

**本科学生实验(实践)报告**

|  |  |
| --- | --- |
| **院 系:** | 计算机学院 |
| **实验课程:** | 编译原理项目 |
| **实验项目:** | minic编译器实验 |
| **指导老师:** | 黄煜廉 |
| **开课时间:** | \_\_2020\_\_**～**\_\_2021\_\_**年度第**\_2\_**学期** |
| **专 业:** | 计算机科学与技术 |
| **班 级:** | 4班 |

**华南师范大学教务处**

**华南师范大学实验报告**

|  |  |  |  |
| --- | --- | --- | --- |
| **学生1姓名**  **与学号** | 梁泽浩20172131086 | **学生2姓名**  **与学号** | 张勇标  20172131077 |
| **专 业** | 计算机科学与技术 | **年级、班级** | 2017级4、5版 |
| **课程名称** | 编译原理项目 | **实验项目** | minic编译器 |
| **实验类型** | **🞎验证🞎设计 🗹综合** | **实验时间** | **2020 年 03 月 02 日** |
| **实验指导老师** | 黄煜廉 | **实验评分** |  |

1. **实验内容**

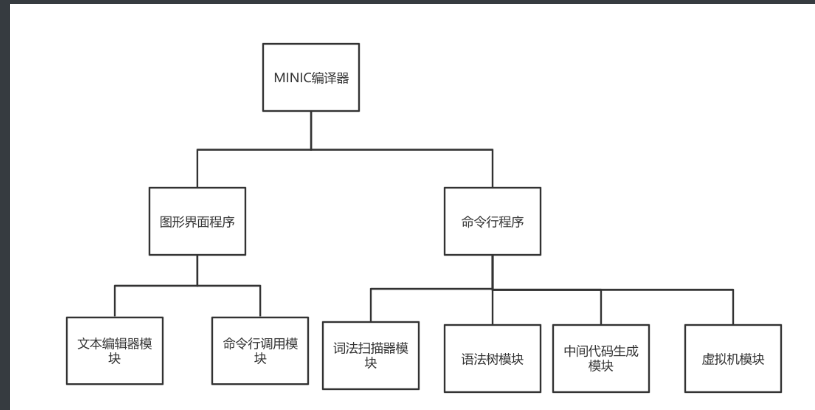
结合设计范例,提出一个实现 Mini C 语言编译程序的构造要求，完成数据结构分析、概要设计、详细设计、程序编制、调试、提出实验报告书

1. **实验目的**

本实验就是为了加强理论联系实际,了解编译程序的概貌和培养其实际动手能力。 进一步提高独立分析问题和解决问题的能力以及提高对中、大型软件的分析和开发能力。

1. **实验文档**

实验设计结构：

****

# 详细文档

## 编译工具

g++和make：windows可使用[mingw64](https://sourceforge.net/projects/mingw-w64/files/Toolchains%20targetting%20Win64/Personal%20Builds/mingw-builds/8.1.0/threads-posix/seh/x86_64-8.1.0-release-posix-seh-rt_v6-rev0.7z/download)，解压后配置环境变量并修改make的名称为make.exe即可;linux平台一般自带了g++和make

flexc++和bisonc++：只在Linux平台上有，可以通过软件仓库安装sudo apt install flexc++ bisonc++(apt是Ubuntu的包安装管理器，其他发行版请自行百度)

## 重要术语

共用体（Union）：C或C++的一种数据类型， 共用体可以定义多个数据类型， 它们共用同一个内存区域， 共用体变量的大小以最大的数据类型为准， 在同一时刻共用体只能被解释成其中一种类型

可变参数：需要include <stdarg.h>,可以把函数的最后一个参数的数据类型定义为...以允许函数有不定长个参数，在函数内通过va\_list ap;获取可变参数，例子如下：

#include<stdarg.h>   
 void yyerror(const char\* message, ...){  
 fprintf(listing, "Error at Line %d Column %d: ", yylloc.first\_line, yylloc.first\_column);  
 va\_list ap; // 指向参数的指针  
 va\_start(ap, message); // 宏初始化  
 vfprintf(listing, message, ap);  
 va\_end(ap);  
 fprintf(listing, ".\n");  
 Error = TRUE;  
 }

## 词法分析

关键字： IF ELSE INT VOID WHILE RETURN

标识符：ID

数字字面量：NUM

运算符=、==、！=、<=、>=、<、>、+、-、×、/：ASSIGN EQ NEQ LTEQ GTEQ LT GT PLUS MINUS TIMES OVER

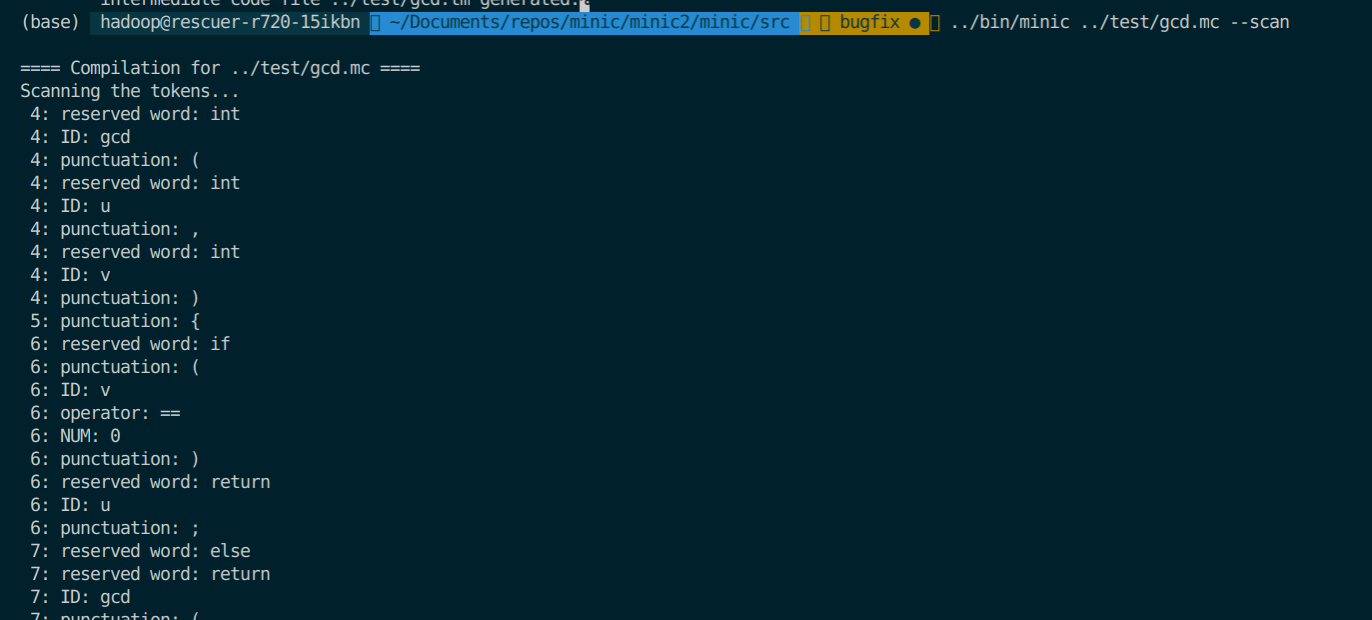
括号符号(、)、[、]、{、}： LP RP LB RB LC RC

冒号和逗号：SEMI COMMA

此处定义的符号就是语法分析用到的终结符

使用flex工具生成， 需要输入一个声明了词法符号（使用正则表达式定义）的lex文件（.l后缀的文本文件）

词法符号每行以符号名 : 符号我类型： 实际值的格式打印，例子如下：



## 语法分析

### Bison工具介绍

使用bison生成语法分析器，通过分析器产生语法树。分析器使用的是LALR(1)分析法， 需要输入一个定义BNF和语义动作的yacc文件（.y后缀的文本文件）。yacc文件的重要关键字介绍如下

#### %union

%union {   
 struct treeNode \* node;  
 int op;   
 int val;   
 char name[256];  
};

有时候需要指定终结符和非终结符的数据类型，这些数据类型就定义在 %union块中

#### %type

%type<node> term

用于声明词法符号，如果加上<node>， 则指定词法符号的数据类型为node，这里的node是在%union中定义的

#### yyerror

#include<stdarg.h>   
void yyerror(const char\* message, ...)  
{  
 fprintf(listing, "Error at Line %d Column %d: ", yylloc.first\_line, yylloc.first\_column);  
 va\_list ap; // 指向参数的指针  
 va\_start(ap, message); // 宏初始化  
 vfprintf(listing, message, ap);  
 va\_end(ap);  
 fprintf(listing, ".\n");  
 Error = TRUE;  
}

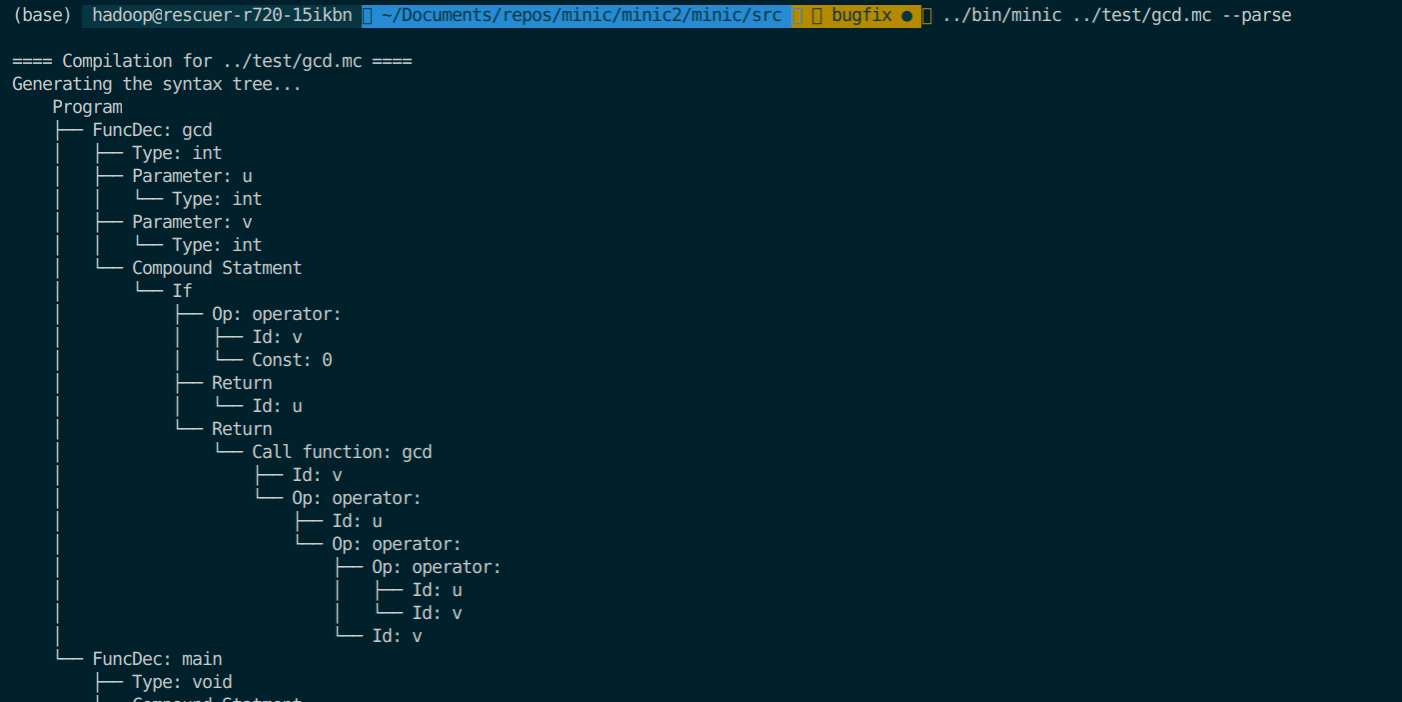
实现yyerror函数，当语法分析出错时执行该函数，一般是打印错误发生的行号位置

yacc源文件内容分为三个部分如下：

符号声明及选项标志，还可以include一段C代码  
%%  
产生式及产生式规则定义  
%%  
自定义C函数

### 语法树定义

语法树示意图如下：



树节点（TreeNode）包含属性如下

* 众孩子结点指针
* 兄弟结点指针
* 行号
* 结点类型
* 结点属性
* 表达式类型（用于语义分析中的类型检查， 可空）
* 描述信息（打印语法树时显示的内容， 可选)

结点类型如下

* 语句结点（StmtK)
* 表达式结点（ExpK）
* 声明结点（DeclK)
* 参数结点(ParamK)
* 类型结点(TypeKind)

每个树节点只能属于一种结点类型， 故将结点类型定义在一个共用体变量中

**语句结点取值**：if-结点（IfK）， while-结点（IterK）， return-结点（RetK）， 复合结点（CompK）

**表达式结点取值**：赋值结点（AssignK）， 调用结点（CallK），操作符结点（OpK）， 常量结点（ConstK)，非数组变量（下面简称变量）结点(IdK)， 数组结点(ArrIdK)

**参数结点取值**：数组参数(ArrParamK)， 非数组参数(NonArrParamK)

**类型结点取值**：类型名(TypeNameK)

各个结点类型的所有取值定义成一个枚举变量， 如enum {IfK, IterK, RetK, CompK} StmtK；表示StmK变量保存了语句节点的具体取值

结点属性如下

* 操作符名
* 数据类型
* 值
* ID名
* 数组属性arr
* 作用域

以上结点属性不能同时存在于一个树节点中，树节点的结点类型决定了使用哪一个结点属性， 故结点属性使用共用体变量保存

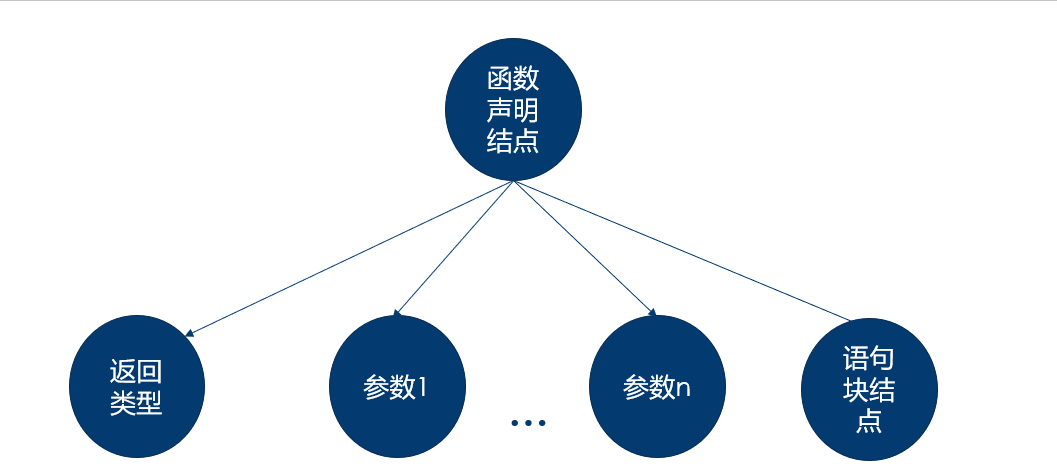
数组属性如下

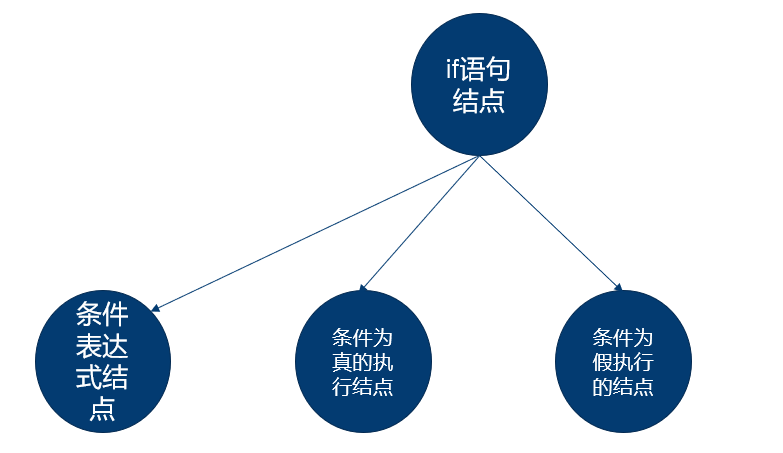
* ID名
* 数据类型
* 规模大小

表达式类型根据需要给结点赋予表达式类型，用于语义分析阶段的类型检查

语法树结构定义如下

语法树是一个森林，返回的根节点包含了兄弟节点的指针。每种结点的结构示意图如下：





### 匹配关键产生式执行的规则

var-declaration→ type-specifier ID ;

1. 左部创建VarK的声明结点
2. 设置结点的行号和ID名
3. 修改孩子指针

var-declaration→ type-specifier ID[NUM];

1. 左部创建ArrVarK的声明结点
2. 给结点设置行号、数组属性的ID名和规模大小
3. 修改孩子指针

type-specifier→ int | void

1. 左部创建TypeNameK的类型结点
2. 设置结点的数据类型

fun-declaration→ type-specifier ID( params ) compound-stmt

1. 左部创建FuncK的声明结点
2. 设置结点行号和ID名
3. 修改孩子指针

params→ void

1. 左部创建TypeNameK的类型结点
2. 设置结点数据类型

param→ type-specifier ID

1. 左部创建NonArrParamK的参数结点
2. 设置ID名
3. 修改孩子指针

param→ type-specifier ID[ ]

1. 左部创建ArrParamK的参数结点
2. 设置ID名
3. 修改孩子指针

compound-stmt→ { local-declarations statement-list }

1. 左部创建CompK的语句结点
2. 修改孩子指针

selection-stmt→ if(expression) statement

| if(expression)statement else statement

1. 左部创建IfK的语句结点
2. 修改孩子指针

iteration-stmt→ while (expression) statement

1. 左部创建IterK的语句结点
2. 修改孩子指针

return-stmt→ return ; | return expression;

1. 左部创建Ret的语句结点
2. 修改孩子指针

expression→ var = expression | simple-expression

1. 左部创建AssignK的表达式结点
2. 修改孩子指针

var→ ID

1. 左部创建IdK的表达式结点
2. 设置ID名

var→ ID[expression]

1. 左部创建ArrIdK的表达式结点
2. 设置数组属性的Id名
3. 修改孩子指针

simple-expression→ additive-expression relop additive-expression

additive-expression→ additive-expression addop term

term→ term mulop factor

1. 左部创建OpK的表达式结点
2. 设置结点的操作符名
3. 修改孩子指针

factor→ NUM

1. 左部创建ConstK的表达式结点
2. 设置值

call→ ID(args)

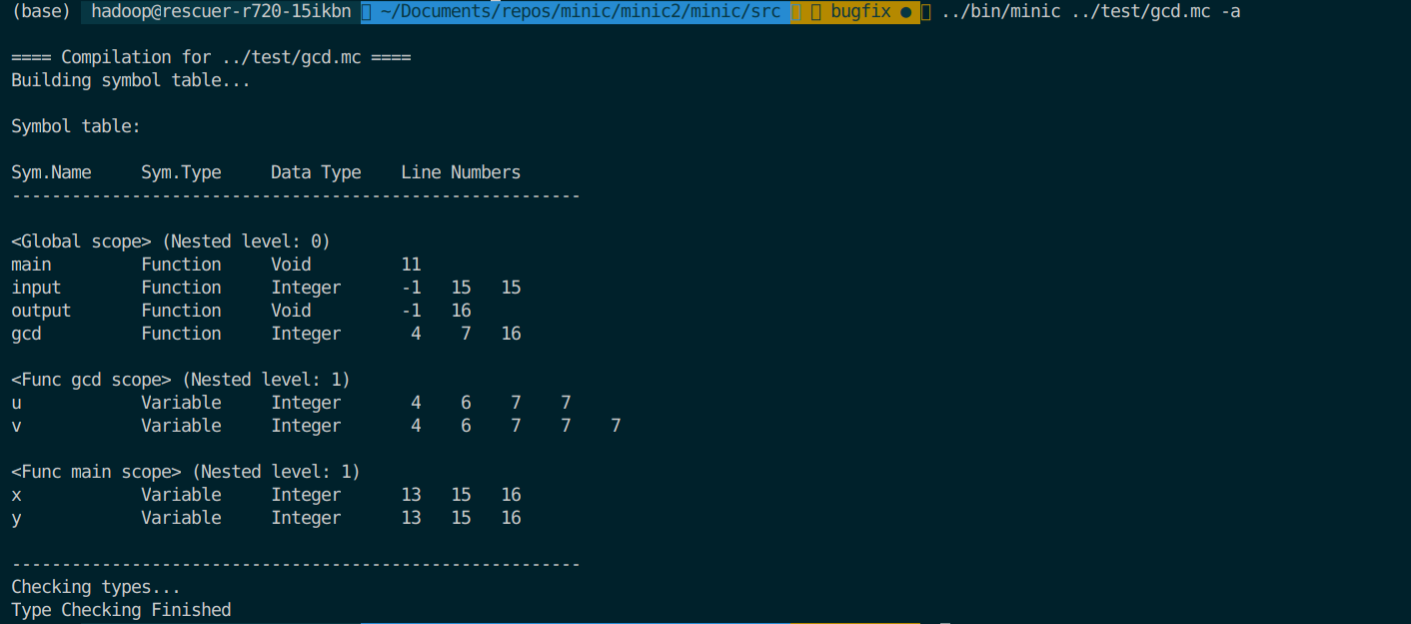
1. 左部创建CallK的表达式结点
2. 设置ID名

### 打印语法树

调用parse函数分析语法并执行上面的语义动作后，语法树的结构已经生成，函数会返回语法树的根节点，通过根节点的深搜可以遍历整个语法树

## 语义分析

主要包含类型检查和生成符号表两个部分，运行结果如下：



### 类型检查

检查内容如下

* minc的input和output函数是minic预定义的，类型检查前需要为input函数和output函数建立语法树节点，并把符号插入到符号表中
* 开始函数应新建一个局部作用域
* 函数体结束后其作用域应失效
* while语句的条件不能为空
* return语句的返回类型应和函数定义的返回类型一致
* 数组不能作为单变量使用
* 单变量不能作为数组使用
* 下标应该是整数（int）
* 数组类型不能为VOID
* 调用的函数应该已经被定义
* 调用时传入的参数个数应和函数定义时的形式参数个数保持一致
* 无参函数不能传入参数
* 形式参数类型应和实际参数类型一致
* 参数数组类型应和
* 函数和变量不能未定义就使用（除了input和output函数）
* 变量和函数不能重复定义
* 计算操作符两边的操作数的类型应同为int（因为minic只定义了int这一种数值类型）

### 生成符号表

在一个作用域内声明的变量都要加到该作用域的符号表中

#### 符号表打印格式

|  |  |  |  |
| --- | --- | --- | --- |
| 符号名 | 符号类型 | 数据类型 | 行号 |
|  |  |  |  |

符号可能会在多处地方出现，每处的行号都要加入符号表中

符号的内存位置是模拟的，只需要保证同一个作用域的符号位置不同就行了，为了方便分配位置，使用一个location表存储每个作用域已分配的变量的位置数量

程序会可能有多个作用域，本项目由于变量声明只能出现在全局或函数体的开头，因此作用域只分全局作用域和某函数作用域两类。每个作用域都有一个符号表，符号表前应有一个标题，指出此符号表的作用域范围和嵌套层次，标题格式为<作用域名称> (嵌套层次)

#### 数据结构选择

符号用一个结构体表示，属性包含标识符名字、存放变量的内存位置、语法树节点(可以得到数据类型)和行号列表

同一个符号的所有行号使用链表存储，每个节点存储行号和下一个节点的指针

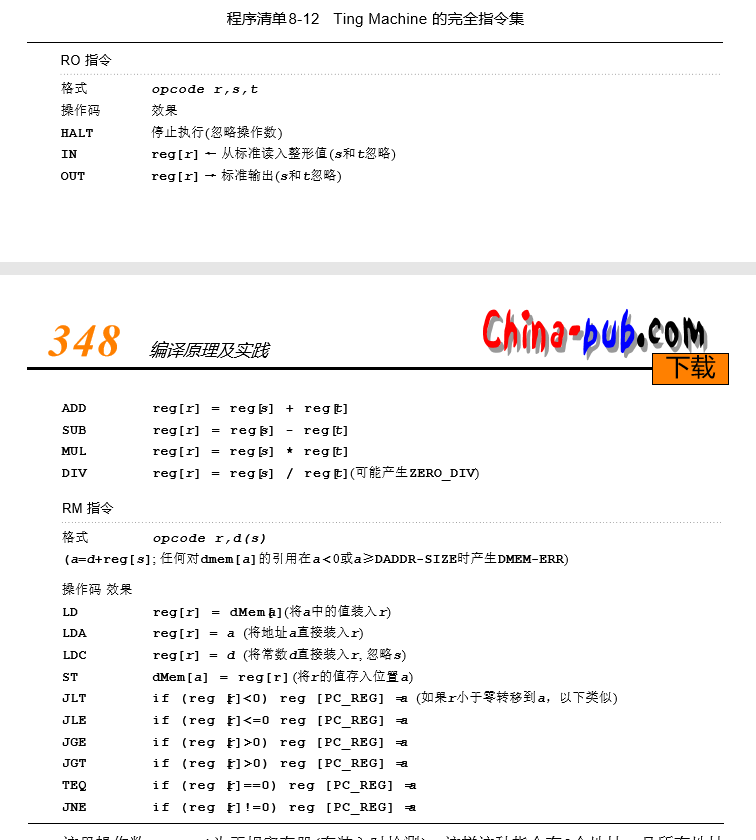
一个作用域有多个符号，用一个结构体表示。包含的成员有存储该作用域声明的所有变量的散列表、作用域的嵌套深度、指向父作用域的指针

多个作用域使用倒排链表+数组，即每个作用域都有一个指向父作用域的指针，每个作用域变量存到一个数组里。为了便于计算嵌套的深度和插入符号， 需要一个LIFO的数据结构，也就是栈来临时存放作用域

## 中间代码生成

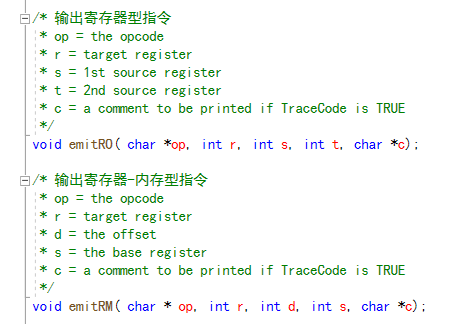
### 中间代码指令集

中间代码使用的是教材Tiny Machine的完全指令集：

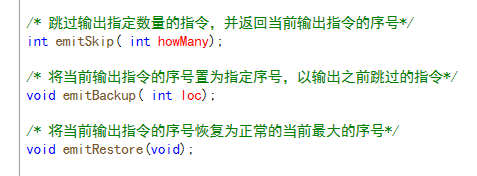


此指令集仅有RO指令和RM指令两种格式，其简便性和规范性为程序的编写和输出带来了极大的优势。

### 输出中间代码



程序分别利用emitRO和emitRM两函数格式化地输出RO和RM指令。

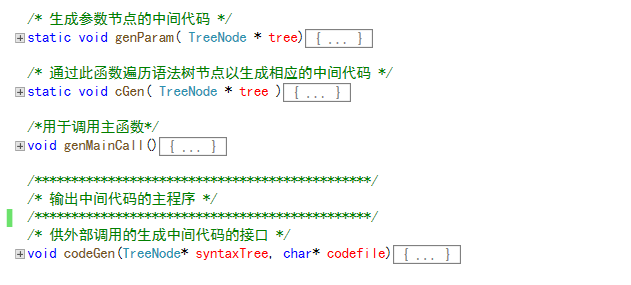


此外，在生成if或while代码段的中间代码时，程序会依次执行emitSkip、emitBackup、emitRestore函数，以达到跳过及回填假测试条件的跳转语句

和立即跳转语句、恢复当前指令序号的目的。

### 中间代码指令集



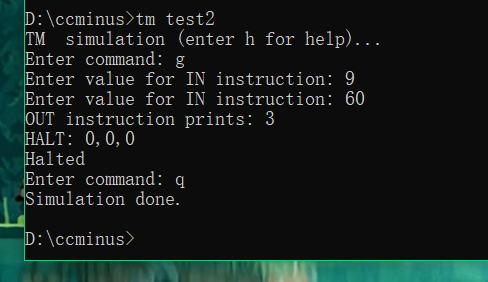


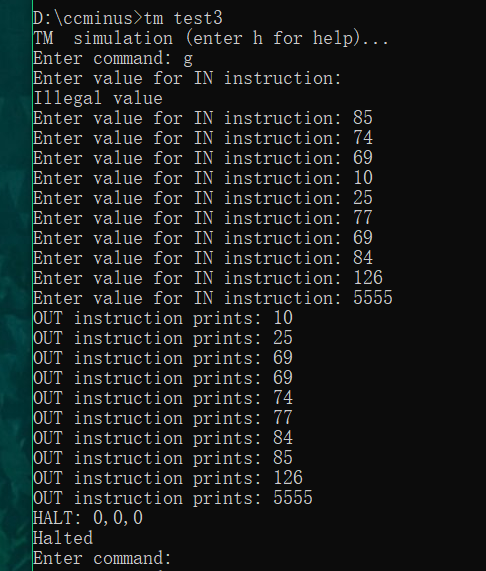
函数codeGen是供外部调用生成中间代码的唯一接口，也是中间代码生成的主程序。

在该函数中，程序首先生成tiny Machine的标准序言、设置启动时的运行环境，然后用cGen函数遍历语法树，对不同类型的语法树节点调用相应的函数生成对应的中间代码。最后调用genMainCall函数，使得程序从main函数处开始执行，并生成程序结束指令“HALT 0,0,0”。

此外，程序还利用了多个静态整型变量来充当全局偏移量和局部偏移量等，配合在代码输出文件定义的指令计数器、全局变量指针、缓存指针等寄存器，使得该中间代码的生成工作得以顺利进行。

### 运行示例





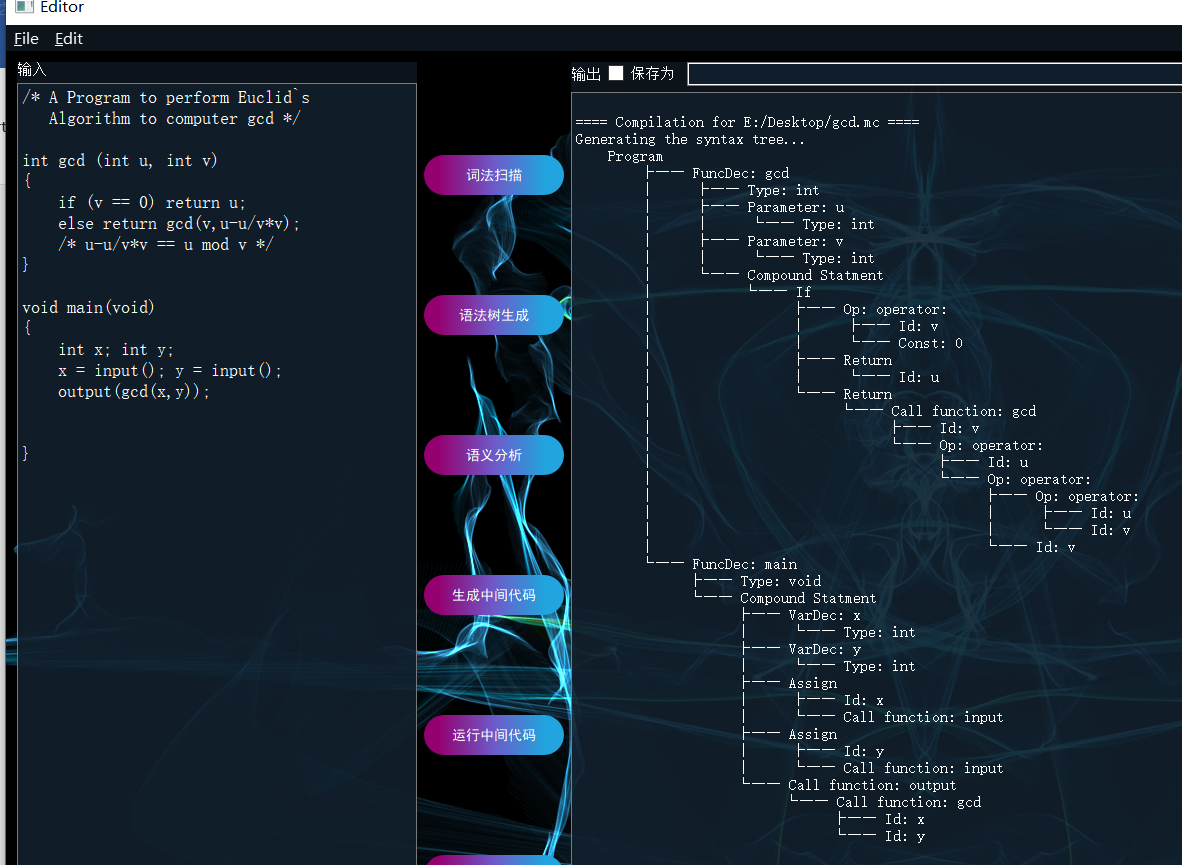
以上为运行两示例函数gcd和sort的结果：以TM指令启动tiny Machine后输入指令g便可成功运行程序，正确输入操作数后程序可正常运作至结束。

## 测试文档

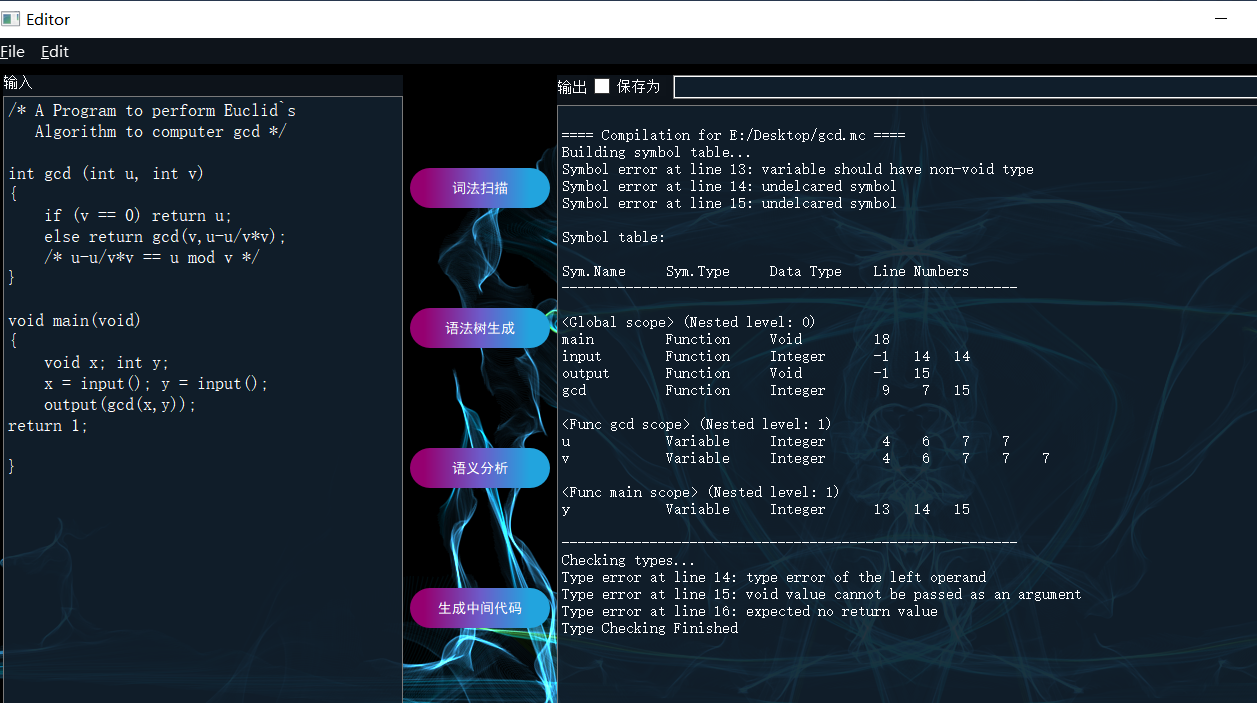
1.组织相应的测试数据来测试 Mini C 的所有词法。



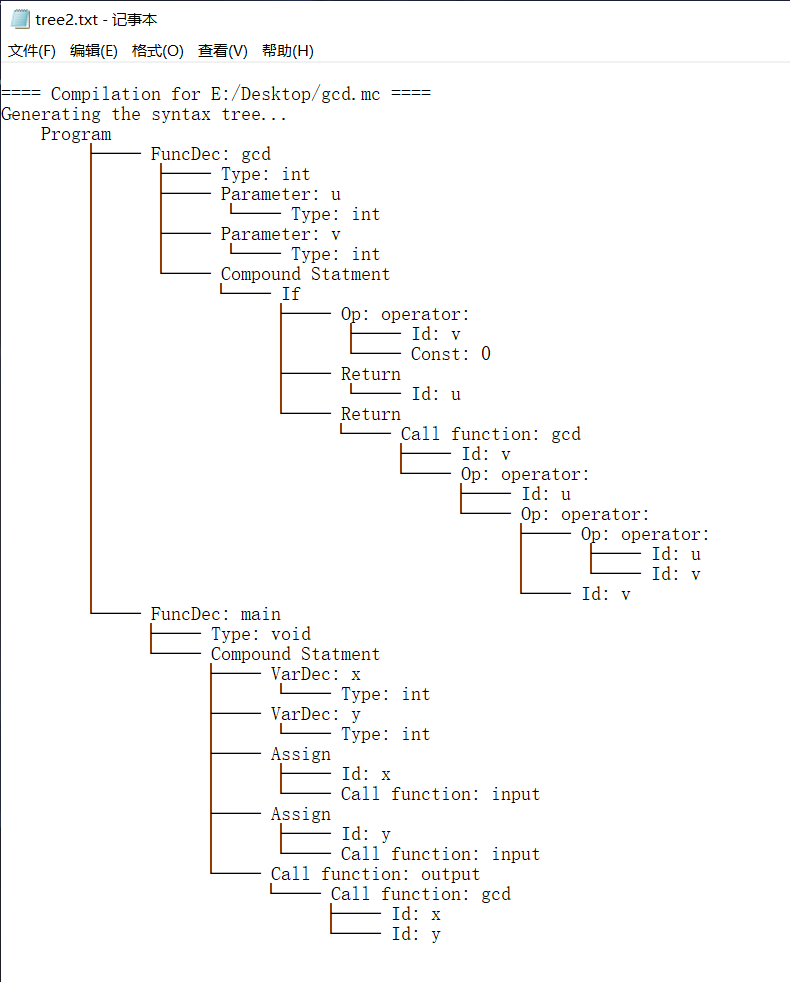
2.组织相应的测试数据来测试 Mini C 的所有语法。



3.组织相应的测试数据来测试 Mini C 的所有语义问题。

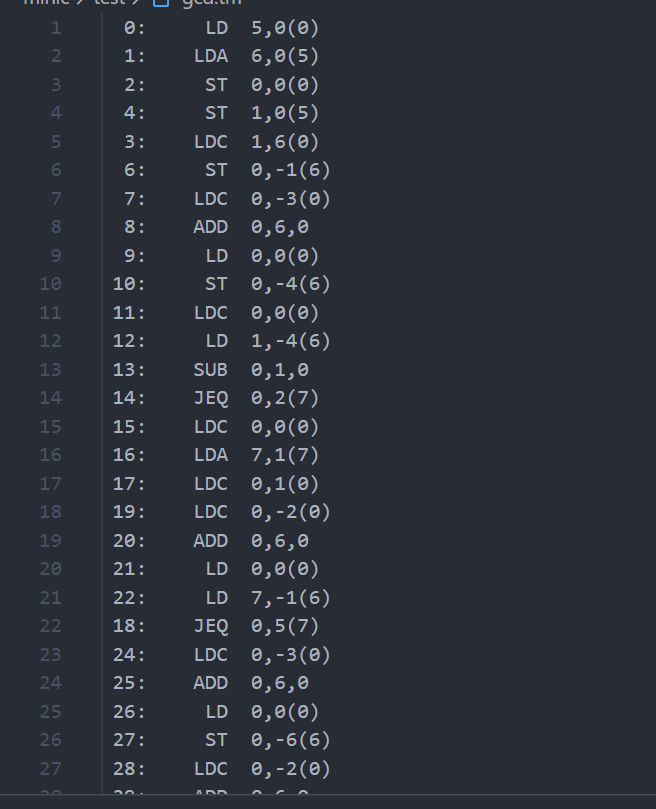


4.使用第三部分的测试用例来测试程序,并显示或保存相应的语法树。

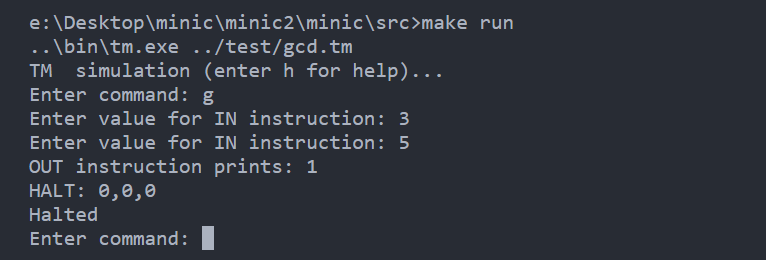


5.使用第三部分的测试用例来测试程序,并显示或保存相应的虚拟机目标代码

生成中间代码，保存为gcd.tm



6.并使用虚拟机解释执行得出程序的执行结果。



1. **实验总结**

梁泽浩同学的心得体会

此次项目加强了对编译原理各个阶段的理解,掌握了lex和bison工具的使用,能够熟练运用命令行工具, 学会了编写Makefile.

此外,后面开始全面使用C语言编写minic编译器代码并使用gcc+make编译, gdb调试,加深了对C语言的理解, 收获了很多以前不熟悉的知识点,例如:

* C语言不允许有同名函数
* C语言没有bool类型
* gdb调试带参数的程序的命令gdb --args program-name arg1 arg2 ...
* strchr(str1, ch)用于查找str1第一次出现ch时的位置
* strcmp(str1, str2)逐字符比较str1和str2如果相等则返回0
* strcpy(str1, str2)将str2复制到str1，返回str1
* strcat(str1, str2)将str2拼接到str1，返回str1
* C的无参函数声明的规范写法应该在括号中加入void关键字

在这个项目中，不断地出现有的没的bug，反复的调试改bug，调试能力有了很大提高。最欣喜的是通过这次的项目，学会熟练的使用flex和bison工具，相信在以后需要做语法分析相关的工作（比如做md解释器）会有很大帮助。

张勇标同学的心得体会

无疑，本次实验受益最大的方面便是在对于编译器编译的理解上。遥想当时初学编译原理的情形，一个最开始的词法分析部分都能让我焦头烂额，后面的内容更是使我云里雾里。

无法想象，今天，我竟然算是从头到尾地演练了一遍编译的过程。编译的虽然只是比tiny语言还要tiny的minic，但从词法到文法再到语义以及代码生成，每个环节无不耗费我大量的精力去理解和运用。过程固然是痛苦的，但这种难得的专注一事让我对编译原理有了更为清晰、甚至是从细节实现到抽象逻辑的更为广度的认知。

通过对flex和bison的运用，让我深刻地认识到对底层知识的吃透并对其进行封装运用，能给广大的开发者带来多大的遍历。同时，这也让我更加喜欢上了产生式和语法树。通过对构造符号表的理解、对类型检查的运用，让我深刻认识到了语义分析的重要性，语义分析做不好，后面的工作根本无法进行。而通过对代码生成的不断调试，让我深谙自己的代码能力需要不断加强，调试真的很想吐，但我也明白自己的代码或是理解能力正是在这份坚持中不断进步着。

通过这次实验，我挖掘出了许多编译原理的乐趣，也让我对编译器有了更深的情谊。我深知这只是编译原理一小部分的魅力，而其中的至宝，我仍需不断地去探索。

1. **参考文献**

编译原理与实践：

P14-P18,P52-P57,P97-P101,P136-P137,P141-P143,P257-P260,P296-P297,P346-P356

flex manual：

(<http://dinosaur.compilertools.net/flex/manpage.html>)

bison manual：

(<http://www.gnu.org/software/bison/manual/bison.html>)

Makefile教程（绝对经典，所有问题看这一篇足够了：<https://blog.csdn.net/weixin_38391755/article/details/80380786>

Qt Installer Framework 使用说明：<https://www.cnblogs.com/oloroso/p/6775220.html>

其他：

<https://github.com/isairz/cminus>

<https://blog.csdn.net/duke56/article/details/100187199>

<https://blog.csdn.net/cwj1412/article/details/104829148>

<https://blog.csdn.net/yyd19981117/article/details/89261108>

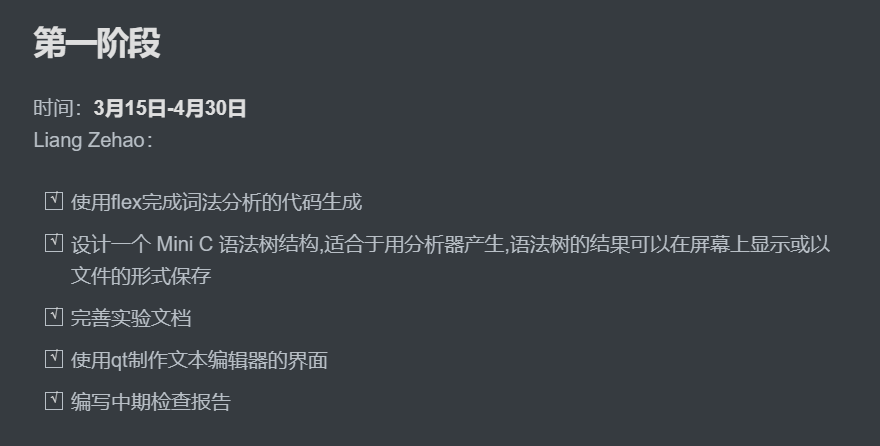
1. **项目以及各成员自评**

1．项目完成情况的自评分数以及原因说明

100：该项目已完成实验要求的全部内容，功能完善，在语义分析阶段调试了不少时间，整个项目重重复复修改了不少bug，目前拥有相当的稳健性。该项目使用了非常简洁且易懂的c语言，变量和函数的命名让人能够迅速理解其意，简洁而又附带注释的代码拥有相当的可读性。同时项目参考tiny源程序，使用了增量编程的思想，便于代码的增添、移植，以及分块运行。最后，该项目拥有简洁大方的界面，使用便捷舒适，体验感很好。gui美观大方，初步具备文本编辑器的功能，代码编辑器gui和minic命令行程序分离，使minic同时可以作为一般的命令行工具使用，文档注释规范，编程风格良好，设计思想清晰，使用钢鞭。

2.各成员自评分数以及原因说明

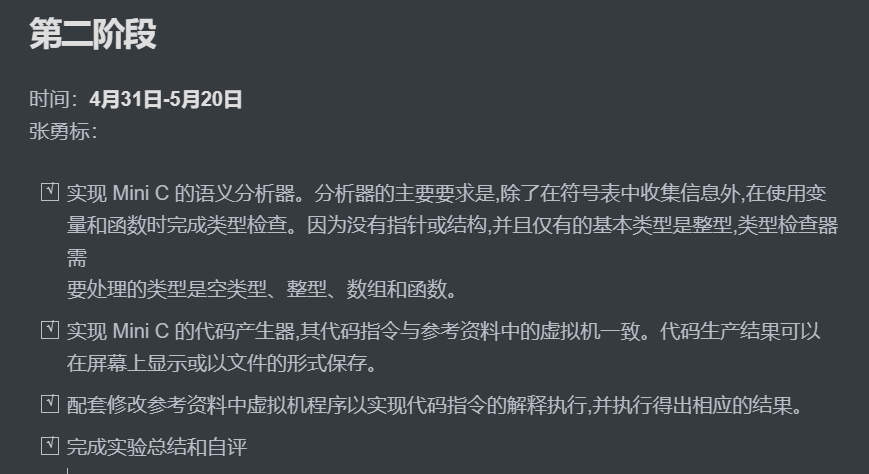
梁泽浩：95。需要完成的分工如下:



实际如期完成，设计了美观大方的界面，并完成了编辑器gui的文件打开新建，保存，另存为等功能，编译器程序路径已设置为可自定义。在语法树分析上花了较长时间，在项目结构管理和版本管理处上花了不少心思，也学习了make，flex，bison工具的使用等。本项目已经上传到github（<https://github.com/rattonlzh/minic>）上，本人目前贡献了43个commit，以下是我的git提交记录：



张勇标：93。需要完成的分工如下：



本人的任务主要集中在该实验的语义分析和代码生成上。目前，该项目的语义分析和代码生成功能运行顺畅，本人的任务圆满完成。本人在这两个功能的调试上着实花了非常多的时间和精力，任务能够正常完成绝非偶然。同时，队友在此项目上也付出了非常多的心血，他的努力我都看在眼里。在项目工作的贡献量上，队友占百分之五十以上、我占百分之四十以上。此外，本人与队友都会积极地沟通和讨论，确保项目的顺利推进，这点我认为是非常重要的。基于以上各点，我给自己评分93。

综上，该项目工作量的总体分布大致为：梁泽浩60%， 张勇标40%。