续的过程。特别是编译优化领域需要不断地学习最新的技术动态，吸收新的理念，并结合实际应用需求进行创新和改进。期待本文的研究能够为学习者在编译器领域的研究和实践提供有价值的参考和启示。

参考文献

- [1] 高云云. C 语 言 编 译 系 统 的 研 究 与 实 现 [D]. 南 京 邮 电 大 学,2020.DOI:10.27251/d.cnki.gnjdc.2019.000701.
- [2] 刘畅,武延军,吴敬征等.RISC-V 指令集架构研究综述[J].软件学报,2021,32(12):3992-4024.DOI:10.13328/j.cnki.jos.006490.
- [3] 刘三献. 基于 ANTLR 的 Gaussian 词法分析器和语法分析器的分析与设计[D]. 兰州大学,2009.
- [4] 李磊. C 编译器中间代码生成及其后端的设计与实现[D]. 电子科技大学,2017.
- [5] 徐金光. C 编译器语法分析的设计与实现[D]. 电子科技大学,2017.
- [6] 刘聪. C 编译器符号表及属性文法的设计与实现[D]. 电子科技大学,2017.
- [7] ANTLR,<https://www.antlr.org/>
- [8] 王爱英. 精简指令系统计算机（RISC）的存储体系[J]. 清华大学学报,2012,4:58-66.
- [9] Alfred V. Aho, Ravi Sethi, Jeffrey D.Ullman .编译原理[M]. 第二版. 赵建华. 北京：机械工业出版社,2012:20-156.
- [10] 谭浩强. C 程序设计(第三版)[M]. 清华大学出版社, 2005:3-5.
- [11] KeithD.Cooper, LindaTorczon. 编译器设计[M]. 人民邮电出版社, 2013: 18-53.
- [12] 张波, 黄涛, 傅远彬, 等. 对象描述语言编译器的设计和实现[J]. 软件学报, 1998, 9(7): 525-531.
- [13] 王馨梅, 王冬芳. 编译器前端自动构造的研究与实现[J]. 微机发展, 2004, 14(4): 82-83.
- [14] 王海燕, 杨鹤标. 基于 ANTLR 的前端编译器若干关键问题的研究与实现[J]. 长春工程学院学报(自然科学版), 2013, 14(3): 104-107.

致谢

日月忽其不淹兮，春与秋其代序。行文至此，好似大梦一场，仿佛昨日还是 2020 年 9 月 12 日，初次到达松山湖大学路 1 号，我满怀憧憬开启本科生涯。四年后的今天，2024 年 4 月 17 日，临近毕业，我对脚下的道路仍然充满希望、保持乐观，但我深知当我撰写完毕业论文就意味对这段特别的经历画上一个句号，再回首，我想要感谢许多人。

明师之恩，诚为过于天地。感谢本科时期遇到的每一位老师，最想感谢的是对自己长期指导和帮助的课题组老师——殷建平和李宽老师。2021 年秋，我刚加入课题组，对专业方向比较迷茫，殷老师让我每周参与组会和定期汇报，那时起我渐渐了解前辈的专业方向与动态，学习如何用科学的方法分析解决科研问题；2022 年夏，借着一次编译器比赛的机会，李老师带我们入门了编译赛道，那是一天中午，老师在会议室对着代码逐行给学生讲解；2023 年夏，在李老师带领下非常荣幸地参加了 2023 全国高性能计算学术年会，那时我已取得一些进步，能听懂大部分报告内容，盛大的学术氛围彻底激发了我在高性能计算、机器学习系统等领域的兴趣。2024 年春，考研失利，未进复试遗憾落榜，但我不愿放弃学习的机会，在殷老师的鼓励和指导下，我重振了士气，接连赶赴重庆和广西参加调剂复试，最终被桂林电子科技大学录取。殷老师作为课题组的领导者，德高望重，明察秋毫，为课题组的发展方向指明了道路。李老师一贯务实的科研作风，潜移默化的影响着课题组的学生，李老师不仅是我的启蒙导师，更是学生不断成长的引路人，尊重学生个性，为学生提供了非常多指导和难得的锻炼机会，学生受益颇多。在本次毕业设计中，李老师指导学生有序地按照要求和节奏去完成毕设和毕业论文，并给出许多修改论文的指导建议，我时常会因为在我们课题组学习而感到幸运和骄傲。

家人闲坐，灯火可亲。感谢父母二十二年对我的悉心培养，感谢家人对我无私的支持和帮助，家庭永远是我内心最坚强的后盾。从小到大，父亲都非常看重我的学习，但我的成绩并不算好，因此一谈成绩父亲的神情就稍显严肃，感觉父亲是一个严厉的人。上了大学，我与父母之间逐渐相互理解，他们时常关心我的英语等级考试、期末考试以及考研备考；父亲会给我分享考研经验的推文，调剂招生的信息；母亲会打电话给我鼓励让我注意身体。但无论事情结果好坏，父母始终没有责怪我，甚至有时会因为不够了解考研而自责。我也很感谢我的妹妹，她比我优秀许多，让父母少操心许多，中学成绩在年级常常名列前茅，是我们家的骄傲。大学四年，每年寒暑假我都毅然决然地选择了留校，只有短暂的春节假期会待在家里，对我们小孩来说，家庭带给我们的感觉就像

家族聚餐的冬夜，等待开饭的时光，是最惬意的，客厅里灯光明亮，流露出动人的暖意，上桌时还滚烫的一大锅热汤，香气随着水雾蒸气袅袅蔓延开，我们愿意这样盯着它快活地发愣，什么也不操心。

莫言相遇易，趣途远有期。感谢本科遇到的每一个朋友，无论我们共同有过酸、甜、苦、辣的经历，它都将是人生中不可代替的一笔，构成我们各自丰富多彩的人生。感谢我的本科舍友彭涓、王春光、彭奇岳，他们是我本科阶段交流最多的朋友，我们了解彼此的性格特点，携手互助走过了四年大学时光，看到不同人身上各自散发的美好光辉。感谢本课题组的师兄师姐肖国键、陈浩源、韩宇飞、徐聘、李言一、敖斌、伍欣、文青、程逸远、邓卓灏、宁卓昊、姚琳薇、龚莨皓，在成长路上给予过我许多帮助和支持，我们一同探索过人间百味，也曾共举杯酒饮微醺。感谢所有同学，愿意在我担任学生干部期间配合和支持我的工作，这份同窗之情将是我最值得留恋的回忆。

江山不负英雄泪，且把利剑破长空。感谢伟大的中国共产党和革命先辈，今日的太平盛世，是前人铸造。我们的历史长河，处处闪耀着光芒，英雄豪杰，铸就了中华民族不灭的光辉，没有先辈的艰苦奋斗，就没有今天强大的中国，更没有现在的我。