

# 西安邮电大学

## 毕业设计（论文）

题目： 轻量级编译器设计与实现

学院： 计算机学院

专业： 计算机科学与技术

班级： 计科 1603

学生姓名： 王宝彤

学号： 04161082

导师姓名： 董梁 职称： 讲师

起止时间： 2019 年 11 月 18 日 至 2020 年 6 月 12 日

# 毕业设计（论文）承诺书

本人所提交的毕业论文《轻量级编译器设计与实现》是本人在指导教师指导下独立研究、写作的成果，论文中所引用他人的文献、数据、图件、资料均已明确标注；对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式注明并表示感谢。

本人深知本承诺书的法律责任，违规后果由本人承担。

签名：王宝彤

日期：2020 年 6 月 12 日

# 西安邮电大学本科毕业设计(论文)选题审批表

申报人	董梁		职称	讲师	学院	计算机学院		
题目名称	轻量级编译器设计与实现							
题目来源	科研				教学	√	其它	
题目类型	硬件设计		软件设计	√	论文		艺术作品	
题目性质	实际应用		√		理论研究			
题目简述	<p>(为什么申报该课题)</p> <p>计算机系统能力培养学生解决复杂工程问题的能力，需要在处理器设计、编译技术、操作系统之间完成垂直打通。编译器是将高级语言程序翻译成处理器可以识别的机器语言，因此编译器是联系操作系统与处理器运行环境的重要支撑软件。计算机专业学生在学习了程序设计语言和编译原理之后，需要掌握实际的编译设计技术，能够将高级语言程序编译生成特定的机器指令程序，从而为后续的升学就业等打下扎实的专业基础。</p> <p>编译器主要分为词法分析、语法分析、语义分析、中间代码生成、代码优化以及目标代码生成等部分。其所需使用的上下文无关文法、语法分析树、形式语言、集合划分等与离散数学有密切联系。词法分析中有限状态机、正则表达式等内容与计算机设计有密切联系。语法分析的自上而下和自下而上的方法，与数据结构等课程有机联系。属性文法和语法制导翻译、语义分析和语言的自动分析有密切联系。中间代码生成和目标代码生成与汇编、机器指令密切相关。而符号表、运行时存储空间组织、代码优化等都与程序语言及设计方法有密切关系。因此，开发一个完整的轻量级编译器本身就是一个复杂的计算机工程问题。而且编译技术对机器学习、人工智能等都有重要的支撑作用。</p>							
对学生知识与能力要求	编译原理与技术 高级语言程序设计 计算机组成原理 微机原理与接口技术 书面表达和团结协作能力							

预期目标	<p>(本题目应完成的工作，题目预期目标和成果形式)</p> <p>设计并实现一个使用面向对象语言开发的轻量级编译器，要求能够将基本 C 语言程序包括：变量、数组、指针、结构体、函数调用等，编译成目标处理器的机器语言程序。该编译器更为符合 32 位 MIPS 处理器设计需求，包括支持 50 条左右常用基本指令、目标代码生成和简单优化。生成的目标代码能够完成总线驱动外设访问以及多处理器之间互联等功能。完成所有开发设计文档的撰写。</p>		
时间进度	<p>2019.11.18 ---- 2019.11.30 根据题目，查找资料，设计整体方案，确定实现的语言范围。</p> <p>2019.12.01 ---- 2019.12.15 根据整体设计方案，实现高级语言子集的词法分析。</p> <p>2019.12.16 ---- 2019.12.31 根据整体设计方案，实现高级语言子集的语法分析。</p> <p>2020.01.01 ---- 2020.02.28 根据整体设计方案，实现高级语言子集的中间代码生成。</p> <p>2020.03.01 ---- 2020.03.31 根据生成的中间代码，完成遍历和语义分析。</p> <p>2020.04.01 ---- 2020.04.30 根据遍历和语义分析的结果，生成对应的机器指令代码。</p> <p>2020.05.01 ---- 2020.05.10 对生成的机器指令代码进行调试并验证结果的正确性。</p> <p>2020.05.11 ---- 2020.05.31 完成各项开发文档和毕业设计论文的撰写修订工作。</p> <p>2020.06.01 ---- 2020.06.12 准备各项文档，完成毕业设计论文答辩。</p>		
系（教研室）主任 签字	2019 年 11 月 5 日	主管院长 签字	2019 年 11 月 7 日

# 西安邮电大学本科毕业设计（论文）开题报告

学生姓名	王宝彤	学号	04161082	专业班级	计科 1603
指导教师	董梁	题目	轻量级编译器设计与实现		

选题目的（课题背景及意义、国内外研究情况）

编译器是将高级语言程序翻译成处理器可以识别的机器语言<sup>[1]</sup>的系统软件，因此编译器是联系操作系统与处理器运行环境的重要支撑软件，若要在处理器设计、编译技术、操作系统之间完成垂直打通，编译器的设计是关键一步。它决定了是否有能力在自己的处理器架构的基础上开发较为复杂的程序软件或者驱动程序<sup>[2]</sup>，如：进一步设计操作系统、系统程序和工具等。

从理论上讲，国内外对于编译涉及到的算法都已有成熟的实现，尤其是词法分析和语法分析算法已经非常成熟<sup>[3]</sup>，甚至词法分析和语法分析阶段均有“分析程序生成器”工具可以自动生成高效率的词法分析器和语法分析器。有大量关于编译的教材和书籍以及网上课程。

但是，大多数书籍和教材大都针对理论进行深入论述，大都难以用于针对具体实践的实际教学。要么缺少针对一个简单编译器的具体实现，要么功能过于庞大难以用于大学教学，要么缺乏针对特定精简指令集的兼容，要么功能过于冗杂难以阅读。国内大多数高校没有开设编译器实现的课程，少数高校虽然有简单编译器的开发教学，但是其选用的算法过于简单，难以适应复杂语法的编译需求，只有极少数高校可能能够提供比较完善的编译器教学。国外有商业化开源代码的 **LLVM** 等，但是代码庞大，功能冗杂，难以用于教学。**GitHub** 上有许多开源编译器项目（国内的和国外的），但是质量良莠不齐，普遍没有详细的文档，不能适应针对特定平台的编译要求，难以用于教学。

从头开发一个编译器有利于加深对大学整体课程的理解，其设计与开发涉及了许多专业课程和理论。计算机程序如何运行依赖于该语言的编译器，因此对编译器的编译结果进行分析是了解编译器如何对运算进行编译的有效方法<sup>[4]</sup>。**C** 语言是计算机从业者学习最广泛的语言之一，编译器源程序的语言是在 **C** 语言的基础上进行简化、修改后的语言，对其编译效果比较熟悉，也利于编译器的设计。

编译器自主开发也对后续开发操作系统等系统软件具有深远影响，其具体表现在正确性和高效性（代码优化）。编译器开发本身是一个复杂的计算机工程问题，编译器的优劣直接影响了源程序是否能翻译成高效的目标语言、目标语言能否正确地工作、目标语言是否与源程序等效。其正确性尤为重要，具体是指编译过程的正确性，编译器能确保其编译过程是正确无误的，不会带来误编译，目标代码能将源代码的特征正确、完整地实现，其语义与源代码保持一致<sup>[5]</sup>。编译器对目标代码的优化关系到运行效率，优化部分分为机器无关的优化和机器相关的优化<sup>[6]</sup>。例如：**GCC** 提供了数种编译选项优化等级，每个优化等级又提供了许多可选项，来适应特定优化需求<sup>[7]</sup>。代码优化带来的收益是可观的，能大幅度提高效率、节约能耗等，研究显示，嵌入式系统中目标程序冗余带来的能耗损失高达 80%<sup>[8]</sup>。如果程序运行时间超出用户的等待时间上限，即使其能够完成某方面的功能，也无法投入实际应用。因此，需要采用有效的性能优化方法，改善程序整体性能水平<sup>[9]</sup>。高级语言编译器的出现，实现了人们使用简洁易懂的编程语言与计算机交流的目的<sup>[10]</sup>。

编译器的开发实现具有很高的教学价值。编译器的开发可以增加大学期间课程之的宏观理解、加强复杂系统软件的开发能力、提升对计算机系统从硬件到软件整理链

路的理解。本项目允许每个人自行设计一门语言，考验了一定的创新能力，具有一定的趣味性。进行代码生成时需要处理器、操作系统、编译器之间进行规则约束和协议约定，使学生有机会体验一次制定关键协议规则的过程，提高全面思考能力。

参考文献：

- [1] Alfred V.Aho/Monica S.Lam/Ravi Sethi/Jeffrey D.Ullman 《Compilers Principles Techniques and Tools 2nd》
- [2] 冯钢,郑扣根.基于 GCC 的交叉编译器研究与开发[J].计算机工程与设计,2004(11):1880-1883.王保胜.何金枝.关于 C 编译器对——运算编译的研究 电脑知识与技术 2010 年 18 期
- [3]陈俊洁,胡文翔,郝丹,熊英飞,张洪宇,张路.一种静态的编译器重复缺陷报告识别方法[J].中国科学:信息科学,2019,49(10):1283-1298.
- [4]王保胜,何金枝.关于 C 编译器对——运算编译的研究[J].电脑知识与技术,2010,6(18):5093-5095.
- [5]刘洋,杨斐,石刚,闫鑫,王生原,董渊.可信编译器构造的翻译确认方法简述[J].计算机科学,2014,41(S1):334-338.
- [6]彭获然,熊庭刚,胡艳明,黄亮.基于国产 GPU 的 GLSL 编译器设计[J].计算机与数字工程,2019,47(06):1502-1506.
- [7] Boma A. ADHI,Tomoya KASHIMATA,Ken TAKAHASHI,Keiji KIMURA,Hironori KASAHARA. Compiler Software Coherent Control for Embedded High Performance Multicore[J]. IEICE Transactions on Electronics,2020,E103.C(3).
- [8]Youcong Ni,Xin Du,Peng Ye,Ruliang Xiao,Yuan Yuan,Wangbiao Li. Frequent pattern mining assisted energy consumption evolutionary optimization approach based on surrogate model at GCC compile time[J]. Swarm and Evolutionary Computation,2019,50.
- [9] 陈雪.C++ 程序设计中的时间性能优化 [J/OL]. 电子技术与软件工程 ,2019(23):232-233[2019-12-11].<http://kns.cnki.net/kcms/detail/10.1108.TP.20191210.1458.262.html>.
- [10]范志东 张琼生.《自己动手构造编译系统:编译、汇编与链接》

前期基础（已学课程、掌握的工具，资料积累、软硬件条件等）

已学课程：

离散数学、C 语言程序设计、C++面向对象的程序设计、数据结构与程序设计、操作系统、Linux 操作系统、嵌入式系统原理与应用、计算机组成原理、微机原理与接口技术、编译原理与技术、FPGA 模型机课程设计。

掌握的工具：

开发平台（iOS）、版本管理工具（git）、集成开发环境（xcode）。

资料积累：

编译原理课程资料。

软硬件条件：

笔记本电脑、软件开发工具、毕业设计的汇编脚本工具。

要研究和解决的问题（做什么）

设计并实现一个使用面向对象语言开发的轻量级编译器，要求能够将基本 C 语言程序包括：变量、数组、指针、结构体、函数调用等，编译成目标处理器的机器语言程序。设计的文法功能包含：变量、结构体、if-else 语句、while 语句、下标运算（支持数组）、函数括号运算符（支持函数调用），普通括号、部分运算符、取地址、指针解引用、break、continue、return 等。不包含：int 以外的基本数据类型、void 类型、const/static 等修饰符、goto 语句、枚举类型、共用体、for 语句、do-while 语句、switch 语句、逗号表达式、long long 等类型、三目运算符、++、--、不定参数、函数声明省略形参的形式（不支持）、强制类型转换等。其他细节：函数必须有参数，只支持一维数组声明，指针声明只支持一层指针（但是预留多层指针解引用的逻辑），32 位 int 为最小识别单元。

该编译器后端要求符合我们自主开发的 MIPS 处理器设计需求，包括支持 50 条左右常用基本指令、目标代码生成和简单优化。生成的目标代码能够完成总线驱动外设访问以及多处理器之间互联等功能。有控制台交互界面，能够完成源代码的输入、保存、打开、编译等功能。能够展示编译出的中间代码，并能够保存文件。能够展示编译出的伪汇编/机器指令代码，并能够保存文件。

工作思路和方案（怎么做）

### 1.工作思路

A.查阅资料，确立需要实现的语言范围，设计文法描述语言格式，设计文法。

B.根据文法设计针对文法的“词法分析器”，求解相关数据，最终算出识别活前缀的自动机的状态转移矩阵。

C.设计符号表结构、虚拟语法树结构，构造语法分析总控程序。

D.调整整体代码结构，编写语法制导翻译的制导函数。

E.确定中间代码表示方式，完成中间代码生成。进行简单代码优化。

F.完成最终代码生成。

### 2.技术方案

使用自动机技术进行词法分析和对文法的识别。

使用构造项目集规范族的方法求解识别活前缀的状态转移矩阵。

使用 SLR(1)算法进行语法分析，使用数据结构——栈进行管理。

使用语法制导翻译进行语义分析和中间代码生成。

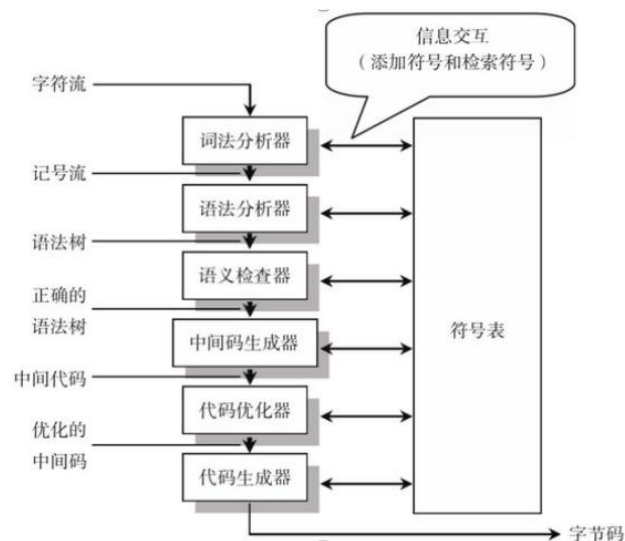


图 1 编译流程图

### 3.可能的困难

第一：工作量大。根据前期研究评估得出部分数据显示，编译器设计工程量巨大。从第一版文法看：有 61 条产生式，生成的识别活前缀的 DFA 有 110 个项目集和 410 条边，每项目集有几十个项目，生成的状态转移矩阵有 831 个状态转移，错误状态约有  $110 \times (23+29) - 831 = 4890$  个，这种规模的图只能使用程序计算，无法手动画图，还有判重、出错、修改文法、解决冲突等工作，均需要大量时间。

第二：实现复杂。符号表、语法树、语法分析自动机这三个紧紧联系，且与工程其他部分耦合度大。此处的设计影响了编译器的全局，决定了编译器整体架构是否合理、工作效率是否高效、开发过程是否容易出错等。

### 4.进度计划

2019.11.18 -- 2019.11.30	根据题目，查找资料，设计整体方案，确定实现的语言范围。
2019.12.01 -- 2019.12.15	根据整体设计方案，实现高级语言子集的词法分析。
2019.12.16 -- 2019.12.31	根据整体设计方案，实现高级语言子集的语法分析。
2020.01.01 -- 2020.02.28	根据整体设计方案，实现高级语言子集的语义分析。
2020.03.01 -- 2020.03.31	根据整体设计方案，完成遍历，生成中间代码。
2020.04.01 -- 2020.04.30	根据中间代码，生成对应的机器指令代码。
2020.05.01 -- 2020.05.10	对生成的机器指令代码进行调试并验证结果的正确性。
2020.05.11 -- 2020.05.31	完成各项开发文档和毕业设计论文的撰写修订工作。
2020.06.01 -- 2020.06.12	准备各项文档，完成毕业设计论文答辩。

指导教师意见

签字：                    年    月    日





## 摘 要

编译器是将高级语言程序翻译成处理器可以识别的机器语言的系统软件。大多数书籍和教材主要针对理论进行深入论述，缺少针对于一个完整的简单编译器的具体实现。国内大多数高校在教学中侧重编译理论，少数高校虽然有简单编译器的开发教学，但是其选用的算法过于简单，难以适应复杂语法的编译需求，只有极少数高校能够提供比较完善的编译器实践教学。而商业化开源代码功能过于强大，代码冗杂，难以用于教学。

本文将对词法分析、语法分析、中间代码生成、汇编代码生成、目标代码生成进行论述，完整地描述将一份源程序编译成二进制可执行代码的完整过程，并进行二进制代码的模拟执行验证。这不仅仅是理论的分析，更是完整地实现一个精简的编译程序。本设计中编写代码简练，语法分析采用功能强大而又容易理解的 SLR(1)分析法，符合简化后的 C 语言的编译需求。本文对方案进行充分的论证，对编译器的重要模块进行详细的描述，对细节进行充分的说明。

本设计源语言为精简过的 C 语言，可用本设计编译简单的操作系统、设备驱动程序，目标代码可兼容自主处理器。本文对精简编译器设计做了详细的介绍，为编译器实现提出了一套有效的解决方案，具有较高的继续开发的价值，为编译原理的实践教学提供了有力支持。

**关键词：**编译器；SLR(1)；语法分析；中间代码生成；目标代码

# ABSTRACT

The compiler is system software that translates high-level language programs into machine language that the processor can recognize. Most books and textbooks mainly focus on theories and lack specific implementations for a simple compiler. Most domestic colleges and universities focus on compilation theory in teaching. Although a few colleges and universities have simple compiler development teaching, the algorithms they choose are too simple to adapt to complex grammar compilation needs. Only a few colleges and universities can provide more complete compiler practice teaching. The commercialized open source code is too powerful and the code is complicated, which is difficult to use for teaching.

This article will describe the complete link of compiling a source program into binary executable code from lexical analysis, grammatical analysis, intermediate code generation, assembly code generation to binary code generation, and simulate the binary code. This is not just a theoretical analysis, but also a complete streamlined compiler. The code written in this design is concise, and the syntax analysis uses the powerful and easy to understand SLR (1) analysis method, which perfectly meets the simplified C language analysis needs. This paper will give a detailed description of the important modules of the compiler, fully explain the details, make a more complete demonstration of the program.

The source language of this design is simplified C language. Simple operating system and device driver can be compiled by this design. The binary executable code can be compatible with independent processor. This paper introduces the design of the compiler in detail, and puts forward a set of effective solutions for the implementation of the compiler. It has a high value of continuous development and provides a strong support for the practical teaching of compilation principles.

**Key words:** Compiler; SLR(1); Syntax analysis; Generate intermediate code; Binary executable code

# 目 录

第一章 绪论.....	1
1.1 研究背景和意义.....	1
1.2 国内外教学用编译器现状.....	1
1.3 本文研究内容.....	1
第二章 编译前端算法.....	3
2.1 概述.....	3
2.2 词法分析.....	3
2.3 语法分析.....	3
2.3.1 文法设计总则.....	3
2.3.2 文法设计与描述.....	4
2.3.3 求解 first 集与 follow 集.....	6
2.3.4 求解识别活前缀的 DFA.....	6
2.3.5 移进-归约表.....	7
第三章 编译前端设计.....	8
3.1 编译前端概述.....	8
3.2 信息中心设计.....	9
3.2.1 符号表与 display 表功能.....	9
3.2.2 流程控制功能.....	10
3.2.3 函数控制功能.....	11
3.3 数据类型.....	11
3.3.1 数据类型识别过程.....	11
3.3.2 广义数据类型模型设计.....	12
3.3.3 结构体的模型设计.....	12
3.3.4 数据类型内存大小计算.....	14
3.4 语法制导和语义分析.....	15
第四章 中间代码.....	16
4.1 中间代码概述.....	16
4.2 数组.....	17
4.3 结构体.....	18
4.4 不同阶段的内存地址.....	18
4.5 临时变量与资源释放问题.....	18
4.5.1 临时变量的释放时机.....	18
4.5.2 临时变量的界限.....	19
4.5.3 解引用操作符与资源释放.....	19
第五章 编译后端.....	21
5.1 编译后端概述.....	21
5.2 信息管理.....	21
5.3 程序结构.....	21
5.4 临时变量的资源分配.....	22

5.4.1 中间代码 if-goto 与临时变量释放问题.....	23
5.4.2 运行时临时变量资源的分配释放的合理性.....	23
5.5 函数上下文保存与中断.....	24
5.6 栈空间申请与栈顶指针.....	25
<b>第六章 测试与仿真.....</b>	<b>26</b>
6.1 汇编器.....	26
6.1.1 汇编格式.....	26
6.1.2 汇编器输出格式.....	26
6.1.3 汇编器程序结构.....	27
6.2 模拟器.....	27
6.2.1 初始化.....	27
6.2.2 模拟器的使用.....	28
6.3 测试.....	28
<b>结束语.....</b>	<b>35</b>
<b>致 谢.....</b>	<b>36</b>
<b>参考文献.....</b>	<b>37</b>
<b>附录一.....</b>	<b>38</b>
<b>附录二.....</b>	<b>40</b>
<b>附录三.....</b>	<b>41</b>

## 第一章 绪论

### 1.1 研究背景和意义

编译器是将高级语言程序翻译成处理器可以识别的机器语言的系统软件<sup>[1]</sup>，因此编译器是联系操作系统与处理器运行环境的重要支撑软件，若要在处理器设计、编译技术、操作系统之间完成垂直打通，编译器的设计是关键一步。

编译器主要分为词法分析、语法分析、语义分析、中间代码生成、代码优化以及目标代码生成等部分。其所需使用的上下文无关文法、语法分析树、形式语言、集合划分等与离散数学有密切联系；词法分析中有限状态机、正则表达式等内容与计算机设计有密切联系；语法分析的自上而下和自下而上的方法，与数据结构等课程有机联系；属性文法和语法制导翻译、语义分析和语言的自动分析有密切联系；中间代码生成和目标代码生成与汇编、机器指令密切联系；而符号表、运行时存储空间组织、代码优化等都与程序语言及设计方法有密切联系。因此，开发一个完整的轻量级编译器本身就是一个复杂的计算机工程问题。而且编译技术对机器学习、人工智能等都有重要的支撑作用。同时，编译技术的发展使得编写大型工程成为可能，同时也带来了许多优化方案，例如：GCC 提供了数种编译选项优化等级，每个优化等级又提供了许多可选项，来适应特定优化需求<sup>[2]</sup>。

### 1.2 国内外教学用编译器现状

大多数书籍和教材重视理论进行深入论述，忽视完整编译器的开发实践，难以用于完整编译器的实践教学。国内大多数高校没有开设编译器具体实现的教学，少数高校虽然有简单编译器的开发教学，但是其选用的算法过于简单，难以适应复杂语法的编译需求，只有极少数高校能够提供比较完善的完整编译器的实践教学。开源编译器代码要么代码过于庞大难以用于大学教学，要么缺乏针对特定精简指令集的兼容。国外有商业化开源代码的 LLVM 等，但是代码庞大、功能冗杂，难以用于教学。GitHub 上有许多开源编译器项目（国内的和国外的），但是质量良莠不齐，普遍没有详细的文档，不能适应针对特定平台的编译要求，难以用于教学。

### 1.3 本文研究内容

编译器的开发实现具有很高教学价值。编译器的开发可以增加大学期间课程之间的宏观理解、加强复杂系统软件的开发能力、提升对计算机系统从硬件到软件整理通路的理解<sup>[3]</sup>。从头开发一个编译器有利于加深对大学整体课程的理解，其设计与开发涉及了许多专业课程和理论。生成汇编代码时要使用《微机原理与接口技术》的知识，对编译器的学习有利于强化对 C 语言的理解<sup>[4]</sup>。C 语言是计算机从业者学习最广泛的语言之一，因此编译器源程序的语言是在 C 语言的基础

上进行简化、修改后的语言，编译生成目标代码为二进制可执行代码。

由于本设计源语言为精简过的 C 语言，因此可用本设计编译简单的操作系统、设备驱动程序，目标代码可兼容自主处理器。本设计过程需具备一定的创新能力，同时又具有一定的趣味性。进行代码生成时需要对处理器、操作系统、编译器之间进行规则约束和协议约定，使学生有机会体验到定制关键语言翻译规则的过程，全面提高设计实践能力。

## 第二章 编译前端算法

### 2.1 概述

编译过程中，词法分析是针对源语言词法进行识别的过程，业界普遍采用词法分析自动机理论进行分析，其过程较为简单本章一带而过。语法分析是针对语法进行识别的过程，在编译算法的发展史上出现过很多的语法分析算法，该算法是编译过程的重点和难点，语法分析的过程更是为语义分析起到了重要的制导功能。语义分析过程更是牵扯到属性文法的设计，与具体实现联系密切，属于编译前端设计内容。本章着重讲述文法设计的部分细节和语法分析算法的实现。

语法分析算法采用 SRL(1)分析法，它比 LR(0)分析法功能强大，比 LR(1)分析法实现简单，其算法要点是如何计算移进-归约表。编译前端读取 lan/lan.txt 中的文法规定，进行针对文法的词法分析，并进行“求解项目集规范族”的运算<sup>[5]</sup>，生成语法分析的移进-归约表（或称之为状态转移矩阵）。

### 2.2 词法分析

词法分析是编译过程中第一个过程，是将源程序读入，将字符流从左到右一个一个字符扫描，根据词法规则识别单词（也称作符号）切割成若干元组的过程，是编译的基础，本项目中使用自动机技术进行词法分析。

单词是一个字符串，单词是源代码的基本组成单位。词法分析的过程就是将源程序单词化的过程。这个过程中，词法分析器还会将单词进行分类，添加标签，以便于语法分析进行进一步处理。

词法分析的输出为包含单词名、真值、坐标、标签等成员的元组，其具体成员由语法分析器和文法的具体设计决定。单词名应与文法中的“终结符”相对应。如文法中定义了<返回语句>->'return'<值>，则词法分析时遇到“return”时，该成员应当填写为“return”而不是“[关键字]”。真值用于保存常量，如词法分析识别到整数 12345，则使用真值成员保存该值，并将名字成员写为统一的“[常量]”名字，方便文法进行统一规定。坐标成员分别记录了该元组在源代码中的行、列，以便后续报错定位使用。标签成员用于标识该单词的属性，用于分类。

### 2.3 语法分析

#### 2.3.1 文法设计总则

在编译项目的开发之前，要首先确定源语言，而源语言的确定就要确定该语言的文法。文法的确定要满足语法分析算法的要求。越是功能强大的语法分析算法实现起来就越复杂，对文法的要求就越弱，文法设计自由度便越高。本设计采用的语法分析算法在工作量与文法自由度两者之间取了平衡，选取 SLR(1)分析



法。该算法对文法的要求主要体现在：

- 允许文法存在左递归和右递归。
- 允许一定程度的移进-归约冲突。
- 不允许文法的二义性，体现在不能出现两个等价的非终结符，否则会出现文法解决的移进-归约冲突。

为了计算方便，额外加入的一条规则：

- 文法不允许使用 $\epsilon$ 符号。

后期进行语法制导时，为降低工作量只进行归约项目的制导而不进行移进项目的制导，需要在设计文法时进行部分特殊设计：

- 文法对归约顺序有强要求时，需按照实际情况对文法进行拆分。

### 2.3.2 文法设计与描述

为满足开发过程中对文法的动态修改需要，要将文法以描述语言的形式保存在文件中。该 lan.txt 文法文件的格式规则<sup>[6]</sup>为：

- 非终结符由尖括号表示，如：<程序>。
- 源程序中出现的终结符字符串由单引号表示，如：'break'。
- 词法分析对应的类别终结符由中括号表示，如：[关键字]、[常量]。
- 产生式的推导符由箭头“->”表示。
- 具有相同左部的产生式进行合并时，使用“|”符号分割。例如  $a \rightarrow b$  和  $a \rightarrow c$  可合并为  $a \rightarrow a|b$ 。

● 一条产生式或者几条产生式以“|”合并后构成一“行”，每行由英文分号表示结束，空格、制表符、换行符等无意义。

● 文法在语言描述上由若干行组成；文法在编译角度上由若干产生式组成。使用“|”合并的一行仍然认为是多条产生式。也就是产生式个数=行数+“|”符号数。

使用上述描述语言对源语言文法进行描述，从而定义了简化后的 C 语言语法。宏观上看，我们将程序看作是若干外部声明的集合，外部声明包括：

- 函数定义或声明（无声明功能）。
- 变量定义或声明（无定义功能）。
- 结构体定义或声明（无声明功能）。

故有文法定义：

```
<程序>-><外部声明>|<外部声明><程序>;
```

```
<外部声明>-><struct 定义>';'|<函数定义>|<变量声明或定义>';';
```

函数的定义是由函数的声明和程序块组成。函数声明与变量声明的规则基本相同，故复用变量声明的规则，在语义分析时区分两者。在定义文法时，要考虑到将来可能增加函数声明的功能，要注意将函数的声明独立出来，有文法定义：

```
<函数定义>-><函数定义头><程序块>;
```

<函数定义头>-><函数声明>;

<函数声明>-><变量声明或定义>;

程序块指的是由花括号包围的语句列表，区分了 C 语言的物理层次和变量的命名空间。有如下定义：

<程序块首部>->'{';

<程序块尾部>->}';

<程序块>-><程序块首部>

<语句列表>

<程序块尾部> | <程序块首部>

<程序块尾部>;

结构体的定义是由结构体的声明和变量声明列表组成。有如下定义：

<struct 定义>-><struct 声明><程序块首部>

<变量声明或定义列表>

<程序块尾部>|<struct 声明><程序块首部><程序块尾部>;

<struct 声明>-><struct 数据类型>;

<struct 数据类型>->'struct'[关键字];

<变量声明或定义列表>-><变量声明或定义>;<变量声明或定义列表>|<变量声明或定义>;

注意：由于在语法分析识别过程中，<struct 定义>符号的归约识别可能晚于该结构体内部变量的声明，也就是存在自己类型的指针成员，如：

```
struct node{
    ... ..
    struct node *next;
    ... ..
};
```

识别 next 成员时，结构体 node 还未定义完毕，符号表中还没有该结构体的完整信息，故需要做到先声明后定义。也就是在语法分析过程中要保证先进行声明的归约再进行成员的归约。在文法中体现为产生式<struct 声明>-><struct 数据类型>的分离。

其他语法成分与上述定义相似，完整文法定义见附录一。

文法定义完毕后，程序将从文件中将文法读入，并将文法字符串离散化：将终结符与非终结符变为数字标号表示；将“|”符号合并的产生式分离，将所有产生式列表求出；对文法进行校验，检查文法的完整性。这样，为后续进一步计算做好了准备。

### 2.3.3 求解 first 集与 follow 集

设计的文法不允许存在终结符 $\epsilon$ ,故求解 first 集和 follow 集的算法有所简化。  
注意：求解上述两个集合时要求文法没有被扩展。

first 集求解算法：

对于文法中的符号  $X \in \text{终结符} \cup \text{非终结符}$ ，其  $\text{first}(X)$  集合可反复应用下列规则计算，直到其  $\text{first}(X)$  集合不再增大为止：

- 1) 若  $X \in \text{终结符}$ ，则  $\text{first}(X) = \{X\}$ 。
- 2) 若  $X \in \text{非终结符}$ ，且具有形如  $X \rightarrow a\alpha$  的产生式，则把  $\text{first}(a)$  加入  $\text{first}(X)$ 。  
也就是把  $X$  能推出的第一个终结符加入  $\text{first}(X)$ 。

follow 集求解算法，求解非终结符  $A$  的随符集  $\text{follow}(A)$ ：

- 1) 对开始符号  $S$ ，将  $\#$  加入  $\text{follow}(S)$ ，然后再按后面的处理。
- 2) 若  $B \rightarrow \alpha A \beta$  是文法的产生式，则将  $\text{first}(\beta)$  加入  $\text{follow}(A)$ 。
- 3) 若  $B \rightarrow \alpha A$  是文法的产生式，则将  $\text{follow}(B)$  加入到  $\text{follow}(A)$ 。
- 4) 反复使用 2) - 3)，直到 follow 集合不再增大为止。

### 2.3.4 求解识别活前缀的 DFA

求解该 DFA 的方法有两套方案：

- 文法  $\rightarrow$  正规式  $\rightarrow$  NFA  $\rightarrow$  DFA。
- 文法  $\rightarrow$  first 集和 follow 集  $\rightarrow$  求项目集规范族(DFA)。

上述两套方案在《编译原理》课程中都是经常考察的，算法上并无明显的优劣之分。本次项目选择第二套方案，是因为：

从项目开发的角度来看，第一套方案的 NFA 和 DFA 需要表示和求解两个与文法等价的图，图的表示和求解具有很大的复杂性，对后期的调试和排错带来有很大的难度。而第二套方案虽然在计算过程中项目集会略显庞大，但只有一个图，难度相对更简。

计算结果规模较为庞大。文法词法分析输入：产生式 78 个；非终结符有 38 个；终结符有 25 个；输出的 DFA：计算出项目集有 133 个，共计 846 条项目；单向边个数为 529 个；

项目集规范族的求解算法如下，设  $S$  为初始符号：

- 1) 对与扩展文法： $G[E]$  进行扩展： $S' \rightarrow E$ ，使此产生式为第一个项目集。
- 2) 由前驱状态  $A \rightarrow a.xb$  增加一个字符  $x$  能得到的  $A \rightarrow ax.b$  状态，加入项目集。
- 3) 对于某个项目集，若  $A \rightarrow a.Bc$  属于项目集，若文法存在产生式  $B \rightarrow xxx$ ，则将  $B \rightarrow .xxx$  加入项目集。

如果构造 DFA 过程中遇到移进-归约冲突，则要尝试解，例如项目集包含如

下产生式：

$A \rightarrow rD.$

$D \rightarrow D.i$

则遇到符号  $i$  时就会发生移进-归约冲突,解决方法为检查符合  $A$  的  $\text{follow}$  集是否与移进符号集合相交,也就是  $\text{follow}(A)$  和  $\{i\}$  是否相交,若不相交就能区分开来。也就是说,可以有多个移进项目,但是如果存在归约项目,则归约项目必须唯一且不能同时存在移进项目(否则就要考虑解决冲突)。从这一用例可以看出,SLR(1)分析法允许左递归的存在,只需要保证  $\text{follow}(A)$  和  $\{i\}$  不相交即可。

该算法本身不会出现移进-移进冲突,如果文法设计合理则不会出现归约-归约冲突。

求解 DFA 的函数为 `int create_prod_set_DFA()`。得到 DFA 后,需要将图转换为移进-归约表(状态转移矩阵)。我们称矩阵每一个格子为 **cell**。每一个非空 **cell** 为一个移进-归约动作,需要绑定一个制导函数用于语义分析(实际开发选择只为归约动作放置制导函数)。空 **cell** 为错误状态,理论上可以为所有空 **cell** 放置错误处理函数的函数指针。但由于 **cell** 数量巨大,对此处报错不予特殊处理。

由于没有给移进动作放置制导函数,编译器并不会从移进动作中获取额外信息。但是实际情况下归约前需要提前获取一些必要的信息,如:

<程序块> $\rightarrow$ ‘{’ ‘}’;

上述产生式识别到‘{’时,信息中心需要立即获取到该信息,将程序层次结构提升一级。这时可以通过改造该文法达到该目的:

<程序块> $\rightarrow$ <程序块首部>‘{’;

<程序块首部> $\rightarrow$ ‘}’;

改造后,识别第二个产生式时即可通过制导函数告知信息中心相关信息。

### 2.3.5 移进-归约表

已知 DFA,还需将 DFA 变为状态转移矩阵,也就是引导语法分析总控程序进行状态转移和归约的移进-归约表。

算法为:

- 所有的边均为移进动作。
- 项目集中所有可归约项目(形如  $A \rightarrow \alpha.$  的项目)均可以分别产生归约动作。
- 若  $S' \rightarrow E$  属于  $S_i$ ,则置  $\text{Action}[S_i, \#] = \text{acc}$ 。

得到矩阵(也就是移进-归约表)为稀疏矩阵,且后续需要查找某个 **cell** 是否存在,因此采用 `vector<map<int, map_cell_t>>` 的数据类型保存该表,该数据结构是二维结构,第一维是线性表,第二维(也就是线性表中每个元素)是红黑树维护的字典。可以通过第一维选中当前状态机的状态,通过第二维字典快速查某个动作是否存在。

### 第三章 编译前端设计

#### 3.1 编译前端概述

编译前端运行前要对文件进行相关运算得出移进-归约表，之后进行针对源程序的读入和词法分析，生成元组列表（`vector<word_node_t>word_lst`），在 `main` 函数中进行一下元组的类型转换（接口转换），将 `word_node_t` 列表转为以 `tree_node_t` 为基本单位的符号栈。

进入语法分析和语义分析阶段，初始化时将语法制导函数(后文称之为 `guider`)绑定到产生式上，进入语法分析的总控程序 `slr1_ctrl` 函数，在语法分析的同时进行语法制导翻译。在语法分析的过程中，产生归约时通过“语法制导函数控制器”调用产生式绑定的 `guider`，完成语义分析。

语义分析过程中需要与信息中心进行交互，`symbol_controller` 单例包含了符号表以及其他必要的信息。流程如图 3.1 所示。

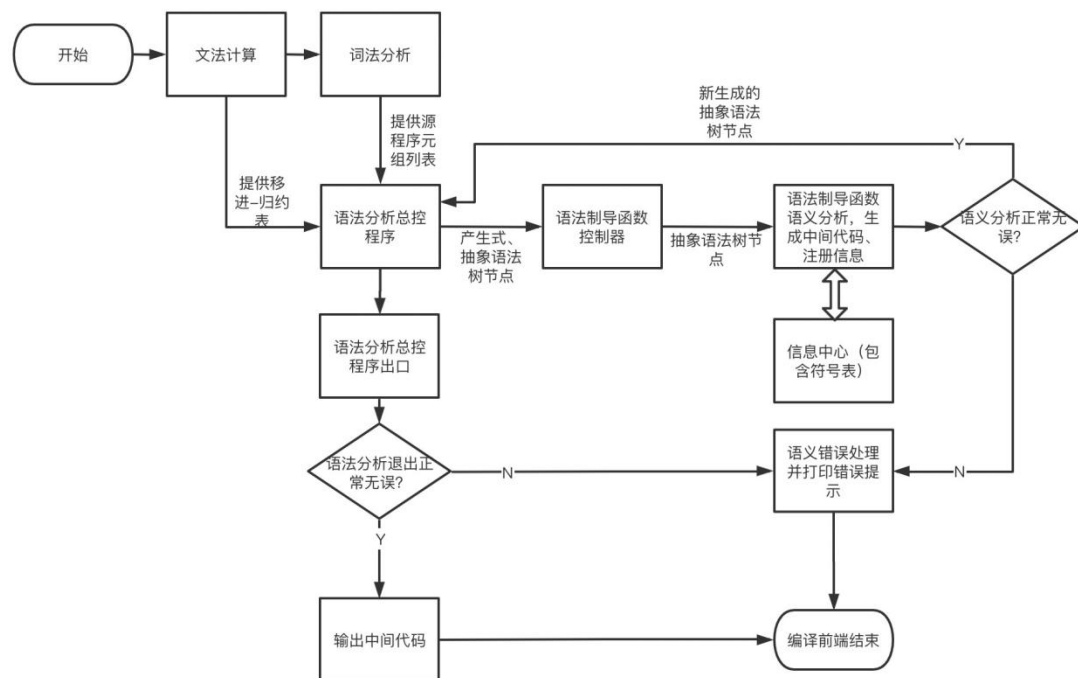


图 3.1 编译流程

其中语法制导函数控制器用于绑定产生式和对应的制导函数，初始化函数中使用产生式签名-制导函数指针注册的方式将两者绑定。这样设计优点有：开发过程中不要求完全实现所有的制导函数，可按需增加制导函数，更为灵活；产生式与制导函数的映射关系可读性较好，不容易出错。

## 3.2 信息中心设计

### 3.2.1 符号表与 display 表功能

信息中心中集成了符号表的功能，符号表的实现是编译器的核心<sup>[7]</sup>。C 语言中不允许嵌套函数，对于 display 表的物理结构管理功能要求不高，故也直接集成在了符号表中。

符号表（集成了 display 表）：symble\_stack 由基本组成单位 Symble\_block\_t 组成。Symble\_block\_t 简称 block，每个 block 对应一个程序块（也就是一个层次）。当编译器识别到程序块首部时，则会创建一个 block 并 push 到符号表中，进行程序块归约后会将 block 从符号表中 pop 出并销毁。每个 Symble\_block\_t 以栈为基本结构，由若干 Symble\_node\_t 组成。

Symble\_node\_t 简称 snode，每个 snode 对应一个符号，包括函数名、全局变量、局部变量、数据类型、struct 数据类型、返回值等，如图 3.2 所示。

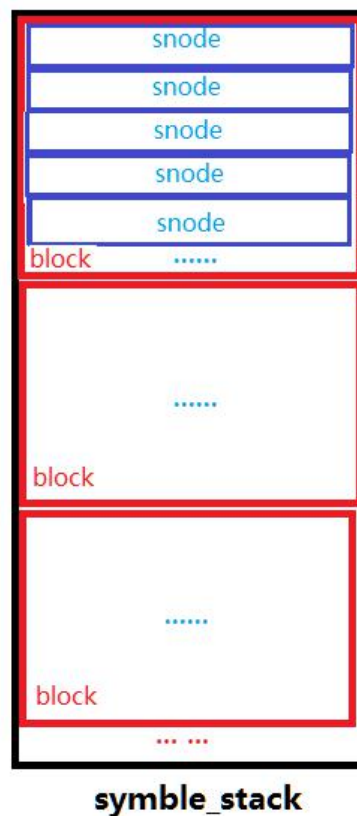


图 3.2 符号表结构图

符号表每一个 block 都可以被称之为一个层次：符号表的第 0 层由构造函数自动 push 进入栈中，命名为“\_\_main\_stack”，该层用于存放全局信息，如全局数据类型、全局变量等。符号表的第 1 层由制导函数 push 进入栈中，该层可以是函数体或结构体定义。符号表的第  $n$  ( $n \geq 2$ ) 层由制导函数 push 进入栈中，这

些层可以是函数内的结构体定义、函数内嵌套的程序块、函数内的分支或循环语句的程序块等，不可能是函数定义。

编译开始时由信息中心初始化 push 第 0 个 block，然后由 main 函数调用各类初始化函数注册各种全局信息（如注册数据类型 int）。之后经由词法分析、语法分析进入语法制导函数，针对语法和语义向符号表栈内 push 或者 pop 层次 block。最终结束编译后，符号表栈内应当形成平衡。

符号表提供的主要功能有：

- 增加和销毁 block 的功能（物理层次和命名空间管理）。
- 向 block 中添加和销毁 snode 和功能（符号管理）。
- 按命名空间查询 snode 的功能（符号与命名空间管理）
- 透传 block 其他功能。

其中 block 除了基本功能外，还为中间代码生成提供了如下功能：

- 获取所有临时变量的总大小。
- 获取所有变量的总大小（不包括临时变量）。
- 获取所有临时变量的 snode。
- 获取所有变量的 snode（不包括临时变量）。
- 获取所有需要释放的临时变量（可用于语句结束的清理工作）。
- 合并两个 block。

### 3.2.2 流程控制功能

流程控制功能功能要旨为：为中间代码的生成提供流程控制语句的合法性检测，和对符号的查询。

在语法制导函数中，在流程控制语句中语义分析需要得知当前所处的特定环境(如 break 语句的分析中需要检测 break 是否是在 while 循环内，并且需要获取 break 的跳转地址)，需要信息中心提供相关的注册和查询功能。

针对循环语句的主要功能有：

- push 一个循环（注册）。
- pop 一个循环。
- 查询是否允许 continue。
- 查询是否允许 break。
- 查询 continue 跳转地址。
- 查询 break 跳转地址。
- 查询循环 then 地址。
- 查询循环 continue 地址。
- 查询循环 break 地址。

针对分支语句的主要功能有：

- push 一个分支（注册）。
- pop 一个分支。
- 查询分支的 then 地址。
- 查询分支的 else 地址。
- 查询分支的 endif 地址。

针对返回语句的主要功能有：

- 注册功能（同 block 的 push 功能）。
- 查询是否可以返回。
- 查询 return 跳转地址。

### 3.2.3 函数控制功能

信息中心对函数控制提供的功能支持包含三个功能：层次控制功能、参数控制功能、返回值控制功能。

层次控制功能：上文已经将符号表对层次（block）的管理说明清楚，对于函数而言层次控制功能的意义就是区分出 block 是否是第 1 层，也就是函数定义层。若是函数定义层，则会自动注册或者销毁该函数。C 语言不允许函数嵌套，也不允许结构体内包含函数，所以函数的函数体必然注册于第 1 层的 block。

参数控制功能：函数定义在语法上包含一个程序块，程序块内部也可以包含程序块，在本项目中程序块的制导函数是复用的。两种程序块不同仅为前者可能会包含参数（也就是存在额外定义的变量），后者没有。当前语法制导的程序结构不容易通过传递参数的方式区分这两点，故需要信息中心提供相应的信息（这是全局的）。

返回值控制功能：编译时遇到 return 语句需要查询当前语义是否允许返回、返回地址、返回值类型（本设计返回值类型必须是 int 型），信息中心应当提供相应的功能。

## 3.3 数据类型

### 3.3.1 数据类型识别过程

数据类型的声明是一个递归的过程，其定义和解析是两组互逆操作，使用栈来构建程序要比递归更加灵活且容易调试。但是语法分析的顺序是由内而外，对于抽象语法树来讲又是一组递归操作，恰好与解析顺序相同。所以，虽然数据类型的分析是一种递归的形式，但最终是用顺序表来保存数据类型各个元素的。

一个数据类型是由若干数据类型相关的符号组成的，比如变量定义：

```
int *p;
```

变量 p 的数据类型为 int \*, 是由符号 int 和 \* 组成的，两者都是数据类型相关的符号。为了方便，我们将组成数据类型的基本元素称为“类型符”，类型符是数



据类型基本组成单位，包括：

int、\*、[]、函数括号()、参数列表、优先级括号()。

### 3.3.2 广义数据类型模型设计

数据类型是由结构体和类型符组成的。每个类型符都有不同的属性，故将每种类型符单独定义成类，但是这样一来，不同的类型符是不同的类，而栈内所有元素要求类型是相同的。故将所有类型符共有的操作抽象成接口，打包成纯虚基类，所有类型符继承该基类，使用该基类指针构造栈结构<sup>[8]</sup>。该纯虚基类为（广义数据类型类）：`general_data_type_t`，包含的接口见文件 `src/data_type.h`。类型符、结构体程序中的定义如表 3.1。

表 3.1 数据类型元素定义表

类型符	类	继承（接口）
int	<code>primary_type_int</code>	<code>general_data_type_t</code>
*	<code>primary_type_pointer</code>	<code>general_data_type_t</code>
[]	<code>primary_type_array</code>	<code>general_data_type_t</code>
函数括号	<code>func_type_t</code>	<code>general_data_type_t</code>
优先级括号	无	无
空类型（用于初始化）	<code>primary_type_undefined</code>	<code>general_data_type_t</code>
结构体	<code>struct_type_t</code>	<code>general_data_type_t</code>

数据类型类为 `data_type_t`。现阶段数据类型类不参与递归结构，暂不继承 `general_data_type_t` 类，但实际上实现了 `general_data_type_t` 内部接口，可随时根据需要继承该虚基类，具有较好的可扩展性。

数据类型类中核心成员为类型符栈 `type_stack`。数据类型语法分析时，先识别最内层的（最靠近变量的）类型符，越靠近变量的类型符就越接近栈底，越接近栈底的类型符就越能描述变量的属性。

### 3.3.3 结构体的模型设计

编译器开发时不可能为每一个结构体类型创建一个类，故所有的结构体类型均为同一个模板，也就是同一个类，结构体关心的内容主要为：

- 成员变量名。
- 成员变量的数据类型。
- 成员变量的相对结构体头部的偏移地址。

但是请注意，我们不能仅仅简单地记录上述三点以及映射，要考虑到一种特殊情况：在 C 语言中允许结构体的成员可以是自己类型的指针，例如最经典的链表节点定义：

```
struct node{
    struct node * next;
};
```

当编译器识别到 `struct node* next` 时，如果符号表中没有 `struct node` 类型的记录，编译器将会不知所措，但此时 `struct node` 类型又没有真正被声明完毕，无法向符号表提交一个完整的类型符号。

那我们是否可以先向符号表提交一个不完整的类型符号，然后在定义彻底完成后修改该符号呢？这种思路在本工程中也是不成立的，因为当识别到 `struct node * next;` 时会记录下 `next` 成员的数据类型，也就是从符号表中取出 `struct node` 的定义并记录下来，而这个记录是不完备的，哪怕将来我们修改符号表中的定义也不会影响这个不完备的记录。所以，解决这一系列矛盾有两种方案：

方案 1：既然修改符号表中的定义不能影响不完备的记录，那在识别 `struct node * next;` 语句时可以考虑记录数据类型的地址（也就是将拷贝改成引用），这样修改符号表中的符号就能同步至所有不完备的记录。

此方案本身无明显的缺陷，但是开发过程中本工程中局部变量、全局变量等的所有数据类型记录均为拷贝（拷贝易于进行其他修改），若要将结构体成员的数据类型改为指针，不符合整个工程对变量操作的统一抽象，影响整个工程对数据类型的处理。

方案 2：大多工程结构性问题都可以使用中间量来解决。我们为真正的结构体类实例前面增加一个“门户”，该门户只记录结构体名，如图 3.3 所示。当识别到结构体声明 `struct node` 语句时，就向符号表中插入该门户，此门户是完备的。后续声明完成时，创建一个真正的结构体类实例，加入到一个字典中。当需要对该数据类型进行操作时，门户类会从字典中查找真正的结构体类实例，将操作透传至该实例并将返回值传回，以达到相同效果，如图 3.4 所示。

方案 2 更契合本工程对数据类型的处理，故采用方案 2。

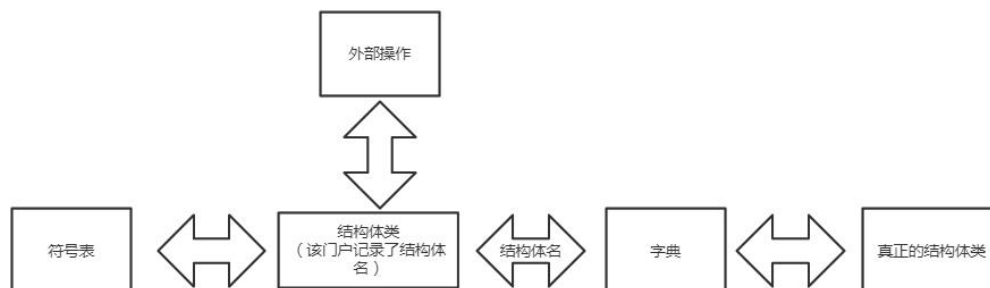


图 3.3 结构体类型宏观图

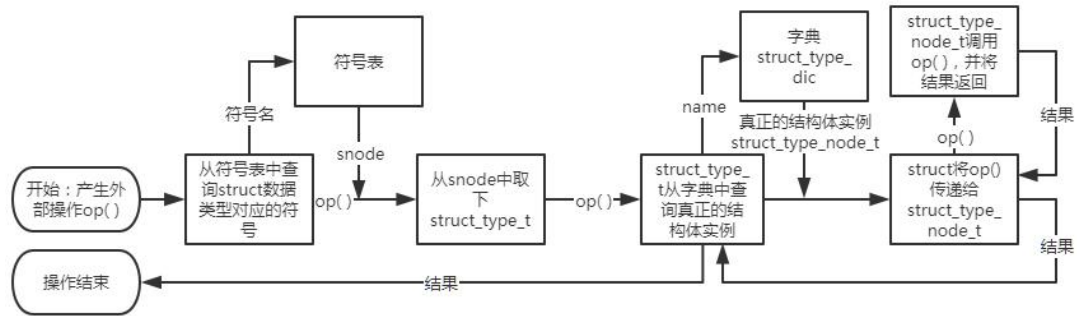


图 3.4 结构体类型操作流程图中

`struct_type_t` 为方案 2 中的“门户”，保存了结构体名，该类在外部被当作真正的结构体类型使用，但实际上它仅仅是个门户。该类继承了 `general_data_type_t`，其实现的所有接口的操作均为：

- 从字典中查找该类的实例。
- 调用该实例对应的接口。
- 将返回值（若有）传回调用者。

`struct_type_node_t`：该类为真正的结构体类型，保存了成员名列表、成员数据类型列表、成员偏移地址列表，不需要继承 `general_data_type_t`，但实际上必须实现 `general_data_type_t` 所有的接口。

`struct_type_dic` 用于保存结构体名到 `struct_type_node_t` 类的映射关系并提供映射服务，即上文中的字典。

### 3.3.4 数据类型内存大小计算

由于数组的存在，为了抽象方便，我们认为所有的变量都由若干单元格（子变量）组成。如：数组 `int arr[2][5]` 中的 `arr` 变量由 10 个格子组成，每个格子都是一个 `int` 类型的变量。再比如 `int *p` 中的 `p` 变量由 1 个格子组成，每个格子都是一个 `int*` 类型的变量。

这样某个时间类型的大小 `sizeof` 的计算就拆分成计算格子数（`cellnum`）与计算每个格子的大小（`cellsize`），而 `sizeof=cellnum * cellsize`。如此划分，有利于计算指针与数字的加法计算，和数组的偏移量计算。

`cellnum` 的计算方法为：

若某变量 `var` 是一个数组，则其数据类型的 `cellnum` 应为数组所有维度容量的乘积；若非数组，`cellnum` 应为 1。若某变量 `var` 是一个数组，则其数据类型中的 `type_stack` 某个非空前缀的所有类型符的 `is_array()` 均应为真，其中只有类型符 `[]` 的 `is_array()` 接口会返回真，其余均为假。若 `var` 是一个数组，则 `cellnum` 等于此前缀中所有类型符 `[]` 对应的的最大容量的乘积。

单元格（`cell`）的数据类型为：若变量 `var` 非数组，则 `var` 的数据类型就是

cell 的数据类型；若变量 var 是数组，则 type\_stack 去除 is\_array() 为真的前缀后，剩余部分即为单元格的数据类型。

cellsize 的计算方法为：

若某变量 var 的单元格是指针（包括普通指针、函数指针等，数组名除外），则 cellsize 等于一个机器字。如 int var[2] 的单元格类型为 int，大小为一个机器字。若非指针，则应当由单元格的数据类型的 type\_stack 栈底的类型符决定。例如 int \*var[2][3]；栈为[2][3]\*int。去除数组前缀后 type\_stack 为\*int，栈底的类型符为\*，其 sizeof() 接口返回值为 4 所以 cellsize=4。

### 3.4 语法制导和语义分析

源程序从语法的角度可以看作是一个语法树的展开。而在 SLR(1) 分析的过程中，总控程序不断进行移进和归约操作，此种移进-归约可看作对语法树的深度优先搜索。这种语法树只存在于抽象逻辑，并不需要在内存中实际创建出来（从语法分析的角度来讲，语法树创建出来就是用来搜索的，现在已经有了搜索过程就不需要多此一举了）。

终结符为语法树的叶子，非终结符为语法树的非叶子节点。我们为语法树的节点构造的类为 tree\_node\_t，它记录了以它为根的子树的语义分析结果（包括了中间代码与各种信息）。

语法分析前，会为每个终结符构造 tree\_node\_t 实例，构成符号栈以供语法分析使用。在进行归约操作时，归约使用的产生式对应的制导函数会被调用，出栈的所有符号作为参数，该制导函数最终返回一个 tree\_node\_t，对应归约后产生的非终结符代表的节点，包含了若干信息，并重新压入符号栈中。

语法制导函数的函数结构大同小异，其功能各不相同。以<成员选择表达式>-><成员选择表达式>.'[关键字]对应的制导函数为例，该制导函数为 guid\_func\_70，传入 3 个参数分别是产生式右边的三个符号对应的语法树上的节点<成员选择表达式>、'.'和[关键字]的三个 tree\_node\_t 实例。返回值为父节点（产生式左部<成员选择表达式>）的 tree\_node\_t 实例，其流程为：

- 取出参数 0 和参数 2，参数 1 无用。
- 初始化返回值 tree\_node\_t 类型实例 ret\_word，设置 ret\_word 可作为左值。
- 取出参数 0 承载的变量 a，并从符号表中将其信息取出。
- 校验 a 是否是结构体。
- 将参数 2 承载的成员名取出，并添加合适的前缀。
- 从结构体 a 中将成员的信息（数据类型和偏移地址）取出。
- 构造中间代码取变量 a 的首地址，并与偏移地址计算得出成员地址。
- 利用成员地址访问成员，申请并向符号表中注册使用的临时变量。
- 向 ret\_word 赋值并返回。

## 第四章 中间代码

### 4.1 中间代码概述

中间代码解决了编译前端和编译后端的多对多的问题，也就是多个源语言 and 不同平台的兼容问题。中间代码在功能上与源程序等价，在语法上是一种简单的线性结构。

中间代码输出文件为 `mid.c` 文件，宏观上由两部分组成：

- 命令：包含了构建程序的信息。
- 四元式：对应若干条汇编代码。

中间代码为编译后端提供了一些信息，这些信息应当包含了最小信息子集。所谓最小信息子集为：编译后端构建程序所需要的最小的信息集合。也就是说中间代码中可以有信息冗余，编译后端可以利用冗余信息更简单、高效、合理地构建目标程序。

应当注意的是，中间变量中不宜含有过多机器硬相关的内容，要最大程度上保证机器无关性。如函数调用时要保存的上下文，应当使用命令的方式来表示，交由后端进行展开，而编译前端不应该填写寄存器具体细节。

命令是指由双冒号开头的语句，可以由多行组成，行数由命令本身的结束标志决定，只有一行的命令由换行符作为结束标志。所有的命令有：

表 4.1 中间代码命令表

命令	功能描述
<code>::push_registers</code>	将上下文相关的寄存器 <code>push</code> 入栈
<code>::push_stack_var</code>	<code>push</code> 若干字节（为栈内变量预留空间）
<code>::var_stack_alloc_begin</code>	声明一些栈内变量（可以是局部变量）
<code>::push_global_func</code>	声明函数
<code>::push_global_var_begin</code>	声明全局变量
<code>::push_stack_tmp</code>	为临时变量 <code>push</code> 栈内空间（未使用）
<code>::push_stack_tmp_unfree</code>	提前为临时变量 <code>push</code> 栈空间（未使用）
<code>::free_tmp_var</code>	释放临时变量占用的寄存器或栈空间
<code>::pop_stack_tmp_unfree</code>	与 <code>::push_stack_tmp</code> 对应
<code>::pop_stack_tmp</code>	<code>pop</code> 临时变量的栈内空间（未使用，仅有正确性检查）
<code>::pop_stack_var</code>	与 <code>::push_stack_var</code> 对应
<code>::pop_registers</code>	与 <code>::push_registers</code> 对应
<code>::tmp_var_stack_alloc_begin</code>	与 <code>::var_stack_alloc_begin</code> 相似，声明的是临时变量
<code>::return</code>	函数返回
<code>::set_func_arg</code>	置函数参数，现阶段只允许一个函数参数
<code>::get_func_arg</code>	取函数参数

四元式由四部分组成：

- 操作符 `op`
- 操作数 `arg1`（可空）
- 操作数 `arg2`（可空）
- 结果 `ans`（可空）

为了提高四元式的可读性，将上述四部分代码设计为如下格式：

`ans = arg1 op arg2`

其中若 `ans` 为空，则省略符号“=”。

例如：

- 无 `ans`：

`goto __sign_1_if_then`

其中 `op` 为 `goto`，`arg1` 为 `__sign_1_if_then`，其余为空。

- 无 `arg2`：

`a = %__ADDRESS_OF__ b`

其中 `op` 为 `%__ADDRESS_OF__`，`arg1` 为 `b`，`ans` 为 `a` 其余为空。

- 无 `op`：

`a = b`

其中 `arg1` 为 `b`，`ans` 为 `a`，其余为空。

- 全部包含：

`a = b + c`

其中 `op` 为 `+`，`arg1` 为 `b`，`arg2` 为 `c`，`ans` 为 `a`。

## 4.2 数组

为了体验不同的设计思路，数组名在本编译器中被当作指针实体使用，标准 `c` 中将数组名作为地址常数而非实体。设一个机器字为 `unit`。

形如 `TYPE arr[H][W]` 的数组，其内存大小为数组的大小加指针的大小。数组大小为 `H*W*sizeof(TYPE)`，指针大小为 `sizeof(指针)=1 unit`。

在编译前端中，`arr` 的大小 `sizeof(arr)` 被认为是 `H*W*sizeof(TYPE) + 1`。在中间代码中，将数组以普通指针相同的实现方式实现，数组初始化时创建指针 `arr`，并声明 `sizeof(arr)` 大小的空间，将 `arr` 的大小重设为 `1 unit`，并使其指向自己，接下来中间代码为 `arr` 赋值，并将其指向它下一个 `unit`，这时 `arr` 指向的内存实体便为数组的首机器字。由于多维数组在内存中最终以线性的形式展开，为了方便理解我们把多维数组 `arr[H][W]` 看成一维数组 `arr[H*W]`，数组保存的内容看作 `val[H*W]`，则数组的内存线性模型如表 4.2 所示。

表 4.2 数组  $\text{arr}[H*W]$  的内存模型

内存	$\text{arr}$ 占用的内存 $= H*W*\text{sizeof}(\text{TYPE}) + 1$					
地址	$\text{arr}[-1]$	$\text{arr}[0]$	$\text{arr}[1]$	$\text{arr}[2]$	...	$\text{arr}[i*H+j]$
含义	$\text{arr}$ 指针	$\text{arr}[0,0]$	$\text{arr}[0,1]$	$\text{arr}[0,2]$	...	$\text{arr}[i,j]$
值	指向 $\text{arr}[0]$	$\text{val}[0,0]$	$\text{val}[0,1]$	$\text{val}[0,2]$	...	$\text{val}[i,j]$

在中间代码与编译后端中，数组实际上使用指针来模拟实现，故中间代码与编译后端只需提供指针的功能，无须增加额外的功能。

### 4.3 结构体

本编译器中使用结构体基地址+成员偏移地址来实现结构体的成员访问，不考虑内存对齐。如 3.3.3 描述，结构体声明时，记录的主要成员信息包括：成员名，成员数据类型，成员偏移地址。

设结构体 `node` 存放起始地址为 `addr`，`node` 有 `n` 个成员，设第 `i` 个成员的数据类型为 `typei`，该数据类型的大小为 `sizeof(typei)`。则有偏移地址 `offset` 公式：

- $\text{offset}(0)=0;$
- $\text{offset}(i)=\text{offset}(i-1)+\text{sizeof}(\text{type}(i-1));$

该公式等价于：

$$\text{offset}(i)=\text{sizeof}(\text{type}0)+\text{sizeof}(\text{type}1)+\dots+\text{sizeof}(\text{type}i)-\text{sizeof}(\text{type}i).$$

与数组的实现相似，结构体在中间代码中使用指针的方式实现，中间代码与编译后端无须增加额外功能。

### 4.4 不同阶段的内存地址

中间代码生成时，要注意编译前端无法获取到某变量存放位置（寄存器、栈空间、全局空间等），更无法获知变量地址（虚地址或物理地址），这是由编译后端决定的。所以编译前端对地址的操作，应使用取地址运算符和中间变量来获取和代替地址。

比如在结构体的成员操作中，需要获取结构体变量的首地址，与成员的偏移地址相加。其中首地址必须由取地址运算符进行取出，并赋值给某中间变量，再使用中间变量与偏移地址相加得到成员变量的真实地址。所谓真实地址，是程序本身使用的地址，可以是物理地址，也可以是虚地址（可由操作系统、硬件转换成物理地址）。

这是一个典型的编译前端时和编译后端时的问题，而不是编译时和运行时的问题。

### 4.5 临时变量与资源释放问题

#### 4.5.1 临时变量的释放时机

临时变量的释放是指：编译后端将模块内存、寄存器分配给某个临时变量，

使用期间这块资源不能被其他变量使用，当临时变量使用完毕后应当将资源释放以备重复利用。

从难度与效率考虑，采用以【c 程序的语句】为单位的释放策略比较恰当。其中 c 程序以一个分号表示语句的结束。所有方案有如下三种：

- 临时变量使用完毕立即释放。
- 当前方案。每条语句结束时释放资源。
- 函数返回时统一释放。

方案 1：资源利用率最高，难度也最高。情况较为复杂，若释放不得当会造成程序错误。对解引用语句（见 4.5.3）等若干部分均有影响。

方案 3：最为简单，可大大降低编译后端的设计难度。但资源利用率过低，尤其是函数越长资源利用率就越低，会造成函数内资源浪费的叠加。

方案 2：资源利用率适中，当每条语句较短时利用率可与方案 1 接近，不会造成资源浪费的叠加，难度适中。

#### 4.5.2 临时变量的界限

临时变量具有一个很重要的特点：其作用域（或者叫做生命周期）最大不会超出一条 c 语句的范围。换句话说临时变量不会同时出现在两条 c 语句的范围内。证明：按照一条语句通过是否会产生运算结果分，有如下几种情况：

- 语句是常规语句无结果：如单纯一个分号。
- 语句包含程序块，如 while 语句。可看成特殊的过程(函数)调用。
- 语句有结果但没保存：如函数调用 funccall();再如 1+2;
- 语句有结果且保存了：如 a=funccall();再如 a=1+2;

不管是什么情况，语句最多可以产生一个最终的运算结果。若语句产生了运算结果但没保存、或者是语句没结果，该语句的计算在语句过后就结束了，对环境的影响也均已完成，故不会遗留中间结果。若语句产生了运算结果且保存起来，那么该语句的计算结果只会保存到变量（局部变量或全局变量或指针指向的变量）里，不可能保存到中间变量里，其运算过程对环境的影响也结束了。故也不会遗留中间结果。

综上所述，语句结束后不会遗留中间结果，也就不会有任何临时变量具有存在价值。

#### 4.5.3 解引用操作符与资源释放

解引用操作符四元式格式为：

`ans = ptr % __MEMORY_OF__ size`

功能为取指针 ptr 指向的内存空间对应的变量(也就是\*ptr)。该变量可作为左值也可作为右值。并将该引用赋给 ans。由于 ans 保留了\*ptr 的左值，故需要记



录下 `ptr` 的真实值。而此 `ptr` 存放位置有若干种情况：

- `ptr` 为立即数。
- `ptr` 存放在寄存器里。
- `ptr` 存放在栈空间里。
- `ptr` 存放在全局空间里。

其中只有情况 1 可以在编译时得知 `ptr` 的具体内容，其他情况需要在运行时取出此地址。故在情况 234 中必须将 `ptr` 本身的地址信息保存下来，以备在运行时将内容取出。

若 `ptr` 非立即数，且 `ptr` 最后出现的位置为上述中间代码语句，由于 `ans` 保存了 `ptr` 各种信息，且后续可能会利用这些信息读取 `ptr` 的内容，就相当于变量 `ptr` 的生命周期以 `ans` 的形式延续了下去。直到对 `ans` 进行了其他操作（取左值、取右值或不再使用），才会结束此种延续。这种特性对变量 `ptr` 的释放带来了难度，有几点我们必须要考虑清楚：

若 `ptr` 是全局变量，那么 `ptr` 本身不需要在程序结束前被释放，自然没有任何问题。

若 `ptr` 是局部变量，那么 `ans` 本身不会在函数外部使用，且 `ptr` 不会在函数结束前被销毁。故不会产生错误的释放。

若 `ptr` 是临时变量，`ans` 的声明周期可能会大于 `ptr` 的声明周期。但 `ans` 的第一次使用（或者不使用）不会超出一条 `c` 语句的范围（也就是 `*ptr` 不会作用于语句之外），故采用“以语句为单位的释放策略”不会因为此种特性产生错误。

## 第五章 编译后端

### 5.1 编译后端概述

编译后端负责将机器无关的中间代码翻译成机器相关的目标代码（汇编代码）。因为中间代码的语法是一种简单的线性结构，所以编译后端相对于编译前端实现起来要简单的多。后端程序结构如图 5.1 所示。

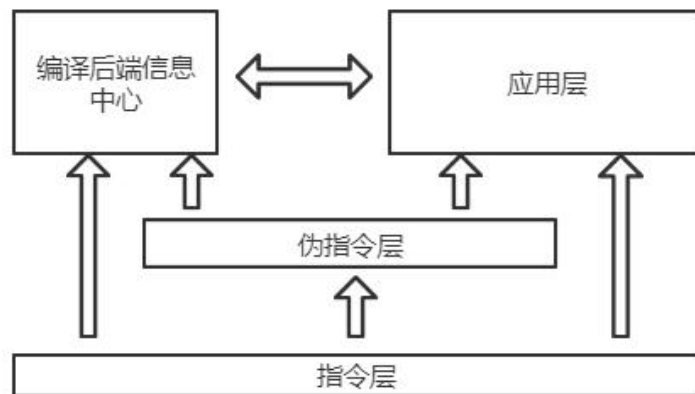


图 5.1 后端程序结构图

对于中间代码的命令部分，与产生式和制导函数绑定的注册模式相同，使用注册的方式进行命令和处理函数的绑定。对于中间代码的四元式部分，使用操作符 `op` 部分进行区分，分别处理即可。

### 5.2 信息管理

编译后端需要对变量信息进行管理。对应的信息中心为 `back_var_ctrl` 类（单例）。提供的功能有：

- 变量的增删查改。
- 返回值管理。
- 栈空间管理。
- 寄存器池管理。
- 资源申请。
- 变量资源的释放。
- 寄存器释放。
- 临时变量释放。
- 部分正确性校验。

`back_var_ctrl` 和编译后端其他场所均使用 `var_node_t` 来记录一个变量信息，主要记录了变量类型、变量存放场所、变量存放位置、左值信息等。

### 5.3 程序结构

编译后端结构上分为三层：指令层、伪指令层、应用层。

指令层代码为 `src/background/instruction.h`，该层对应汇编指令，用于生成目标代码，与目标平台强相关。

伪指令层位于 `src/background/macro_inst.h`，该层是对指令层的一层封装，与信息中心配合完成更复杂的功能。如：

- 变量各种类型值传送的封装。
- 复杂操作符的实现。

其中复杂操作中的逻辑运算符（`||`和`&&`）的设计与标准 C 语言的设计不相同。编译后端以 `sltu` 指令（无符号小于号）和加减法来实现布尔逻辑判断，不以某个运算结果影响后续计算是否执行。若要实现标准 c 的计算简化（`||`第一个操作符若为真，则不进行第二个操作符的计算，`&&`同理），需要使用流程控制指令实现。`sltu` 实现法有利于提高运算效率，但不利于运算的简化，使用流程控制语句效果相反。

应用层决定了编译后端提供的功能，工作逻辑为：

- 逐行识别中间代码。
- 区分命令和四元式。
- 若是命令则送入命令管理器执行相应的处理函数，若是四元式则送入四元式处理函数。
- 重复上述操作直到所有中间代码处理完毕。

编译后端处理逻辑如图 5.2 所示。

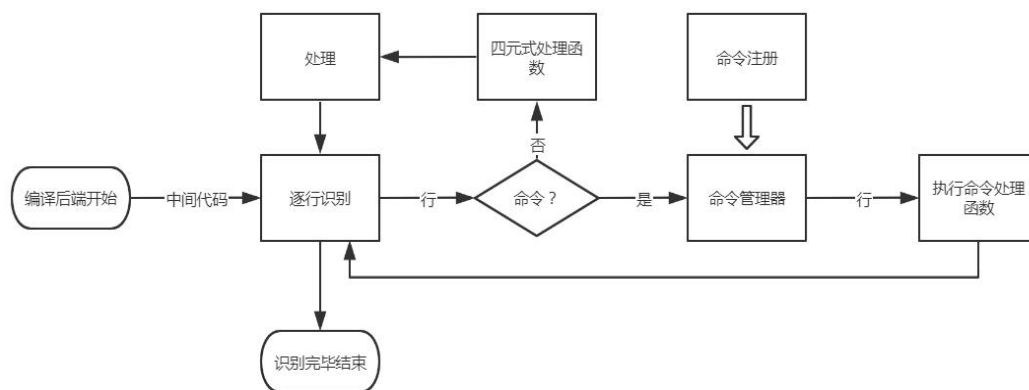


图 5.2 编译后端处理逻辑图

## 5.4 临时变量的资源分配

临时变量由编译前端产生，由中间代码描述，由编译后端管理。作用域不超过一条 c 语句。临时变量与全局变量、局部变量有本质的区别，它可以与普通变量有相同的处理逻辑，但不能被程序员手动操作或管理。

当遇到一个新的临时变量时，编译后端需要为其分配临时资源以存放数据。

临时资源可以是寄存器或者栈空间，后端应当尽可能地将临时变量存放至寄存器中，如果寄存器空间不足可以存放在栈空间中，甚至可以预留一些全局内存用于存放临时变量（本设计中临时变量不涉及全局内存）。

当进入一条 c 语句时，可能会产生若干临时变量，并为其申请资源；当一条 c 语句结束时，需要清理其使用过的所有的临时变量，释放临时变量的资源。存放在寄存器中的临时变量无需额外的指令申请和释放寄存器资源，只需要编译后端记录下来防止冲突即可<sup>[9]</sup>。存放在栈内的临时变量需要额外的指令维护栈顶指针寄存器，保证申请前和释放后栈顶指针均指向正确的位置。全局内存的临时变量应与寄存器变量的逻辑相同，本设计中不涉及全局临时变量。

### 5.4.1 中间代码 if-goto 与临时变量释放问题

中间代码的 if-goto 设计时遇到过问题，现介绍如下：

缺陷方案：

```
if cond goto sign_then
```

含义为：如果 cond（变量或者立即数）为真，就跳转到标号 then。此语句在 cond 变量的释放上会出现问题。若：

- cond 为临时变量。
- cond 存放在栈里（只有栈内临时变量才需要插入代码进行释放）。
- cond 最后出现的位置就是此语句。

则：在本语句结束后 cond 需要使用出栈代码进行释放，但是本语句可能会跳转到 sign\_then 代码处，这就会导致释放代码无法生效，会导致栈指针不平衡的严重后果。要注意的是，不能在 sign\_then 处插入 cond 的释放代码，因为本语句不一定是 sign\_then 的唯一前驱代码<sup>[10]</sup>。

故改正后的方案为：

- 将上述 if-goto 语句中插入释放逻辑，若 cond 不满足上述三个条件，格式为：

```
if cond
goto sign_then
```

- 若 cond 满足上述三个条件，则格式为：

```
if cond
::free_tmp_var cond
goto sign_then
```

如上，在跳转生效前执行栈空间释放代码，解决了该问题。

除了 if-goto 以外，return、continue、break 产生的 goto 语句均会导致该问题的发生，故要在相关跳转之前均需插入::free\_tmp\_var 命令来释放临时变量。

### 5.4.2 运行时临时变量资源的分配释放的合理性

编译后端编译时是将中间代码当作线性结构从上到下进行分析，而运行时中间代码具有复杂的执行顺序。线性的资源分配与释放在复杂执行顺序下依然是正确的<sup>[11]</sup>，理由如下：

不需要考虑全局变量的资源问题，因为全局变量在程序开始时分配，结束后释放，不会有任何问题。不需要考虑局部变量的资源问题，因为函数上下文的保存和语法特性保证不会产生问题。所以只需要考虑临时变量的资源问题。

首先明确：临时变量的资源分为寄存器和栈空间两种情况，临时变量的作用范围不会超过一条 c 语句。在一条 c 语句内部进行资源分配，c 语句结束后即刻进行资源释放。所以除了跳转语句外，普通 c 语句间结构的复杂性不会影响资源分配与释放的完整性。

所谓跳转语句有四种：

- 分支 if 语句。
- 循环 while 语句。
- 流程控制语句。

其中分支语句、循环语句和跳转语句在中间代码里均由 if-goto、goto 来实现。故只需要考虑中间代码 if-goto、goto 的情况。因为所有跳转语句的跳转操作都是最后进行的，故资源的申请不会出现任何问题<sup>[12]</sup>，又因为第 4.5 节、5.4.1 节对栈内的情况作了充足的说明，也不会出现问题。故只剩下 if-goto、goto 对寄存器临时变量的释放的讨论。寄存器的申请和释放不需要插入代码，只需要编译后端“知道”即可，又因为::free\_tmp\_var 紧贴于跳转语句之后且跳转操作已经是 c 语句的最后一个操作，故跳转时该临时变量的寄存器已经无用了，跳转语句也不会使用额外的可用于承载临时变量的寄存器资源，故编译后端可正常地、及时地释放寄存器资源。即，函数内部寄存器临时变量资源的申请和释放是纯编译时的问题，不受运行时的任何影响<sup>[13]</sup>。

综上所述，当前临时变量的分配释放策略是正确无误的，不会因为运行时执行顺序的复杂性而产生错误。

## 5.5 函数上下文保存与中断

函数保存的上下文中的寄存器如下所示(寄存器定义见附录 2)：

```
r_ra,    //函数返回地址
r_ax,    //通用寄存器
r_bx,    //通用寄存器
r_cx,    //通用寄存器
r_dx,    //通用寄存器
r_di,    //通用寄存器
r_si,    //通用寄存器
```

```
r_ex, //通用寄存器
r_fx, //通用寄存器
r_fp, //栈帧指针寄存器, 指向栈底
r_tmp_1, //临时寄存器, 编译后端用于存放运算中间结果
r_tmp_2; //临时寄存器, 编译后端用于存放运算中间结果
```

CPU 运算器内部 `hi` 和 `lo` 寄存器在常规函数调用时无需保存, 因为编译器可以保证乘除法等使用了 `hi` 和 `lo` 寄存器的语句, 在 `hi` 和 `lo` 被取出之前不会插入污染这两个寄存器的指令。但是中断条件下可能会在 `hi` 和 `lo` 被取出之前就打断程序, 这是不可控的, 故需要在中断处理程序中额外保存 `hi` 和 `lo`。多线程条件下, 与中断大致相同。

## 5.6 栈空间申请与栈顶指针

栈顶指针 `r_sp` 必须始终指向栈顶, 任何瞬间程序对栈空间的使用都不能超越栈顶指针。例如代码段 (栈为空递减):

代码段 1:

```
[r_sp]=0x00
r_sp = r_sp - 4
和
```

代码段 2:

```
r_sp = r_sp - 4
[r_sp]=0x00
```

两个代码段并非等价。原因是两条语句中间可能会发生中断, 若此种情况使用了代码段 1, 那么中断处理程序可能会污染 `r_sp` 指向的内存而导致错误, 这实际上违反了应用程序二进制接口规范。

## 第六章 测试与仿真

为了能够更好地进行测试与仿真，本设计开发了配套的汇编器与模拟器。汇编器能将编译后端生成的汇编代码汇编成机器指令。模拟器读入机器指令仿真执行，提供了一定的调试功能，降低调试难度。

### 6.1 汇编器

#### 6.1.1 汇编格式

该汇编代码格式由两个部分组成：

- 指令助记符。
- 标号。

与传统汇编相比，该格式较为简单，基本没有伪指令。其中标号格式为 `sign:+` 标号名，例如：

`sign: main`

使用时，使用 `sign_+` 标号名来代替该地址，如：

`j sign_main`

执行助记符基本格式为：

指令 目的地址 操作数 2 操作数 3

如：`lw 24 30 0` 表示  $r24 = [r30 + 0]$ ，其中目的地址为 24 号寄存器，基地址为 30 号寄存器，偏移地址为立即数 0。

#### 6.1.2 汇编器输出格式

汇编器输出代码为 MIPS CPU 能识别的 32 位二进制代码，该代码格式为容易兼容 Verilog 代码的格式，输出文件为 `run.c`。

文件首部为一系列注释，为读者提供标号与地址的对应关系。接下来为一系列二进制代码，如输入文件第 9 行的汇编代码 `sw 29 0 31`，输出二进制代码如表 6.1 所示。

表 6.1 汇编格式表

格式	Verilog 数字头部	十六进制代码	结束符	注释开始	命令说明	行说明
举例	32'h	AFBF0000	;	//	sw [r29+0]=r31	[9]

其中：Verilog 数字头部部分固定不变；十六进制代码为二进制指令；结束符固定不变；注释开始表注释开始，固定不变；命令说明，便于阅读代码和调试；行说明中，数字为本行命令对应输入文件的行数，由于输入文件可能存在注释行，加入此信息大大降低了调试时代码回溯难度。

### 6.1.3 汇编器程序结构

汇编程序负责将汇编源程序翻译成可执行的二进制目标代码，汇编流程如图 6.1 所示。

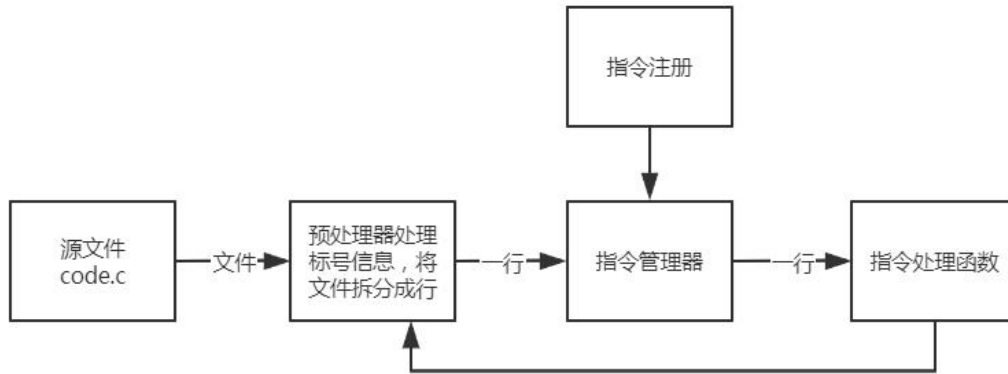


图 6.1 汇编流程图

asslan/order 文件夹下每个源文件对应一条指令的处理函数。每个处理函数在 asslan/order.h/order\_regist() 函数中进行注册（将命令与处理函数绑定）。asslan/head.h 中为指令处理函数框架。输出代码文件为 run.c，由指令处理函数直接输出重定向获取。

## 6.2 模拟器

### 6.2.1 初始化

模拟器为对 CPU、RAM、ROM 的功能模拟，但是纯硬件无操作系统难以操作和调试程序，故在硬件功能的基础上额外增加了部分功能，如环境初始化功能、调试功能<sup>[14]</sup>。

模拟器为寄存器和内存设置了初始状态：

- 寄存器初始为全 0。
- 内存初始为全 0。

为了使程序能在无操作系统的裸机上运行、方便调试等，需要对程序运行环境进行初始化：

- 置 r\_sp 为 1023\*4。
- 置 r\_fp 为 1024\*4。
- 置 r\_ra 为 TREM\_ADDR。

上述 1 和 2 表现为栈空间与全局空间最大为 4KB（再去除 0 号内存不能使用）。上述 3 表现为，程序结束后 pc 应为 TREM\_ADDR，用于检测程序结束和程序是否正常退出。



## 6.2.2 模拟器的使用

模拟器一开始显示的数字为读入的二进制程序。

当看到”====>”提示时可以输入调试命令，调试命令如表 6.2 所示。

表 6.2 调试命令表

命令格式	功能	实例
c	运行，直到程序结束或者异常退出。	c
直接回车	执行一条命令	回车
r	执行一条命令	r
rr	执行 100 条命令	rr
rrr	执行 1000 条命令	rrr
r %d	执行%d 条命令（%d 为十进制数字。下同）	r 30
pm %d %s	设置内存监控。%d 为内存地址，%s 为给该内存起一个名字	pm 4 ret
ps %d %s	设置栈内存监控。%d 为距离栈底 r_fp 的偏移量。%s 为给该内存起一个名字。	ps 4 i
pr %d	设置寄存器监控，%d 为寄存器	pr 30
sh	显示监控的内容	sh
exit	退出模拟器	exit
sr	查看所有寄存器	sr
bk %d	设置断点	bk 32
dbk %d	删除断点	dbk 32
bc %ud	将一个 32 位无符号数字转为有符号数字	bc 4294967284

二进制程序运行完毕后，若打印：

success:pc == TREM\_ADDR

exit with return code:某个数字

则程序正常退出，该数字为源程序 main 函数的返回值。其他情况均为执行错误，尤其若 pc != TREM\_ADDR，则说明 main 函数未能正确返回，比如栈顶指针不平衡会造成 main 函数返回地址错误。

## 6.3 测试

综合测试代码见附录三，该综合测试代码实现了经典的广度优先搜索走迷宫算法。该算法使用了队列的数据结构，代码实现过程中涉及到的功能有：局部变量，全局变量，多层函数调用，多维数组（原计划只支持一维），分支，循环，分支循环的任意嵌套，结构体，指针，注释，无参函数（原计划不支持），带一个参数的函数，函数返回值，任意组合的计算式，复杂运算符（点运算符，逻辑与，逻辑非），部分其他运算符，程序块，默认类型转换等。

运行编译项目，在读入源程序之前编译器读入 C 语言文法（见附录一），计算生成移进-归约表，部分文法处理结果日志如图 6.2 所示：

```

129 , '-' = R72
129 , '.' = R72
<if头部>->'if'('(<值>'))'.
130 , ';' = R50
130 , '{' = R50
130 , [数据类型] = R50
130 , 'struct' = R50
130 , [关键字] = R50
130 , '*' = R50
130 , '(' = R50
130 , 'continue' = R50
130 , 'break' = R50
130 , 'if' = R50
130 , 'while' = R50
130 , '&' = R50
130 , [常量] = R50
130 , 'return' = R50
<循环头部>->'while'('(<值>'))'.
131 , '{' = R51
<实参列表>-><值>','<实参列表>.
132 , ')' = R53
acc: 2
有1209个cell
非终结符有:38

```

图 6.2 部分文法处理结果

编译程序接下来读入源程序，进行词法分析，部分词法分析结果日志如图 6.3 所示。

```

=====
;
type=4分隔符
line=510 idx=14
name=';'
int_val=0
str_val=
=====
int
type=3保留字(关键字)
line=511 idx=0
name=[数据类型]
int_val=0
str_val=int
=====
h
type=1标识符(变量, 函数名, 等)
line=511 idx=4
name=[关键字]
int_val=0
str_val=h
=====
;
type=4分隔符
line=511 idx=6
-----

```

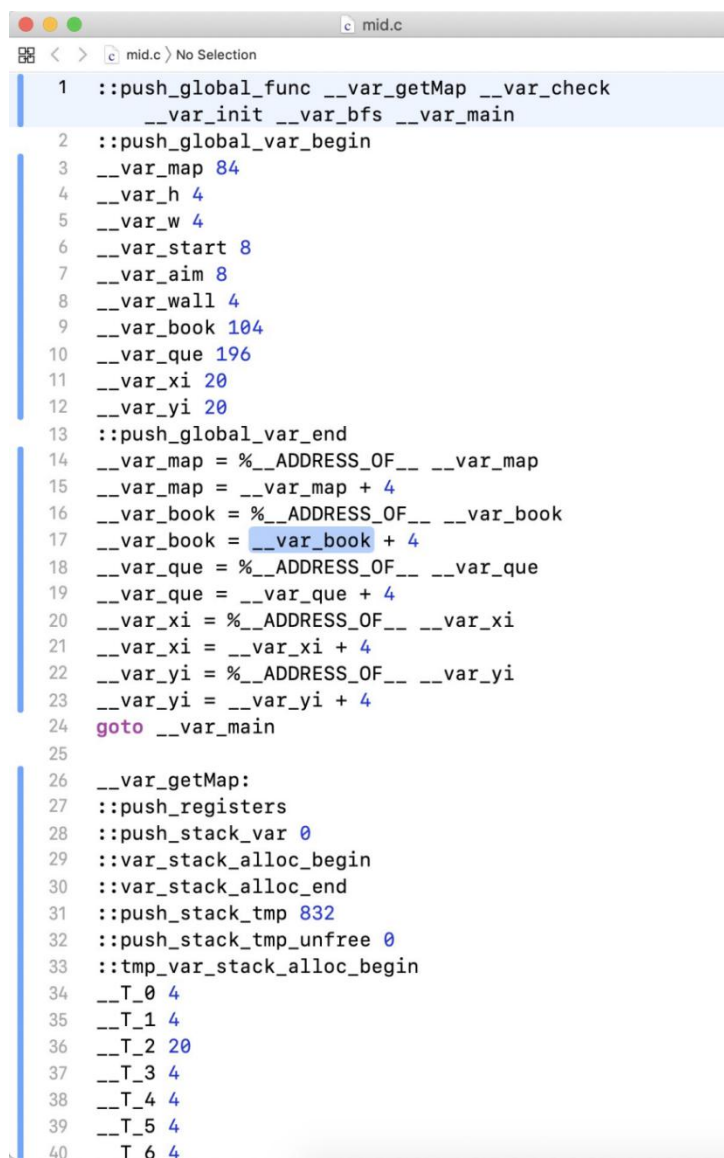
图 6.3 部分词法分析处理结果

接下来编译器进行语法分析，语法分析过程中调用语法制导函数，进行语义分析，对应部分日志如图 6.4 所示。

```
prod.sign=<struct数据类型>->'struct'[关键字]
[关键字]出栈
'struct'出栈
call func:23
<struct数据类型>入word_stack:
<struct数据类型>
=====
现在: 15 <struct数据类型> cond=3
移进
状态入栈: cond=9
=====
现在: 10002 '{' cond=10
归约, 产生式:
prod.sign=<struct声明>-><struct数据类型>
<struct数据类型>出栈
call func:24
数据类型类型
<struct声明>入word_stack:
<struct声明>
=====
现在: 16 <struct声明> cond=3
移进
状态入栈: cond=10
```

#### 6.4 语法分析和语法制导日志

语义分析过程中生成了中间代码，输出至 mid.c 文件，部分代码如图 6.5 所示。



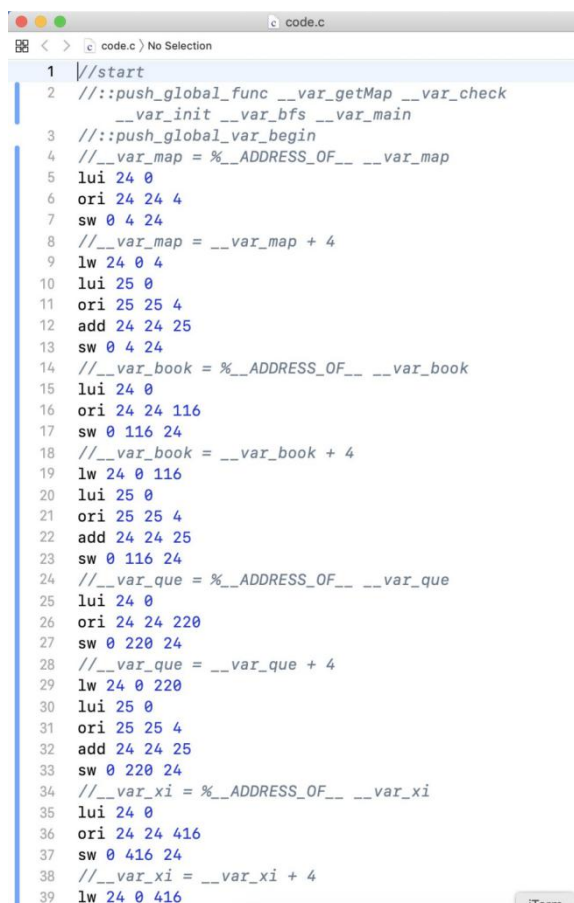
```

1  ::push_global_func __var_getMap __var_check
   __var_init __var_bfs __var_main
2  ::push_global_var_begin
3  __var_map 84
4  __var_h 4
5  __var_w 4
6  __var_start 8
7  __var_aim 8
8  __var_wall 4
9  __var_book 104
10 __var_que 196
11 __var_xi 20
12 __var_yi 20
13 ::push_global_var_end
14 __var_map = %__ADDRESS_OF__ __var_map
15 __var_map = __var_map + 4
16 __var_book = %__ADDRESS_OF__ __var_book
17 __var_book = __var_book + 4
18 __var_que = %__ADDRESS_OF__ __var_que
19 __var_que = __var_que + 4
20 __var_xi = %__ADDRESS_OF__ __var_xi
21 __var_xi = __var_xi + 4
22 __var_yi = %__ADDRESS_OF__ __var_yi
23 __var_yi = __var_yi + 4
24 goto __var_main
25
26 __var_getMap:
27 ::push_registers
28 ::push_stack_var 0
29 ::var_stack_alloc_begin
30 ::var_stack_alloc_end
31 ::push_stack_tmp 832
32 ::push_stack_tmp_unfree 0
33 ::tmp_var_stack_alloc_begin
34 __T_0 4
35 __T_1 4
36 __T_2 20
37 __T_3 4
38 __T_4 4
39 __T_5 4
40 T 6 4

```

图 6.5 部分中间代码

编译后端读入中间代码，编译后生成汇编代码输出至 `code.c` 文件中，部分汇编代码如图 6.6 所示。



```

1 //start
2 //::push_global_func __var_getMap __var_check
  __var_init __var_bfs __var_main
3 //::push_global_var_begin
4 //__var_map = %_ADDRESS_OF__ __var_map
5 lui 24 0
6 ori 24 24 4
7 sw 0 4 24
8 //__var_map = __var_map + 4
9 lw 24 0 4
10 lui 25 0
11 ori 25 25 4
12 add 24 24 25
13 sw 0 4 24
14 //__var_book = %_ADDRESS_OF__ __var_book
15 lui 24 0
16 ori 24 24 116
17 sw 0 116 24
18 //__var_book = __var_book + 4
19 lw 24 0 116
20 lui 25 0
21 ori 25 25 4
22 add 24 24 25
23 sw 0 116 24
24 //__var_que = %_ADDRESS_OF__ __var_que
25 lui 24 0
26 ori 24 24 220
27 sw 0 220 24
28 //__var_que = __var_que + 4
29 lw 24 0 220
30 lui 25 0
31 ori 25 25 4
32 add 24 24 25
33 sw 0 220 24
34 //__var_xi = %_ADDRESS_OF__ __var_xi
35 lui 24 0
36 ori 24 24 416
37 sw 0 416 24
38 //__var_xi = __var_xi + 4
39 lw 24 0 416

```

图 6.6 部分汇编代码

执行汇编器，读入汇编代码，输出机器指令，部分机器指令如图 6.7 所示。



```

25 //sign___sign_9_loop_break=:162/
26 //sign___sign_8_loop_break=:1634
27 //sign___block_8_exit=:1637
28 //sign___sign_7_func_return=:1637
29 //sign___var_main=:1664
30 //sign___block_13_exit=:1699
31 //sign___sign_12_func_return=:1699
32 'h3C180000; // lui r24=0<<16 [5]
33 'h37180004; // ori r24=r24 or 4 [6]
34 'hAC180004; // sw [r0+4]=r24 [7]
35 'h8C180004; // lw r24=[r0+4] [9]
36 'h3C190000; // lui r25=0<<16 [10]
37 'h37390004; // ori r25=r25 or 4 [11]
38 'h0319C020; // add r24=r24 + r25 [12]
39 'hAC180004; // sw [r0+4]=r24 [13]
40 'h3C180000; // lui r24=0<<16 [15]
41 'h37180074; // ori r24=r24 or 116 [16]
42 'hAC180074; // sw [r0+116]=r24 [17]
43 'h8C180074; // lw r24=[r0+116] [19]
44 'h3C190000; // lui r25=0<<16 [20]
45 'h37390004; // ori r25=r25 or 4 [21]
46 'h0319C020; // add r24=r24 + r25 [22]
47 'hAC180074; // sw [r0+116]=r24 [23]
48 'h3C180000; // lui r24=0<<16 [25]
49 'h371800DC; // ori r24=r24 or 220 [26]
50 'hAC1800DC; // sw [r0+220]=r24 [27]
51 'h8C1800DC; // lw r24=[r0+220] [29]
52 'h3C190000; // lui r25=0<<16 [30]
53 'h37390004; // ori r25=r25 or 4 [31]
54 'h0319C020; // add r24=r24 + r25 [32]
55 'hAC1800DC; // sw [r0+220]=r24 [33]
56 'h3C180000; // lui r24=0<<16 [35]
57 'h371801A0; // ori r24=r24 or 416 [36]

```

6.7 部分机器指令

执行模拟器，读入二进制程序。

使用“c”命令执行程序，模拟器打印指令执行情况。执行完毕后模拟器打印如下字段：

```
success:pc == TREM_ADDR
exit with return code:9
```

其中 success:pc == TREM\_ADDR 表示程序 pc 指针内容指向正确，间接说明栈顶指针指向正确，初步判断程序运行正确。其中 exit with return code:9 表示程序 main 函数返回值为 9，与源程序算法预期返回值一致，进一步判断程序运行正确，初步说明编译器、汇编器、模拟器工作正确无误，如图 6.8 所示。

```
1656 //32'h8FAA0000; lw r10=[r29+0][2177]
1657 //32'h23BD0004; addi r29=r29 + 4 [2178]
1658 //32'h8FA90000; lw r9=[r29+0] [2179]
1659 //32'h23BD0004; addi r29=r29 + 4 [2180]
1660 //32'h8FA80000; lw r8=[r29+0] [2181]
1661 //32'h23BD0004; addi r29=r29 + 4 [2182]
1662 //32'h8FBF0000; lw r31=[r29+0][2183]
1663 //32'h03E00008; jr r31 [2185]
1695 //32'h00407025; or r14=r2 | r0[2233]
1696 //32'hAFCE0000; sw [r30+0]=r14[2235]
1697 //32'h8FC20000; lw r2=[r30+0] [2238]
1698 //32'h080006A3; j 1699 [2240]
1699 //32'h03A0F025; or r30=r29 | r0[2248]
1700 //32'h23BD0004; addi r29=r29 + 4 [2249]
1701 //32'h23BD0004; addi r29=r29 + 4 [2251]
1702 //32'h8FB90000; lw r25=[r29+0][2252]
1703 //32'h23BD0004; addi r29=r29 + 4 [2253]
1704 //32'h8FB80000; lw r24=[r29+0][2254]
1705 //32'h23BD0004; addi r29=r29 + 4 [2255]
1706 //32'h8FBE0000; lw r30=[r29+0][2256]
1707 //32'h23BD0004; addi r29=r29 + 4 [2257]
1708 //32'h8FAF0000; lw r15=[r29+0][2258]
1709 //32'h23BD0004; addi r29=r29 + 4 [2259]
1710 //32'h8FAE0000; lw r14=[r29+0][2260]
1711 //32'h23BD0004; addi r29=r29 + 4 [2261]
1712 //32'h8FAD0000; lw r13=[r29+0][2262]
1713 //32'h23BD0004; addi r29=r29 + 4 [2263]
1714 //32'h8FAC0000; lw r12=[r29+0][2264]
1715 //32'h23BD0004; addi r29=r29 + 4 [2265]
1716 //32'h8FAB0000; lw r11=[r29+0][2266]
1717 //32'h23BD0004; addi r29=r29 + 4 [2267]
1718 //32'h8FAA0000; lw r10=[r29+0][2268]
1719 //32'h23BD0004; addi r29=r29 + 4 [2269]
1720 //32'h8FA90000; lw r9=[r29+0] [2270]
1721 //32'h23BD0004; addi r29=r29 + 4 [2271]
1722 //32'h8FA80000; lw r8=[r29+0] [2272]
1723 //32'h23BD0004; addi r29=r29 + 4 [2273]
1724 //32'h8FBF0000; lw r31=[r29+0][2274]
1725 //32'h03E00008; jr r31 [2276]
end
=====
success:pc == TREM_ADDR
exit with return code:9
=====
pc=1048575
=====>
```

图 6.8 程序执行结果

接下来查看结构体数组内保存的路径值，经计算数组的起始地址为 232，每个结构体有 x 成员和 y 成员记录了坐标，每个成员占 4 字节。使用 pm 指令查看对应内存内容，参数 1 为内存地址，参数 2 为内存名（给这块内存起个名字），输入如图 6.9 所示的调试指令。

```
c
pm 232 que0.x
pm 236 que0.y
pm 240 que1.x
pm 244 que1.y
pm 248 que2.x
pm 252 que2.y
pm 256 que3.x
pm 260 que3.y
pm 264 que4.x
pm 268 que4.y
pm 272 que5.x
pm 276 que5.y
pm 280 que6.x
pm 284 que6.y
pm 288 que7.x
pm 292 que7.y
pm 296 que8.x
pm 300 que8.y
pm 304 que9.x
pm 308 que9.y
sh
```

图 6.9 调试命令

```
<232> que0.x 0
<236> que0.y 1
<240> que1.x 1
<244> que1.y 1
<248> que2.x 2
<252> que2.y 1
<256> que3.x 2
<260> que3.y 0
<264> que4.x 3
<268> que4.y 0
<272> que5.x 4
<276> que5.y 0
<280> que6.x 4
<284> que6.y 1
<288> que7.x 4
<292> que7.y 2
<296> que8.x 4
<300> que8.y 3
<304> que9.x 0
<308> que9.y 0
pc=1048575
```

图 6.10 路径坐标

输出结果如图 6.10 所示，每行都是一条内存结果展示，第一列<232>表示对应的内存地址，第二列 que0.x 为自己起的内存名，第三列为内存值，这样便查询出全局数组 que 内保存的所有路径坐标，经验证符合预期，说明程序执行正确，最终证明编译器、汇编器、模拟器均正确无误。

## 结束语

编译项目完成了从文法设计、算法计算到二进制程序的生成、模拟计算整套流程。纵向看比预期额外实现了汇编器和 MIPS 模拟器；横向看比计划额外完成了多维数组等部分功能。

在用于教学的角度上来看，本项目将编译程序分割成：文法设计、词法分析、语法分析、符号表设计、语义分析、中间代码设计、编译后端设计、汇编器、模拟器。每一部分均可单独拿出来用于教学，可以用作《编译原理》课程的理论与实践教学，可以用作软件工程等大型项目的实践设计，可以用作《计算机组成原理》的模型机指令设计的教学，可以用作《数据结构与程序设计》的实践教学，也可以用作《微机原理与接口技术》的汇编与计算机模型教学，具有较高的灵活性和较高的教学价值，是一个较为大型的综合性项目。与其他开源编译软件相比，本设计具有如下优势：

- (1) 是一个完整的 C 编译器，对基础的 C 语言都可以编译，目标代码可以放到 MIP 平台执行，工程质量有保证，是对编译器设计教学的有益补充。
- (2) 松耦合，中间代码独立，可以在不修改编译前端的情况下支持其他目标平台和指令集，仅修改编译后端，工作量较小。
- (3) 有详细的说明文档，降低了代码阅读难度，方便在此基础上进行扩展设计与开发。

虽然此项目工程功能强大，但依然存在部分不足。从项目设计的角度看，部分模块依然存在设计不合理的情况，存在一定程度的重构改进的空间。作为大型 C++ 项目，它存在一定的内存泄漏的问题，需要进一步改进方案。

本设计源语言为精简过的 C 语言，后续可在教学过程中不断完善：加入 C 语言的剩余功能，提高语言的完整性；融入新的语法功能，不断提高语言的可用性和灵活性，例如加入协程支持以获取单核 CPU 下的并发能力；加入多文件和静态库支持，可用本设计开发简单的操作系统、自行编写设备驱动、可兼容自研的硬件平台等。本设计具有较高的继续开发的價值，为完整编写编译器的教学积累了经验，为编译原理的教学提供了有力支持。



## 致 谢

本论文和毕业设计是在指导老师董梁老师帮助下完成的，董老师在这个过程中给予了许多宝贵指导和意见，他严谨的治学态度、精益求精的工作作风、平易近人的人格魅力对我影响深远。从选题开始到论文编写完成，每一步董老师都倾注了大量心血，再此谨向董梁老师致以崇高的敬意和衷心的感谢。同时还要向在大学四年的生活中，给予我关心和帮助各位老师和同学，表示我深深的感谢！

## 参考文献

- [1]冯钢,郑扣根.基于 GCC 的交叉编译器研究与开发[J].计算机工程与设计,2004(11):1880-1883.王保胜.何金枝. 关于 C 编译器对——运算编译的研究 电脑知识与技术 2010 年 18 期
- [2] Youcong Ni,Xin Du,Peng Ye,Ruliang Xiao,Yuan Yuan,Wangbiao Li. Frequent pattern mining assisted energy consumption evolutionary optimization approach based on surrogate model at GCC compile time[J]. Swarm and Evolutionary Computation,2019,50.
- [3] 黄贤英,曹琼.新工科背景下《编译原理》的重定位思考与实施[J].福建电,2017,33(09):78+8
- [4] 陈俊洁,胡文翔,郝丹,熊英飞,张洪宇,张路. 一种静态的编译器重复缺陷报告识别方法[J]. 中国科学:信息科学,2019,49(10):1283-1298. 0.
- [5]张惠艳.编译原理中 LR 分析的教学探讨[J].福建电脑,2010,26(01):205-206.
- [6]刘斌.基于 C 语言的计算机编程技术探讨[J].数字通信世界,2018(04):88.
- [7]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,2016.
- [8]王照.C 语言中指针用法解析[J].中国新通信,2014,16(22):5-6.
- [9]Boma A. ADHI,Tomoya KASHIMATA,Ken TAKAHASHI,Keiji KIMURA,Hironori KASAHARA. Compiler Software Coherent Control for Embedded High Performance Multicore[J]. IEICE Transactions on Electronics,2020,E103.C(3).
- [10]Srećko Stamenković,Nenad Jovanović,Pinaki Chakraborty. Evaluation of simulation systems suitable for teaching compiler construction courses[J]. Computer Applications in Engineering Education,2020,28(3).
- [11]范志东 张琼生.《自己动手构造编译系统:编译、汇编与链接》
- [12]彭获然,熊庭刚,胡艳明,黄亮.基于国产 GPU 的 GLSL 编译器设计[J].计算机与数字工程,2019,47(06):1502-1506.
- [13]王保胜,何金枝.关于 C 编译器对——运算编译的研究[J].电脑知识与技术,2010,6(18):5093-5095.
- [14]Alfred V.Aho/Monica S.Lam/Ravi Sethi/Jeffrey D.Ullman 《Compilers Principles Techniques and Tools 2nd》

## 附录一

源语言文法完整定义 lan.txt 为:

```

<程序>-><外部声明>|<外部声明><程序>;
<外部声明>-><struct 定义>|<函数定义>|<变量声明或定义>;
<函数定义>-><函数定义头><程序块>;
<函数定义头>-><函数声明>;
<函数声明>-><变量声明或定义>;
<程序块首部>->'{'
<程序块尾部>->'}';
<程序块>-><程序块首部>
    <语句列表>
<程序块尾部>    |    <程序块首部>
<程序块尾部>;
<参数列表>-><广义数据类型><类型计算式>;

<广义数据类型>->[数据类型]|<struct 数据类型>;
<struct 定义>-><struct 声明><程序块首部>
    <变量声明或定义列表>
<程序块尾部>|<struct 声明><程序块首部><程序块尾部>;
<struct 声明>-><struct 数据类型>;
<struct 数据类型>->'struct'[关键字];
<变量声明或定义列表>-><变量声明或定义>;<变量声明或定义列表>|<变
量声明或定义>;
<变量声明或定义>-><广义数据类型><类型计算式>;

<类型计算式>->'*'<类型计算式>|<中括号运算>;
<中括号运算>-><中括号运算>['<值>']|<函数括号运算>;
<函数括号运算>-><函数括号运算>'(<参数列表>)'|<函数括号运算>'()'<括
号运算>;
<括号运算>->'(<类型计算式>)'[关键字];

<语句列表>-><语句>|<语句列表><语句>;
<语句>-><变量声明或定义>;|<struct 定义>;|<程序块>|<if 语句>|<循环语
句>|<值>;|;"continue";|<break";|<返回语句>;

```

---

```

<if 语句>-><if 头部><程序块>'else'<语句>
    |<if 头部><语句>;
<if 头部>->'if'('<值>');
<循环语句>-><循环头部><程序块>;
<循环头部>->'while'('<值>');
<实参列表>-><值>|<值>','<实参列表>;

<值>-><值>'='<逻辑或表达式>|<逻辑或表达式>;
<逻辑或表达式>-><逻辑或表达式>'||'<逻辑且表达式>|<逻辑且表达式>;
<逻辑且表达式>-><逻辑且表达式>'&&'<比较表达式>|<比较表达式>;
<比较表达式>-><比较表达式>'<'<加法表达式>|<比较表达式>'=='<加法表达式>|<加法表达式>;
<加法表达式>-><加法表达式>'+<强制类型转换表达式>|<加法表达式>'-'<强制类型转换表达式>|<强制类型转换表达式>;
<强制类型转换表达式>-><解引用表达式>;
<解引用表达式>->'*<解引用表达式>|&'<解引用表达式>|<成员选择表达式>;
<成员选择表达式>-><成员选择表达式>'.'[关键字]|<成员选择表达式>'['<值>']|<函数括号表达式>;
<函数括号表达式>-><函数括号表达式>'(')|<函数括号表达式>'('<实参列表>')|<括号表达式>;
<括号表达式>->'('<值>')|<元素>;
<元素>->[关键字][常量];
<返回语句>->'return'<值>;

```

## 附录二

寄存器定义为:

```
enum reg_t{
    r_undefined=-1,
    r_zero=0,    //零寄存器
    //r_at=1,      //为汇编器预留
    r_ret_1=2,   //返回值
    r_ret_2=3,
    r_arg_1=4,   //函数参数
    r_arg_2=5,
    r_arg_3=6,
    r_arg_4=7,

    r_ax=8,      //通用寄存器
    r_bx=9,
    r_cx=10,
    r_dx=11,
    r_di=12,
    r_si=13,
    r_ex=14,
    r_fx=15,

    //16-23 考虑到可能有特殊含义，暂未使用
    r_tmp_1=24,   //通用寄存器，编译器后端使用
    r_tmp_2=25,
    //26-27 预留给中断
    //r_gp=28,    //全局指针，先预留下来
    r_sp=29,      //栈顶指针寄存器
    r_fp=30,      //栈帧指针寄存器
    r_ra=31,      //函数返回地址 ret addr
};
```

## 附录三

测试样例代码：广度优先搜索走迷宫

```
int map[4][5];
int h;
int w;
/*
01000
00010
10100
00100
*/
struct coor{
    int x;
    int y;
};
struct coor start;
struct coor aim;//aim cannot equals to start
int wall;
int getMap(){
    h=4;
    w=5;
    wall=1;
    map[0][0]=0;
    map[0][1]=1;
    map[0][2]=0;
    map[0][3]=0;
    map[0][4]=0;

    map[1][0]=0;
    map[1][1]=0;
    map[1][2]=0;
    map[1][3]=1;
    map[1][4]=0;
```

```
    map[2][0]=1;
    map[2][1]=0;
    map[2][2]=1;
    map[2][3]=0;
    map[2][4]=0;

    map[3][0]=0;
    map[3][1]=0;
    map[3][2]=1;
    map[3][3]=0;
    map[3][4]=0;
    start.x=0;
    start.y=0;
    aim.x=4;
    aim.y=3;
}
int book[5][5];
int check(int ptr){
    struct coor *p;
    p=(struct coor*)ptr;
    int x=(*p).x;
    int y=(*p).y;
    if(x<0 || y<0 || w-1<x || h-1<y){
        return 0;
    }
    if(book[y][x]){
        return 0;
    }
    if(map[y][x]==wall){
        return 0;
    }
    return 1;
}
struct coor que[24];
```

```
int xi[4];
int yi[4];
int init(){
    xi[0]=0;
    xi[1]=0;
    xi[2]=1;
    xi[3]=-1;
    yi[0]=1;
    yi[1]=-1;
    yi[2]=0;
    yi[3]=0;
}
int bfs(){
    int i;
    int head;
    int end;
    head=0;
    end=1;
    que[0]=start;
    book[start.y][start.x]=1;
    while(head<end){
        struct coor cur;
        i=0;
        while(i<4){
            cur=que[head];
            cur.x=cur.x+xi[i];
            cur.y=cur.y+yi[i];
            i=i+1;
            if(check((int)&cur)){
                que[end]=cur;
                book[cur.y][cur.x]=1;
                if(cur.x==aim.x && cur.y==aim.y){
                    return 1;
                }
            }
            end=end+1;
        }
    }
}
```



```
        }
    }
    head=head+1;
}
return 0;
}
int main(){
    getMap();
    init();
    return bfs();
}
```