



# 本科毕业设计（论文）

**基于ANTLR的C语言编译器**

学 院 计算机学院

专 业 网络工程

年级班别 2013级（4）班

学 号 3113006584

学生姓名 吴同球

指导教师 刘添添

2017年 5 月

#### 摘 要

三号黑体加粗

传统的C语言编译器的开发流程中一个重要的步骤是要求开发者自己手写词法和语法分析器，而这个过程对开发者的底层语言了解程度要求很高而且工作量很大。因此本系统开发的目的主要是解决手写词法和语法分析器的过程，不仅使得开发一个编译器的工作量大大减少而且对开发者的底层语言了解程度要求也降低。

本系统的实现所采用的技术是基于ANTLR v4工具的基础上，通过定义C语言的词法和语法规则，利用ANTLR v4自动生成词法和语法分析器，而最后的语义分析则利用生成的词法和语法分析器生成类低级语言代码以及解释执行低级语言代码。

本系统最终实现一个与用户交互良好的编译器。不仅有简洁明了的交互界面，也实现一个编译器的基本功能。实现了词法分析器，将源代码识别成一个个的“单词”；语法分析器，将经过词法分析器得到的标记流进行语法解析，生成语法树；语义分析器，对语法树的结构进行解析，通过指令集实现源代码的运行。

**关键词**： ANLTR，C语言，编译器

**Abstract**

An important step in the development process of the traditional C language compiler is that the developer himself requires handwritten lexical and parser, and this process requires a high degree of knowledge and a great workload for the developer's underlying language. Therefore, the purpose of this system development is to solve the handwritten lexical and parser process, not only makes the development of a compiler to greatly reduce the workload and the developer's underlying language understanding requirements are also reduced.

The technology used in the implementation of this system is based on the ANTLR v4 tool, through the definition of C language lexical and grammar rules, the use of ANTLR v4 automatically generate lexical and parser, and the final semantic analysis is the use of generated lexical and grammar The parser generates class low-level language code as well as interprets execution of low-level language code.

The system eventually implements a compiler that works well with the user. Not only a concise and clear interface, but also to achieve a basic function of the compiler. The lexical analyzer is used to identify the source code as a "word". The parser parses the markup stream obtained by the lexical analyzer to generate a syntax tree; a semantic parser that parses the structure of the syntax tree, through the instruction set to achieve the operation of the source code.

**Key words**：ANLTR, C language, Compiler

目录

[1 绪论 1](#_Toc482698466)

[1.1 研究背景 1](#_Toc482698467)

[1.2 研究现状 1](#_Toc482698468)

[1.3 研究目标 2](#_Toc482698469)

[2 ANTLR开发技术 3](#_Toc482698470)

[3 需求分析和概要设计 5](#_Toc482698471)

[3.1 界面设计 5](#_Toc482698472)

[3.2 需求分析 5](#_Toc482698473)

[3.2.1 功能需求 5](#_Toc482698474)

[3.2.2 性能需求 6](#_Toc482698475)

[3.2.3 开发环境需求 6](#_Toc482698476)

[3.3 系统可行性分析 7](#_Toc482698477)

[3.3.1 技术可行性 7](#_Toc482698478)

[3.3.2 经济可行性 7](#_Toc482698479)

[3.3.3 操作可行性 7](#_Toc482698480)

[3.4 功能模块分析 8](#_Toc482698481)

[3.4.1 词法分析器(Lexer) 8](#_Toc482698482)

[3.4.2 语法分析器(Parser) 8](#_Toc482698483)

[3.4.3 语义分析器 9](#_Toc482698484)

[3.4.4 解释器 9](#_Toc482698485)

[3.5 结构设计 9](#_Toc482698486)

[4 详细设计 11](#_Toc482698487)

[4.1 词法分析器 11](#_Toc482698488)

[4.1.1 变量类型，函数类型和标识符 11](#_Toc482698489)

[4.1.2 数字，字符和字符串 11](#_Toc482698490)

[4.1.3 运算符 11](#_Toc482698491)

[4.1.4 其他 11](#_Toc482698492)

[4.2 语法分析器 11](#_Toc482698493)

[4.2.1 变量定义设计 12](#_Toc482698494)

[4.2.2 函数定义设计 12](#_Toc482698495)

[4.2.3 语句设计 12](#_Toc482698496)

[4.2.4 表达式设计 12](#_Toc482698497)

[4.3 解释器 13](#_Toc482698498)

[4.3.1 内存 13](#_Toc482698499)

[4.3.2 寄存器 13](#_Toc482698500)

[4.3.3 指令集 14](#_Toc482698501)

[5 系统实现 15](#_Toc482698502)

[5.1 词法分析器 15](#_Toc482698503)

[5.1.1 标识符 15](#_Toc482698504)

[5.1.2 数字，字符和字符串 15](#_Toc482698505)

[5.1.3 注释 15](#_Toc482698506)

[5.2 语法分析器 16](#_Toc482698507)

[5.2.1 变量定义语法 16](#_Toc482698508)

[5.2.2 函数定义语法 16](#_Toc482698509)

[5.2.3 语句 16](#_Toc482698510)

[5.2.4 表达式 18](#_Toc482698511)

[5.3 语义分析 20](#_Toc482698512)

[5.4 解释器 20](#_Toc482698513)

[6 运行与测试 22](#_Toc482698514)

[6.1 测试流程 22](#_Toc482698515)

[6.1.1 主界面和导入源代码模块 22](#_Toc482698516)

[6.1.2 词法分析模块 23](#_Toc482698517)

[6.1.3 语法分析模块 24](#_Toc482698518)

[6.1.4 语义分析模块 27](#_Toc482698519)

[6.1.5 解释器模块 27](#_Toc482698520)

[6.2 测试结果反馈 31](#_Toc482698521)

[6.2.1 测试总体情况 31](#_Toc482698522)

[6.2.2 系统需要优化的地方 31](#_Toc482698523)

[6.2.3 对系统的评价 31](#_Toc482698524)

[结 论 32](#_Toc482698525)

[参 考 文 献 33](#_Toc482698526)

[致 谢 34](#_Toc482698527)

# 1 绪论

## 研究背景

C语言作为计算机学科学生的一门基础而且举足轻重的学科，除了熟悉C语言的基本知识外，每位学生更应该去了解C程序的编译过程，从底层去加深对计算机编程的理解。然而现在主流的C语言编译器，其工程的复杂度非常高，编译过程对用户也是不可见的，不适合学生去直接学习；对于C语言编译器的开发，传统的开发流程工作量非常大，对开发者的底层语言和计算机编程的掌握程度要求也较高，这也加大了学生开发C语言编译器的难度。

ANTLR作为语言识别的一种工具，近几年来发展迅速，在语言识别方面受到了开发者越来越多的青睐。ANTLR工具能够识别和处理编程语言，即通过开发者自定义词法记号和语法规则，自动生成目标语言的词法和语法分析器，而这个过程在传统的编译器开发过程中工作量是非常大的，但是利用ANTLR工具却能够大大削弱这个过程使得开发难度降低。此外，ANTLR工具还能够生成抽象语法树，不仅使得最后的语义分析和中间代码的生成变得轻松，还增强了与用户的交互性，可提高用户的开发积极性。

在多年的编程语言排行榜中，Java语言和C语言基本占据前两名，而ANTLR工具不仅是基于Java语言开发的，而且也可以在Eclipse中生成词法和语法分析器的Java代码，因此具有跨平台性，能够被开发者广泛利用，也足以说明其未来的发展前景。

基于以上的背景分析，如果将ANTLR工具融入到C语言编译器的开发中将是一项非常具有现实意义的突破。因此本系统的设计一是为了帮助学生更深层次的理解C语言和学习编译过程，二是基于ANTLR开发的C语言编译器能够减少开发的工作量和降低开发难度，使学生开发和理解变得更加容易。

## 研究现状

有关ANTLR相关的知识，国内的研究还是相对较少，很多也只是翻译国外的著作。目前ANTLR工具在编译器方面的应用能够比较完整的对程序进行编译的包括Java，C++，C#等少数语言。而对于C语言，包括国外和国内的一些现有的资料显示，没有发现对C语言程序的一个比较完整的编译过程，即还没有扩展到对C语言语句比较完整编译的过程。目前基于ANTLR的C语言编译器的相关例子有：标准C语言词法和语法规则定义g4文件和一些简单的表达式的四则运算，而这些只是一个编译器的一小部分。本系统与现有的一些例子相比更具有先进性，除了实现了现有例子的一些功能外，还增加识别和编译更加复杂的C语言的语句。

## 研究目标

基于以上的背景和现状，本系统旨在实现一个不仅能够帮助学生更深层次的理解C语言的编译过程，而且还能够减少开发的工作量和降低开发难度，使学生开发和理解变得更加容易的编译器。此C语言编译器基于ANTLR工具基础上，能够实现编译运行较完整的C语言程序，功能上能够实现词法分析器、语法分析器、实现指令集和实现程序的运行环境。

# 2 ANTLR开发技术

本系统的开发是基于ANTLR[1]工具进行的，下面将详细介绍ANTLR技术：ANTLR是Another Tool for Language Recognition 的缩写，简单的说就是“又一个语言识别工具”，说明在此之前已经存在其他的语言识别工具（如LEX, YACC），而ANTLR因其简洁的语法和高效的运行效率在这些语言识别工具中受到开发者的喜爱。ANTLR被定义成是一种可以嵌入如Java，C#，C++等辅助代码段的文法，来构建出相对该文法的识别器，编译器或翻译器的一种语言工具框架。然而随着ANTLR的不断发展，到现在已经是第四代了，以上的定义在第四代中不提倡在文法中嵌入如Java，C#，C++等辅助代码段。因此ANTLR的功能就是根据给定文法自动生成编译器，也就是根据我们自己编写的相应语言的文法自动生成词法和语法分析器，然而事实上ANTLR知识生成相应语言编译器的源代码，我们最终还是要编译它。本系统所使用到ANTLR工具是第四代的，也就是ANTLR v4。ANTLR v4的工作流程如图2.1所示。



图2.1 ANTLR v4工作流程图

既然ANTLR v4工具是根据语言规则生成翻译器的，那么它的语言规则的编写就有一定的格式和要求了，下面将介绍关于在ANTLR v4中语言规则文件的要求。在v4版本中，语言规则的文件名以g4[2-3]为后缀，在该文件中定义的文法规则有如下几点：

（1）grammar名称要与g4文件名一致；

（2）词法规则以大写字母开头，语法规则以小写字母开头；

（3）若有规则出现冲突，则先出现的规则优先匹配；

（4）每天规则以规则名+冒号+规则+分号形式定义；

（5）在产生式后面 # label 可以给某条产生式命名，在生成的代码中即可根据标签名分辨不同产生式，但是若有一条产生式带有标签，那么该产生式对应的规则下面的所有产生式都要带有标签。

（6）| 用于分隔两个产生式， ( ) 括号用于指定子产生式， ?，+，\* 用法同正则表达式；

ANTLR v4的一个新特性就是能够处理左递归规则，在实现左规则递归时，ANTLR v4提供语法分析树访问者和监听器[2] [4]两种设计模式来实现语言应用，以保持语法的整洁。

（1）访问者模式（Visitor）

文法规则中定义的规则下面的选项后面的标签都会生成一个与该便签名相同的visitlabel方法，如果选项没有便签的话，只会为每条规则生成一个visit方法。正是由于生成了visit方法，每种输入短语才能得到不同的事件相应。所有的visit方法都会生成在BaseVisitor类中，该类中定义使用的是Java泛型，visit方法的返回值为参数化类型。一般情况下在实现我们编译器的时候都会定义自己的visitor类去重写BaseVisitor类的visit方法，由于visit方法的返回值为参数化类型，我们可以根据表达式计算结果返回值的类型去设定实现的泛型参数。

（2）监听器模式（Listener）

类似于访问者模式监听器模式也会为每一条规则和选项生成enter和exit方法，不同的是这两种方法都没有返回值，而且监听器模式不必要我们去做任何的树遍历。所有的enter和exit方法都会生成在BaseListener类中，在与语法规则匹配的短语开始和结束时，BaseListener类得到通知，ANTLR内建的树遍历器就会触发在监听器中enter和exit这样的一串回调方法，如同它对语法分析树执行了一次深度优先遍历。监听器机制有效地隔离了语法和语言应用，使语法可以被其它应用再次使用。

# 3 需求分析和概要设计

## 界面设计

作为一个与用户交互良好的编译器，有一个直观明了，简洁的界面是必不可少的。本编译器的界面设计主要采用Java Swing 中JTabbedPane选项卡技术，类似Eclipse编辑界面，每一个功能的结果将在不同的选项卡中显示。用户可以将已编辑好的源代码文件导入其中，在本编译器的界面中可以编辑导入的源代码，通过点击不同功能的按钮来展示结果。

## 需求分析

### 3.2.1 功能需求

本编译器需要实现的功能有：词法分析、语法分析、语义分析和解释器四部分功能，同时需将每部分分析的运行过程和结果显示出来。

其中在词法和语法分析功能中要求编译器能够识别出源代码中每一个有意义的单词，能够认出源代码中具有特殊含义的字符，将每个单词的信息反馈给用户。这里的识别单词包括能够识别void、int、char、float、double在内的基本数据类型名和用户自定义变量，变量又包括普通变量、一维数组和二维数组；能够理解源代码中具有特定结构的字符串，也就是能够理解具有语法的句子，将经过语法处理之后的语法树[1-2] [5]反馈给用户。具有语法的句子包括能够识别表达式、赋值语句、for语句、if语句、while语句、do while语句、return语句和函数调用语句等[6]。

作为一个学生学习编译过程的编译器，在语义分析中要实现的功能就是将表达式、赋值语句、for语句、if语句、while语句、do while语句和return语句这些跳转语句转化成中间代码，也就是用四元式代码[7]的方式反馈给用户，使用户能够简化代码，更加直观清楚看到各个语句之间的关系。

当编译器能够正确识别和理解源代码的内容时，最后也是最重要的一步就是解释器功能，即要完成源代码中所要表达的意思。为了展示一个编译器的运行过程，在完成这一功能时，在内部上需要定义一个指令集，这个指令集模拟了编译器对源代码的语义做出反应的每个过程。另外程序运行是需要环境的，包括内存段，代码段。因此内部需要实现一个虚拟机来实现这个功能，而这个是用户无法直接控制的模块。

### 3.2.2 性能需求

编译器编译速度一直是工程项目的一个难以避免的时间消耗，在大项目中，即便编译器有着优秀的编译性能，也无法避免消耗较长时间。编译器尽可能以优秀的编译速度方案来开发。由于项目功能的限制，对于源代码的读取无法跳过源代码的任何字符信息。所以期望的运行时间在常数级别，即复杂度[5]为O(n)。

项目各组件能否达到性能指标，将编译器运行分为两个流程来考虑，第一个是词法分析和语法分析阶段，由于预先制定了词法规则和语法规则，词法分析和语法分析可以交叉在一趟遍历源代码的流程中完成，假设源代码的总字符数为m，则完成词法语法分析的时间为O(m)。第二个流程是执行指令以及输出，由于虚拟机在工程上可以做成单例模式[8]，编译器程序运行环境再初始化的时候已经完成。一条源程序可能生成k条指令，假设程序有n条句子，则运行指令的时间复杂度为O(n\*k)。综上所述，编译器执行编译所需要的时间复杂度为O(m)+O(n\*k)。可以看出，编译时间随着编译文件的大小是正相关的。

在内存消耗上，编译器主要的的内存消耗在模拟程序运行环境上，故设计了一个参数MemorySize=256\*1024来作为单个内存区的内存大小，该项目设计了三个内存区一共 (2\*256\*1024\*4+256\*1024)/1024=2304KB=2.25MB的内存，加上少量指令集所消耗的内存不计。故编译器在占用系统资源的方面是非常小的，对其他程序的运行没有影响。

综上所述，编译器在运行速度和资源消耗两个方面都表现较优，应该是满足该项目的性能需求的。

### 3.2.3 开发环境需求

项目构建在Windows 10操作系统，Inter Core i5四核处理器下，由于项目在Eclipse中开发，需要电脑安装Eclipse luna的版本，这一步之前要求电脑已安装了Java的运行环境jdk1.7。本项目是在ANTLR工具的基础上开发，因此在Eclipse中需要安装ANTLR v4版本的插件[9]。

## 系统可行性分析

结合以上的背景和需求分析，下面将从技术可行性，经济可行性和操作可行性三个方面去阐述系统的可行性。

### 3.3.1 技术可行性

在传统的编译器开发流程中，开发的难度和工作量是非常大，特别是词法分析和语法分析，其中很多的细节问题还是比较麻烦。因此对开发人员的技术要求还好比较高，这样一来开发一款编译器的技术成本就会很高。但是利用ANTLR v4工具，工作量和对技术开发人员的技术要求就会大大降低。ANTLR v4工具能够帮助我们处理词法和语法分析的细节，这样的话开发人员只需要关注如何构建语言的文法即可。再加上计算机学科的学生有着编译原理和数据结构的基础，对于语义分析阶段的指令集和虚拟机的构建相对来说就会容易一些。同时，ANTLR v4工具是Java开发的，作为主流语言Java，有着面向对象的特性以及众多的包调用使得开发的难度又减小了。因此从技术可行性上考虑，基于ANTLR的C语言编译器是可以实现的。

### 3.3.2 经济可行性

作为一个学生学习使用的一款小型编译器，没有涉及到服务器，没有涉及到数据库，不需要高价昂贵的机器，只需要一台电脑和网络即可。在电脑和网络普及的信息时代，从经济可行性上说，开发编译器是绝对可行的。

### 3.3.3 操作可行性

本系统的操作相对来说比较简单，用户只需要导入源代码的文件或是编辑导入的源代码的文件，执行编译器得到结果即可。在这个过程中用户仅仅要关注的是导入的源代码的正确性，而内部的复杂流程用户是不需要关心的。因此从操作 上来说是可行的。

基于以上的几点可行性分析，实现基于ANTLR的C语言编译器的各方面的代价都是很低的，系统的可行性成立。

## 功能模块分析

本系统的功能模块相对来说比较少，主要分为词法分析器，语法分析器和语义分析器和解释器模块。系统的功能模块设计如图3.1所示。

图3.1 功能模块图

### 3.4.1 词法分析器（Lexer）

作为编译器第一步也是最基础的一步，词法分析器的作用就是对源代码进行预处理，可以过滤掉编译过程不关心的内容，比如滤掉空白、注释和特殊字符等，大大提高编译的效率。因为直接从源代码编译成汇编代码是很困难的，所以先将源代码的各个字符或字符串处理成标记流再进行语法分析就会变得很容易了。在词法分析中，各个关键字，限定符等都被处理成一个个的“单词”。

### 3.4.2 语法分析器（Parser）

语法分析器的作用就是将经过词法分析器之后得到的标记流根据定义的语法规则进行语法检查，并构建由这些“单词”组成的数据结构。最终经过语法检查之后得到的结果生成语法树。词法和语法分析器的流程如图3.2所示。

图3.2 词法和语法分析器的工作流程

### 3.4.3 语义分析器

在语义分析器中要完成的工作就是对每一条表达式或是各个跳转语句转化成中间代码，即转化成四元式结构。此阶段的主要目的就是简化和优化代码，通过对程序中的表达式和if、while、do-while、for等跳转语句转化成一条条的四元式代码，使用户能够一目了然地看到程序的执行过程。通过展示四元式代码，提高与用户的交互性，也能让用户更清楚的了解整个的编译过程。

### 3.4.4 解释器

解释器所要完成的工作就是针对经过语法分析器得到的语法树进行分析。而此过程最重要的一个工作就是构建虚拟机来模拟指令，该虚拟机[10]制订了程序的内存模块以及各种类汇编指令，用于模拟程序进程的结构，以及类汇编代码的执行。通过词法和语法分析的流程之后，虚拟机已经将源代码中的相关数据结构以及类汇编代码构建起来了，再通过虚拟机的各种指令运行代码，从而实现源代码的运行。

## 结构设计

本系统的设计按照功能模块来划分系统结构，一共分为3层，第一层是输入层，或是文件导入层，用于获取用户的源代码；第二层是翻译器层，该层分为两个方面，是由ANTLR v4生成的词法和语法分析器组成；第三层是语义分析层，也就是虚拟机层，该层主要输出执行代码和执行结果。结构设计如图3.3所示。



图3.3 编译器结构图

# 4 详细设计

## 词法分析器

在详细设计阶段，词法分析器主要是定义本编译器的词法规则，下面将从几方面介绍本系统所定义的几种类型的词法规则，其对应文法定义在g4文件中。

### 4.1.1 变量类型，函数类型和标识符

本编译器只定义C语言的几种基本数据类型包括int、char、float、double以及函数类型void。标识符的定义按照C语言的标识符的定义，为ID: [a-zA-Z\_][a-zA-Z0-9\_]\*，只能匹配下划线，数字和字符组成的标识符。

### 4.1.2 数字，字符和字符串

数字包括整形和实数的定义，整形的定义为NUM: [0-9]+，实数的定义为NUM\_float: [0-9]+.[0-9]+；字符表示为单引号之间的内容，字符串表示为双引号内的内容。需要注意的是”\n”,”\t”这种特殊字符，这种换行或空格的字符需要将其定义为Whitespace : [ \t]+ -> skip ;和Newline : ( '\r' '\n'? | '\n' ) -> skip ;。

### 4.1.3 运算符

运算符的种类比较多，包含了C语言的大部分运算符。每一种运算符对应一种标识。例如以下的几种运算符：Assign: '='; 乘号：Mul : '\*'; 除号：Div : '/';等等。

### 4.1.4 其他

定义了两种的注释方式。一种是单行注释为LineComment : '//' ~[\r\n]\* -> skip ;另一种为多行注释为BlockComment : '/\*' .\*? '\*/' -> skip ;。用于实现C语言中的两种注释规则，当程序匹配到这个规则的时候什么都不做，直接跳过。

## 语法分析器

语法分析器主要定义能够识别C语言语法句子的各种规则。

### 4.2.1 变量定义设计

本系统的变量定义主要分为两种，一种是枚举类型的变量定义，另一种是一般变量类型定义。枚举类型变量形如以下方式定义：enum：标识符 [= 数字]？，{，标识符 [= 数字]？\*}；。而一般变量定义则形如：类型 标识符 {，标识符}\*；。变量的识别中通过将分析出来的变量的相关信息存在符号表中，但需要调用的时候查询符号表即可获得该变量的信息。

### 4.2.2 函数定义设计

函数的定义相对来说比较复杂一点，分为三部分去解析一个函数语句。

第一部分是整体函数定义：类型 标识符 (参数列表) {函数体}。

第二部分就是参数列表的定义，类似于变量定义的设计：类型 标识符 {，类型 标识符}\*。

第三部分为函数体的定义：变量定义\* 语句定义\*。

### 4.2.3 语句设计

语句的设计代表着一个编译器的难度和深度，本系统主要完成了C语言中比较常见的几种语句，包括if语句、while语句、do while语句、for语句以及return语句，下面将一一介绍这几种语句。

赋值表达式：表达式 = 表达式。

if语句的定义：if (表达式) 语句 (else 语句)？。

while语句的定义：while(表达式) 语句。

do while语句：do 语句while(表达式)；。

for语句：for(表达式？；表达式？；表达式？) 语句。

return语句：return 表达式？；。

### 4.2.4 表达式设计

表达式设计涉及类型比较多，根据表达式的优先级顺序来定义表达式，把常量，变量，数字作为主表达式，可以划分为以下几种情况。

把基于住表达式的类型转换和函数调用，后缀的自增自减划分为后缀表达式。例如后缀的自增自减：表达式 操作=(++|--)；

把一元运算符带上后缀表达式的表达式划分为前缀表达式，主要有逻辑取反，按位取反，正负号，前缀的自增自减。例如前缀的自增自减：操作=(++|--) 表达式。

把前缀表达式与二元运算符构成的表达式划分为运算表达式，包括加减乘除，求余取模，大于小于，赋值操作等等。例如乘除求余：表达式 操作=(\* | / | %) 表达式；或是赋值表达式：表达式 操作== 表达式。

## 解释器

解释器主要的任务就是完成对程序的执行，而此过程中要实现对程序的模拟执行，因此要定义虚拟机来完成此过程。在虚拟机中要关注的三个部件是CPU，寄存器和内存。汇编指令以二进制的方式保存在内存中；CPU要从中取出一条条的指令进行执行；而程序运行的状态则需要保存在寄存器中。

### 4.3.1 内存

进程的内存将会分为以下几段：（1）代码段，用于存放指令；（2）数据段，用于存放初始化了的数据；（3）未初始化数据段，用于存放未初始化的数据；（4）堆，用于为程序分配动态内存空间。（5）栈，用于处理函数调用的相关数据或函数的局部变量。在本系统中，定义了并模拟了三个相关的内容，代码段（TextSegment），数据段（DataSegment）以及栈（StackSegment）。

### 4.3.2 寄存器

寄存器用于保存程序的运行状态，本系统模拟了4个寄存器。分别是（1）PC程序寄存器，用于存放下一条要执行指令的地址。（2） SP指针寄存器，永远指向栈顶。（3） BP基地址指针，用来记录栈位置，函数调用的时候会使用到。（4） AX 通用寄存器，存放指令的执行结果。四个寄存器相辅相成，对于程序的运行模拟起到了关键的作用。

### 4.3.3 指令集

在本系统中定义了19种常规指令：

LEA, IMM, JMP, CALL, JZ, JNZ, ENT, ADJ, LEV, LI, LC, SI, SC, LIFD, LCFD, SCTD, SITD, PUSH, EXIT

以及16种运算指令：

OR, XOR, AND, EQ, NE, LT, GT, LE, GE, SHL, SHR, ADD, SUB, MUL, DIV, MOD

虚拟机将给予35种指令具体的含义，并将之处理成4个寄存器对三大内存模块的读写。

# 5 系统实现

## 词法分析器

在这部分，词法分析器需要对源代码中特定的字符串生成对应的映射，用来提供给语法分析器构建语法树。在C语言中，关键词以及运算符毫无疑问需要被提取出来特殊对待，如Eq : ’==’; Assign : ‘=’;ADD : ‘+’ 等映射关系。在不借助其他工具的情况下，在实现这个词法分析器需要定义一个函数一边遍历源代码一遍找到下一个目标单词。当然，借助工具可以简化词法分析器的实现过程，使注意力能够更多的放在词法规则定义上面。本设计使用ANTLR4工具，只需在g4文件中制定需要的词法规则。则ANTLR4工具框架能编译出token文件来进行词法分析。

### 5.1.1 标识符

定义标识符的规则为ID : [a-zA-Z\_][a-zA-Z0-9\_]\*，当匹配成功程序将计算该ID的哈希值，然后在符号表中查询是否存在该ID，若存在则报重定义的错误，若不存在则加入到符号表中。

### 5.1.2 数字，字符和字符串

定义数字的规则为NUM:[0-9]+，当匹配成功的时候将向代码段写入IMM指令以及数值，当程序跑起来的时候会将值放入ax通用寄存器中。当匹配到单引号括住的字符时，程序直接向代码段写入IMM指令以及字符。当匹配到双引号括住的字符串时，程序将记录数据段的指针位置，并将字符串写入数据段，再向代码段写入IMM指令以及指针位置，用于运行时获取字符串的内容。

### 5.1.3 注释

单行注释LineComment : '//' ~[\r\n]\* -> skip以及注释块BlockComment : '/\*' .\*? '\*/' -> skip，用于实现C语言中的两种注释规则，当程序匹配到这个规则的时候什么都不做，直接跳过。

## 语法分析器

语法分析器需要在ANTLR 的g4文件中写下C语言各种语法的文法。包括变量定义语法、函数定义、语句和表达式语法。

### 5.2.1 变量定义语法

遍历语法树时，若获取到一般变量定义的语法规则：类型 标识符 {， 标识符}\*;，首先处理基本数据类型，判断是int型、char型、还是void型等，然后在符号表搜索标识符是否存在，若存在则报重定义的错误，若不存在则把标识符的相关信息加入符号表。

若获取到枚举变量定义的语法规则：enum 标识符? { 标识符 [= 数字]?, {, 标识符 [= 数字]? }\* };，则将枚举变量当成是常变量来处理，此时在符号表搜索标识符是否存在，若存在则报重定义的错误，若不存在则把标识符以及数值加入符号表。

### 5.2.2 函数定义语法

遍历语法树获取到函数定义语法规则：类型 标识符 (参数列表) {函数体}时，首先处理返回值类型，接着搜索标识符存入符号表，然后让语法树遍历参数列表的语法规则：类型 标识符 {, 类型 标识符}\* ，同样是将数据类型以及标识符信息存入符号表中，同时我们需要记录下每个形参的栈偏移量，用于函数调用的时候能够通过偏移量找到对应的形参。此时我们在符号表中的value字段中存放的是形参的偏移量。当参数规则匹配完成后就轮到匹配函数体规则：变量定义\* 语句\* 了，也就是意味着函数体内可以定义局部变量，还有任意的语句，在此时定义的变量皆为局部变量，函数规则匹配的最后用符号表中B的字段来存放和更新局部变量的对应信息。

### 5.2.3 语句

语句规则，其中划分为选择类的语句if，循环语句while、for和do-while，以及返回语句return。为了方便下面的介绍，先引入两个变量a和b，用来记录代码段的位置。变量a的作用是跳回，即当进行一次循环语句之后需要重新回到的位置。变量b的作用是记录循环语句部分和选择语句部分之后的的位置，用于跳出循环和选择。

（1）if语句

遍历语法树匹配if规则：if ( expression ) statement (else statement)?。这里要对表达式和两个语句段进行语法规则匹配并写入代码段，具体操作如下：

先遍历表达式语法expression规则

1.JZ b 如果条件为0则跳转到b，b设置为代码段下一位置

遍历语句语法规则

2.代码段b位置的内容赋值为代码段末的后面三位地址 若有else语句

3.JMP 若有else语句

4.b的值设置为代码段末尾位置 若有else语句

5.遍历else后的语句语法规则 若有else语句

6.代码段b位置的内容赋值为代码段末后面一位地址

（2）while语句

遍历语法树匹配While规则：while ( expression ) statement。按照语句顺序来处理表达式和语句，并向代码段写入指令：

1.a赋值为代码段末的下一位置 用于跳回循环

遍历表达式语法规则

2.JZ b 判断表达式为零时，跳转到b记录的地址

3.b赋值为代码段末的位置，并空出位置回写地址

遍历语句语法规则

4.JMP a 无条件跳转到a记录的位置

2.代码段b处的值改为代码段末尾位置的后一位置用于跳出循环

（3）for语句

for语句比较复杂，其规则为：for ( expression? ; expression? ; expression? ) statement。类似while语句，我们需要记录一个返回循环的位置a和一个跳出循环的位置b。但是这里的处理顺序为表达式1，表达式2，语句块，表达式3，然后向代码段写入指令：

先遍历第一个表达式的语法规则

1.a赋值为代码段末端的下一位置，用于跳回循环判断条件中

再遍历第二个表达式的语法规则

2.JZ b 判断条件为零，则跳到b中存的地址，同时b赋值为代码段末下一位

遍历语句的语法规则，遍历第三个表达式的语法规则

3.JMP a 无条件跳转到a记录的位置

4.代码段b中的值改为代码段末尾位置的后一位置

（4）do-while语句

C语言中do-while语句是先执行一遍循环体再判断条件语句，故其语法规则为do statement while ( expression ); 。依然按照顺序来处理语句和表达式，并增加对应的代码段指令：

1.a赋值为代码段末的后一地址，用于跳回循环体

首先遍历语句语法规则

再遍历表达式语法规则

2.JZ b 条件为零时跳转到b记录的地址，b记录代码段末后一位地址

3.JMP a 无条件跳转到a记录的地址

4.代码段b位置的值设置为代码段末后一位地址用于跳出循环

（5）return语句

return语句的语法规则为：return expression? ;。 先遍历表达式规则，再向代码段写入LEV指令即可。

### 5.2.4 表达式

（1）后缀表达式

目前实现了自增自减的后缀表达式，比如a++或a--这样的表达式。要实现这样的表达式，由于只有一个ax通用寄存器，所以先将主表达式的值进行自增或自减操作之后存回主表达式中，再拿一次主表达式的值做逆操作放到ax寄存器中，不再保存回主表达式中。因此先遍历主表达式，向代码段写入如下序列的指令：

先遍历主表达式的表达式规则，然后

1.PUSH 将主表达式的值地址(此时在ax中)存入栈中

2.LC/LI/LCFD/LIFD 将根据表达式的全局局部性，变量类型从ax的地址中取值

3.PUSH 将值存入栈中

4.IMM 1/4 根据基本类型/指针设置增量（指针为偏移量）放入到ax中

5.ADD/SUB 自增或自减操作

6.SC/SI/SCTD/SITD 将值存回主表达式中

7.PUSH

8.IMM 1/4

9.SUB / ADD 7 - 9 则是将表达式做逆操作变回原值

（2）前缀表达式

有两类情况，一类是前缀自增自减，另一类是其他一元运算符。类似后缀表达式的自增自减操作，前缀自增自减不需要做逆操作，指令如下：

先遍历后缀表达式的语法规则，然后

1.PUSH 将前缀表达式的值地址(此时在ax中)存入栈中

2.LC/LI/LCFD/LIFD 将根据表达式的全局局部性，变量类型从ax的地址中取值

3.PUSH 将值存入栈中

4.IMM 1/4 根据基本类型/指针设置增量（指针为偏移量）放入到ax中

5.ADD/SUB 自增或自减操作

6.SC/SI/SCTD/SITD 将值存回主表达式中

其他一元运算符则向代码段写入如下指令：

1.PUSH 将前缀表达式的值地址(此时在ax中)存入栈中

2.IMM 0/-1/1

3.PUSH MUL 若是前缀+/- 则与-1 / 1 做乘法

4.EQ/XOR 若是前缀！/~ 则与0 / 1 做与和非操作

运算表达式：描述的是二元操作的表达式，如加减乘除，求模取余，大于小于，左移右移，判断操作，位操作和赋值操作。处理方法如下：

先遍历符号左边的表达式的语法规则

1.PUSH 将表达式的值地址(此时在ax中)存入栈中

2.JZ / JNZ <addr> 如果是 && 或 || 操作

再遍历符号右边的表达式的语法规则

3.MUL/DIV/MOD/ADD/SUB/SHL/SHR/LT/GT/LE/GE/EQ/NE/AND/XOR/OR

各种二元运算操作

4.SC/SI/SCTD/SCTI 赋值表达式的指令

## 语义分析

语义分析阶段主要是为了展示四元式中间代码，本系统主要这几了四则运算表达式，大于小于号，赋值表达式，跳转语句的四元式代码。每条四元式形如（op，a，b， dr）

op为操作类型。

a为第一个表达式。

b为第二个表达式。

dr表示中间存储量。

其中op中的JZ代表着有条件跳转到该语句结束后的第一条语句。JMP为无条件跳转回该语句第一条语句。

## 解释器

本设计将虚拟机在代码中做成了一个单例模式，即整个编译程序运行的时候虚拟机始终是存在的。在词法语法分析之后的数据结构是存放在虚拟机之中的。虚拟机将指令集和符号表的字段相关内容定制成常量的形式。此外定义了3个段分别是文本段（TextSegment），数据段（DataSegment）和栈（StackSegment），与一个符号表（Symbols）。定义了四个寄存器ax通用寄存器、bp基地址指针、sp指针寄存器和pc程序寄存器。

虚拟机构建之初，会给3个内存段与符号表分配一个MemorySize(256\*1024)的内存空间用来收纳源代码分析出来的内容。当词法语法分析器分析完毕之后，主程序回调用虚拟机运行程序，即通过4个寄存器对指令进行处理。在这里定义了每个命令的处理方式，再采用PC程序寄存器顺序执行代码段中的指令来达到运行程序的效果。下面我们来介绍各个指令的含义：

1.IMM <num> 将 <num> 放入寄存器 ax 中。

2.LC 将对应地址中的字符载入ax中，要求ax中存放栈地址。

3.LI 将对应地址中的实数载入ax中，要求ax中存放栈地址。

4.SC 将ax中的数据作为字符存放入地址中，要求栈顶存放栈地址。

5.SI 将ax中的数据作为实数存放入地址中，要求栈顶存放栈地址。

6.LCFD 将对应地址中的字符载入ax中，要求ax中存放data段地址。

7.LIFD 将对应地址中的实数载入ax中，要求ax中存放data段地址。

8.SCTD将ax中的数据作为字符存放入地址中，要求栈顶存放data段地址。

9.SITD 将ax中的数据作为实数存放入地址中，要求栈顶存放data段地址。

10.PUSH 将ax中的数据放入栈中。

11.JMP<addr> 跳转指令，无条件将当前PC寄存器指定为<addr>。

12.JZ/JNZ 条件跳转指令，根据条件是否为0判断跳转情况。

13.CALL<addr> 函数调用指令，将当前PC寄存器指定为<addr>的子函数。

14.LEV从子函数返回.

15.ENT<size> 保存当前的栈指针，并在栈上保留<size>的空间。

16.ADJ<size> 将调用子函数时压入栈的数据清除。

17.LEA<offset> 用于调用子函数传入的参数。

18.EXIT 输出ax中的数据并退出程序。

运算符指令:

OR ,XOR ,AND ,EQ ,NE ,LT ,GT ,LE ,GE ,SHL ,SHR ,ADD ,SUB ,MUL ,DIV ,MOD则比较直接，定义op代表上述指令运算符。运算指令针对的是sp栈顶指针和ax寄存器之间的运算，最终会把结果存到ax寄存器中，虚拟机将会对运算符指令处理为：ax = StackSegment[sp++] op ax 的形式。

# 6 运行与测试

## 测试流程

### 6.1.1 主界面和导入源代码模块

初始界面如图6.1所示，点击“浏览…”按钮时弹出文件选择框用户即可选择要编译和运行的源代码文件导入，然后点击“确定”按钮，在界面上就会显示源代码，如图6.2所示。

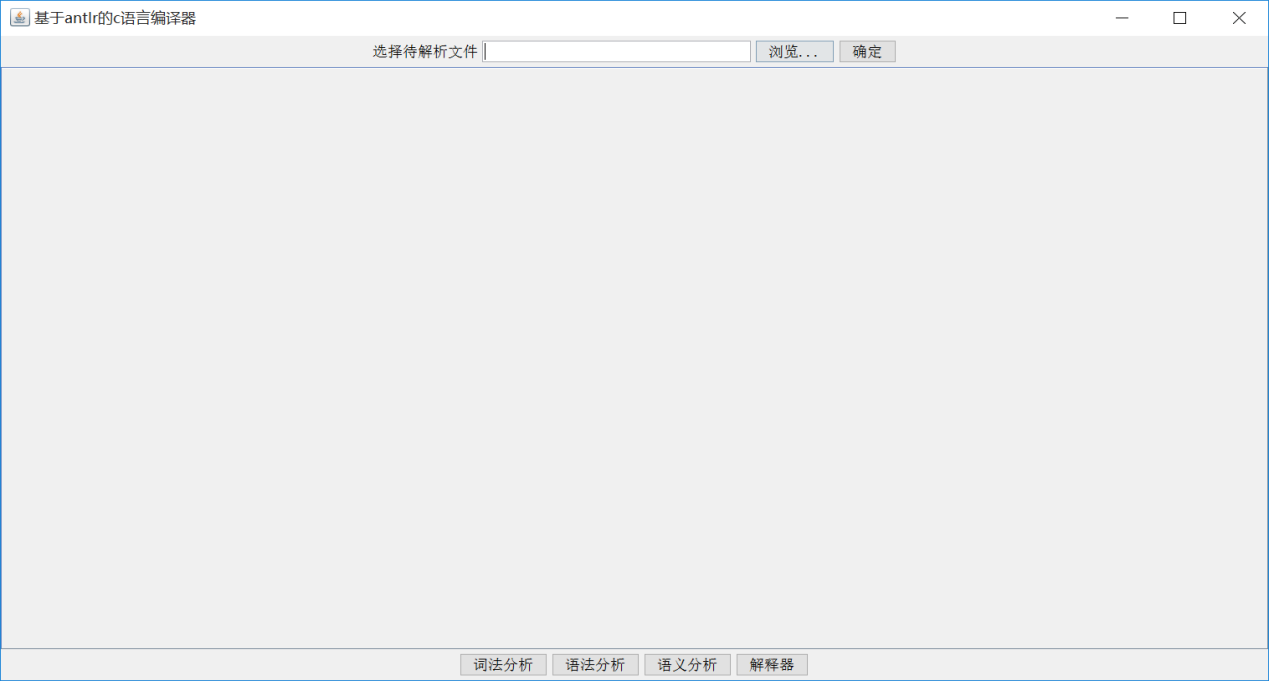


图6.1 初始界面图

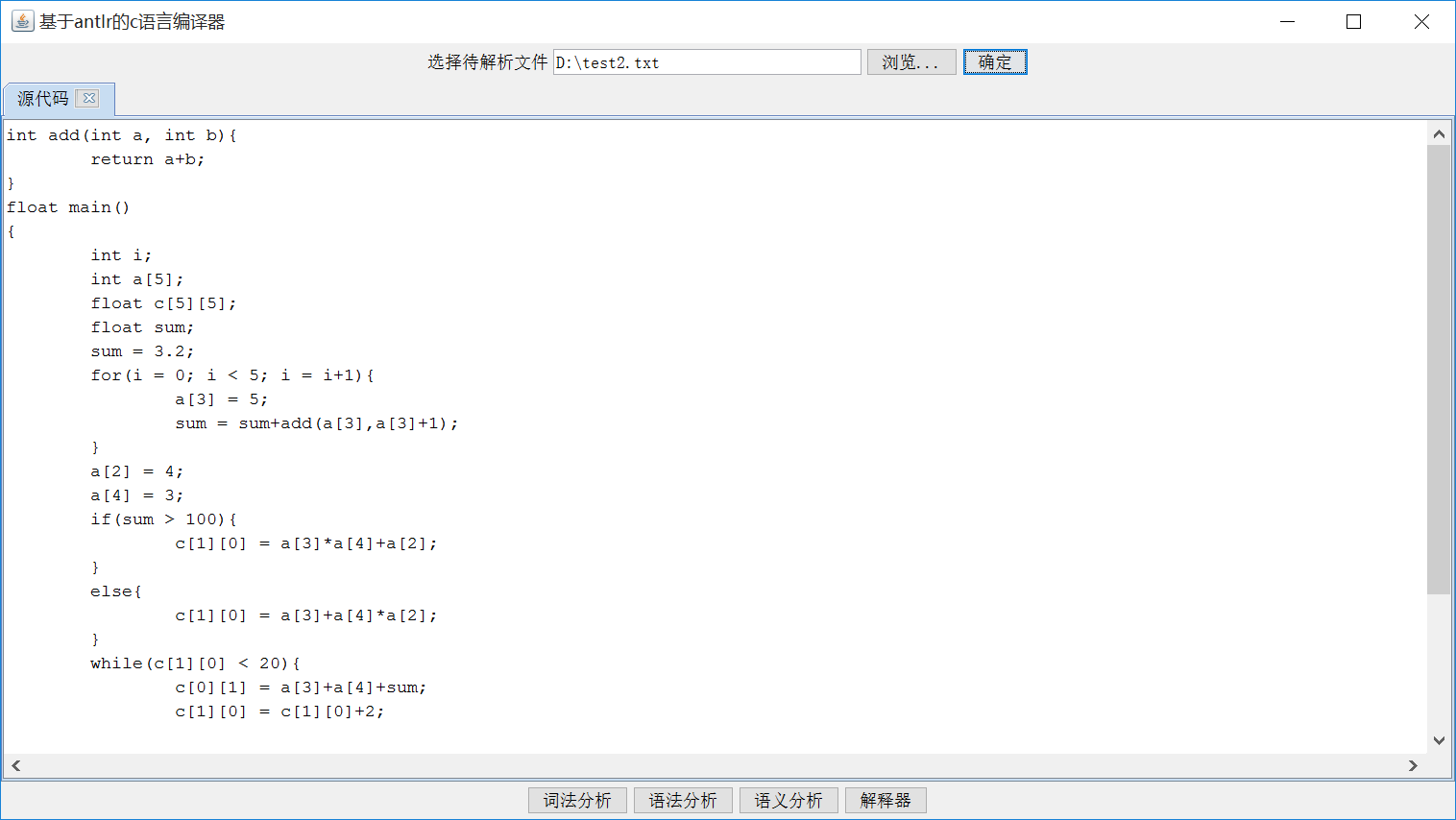
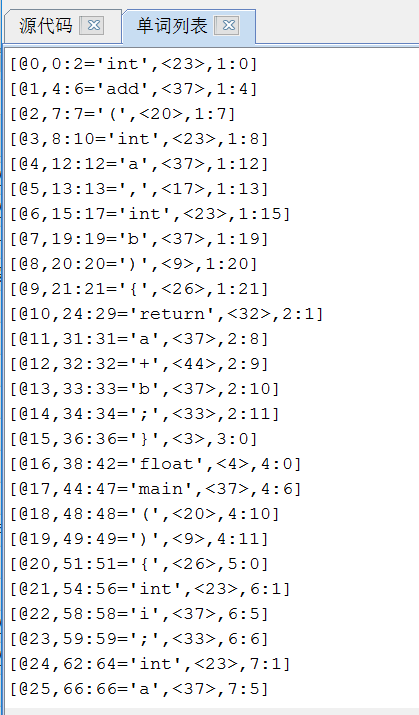
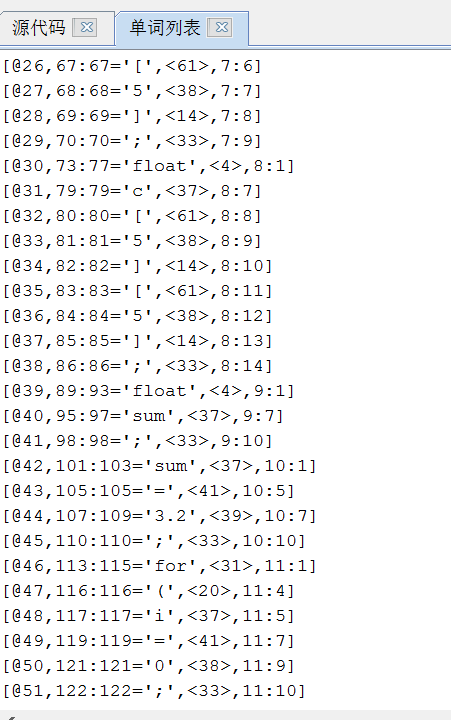


图6.2 源代码导入图

### 6.1.2 词法分析模块

点击底下面板中的“词法分析”按钮之后，界面中就会自动添加一个“单词列表”选项卡来显示经过词法分析之后的结果。如图6.3所示。

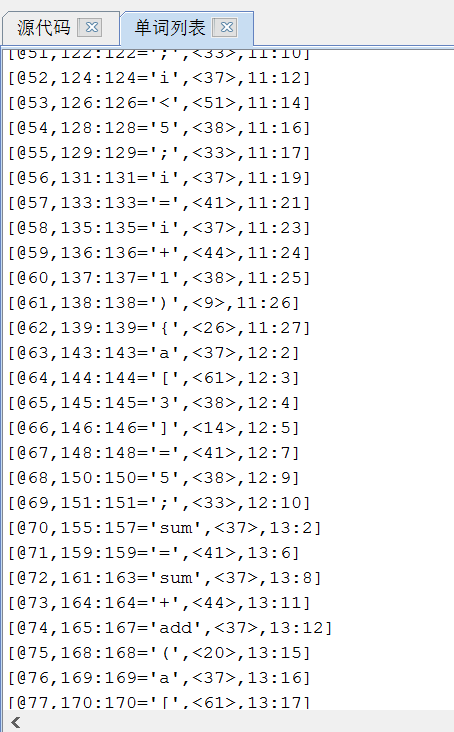
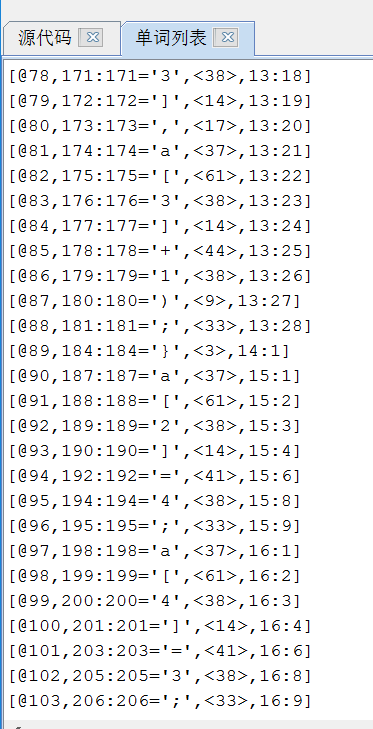
 

图6.3 词法分析结果图

### 6.1.3 语法分析模块

点击底下面板中的“语法分析”按钮之后，界面中就会自动添加一个“语法树”选项卡来显示经过语法分析之后的结果。

（1）变量以及函数类型和一维数组以及二维数组的识别，如图6.4所示。

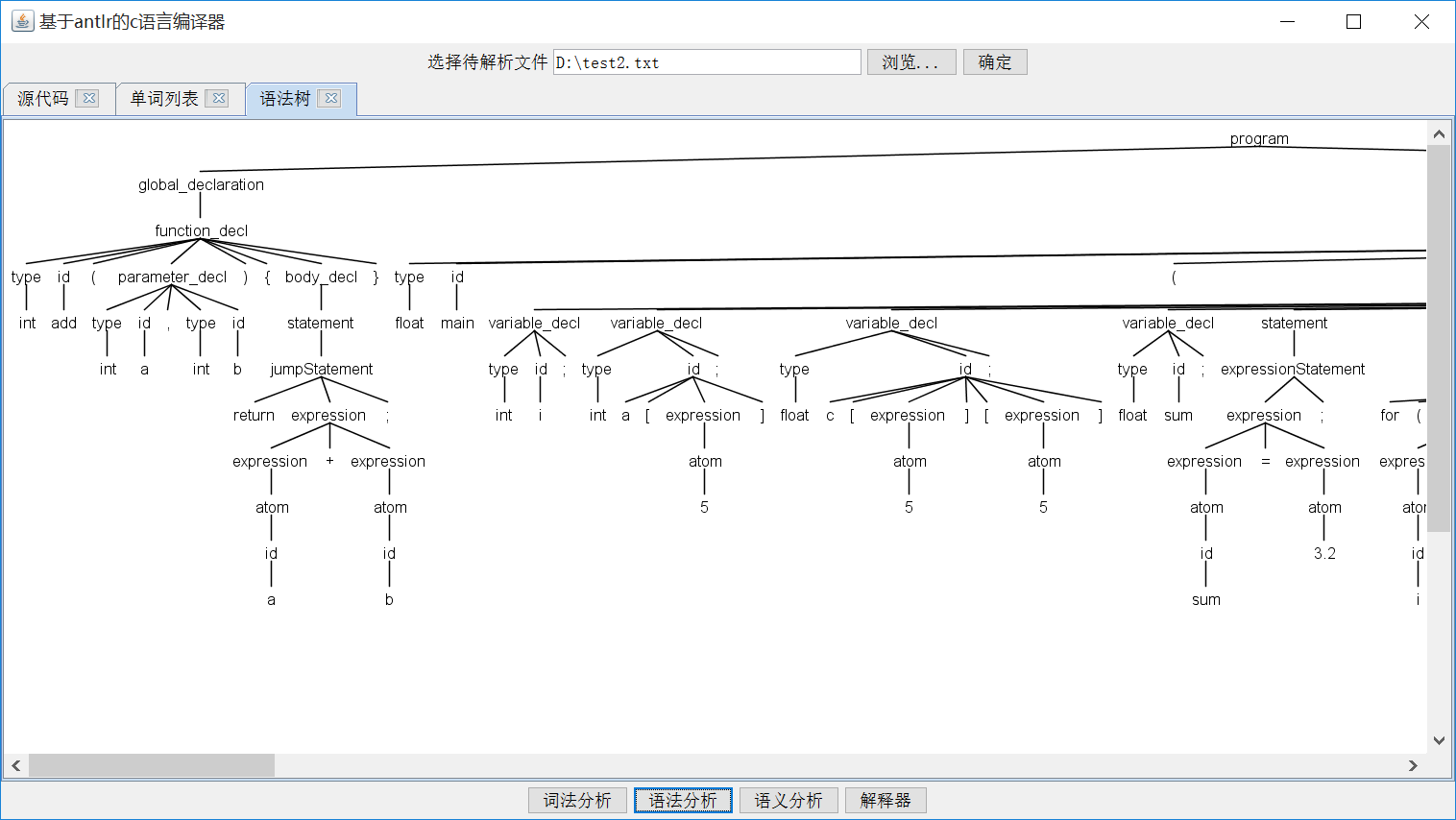


图6.4 变量定义以及类型识别图

（2）for语句的语法结构，如图6.5所示。

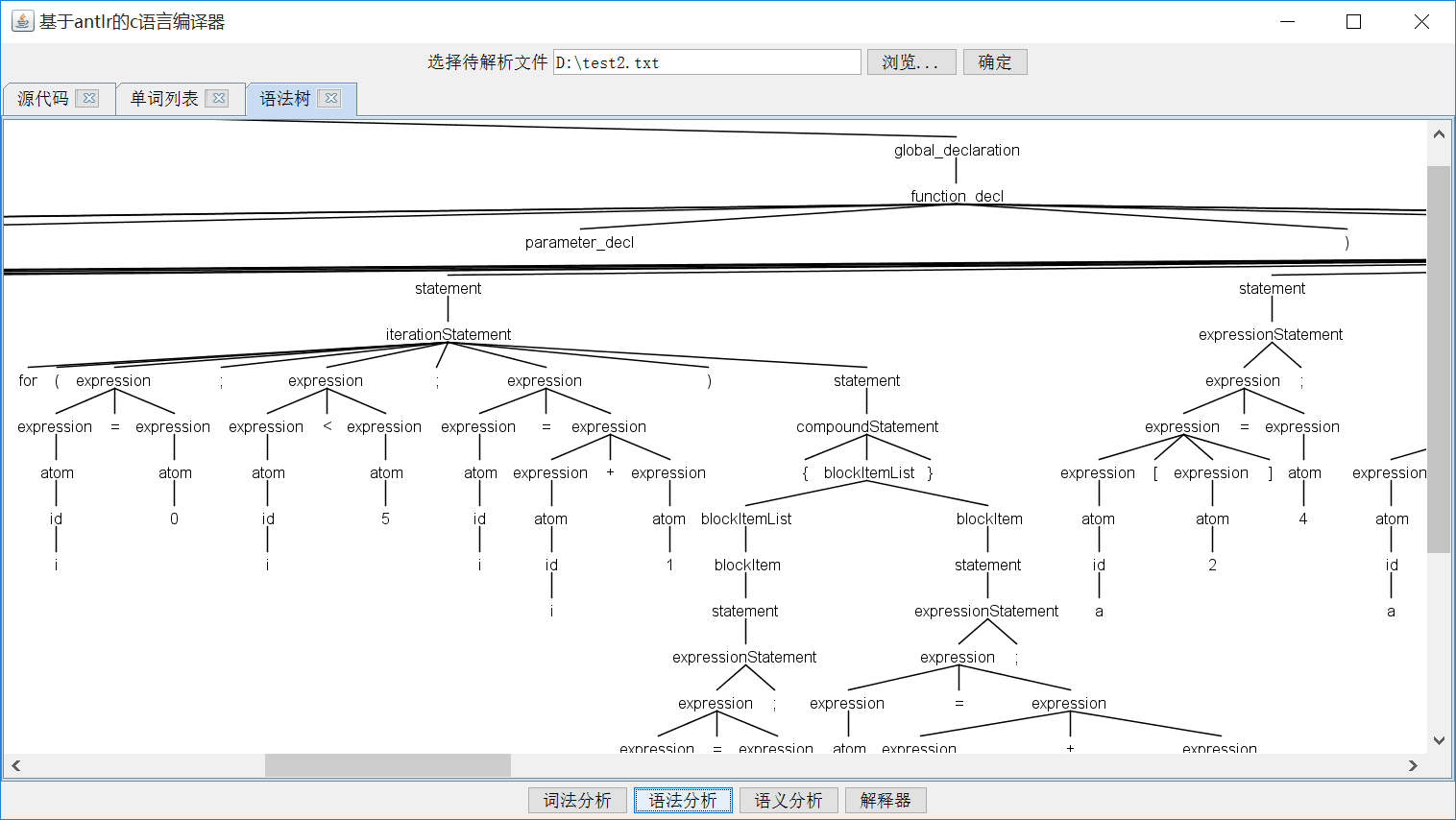


图6.5 for语句的语法结构图

（3）if语句的语法结构，如图6.6所示。

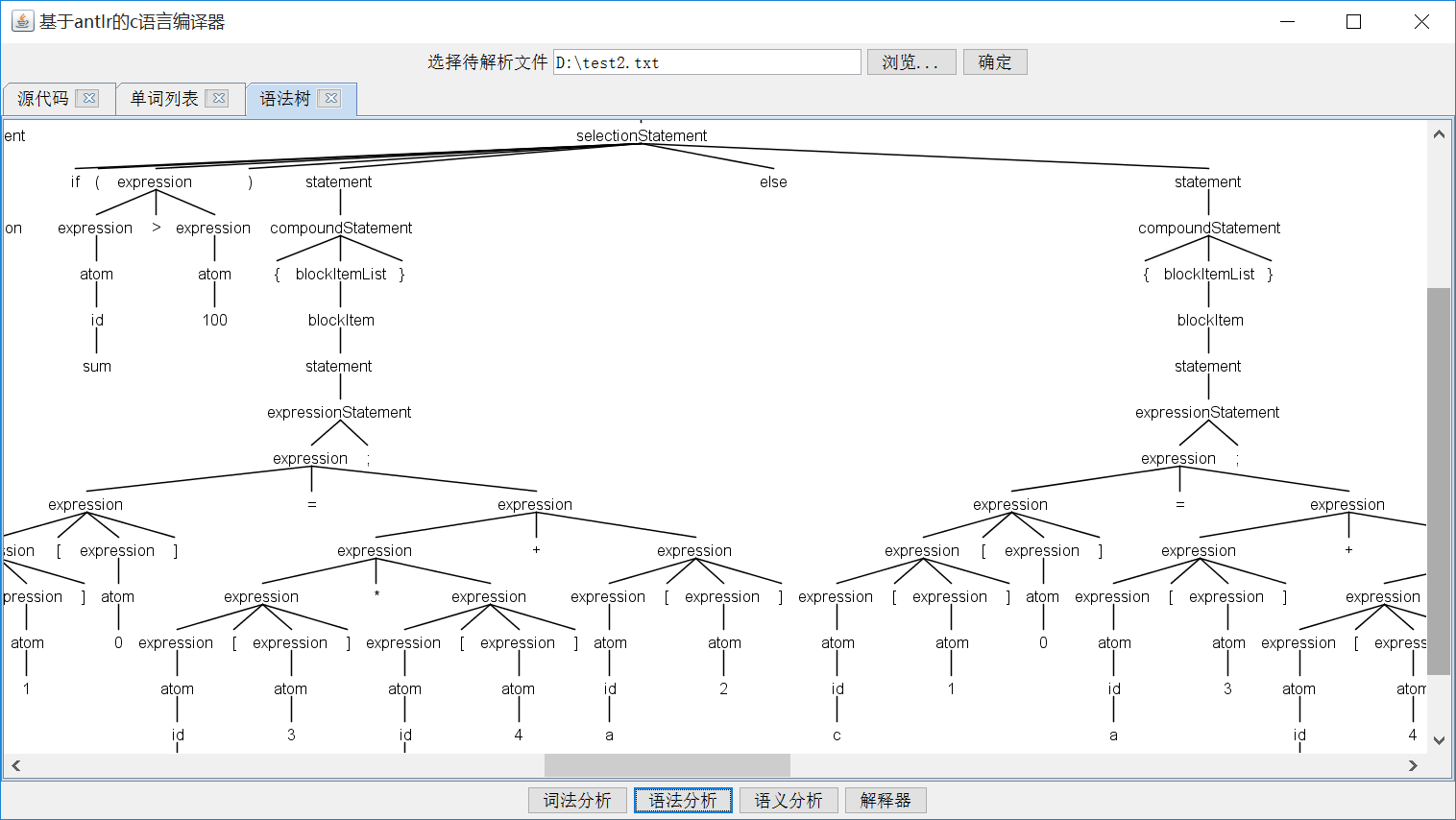


图6.6 if语句的语法结构图

（4）while语句的语法结构，如题6.7所示。

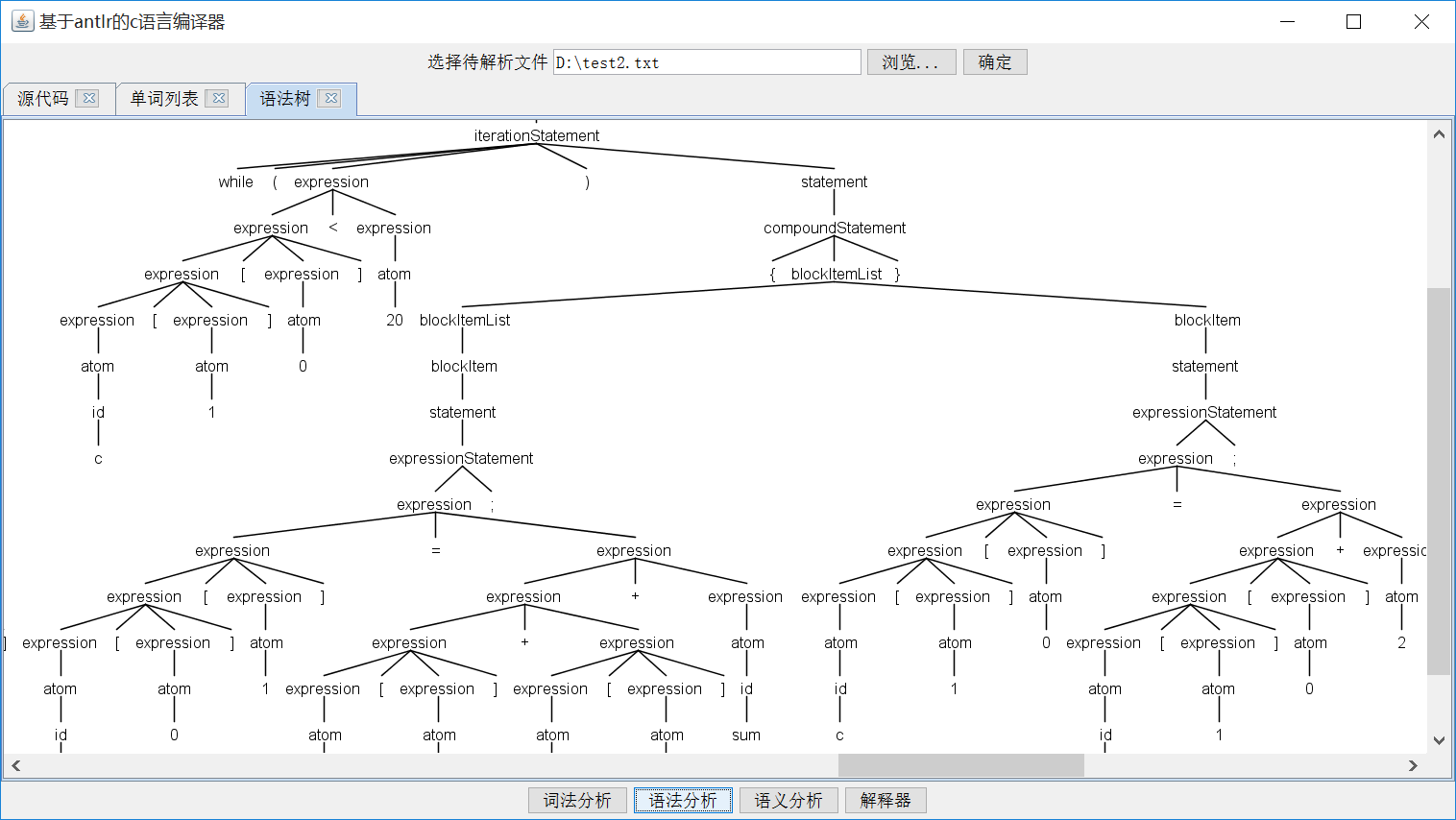


图6.7 while语句的语法结构图

（5）do-while语句的语法结构，如图6.8所示。

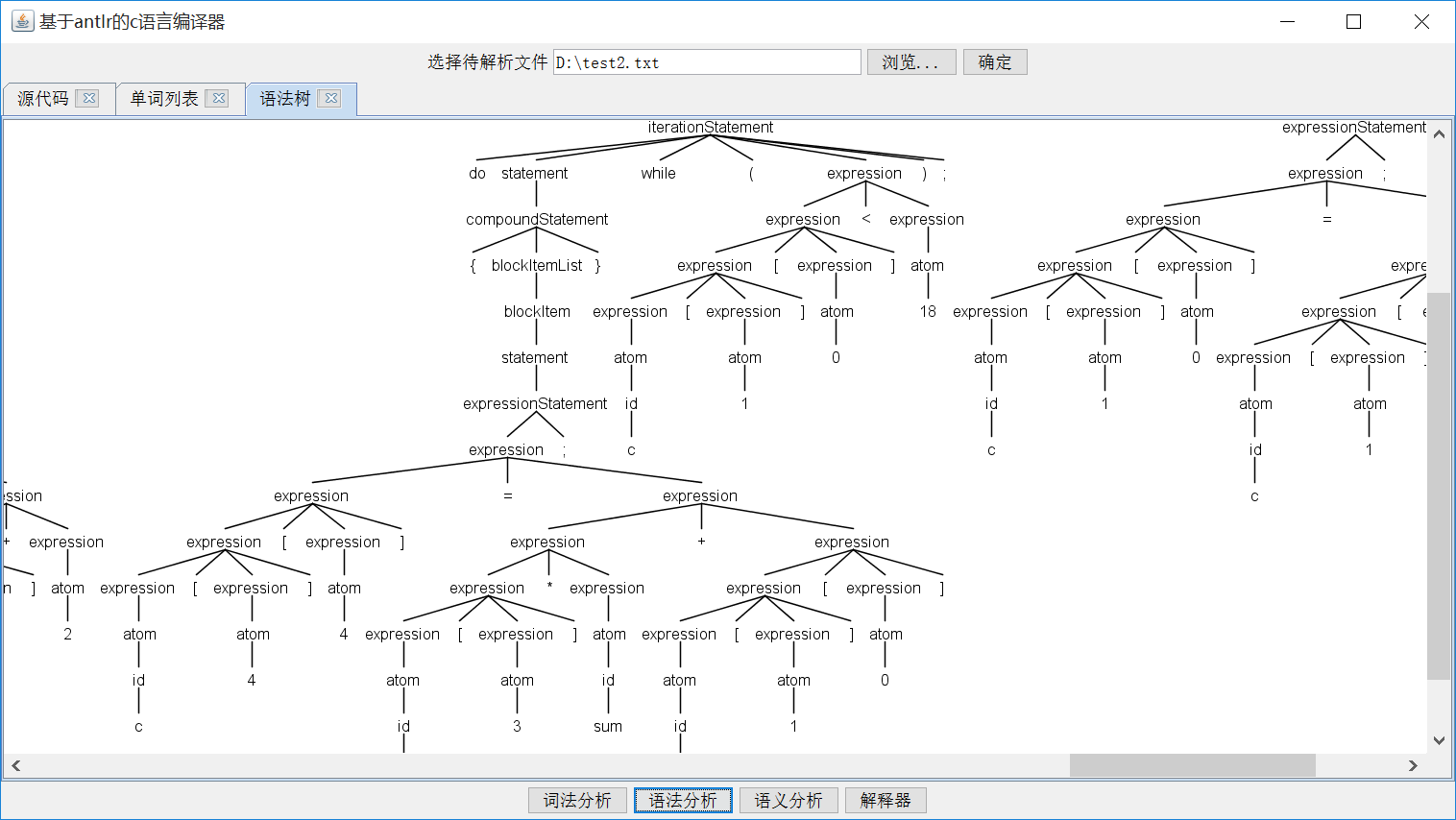


图6.8 do-while语句的语法结构图

（6）return语句的语法结构，如图6.8所示

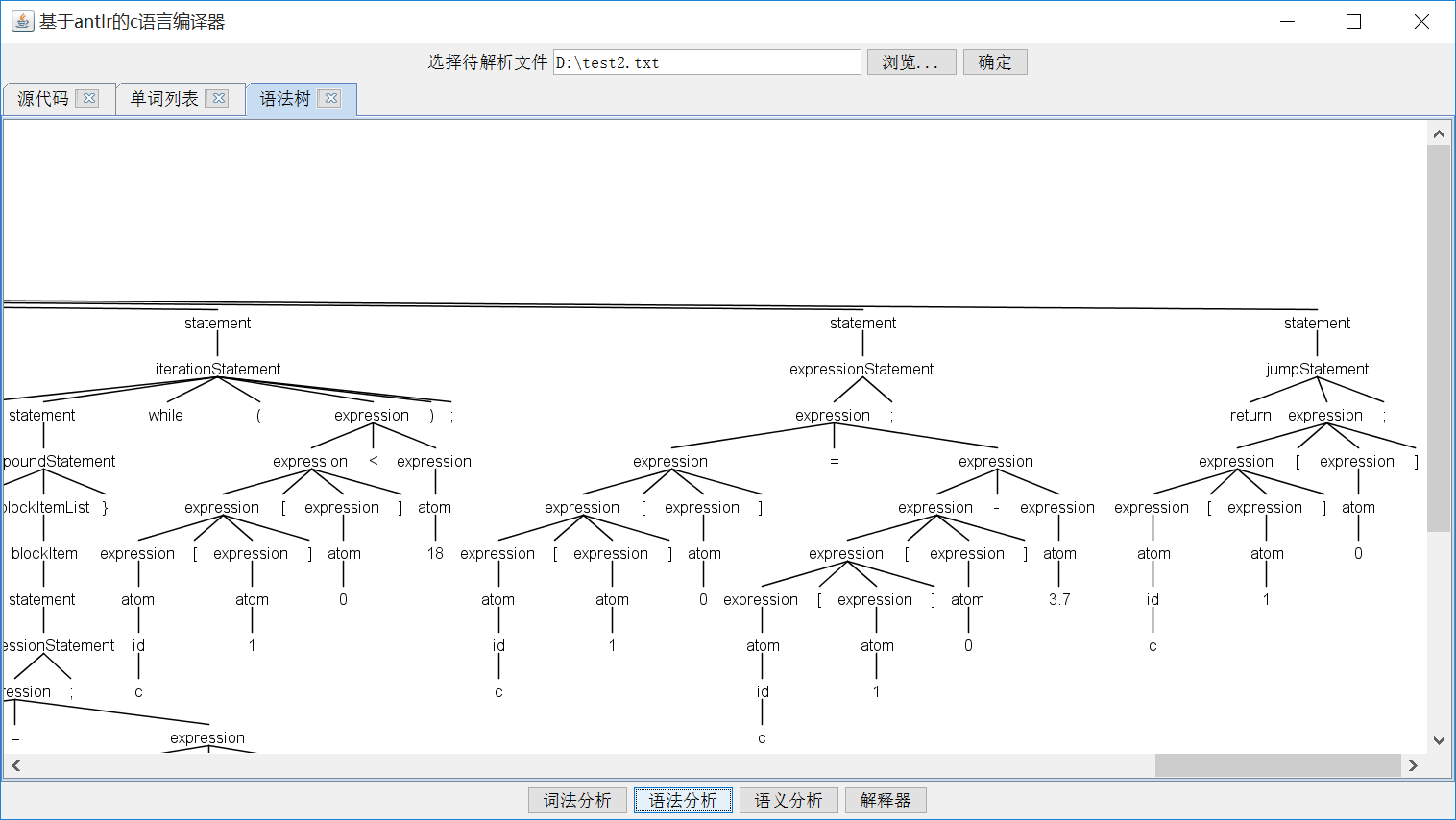


图6.8 return语句的语法结构图

### 6.1.4 语义分析模块

点击底下面板中的“语义分析”按钮之后，界面中就会自动添加一个“中间代码”选项卡来显示经过语义分析之后的结果。如图6.9所示。

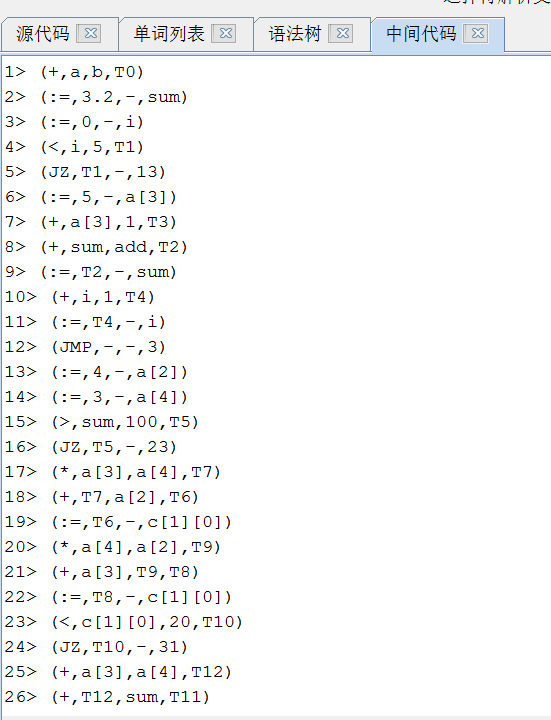
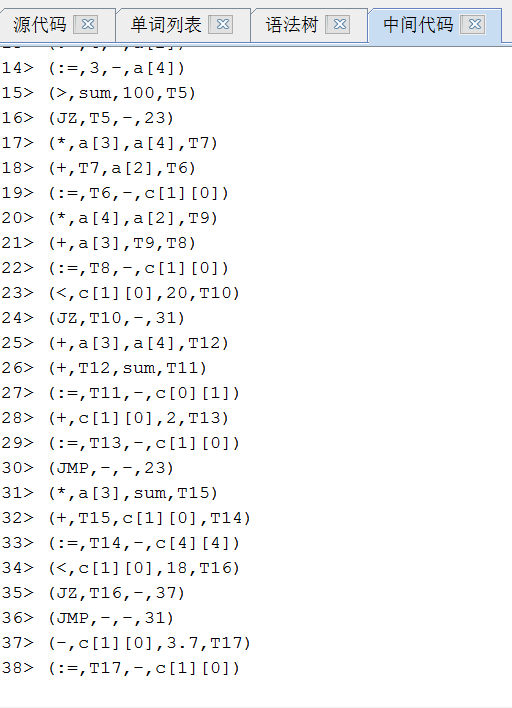
 

图6.9 中间代码图

### 6.1.5 解释器模块

点击底下面板中的“解释器”按钮之后，界面中就会自动添加一个“目标代码”选项卡来显示运行过程以及运行结果。

（1）赋值语句 sum = 3.2的目标代码，如图6.10所示。

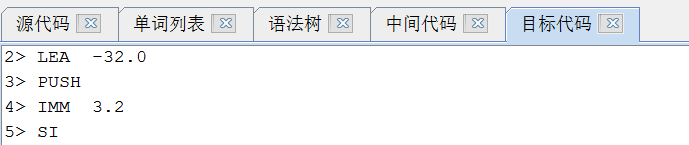


图6.10 赋值语句的目标代码图

（2）for语句的目标代码，如图6.11所示。

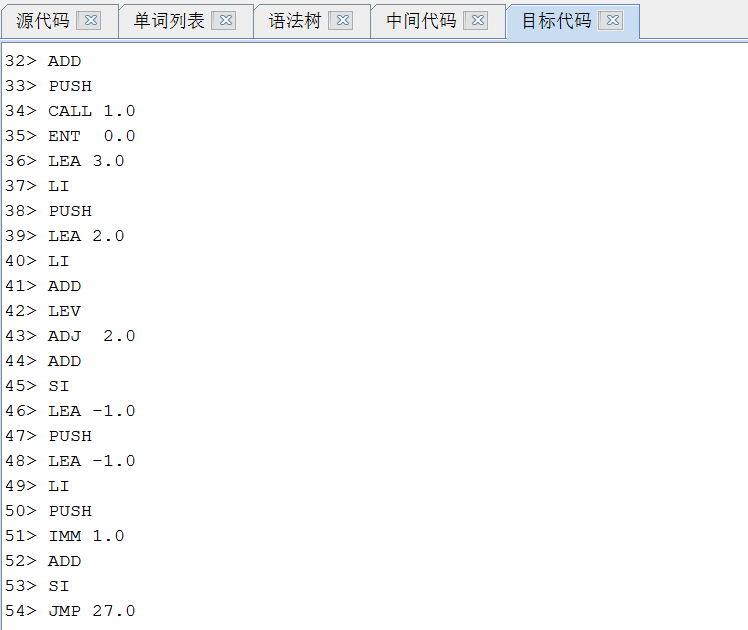
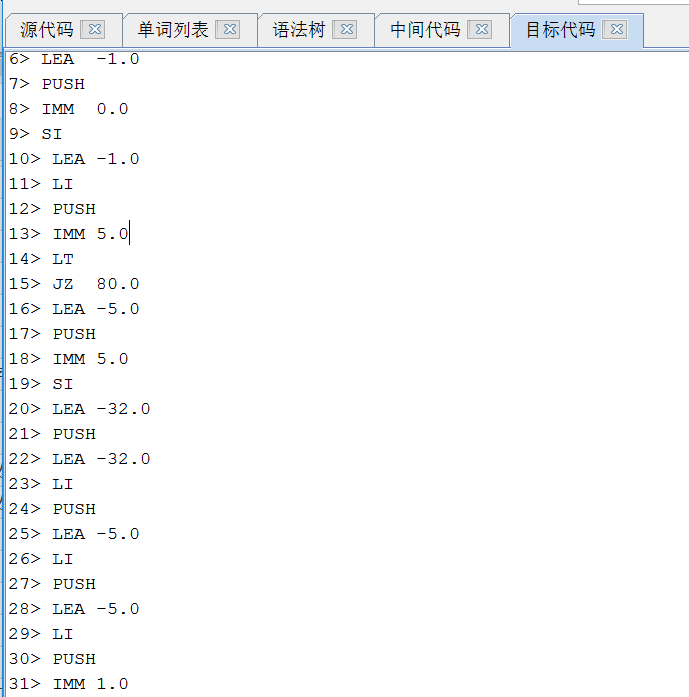


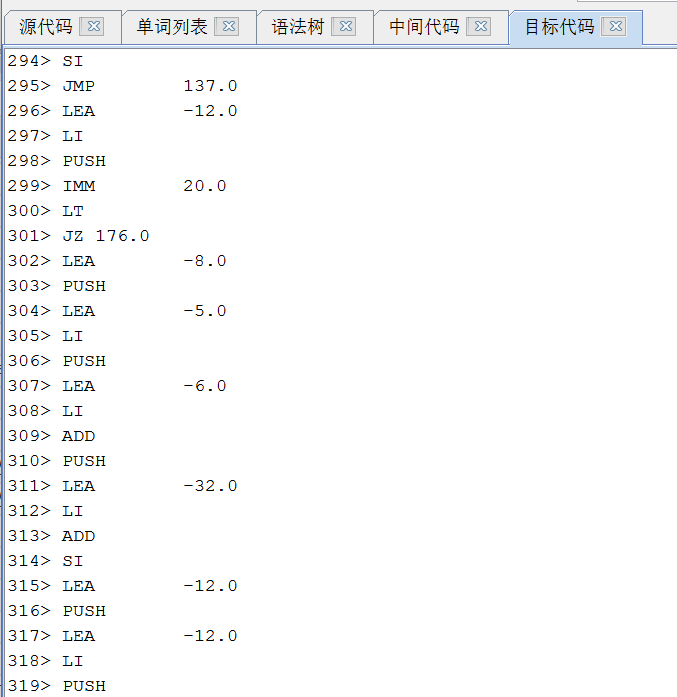
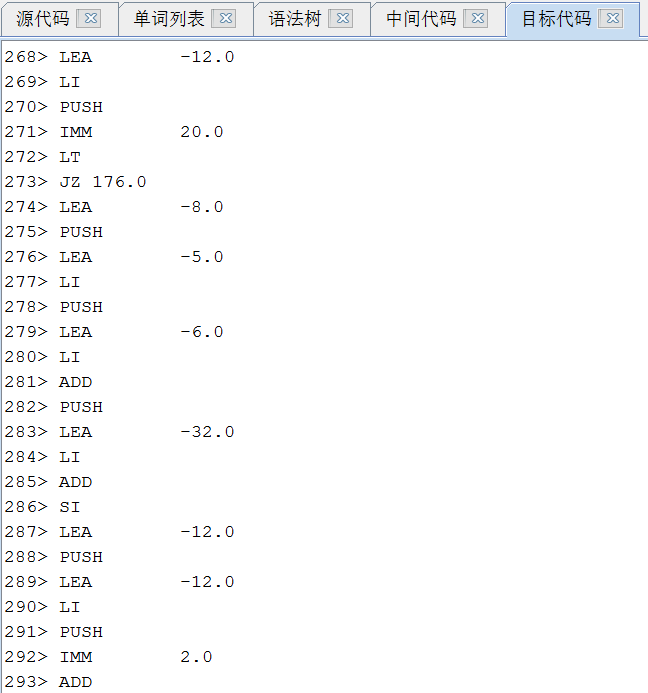
图6.11 for语句的目标代码图

（3）if语句的目标代码，如图6.12所示



图6.12 if语句的目标代码图

（4）while语句的目标代码，如图6.13所示。



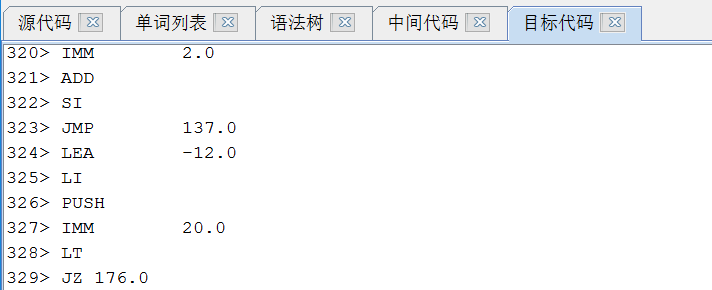


图6.13 while语句的目标代码图

（5）do-while 语句的目标代码，如图6.14所示。

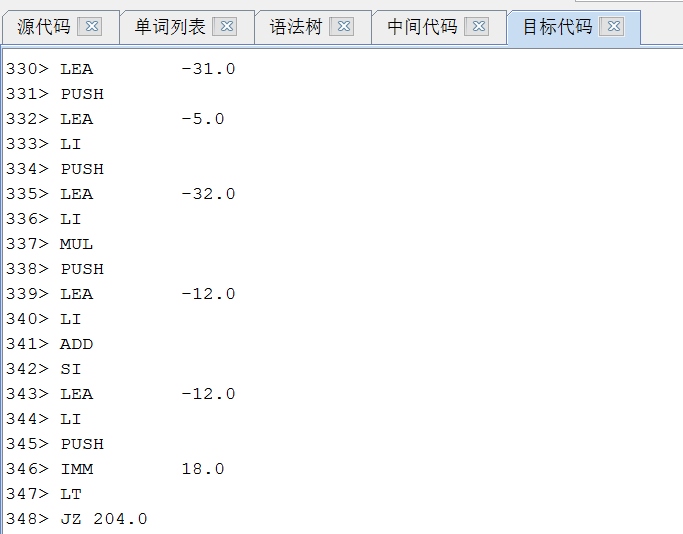


图6.14 do-while语句的目标代码图

（6）return语句的目标代码以及最终运行结果，如图6.15所示。

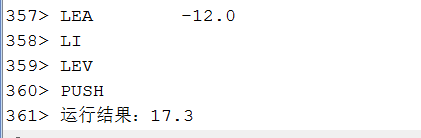


图6.15 运行结果图

## 测试结果反馈

### 6.2.1 测试总体情况

经过多轮的测试，对于需求中的功能都能通过测试。

### 6.2.2 系统需要优化的地方

（1）对于界面处理，点击一个功能按钮出现该功能的选项卡是，理论上应该直接跳转该选项卡界面，但目前的系统只能通过手动的方式点击该选项卡才能跳转。

（2）还没有实现ANTLR的出错机制，也就是还没能运行过程中出现的词法和语法的错误的具体信息反馈到用户界面。

（3）本系统实现的只是C语言的一个子集的编译，还有很多C语言中比较复杂的数据结构和语句还没能实现。

### 6.2.3 对系统的评价

（1）交互性好，可移植性高，具有与用户良好交互的界面，将编译和运行的每个过程的结果展现给用户，供学生更加方便和直观的学习。

（2）实现了一个C语言子集的编译器，完成了一个完整的C语言源程序的编译和运行，包括几种for，if，while等较复杂语句的识别和编译。

（3）很好的模拟了虚拟机运行的过程，通过定义各种指令来模拟虚拟机的运行过程，是学生能够从底层去更加深刻了解和学习编译运行过程。

（4）当然也存在一些问题，比如没有实现出错机制，也只是停留在C语言基本语句的处理上，还没能处理更高级别的C语言的数据结构和语句，这也是本系统的一个不足之处也是今后努力的方向。

# 结 论

本系统的设计是基于ANTLR工具的基础上实现的C语言的编译器，实现了源代码的输入以及每个模块分析结果的输出，其中所实现的功能包括词法分析、语法分析、语义分析和解释器四大模块。

（1）在词法分析模块中完成了对void、int、char、float、double基本数据类型的变量和函数，以及一维数组和二维数组的识别和处理；

（2）在语法分析模块中对于C语言的语句，实现了赋值语句、条件语句

（if）、循环语句（for，while，do-while），返回语句return以及函数调用语句的编译和运行过程；

（3）在语义分析模块中对语法分析中出现的语句设计四元式代码，简化源代码，是用户更加直观理解源代码的意思；

（4）在解释器模块中构建虚拟机通过定义19中常规指令和16中运算指令完成程序的执行，最后将程序运行结果显示出来。

通过对各个功能模块的运行测试，本系统所完成的功能均达到预期的效果。同时，可以预见本系统将会对学生学习编译原理和编译过程起到极大的帮助，提高学生的学习效率和增强学生的动手能力。

当然本系统还存在很多不足之处。比如设计上还不太完善，语言结构剖析还不够精细，代码冗余还有存在。还有很多复杂的特性没有实现，目前也只是C语言完整编译器较小的一个子集。以上种种问题，都有待后续的重构和改善。因此对于本编译器的后续扩展的话应该基于以上几点：

（1）完善词法和语法规则的定义，使其能够识别更多和更加复杂的C语言结构；

（2）增加语法检查以及报错功能，实现更加友好的用户交互；

（3）标准库的实现以及更高级的汇编指令都是要继续拓展和改进的部分。

# 参 考 文 献

[1] The Definitive ANTLR 4 Reference. Terence Parr[M] . USA: Pragmatic Bookshelf, 2013: 21-31，57-149.

[2] ANTLR 4权威参考读书笔记[EB/OL]. <http://codemany.com/tags/antlr/>

[3] teverett. C.g4[EB/OL]. <https://github.com/antlr/grammars-v4/blob/master/c/C.g4>

[4] Language Implementation Patterns. Terence Parr [M] .USA: Pragmatic Bookshelf, 2010: 20-65，88-146，290-323.

[5] 吴伟民，李小妹，刘添添，黄剑锋等. 数据结构 [M] . 广东：广东工业大学计算机学院，2015：90-104，139-157.

[6] 谭浩强. C程序设计（第四版）[M] .北京：清华大学出版社，2010:378,379.

[7] 张素琴. 编译原理（第二版）[M] .北京：清华大学出版社，2015: 75-96，169-202

[8] Erich Gamma.李英军[译].设计模式-可复用面向对象软件基础[M] .北京：机械 工业出版社，2000: 84-90.

[9] antlr4+eclipse环境搭建以及创建实例[EB/OL]. <http://blog.csdn.net/zjq_1314520/article/details/65935718>

[10] 周志明. 深入理解Java虚拟机（第2版）[M] . 北京：机械工业出版社，2013：302-358

# 致 谢

本系统的设计和论文的编写是在我的指导教师刘添添老师的亲切关怀和悉心指导下完成的。在题目的选择上感谢刘添添老师给予的引导和帮助；在系统的完成过程中，非常感谢刘添添老师对我的指导，对系统功能的引导，逻辑的纠正，刘添添老师给予了我极大的帮助；在论文的完成阶段，更是感谢刘添添老师的耐心指导和修正。在此向刘添添老师表示深深的感谢和崇高的敬意。

最后再次感谢在开发系统和编写毕业论文过程中给予我帮助的所有人以及提供参考文献的所有人，感谢你们对我的帮助和支持。