**课 程 设 计 报 告**

设计题目：一个面向对象类java语言的编译器

及其文本编辑器

班 级： 计算机1609

组长学号： 20162514

组长姓名： 窦洹彰

指导教师： 王会珍

设计时间： 2018.12

设计分工

组长学号及姓名：20162514 窦洹彰

分工：整体项目设计、语义分析、目标代码设计和生成、部分文法设计、汇编指令生成（虚拟机）、运行界面实现GUI、具有高亮功能的文本编辑器的实现、整体程序组织、符号表的重新设计、报错的整体设计、Token及AST数据结构优化、API编写（用于功能扩展如IO实现和MATH函数等）

组员1学号及姓名：20154397 张明阳

分工：文法的设计与完善、（部分语义分析）语法树（AST）的实现、LL1语法分析（用于检验文法）、递归下降法的实现、条件编译优化、语法错的处理

组员2学号及姓名：20154379 刘滋雨

分工：词法分析器、常数处理机、错误的接口以及实现

**摘 要**

编译原理是计算机科学与技术专业一门重要的专业课，它具有很强的理论性与实践性，目的是系统地向学生介绍编译系统的结构、工作原理以及编译程序各组成部分的设计原理和实现技术，在计算机本科教学中占有十分重要的地位。计算机语言之所以能由单一的机器语言发展到现今的数千种高级语言，就是因为有了编译技术。编译技术是计算机科学中发展得最迅速、最成熟的一个分支，它集中体现了计算机发展的成果与精华。

本课程设计实现了一个完整的仿java语言风格面向对象的编译器及其带有高亮功能等文本编辑器：文法支持多种声明方式、赋值、算术表达式、逻辑表达式、if/else语句、while语句、do-while语句、for语句、数组的声明，使用，整体引用、函数的声明和调用，类的声明和调用，类方法的声明和调用。编译器本身前后端功能完备，从编辑器输入，经词法分析（带双缓冲机制）、语法分析、语法树生成、优化直到后端目标代码的输出，具有报错功能（报类型和行号），并且能够通过可视化界面直观清晰地输出每一步的结果。

在实现方面，采用了自行编写的虚拟机，并且根据寄存器分配算法实现了多寄存器的虚拟机。通过目标代码产生的指令，可在虚拟机上实现IO、MATH函数，Sys函数等许多功能，因此可以实现类似如快速排序、GCD等递归调用的程序，或者更复杂的程序。

在可视化方面，将c++产生的编译器在java中通过进程运行，从而可以实现java标准的可视化界面，输出了所有的中间结果，并且在文本编辑器中实现了全局查找、替换、高亮等和词法分析相关的功能。

整个项目运行机制和java类似，先编译产生目标代码，然后在虚拟机上运行（在控制台），风格创新，有趣，界面友好，功能完善，运行正确。这次课程设计也给组员们很多的成长，为日后的项目经验奠定了良好的开端。

**关键词**： 前端 后端 类 数组 循环 可视化界面 高亮 文本编辑器 虚拟机

目录

[1.概述 5](#_Toc533628034)

[2. 课程设计任务及要求 7](#_Toc533628035)

[2.1 设计任务 7](#_Toc533628036)

[2.2 设计要求 11](#_Toc533628037)

[3. 算法及数据结构 12](#_Toc533628038)

[3.1算法的总体思想（流程） 12](#_Toc533628039)

[3.2词法分析器模块 13](#_Toc533628040)

[3.2.1功能 13](#_Toc533628041)

[3.2.2数据结构 16](#_Toc533628042)

[3.2.3算法 18](#_Toc533628043)

[3.3 语法分析模块 21](#_Toc533628044)

[递归下降语法分析 21](#_Toc533628045)

[3.3.1.1功能 21](#_Toc533628046)

[3.3.1.2数据结构 22](#_Toc533628047)

[3.3.1.3算法 26](#_Toc533628048)

[2）LL1语法分析 41](#_Toc533628049)

[3.3.2.1 功能 41](#_Toc533628050)

[3.3.2.2 数据结构 42](#_Toc533628051)

[3.3.2.3 算法 43](#_Toc533628052)

[3.4 中间代码产生器及错误判断模块 43](#_Toc533628053)

[3.4.1 功能 43](#_Toc533628054)

[3.4.2 数据结构 44](#_Toc533628055)

[3.4.3 算法 46](#_Toc533628056)

[3.5 符号表模块 47](#_Toc533628057)

[3.5.1 功能 47](#_Toc533628058)

[3.5.2 数据结构 49](#_Toc533628059)

[3.5.3 算法 51](#_Toc533628060)

[3.6优化器模块 52](#_Toc533628061)

[3.6.2数据结构： 53](#_Toc533628062)

[3.6.3算法： 53](#_Toc533628063)

[3.7目标代码生成器模块 55](#_Toc533628064)

[3.7.1功能 55](#_Toc533628065)

[3.7.2数据结构 56](#_Toc533628066)

[3.7.3算法 57](#_Toc533628067)

[4. 程序设计与实现 63](#_Toc533628068)

[4.1 程序流程图 63](#_Toc533628069)

[4.2 程序说明 63](#_Toc533628070)

[4.3 实验结果 65](#_Toc533628071)

[5. 结论 72](#_Toc533628072)

[6. 参考文献 73](#_Toc533628073)

[7. 收获、体会和建议 73](#_Toc533628074)

# 1.概述

在计算机上执行一个高级语言程序其中的一种方式要分为两步：第一步，用一个编译程序把高级语言翻译成目标代码（相对低级）；第二步，运行所得的机器语言程序求得计算结果。而我们此次采用的主要是：第一步，通过将源代码输入到编译器中产生目标代码；第二步，将目标代码在指令依赖的虚拟机上运行。

在学习《编译原理》课程过程中，我们逐渐掌握各章节构造编译程序的基本理论，并能独立完成词法分析器，语法分析器和语义分析器实验，在基本实验完成的基础上，编译器的前端实际上是基本搭建好的，我们需要做的就是课程设计中后端的设计和其他的扩充部分。

编译程序的工作过程一般可以划分为五个阶段：词法分析、语法分析、语义分析、中间代码产生、优化和目标代码生成。（课内中的样例）

第一阶段，词法分析。词法分析的实质是识别出来整个源程序中最小的词法单位。词法分析的任务是：输入源程序，对构成源程序的字符串进行分解和扫描，识别出一个个的单词或符号。一般需要设计词法分析器类的一个Vector容器存放Token序列，但是我们对数据结构实现了改进，实现双缓冲机制来存放Token使得可以快速访问和撤销访问，这种思想也在后期的文法编辑器高亮实现过程中发挥的巨大作用。其中包括在第几个表，表中偏移量和在源程序中的行数。每读出一个单词后判断其是否合法，然后判断他在符号表中的位置，将其加入，对于标识符变量Vector容器，如果新加元素已经存在，将新元素的指针指向原来出现的位置，保障表的非冗余性。并且设计好接口，为之后的语法分析中的调用来提供基础。

第二阶段，语法分析。语法分析的实质是识别出源程序中的每个语法单位。在词法分析的基础上，根据语言的语法规则，把单词符号串分解成各类语法单位。通过语法分析，确定整个输入串是否构成语法上正确的“程序”，实验中是输出yes or no，而课设中我们可以实现动态的错误处理（通过在token中增加row这个行号来记录发生错误的时候）。我们实现了语法分析器，能够使用递归下降分析法识别算术表达式，赋值语句，判断语句，循环语句，输出语句，数组结构、类、函数语法单位等。第三阶段，语义分析和中间代码产生。对语法分析所识别的各类语法范畴，分析其含义，并进行初步翻译(产生中间代码）。这一阶段包括两个方面的工作。首先，如果发生语法错误，则识别出错误类型并定位到错误发生的行号。如果语义正确，则依循语言的语义规则进行中间代码的翻译并生成相应的语法树。

第四阶段，优化，由于我们的中间代码的结构采用的是语法树，所以我们的优化工作主要在数据结构和条件编译上。由于我们实现了API功能（事先写好IO等模块供自己的程序使用），因此我们优化了条件编译，只编译用到的API。此外，我们还可以优化树的结构（平行结构），增加了next域，因此可以进行快速访问下一棵子树，避免了回溯带来的效率低下问题。

第五阶段，目标代码生成，把中间代码变换成特定机器上的低级语言代码，有赖于硬件系统结构和机器指令含义来实现最后的翻译。我们的编译器一套虚拟机环境下的汇编指令：可以对一个语法树中的信息进行深度的标注（类似于四元式的活跃标注），之后根据寄存器分配算法实现寄存器分配的优化，并维护了一个语义栈，用以实现循环和分支结构的跳转，从而实现了一套完整的，可以运行的目标代码。

第六阶段，实现一个文本编辑器和一个可视化界面，对用户更加友好地呈现出我们的结果，并且可以实现高亮、查找替换的功能，实现了课设内容的创新，因此我们对词法分析，语法分析有了更加宽广的应用，做到了学有所用。

通过对编译器的设计实现，一方面再次熟悉了java语言的编程方法及思想，并且重新认识了编程语言是如何执行的。另一方面加深了而对所学编译知识的掌握和理解，也深刻的理解了编译器的思想和实现方法；从词法分析到语法分析，再到语义分析，整个独立而又紧密联系的环节，紧紧相扣，整体的实现理解的更加透彻。并且可以做到了用现有的知识进行一系列实际的应用，丰富了学习兴趣，同时也拓展了自己的应用能力和创新能力。

# 2. 课程设计任务及要求

## 2.1 设计任务

**任务内容：**

1）定义一个简单程序设计语言文法（包括变量说明语句、算术运算表达式、赋值语句；扩展包括逻辑运算表达式、If语句、While语句、数组使用、类使用、类方法使用、函数的定义和调用等）；

2）扫描器设计实现；

3）语法分析器设计实现；

4）中间代码设计；

5) 翻译文法设计；

6) 符号表填写；

7) 错误判断及错误处理；

8）中间代码生成器设计实现；

9）中间代码优化；

10）生成目标代码。

11）实现界面

12）实现带高亮和查找功能的文本编辑器

13）目标代码的运行

14）…

**分析完任务内容，我们制定出一套满足老师要求的语句的文法结构，具体内容如下：**

program -> classlist

classlist -> classlist class

| class

class -> class ID { classVarDecList subroutineDecList }

classVarDecList -> classVarDecList classVarDec

|

classVarDec -> static type varNameList ;

| field type varNameList ;

varNameList -> varNameList , ID

| ID

type -> int

| float

| char

| boolean

| void

| ID

subroutineDecList -> subroutineDecList subroutineDec

|

subroutineDec -> constructor type ID ( params ) subroutineBody

| function type ID ( params ) subroutineBody

| method type ID (params ) subroutineBody

params -> paramList

|

paramList -> paramList , param

| param

param -> type ID

subroutineBody -> { varDecList statements }

varDecList -> varDecList varDec

|

varDec -> type varNameList ;

statements -> statements statement

|

statement -> assign\_statement

| if\_statement

| while\_statement

| return\_statement

| call\_statement ;

assign\_statement -> leftValue = expression ;

leftValue -> ID

| ID [ expression ]

if\_statement -> if ( expression ) statement

| if ( expression ) statement else statement

while\_statement -> while ( expression ) { statement }

return\_statement -> return ;

| return expression ;

call\_statement -> ID ( expressions )

| ID . ID ( expressions )

expressions -> expression\_list

|

expression\_list -> expression\_list , expression

| expression

expression -> expression & boolExpression

| expression | boolExpression

| boolExpression

boolExpression -> additive\_expression relational\_operator additive\_expression

| additive\_expression

relational\_operator -> <=

| >=

| ==

| <

| >

| !=

additive\_expression -> additive\_expression + term

| additive\_expression – term

| term

term -> term \* factor

| term / factor

| factor

factor -> - positive\_factor

| positive\_factor

positive\_factor -> ~ not\_factor

| not\_factor

not\_factor -> INT\_CONST

| CHAR\_CONST

| STRING\_CONST

| keywordConstant

| ID

| ID [ expression ]

| call\_expression

| ( expression )

keywordConstant -> true

| false

| null

| this

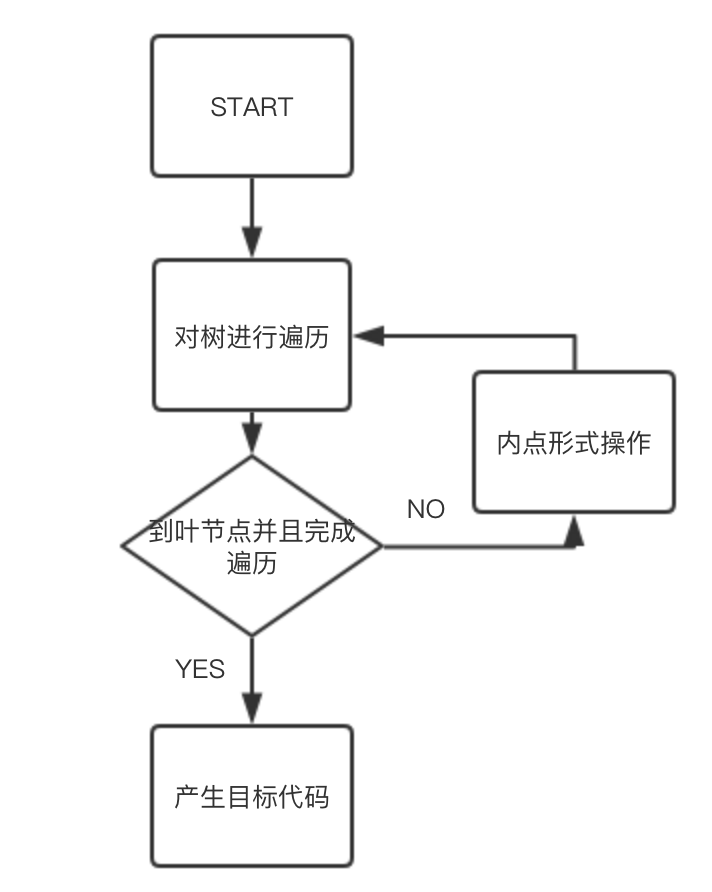
call\_expression -> ID ( expression )

| ID . ID ( expression )

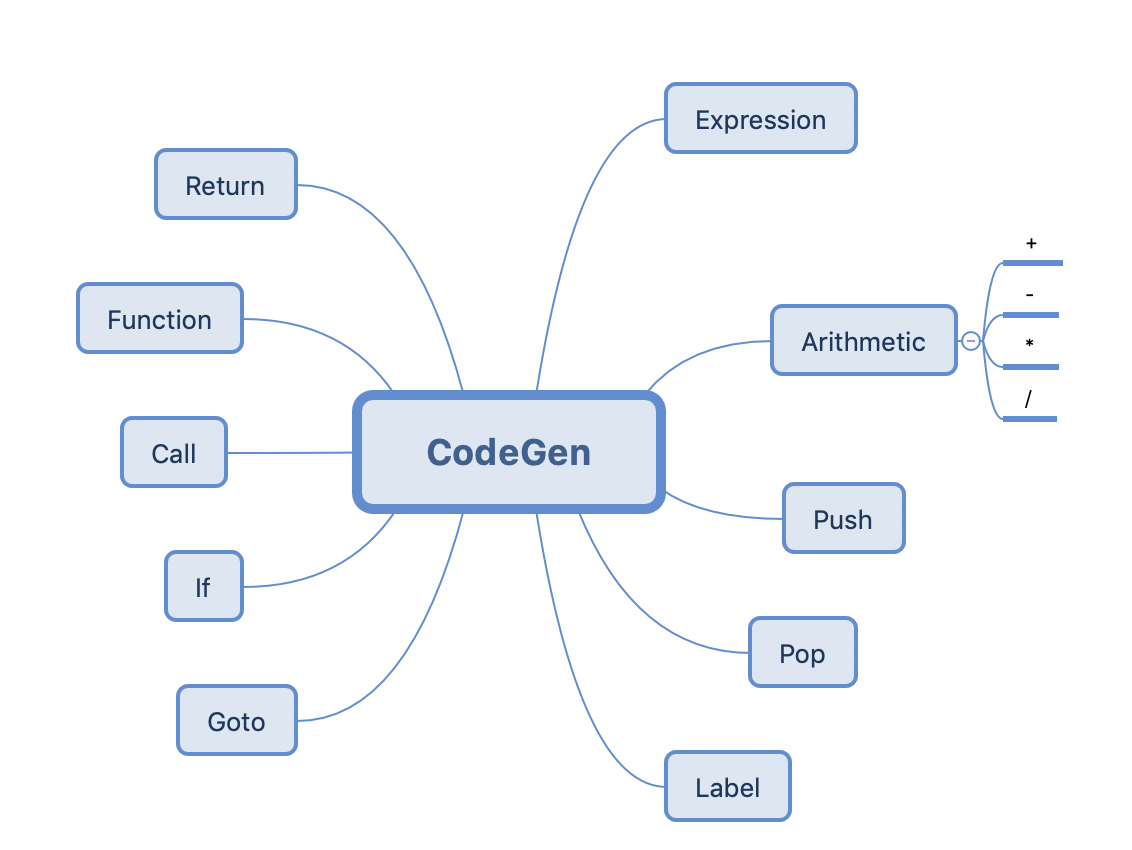
该文法可以识别1.2e3等指数型的数字以及数组。

**在此文法基础上，我们构造出了翻译文法，并在实现的过程中不断完善翻译文法。翻译文法内容如下**：

在翻译文法的设计上，由于我们的中间代码采用的是语法树的形式，因此在中间代码的产生时，实际上是涉及到树的遍历，和在内节点和叶子结点的操作。操作流程图如下：



每次产生一句目标代码实际上是操作了一个子函数，因此根据虚拟机的指令集，我们设计了如下的指令，并且设计了index值来确定当前信息的索引值，因为考虑到8086/8088在具体执行的时候有很大的限制，于是我们重新设计了指令集，以此来可以运行更加稳健和高级的功能，指令集如下图所示：

****

## 2.2 设计要求

1、在深入理解编译原理基本原理的基础上，对于选定的题目，以小组为单位，先确定设计方案；

2、设计系统的数据结构和程序结构，设计每个模块的处理流程。要求设计合理；

3、编程序实现系统，要求实现可视化的运行界面，界面应清楚地反映出系统的运行结果，并且实现文本编辑器，以此来挂载一个成熟的编译器。

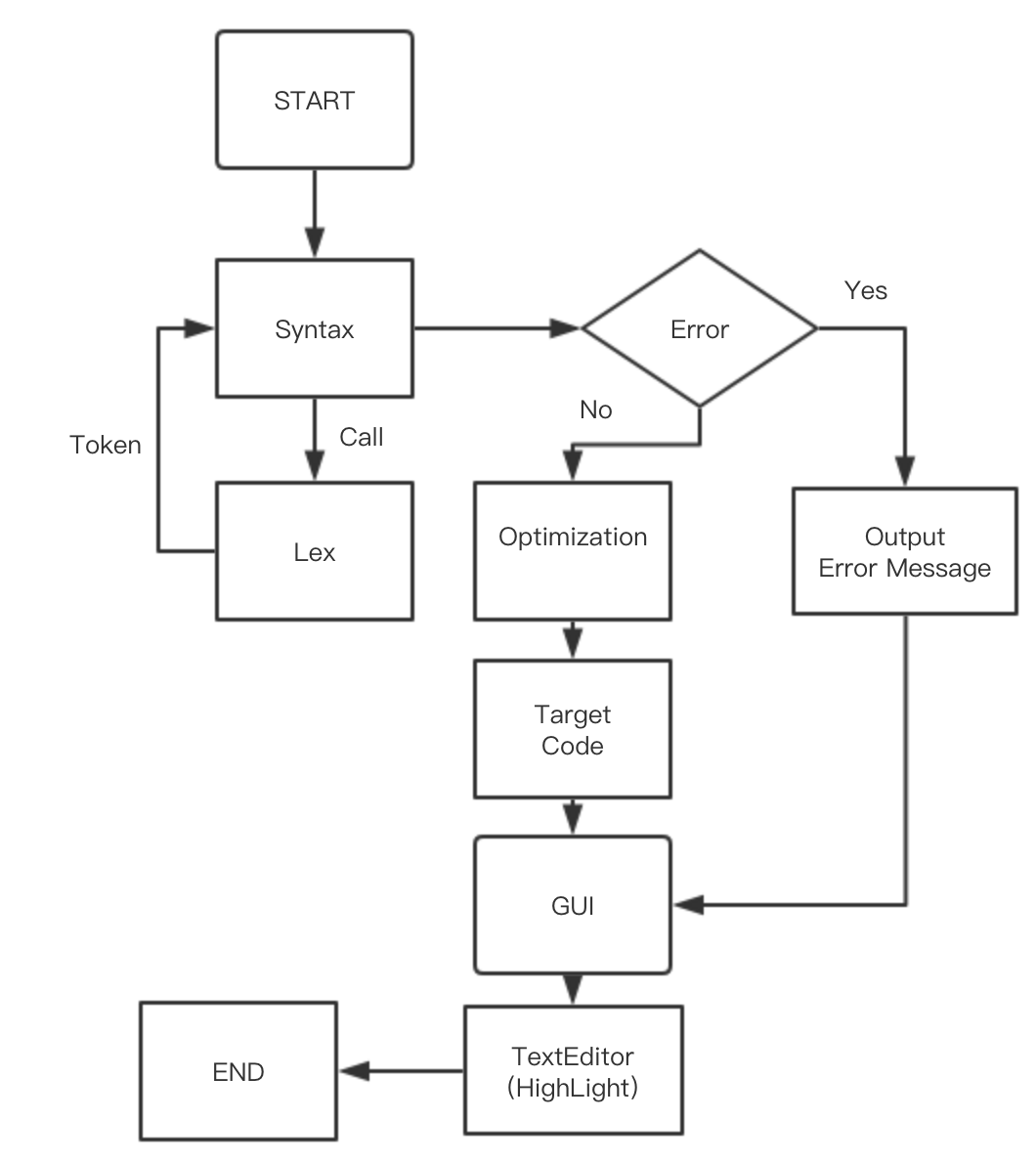
4、确定测试方案，选择测试用例，对系统进行测试；

5、运行系统并要通过验收，讲解运行结果，说明系统的特色和创新之处，并回答指导教师的提问；

6、提交课程设计报告。

# 3. 算法及数据结构

## 3.1算法的总体思想（流程）



本编译器的入口是语法分析器。对于一组用户输入的程序，语法分析器首先调用词法分析器，词法分析器切出一个token返回给语法分析器，语法分析器对词法分析器返回的token逐个进行分析，根据语法规则进行推导，识别出各类语法单位，最终判断输入串是否构成语法上正确的“程序”，同时填写符号表的相关表项。之后进行语义分析，按照语义规则对语法分析器推导出的语法单位（AST）进行语义分析并把它们翻译成一定形式的中间代码（AST），同时进行错误处理：语法错。如果发现错误则停止分析，获取错误的类型以及错误发生的行号，并对错误进行输出。若没有错误则完成中间代码生成，并将完整的中间代码和符号表传给优化器，开始进行优化处理。优化处理将目标代码分块，并对每一块的块内代码进行公共子表达式的提取，删除无用代码等优化。最后，将优化后的中间代码传给目标代码生成器，生成器对每个基本块内的代码标注活跃信息，并且通过优化后的四元式和符号表内的相关信息最终生成汇编代码。最后相关的结果在自行设计的GUI上进行运行，并且在自行设计的文本编辑器上可以用高亮的形式来显示。

## 3.2词法分析器模块

### 3.2.1功能

词法分析（英语：**lexical analysis**）是计算机科学中将字符序列转换为单词（Token）序列的过程。进行词法分析的程序或者函数叫作词法分析器（Lexical analyzer，简称Lexer），也叫扫描器（Scanner）。词法分析器一般以函数的形式存在，供语法分析器调用。 完成词法分析任务的程序称为词法分析程序或词法分析器或扫描器。 从左至右地对源程序进行扫描，按照语言的词法规则识别各类单词，并产生相应单词的属性字。词法分析程序作为编译器基础部分，其主要功能是识别单词并翻译单词。其中识别单词是将源程序中的单词切分出来，而翻译单词则是将切分出来的单词转化为token串，便于后续相关功能的实现。在本次课程设计中，我们将所识别的单词的语法功能分为以下种：

（1）关键字：程序语言定义的具有固定意义的标识符。有时又将这些标识符称为保留字或者基本字。如class 、constructor 、 function 、 method 、 field 、 static 、 int 、 if等。

（2）界符：分为单字符界符和双字符界符。

单字符界符：+ 、- 、\* 、/ 、> 、< 、 算术逻辑运算符以及{ 、 } 、 ， 、 ；等程序分隔符

双字符界符：>= 、<= 、== 、!= 、|| 、&&逻辑运算符

1. 标识符：用户给变量起的一些名字，用以标识各种名字，如变量名、数组名、函数名、字符常量以及字符串常量等。
2. 常数：程序中出现以自身形态面对用户和系统的数值，包括整数、小数以及指数。
3. 字符常量：用单引号括起来的单个普通字符或转义字符。
4. 字符串：用双引号双引号括起来的0个或者多个字符组成的序列。

在翻译单词的过程中，我们将token串定义为如下形式，使切分后单词长短统一、语法语义信息加以区分：

|  |  |  |
| --- | --- | --- |
| 类别kind | 值  token Value | 所在行位  row |

**其中类别包括：**

关键字：KEY\_WORD

标识符：ID

界符：SYMBOL

整数：INT

字符：CHAR

字符串：STRING

**整数处理：**

在START\_STATE开始状态下遇到[0-9]表示进入整数INT\_STATE状态再遇到[0-9]进入INT\_STATE状态循环，遇到其他合法字符进入整数识别结束状态。

**标识符与关键字处理：**

在START\_STATE开始状态下遇到\_ || [a-z] || [A-Z]表示进入标识符ID\_STATE状态再遇到\_ || [a-z] || [A-Z]进入标识符ID\_STATE状态循环，遇到其他合法字符进入标识符识别结束状态，最后调用searchReserved()函数实现是否为关键字判断，是关键字则修改token.kind = KEY\_WORD实现关键字与标识符的判断。

**字符处理：**

在START\_STATE开始状态下遇到一个 ’表示识别字符常量的开始进入CHAR\_STATE状态，再遇到非 \ 且非 ’字符时进入普通字符识别状态CHAR\_STATE\_A再遇到 ’进入DONE\_STATE字符识别结束状态表示普通字符识别完毕。在START\_STATE开始状态下遇到一个 ’表示识别字符常量的开始进入CHAR\_STATE状态，再遇到 \ 表示进入CHAR\_STATE\_B转义字符识别状态,在遇到转义字符a,b,n,r等进入CHAR\_STATE\_C转义字符识别将结束状态,再遇到 ’进入DONE\_STATE转义字符识别完毕状态.实现普通字符与转义字符的识别。

**字符串处理：**

在START\_STATE开始状态下遇到一个 “ 表示识别字符常量的开始进入STRING\_STATE状态，在遇到非 \ 且非 “进入STRING\_STATE不含转义字符字符串识别，遇到 \ 进入S\_STRING\_STATE含转义字符字符串再遇到任意字符进入STRING\_STATE状态，再遇到 ” 进入字符串识别结束DONE\_STATE状态。实现不含转义字符字符串与含转义字符字符串的识别。

**注释处理：**

在SYMBOL\_STATE界符状态下token.tokenValue = ‘/’&& ch==’\*’时进入INCOMMENT\_STATE注释状态，再遇到 \* 进入P\_INCOMMENT\_STATE注释将要结束状态在遇到 / 返回开始状态实现且过程中的识别序列不记录token中若遇到非 / 字符返回注释状态。实现注释识别。

**报错定位：**

所在行位置即为该单词在源程序中出现的行位置，在报错模块方便实现定位报错。

**主要函数：**

closeFile()实现文件关闭，

initKeyWord()实现关键字表的初始化，

initSymbols()实现界符运算符表初始化，

词法分析实现了Scanner接口，实现Scanner nextToken()，不断调用此函数就可以获得所有的记号流。记录所有token序列信息。

OpenFile()实现文件打开，

searchReserved()实现关键字判断

nextChar()实现获取下一字符。

Scanner()实现词法分析初始化操作。

至此词法分析构造基本结束。

### 3.2.2数据结构

对于给定的一种程序语言，其关键字以及界符都是系统设定的，为静态表，而标识符以及常数则会随着源程序动态变化，为动态表。由于我们采用了c++语言来编写该项目，因此我们可以使用c++中自带的STL进行数据结构的优化，比如使用了静态表考虑到效率就采用数组，动态表采用了set、map、vector等数据结构。考虑到后期需要对变量等标识符进行相应操作，故在词法分析阶段对符号表进行建立以及填写。具体如下：

对于给定的一种程序语言，其关键字以及界符都是系统设定的，为静态表，我采用set容器来建立动态表，而标识符以及常数则会随着源程序动态变化，为动态表。在符号表中建立与填写，这样我们易于动态拓展语言的功能，从而使得本编译器的最底层词法分析的可拓展性。

**静态表：（逻辑上的）**

**关键字表：**

set<string> keyWords;

void Scanner::initKeyWords() {  
 keyWords.insert("class");  
 keyWords.insert("constructor");  
 keyWords.insert("function");  
 keyWords.insert("method");  
 keyWords.insert("field");  
 keyWords.insert("static");  
 keyWords.insert("int");  
 keyWords.insert("char");  
 keyWords.insert("boolean");  
 keyWords.insert("void");  
 keyWords.insert("true");  
 keyWords.insert("false");  
 keyWords.insert("this");  
 keyWords.insert("if");  
 keyWords.insert("else");  
 keyWords.insert("while");  
 keyWords.insert("return");  
}//关键字表初始化操作

**界符运算符表：**

set<string> symbols;

void Scanner::initSymbols() {

symbols.insert("{");

symbols.insert("}");

symbols.insert("(");

symbols.insert(")");

symbols.insert("[");

symbols.insert("]");

symbols.insert(".");

symbols.insert(",");

symbols.insert(";");

symbols.insert("+");

symbols.insert("-");

symbols.insert("\*");

symbols.insert("/");

symbols.insert("&");

symbols.insert("|");

symbols.insert("~");

symbols.insert("<");

symbols.insert(">");

symbols.insert("=");

symbols.insert(">=");

symbols.insert("<=");

symbols.insert("==");

symbols.insert("!=");

}//界符表初始化操作

对于token，其相应的数据结构如下：

enum TokenType {

KEY\_WORD,

ID,

INT,

BOOL,

CHAR,

STRING,

SYMBOL,

FLOAT,

NONE,

ERROR, //The error to be outputed

ENDOFFILE //The end of the file

};

struct Token {

TokenType kind; // Token的类型

string tokenValue; // Token的值

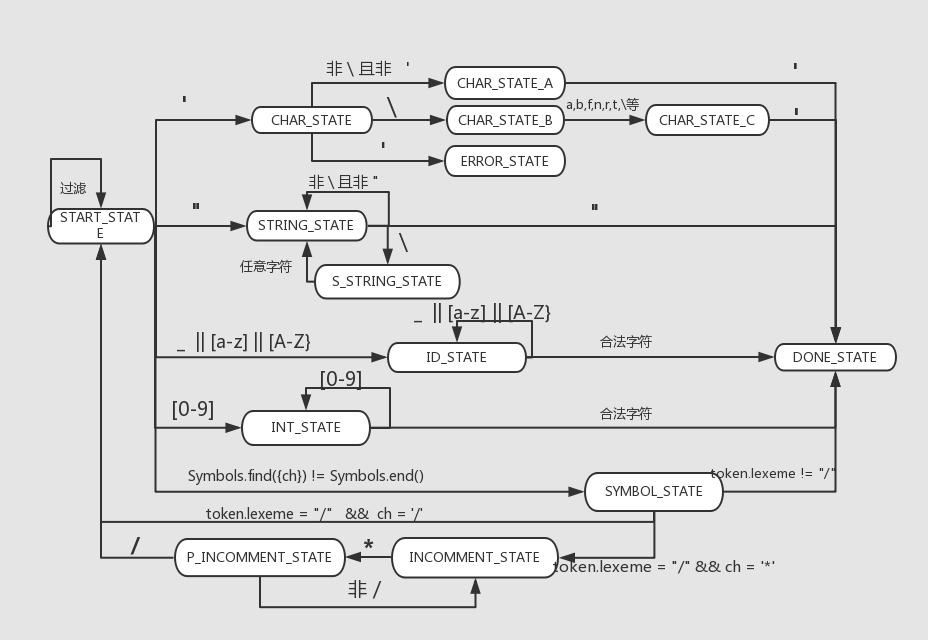
unsigned row; // 当前行

};

### 3.2.3算法

本词法分析程序基于有限自动机，其整体流程如下：

1、识别器(状态转换图)



状态转换图

其中：

过滤：代表空格、回车、换行的滤掉操作；

enum State // 转移图中的状态

{ START\_STATE, // 开始状态

ID\_STATE, // 标识符状态

INT\_STATE, // 整型数状态

CHAR\_STATE, // 字符状态

CHAR\_STATE\_A,

CHAR\_STATE\_B, // 转义字符状态

CHAR\_STATE\_C,

FLOAT\_STATE, // 浮点数状态

D\_FLOAT\_STATE, // 接近带小数点的浮点数状态

E\_FLOAT\_STATE, // 接近科学技术法的浮点数状态

STRING\_STATE, // 字符串状态

S\_STRING\_STATE, // 含有转义字符的字符串

SYMBOL\_STATE, // 界符状态

INCOMMENT\_STATE, // 注释状态

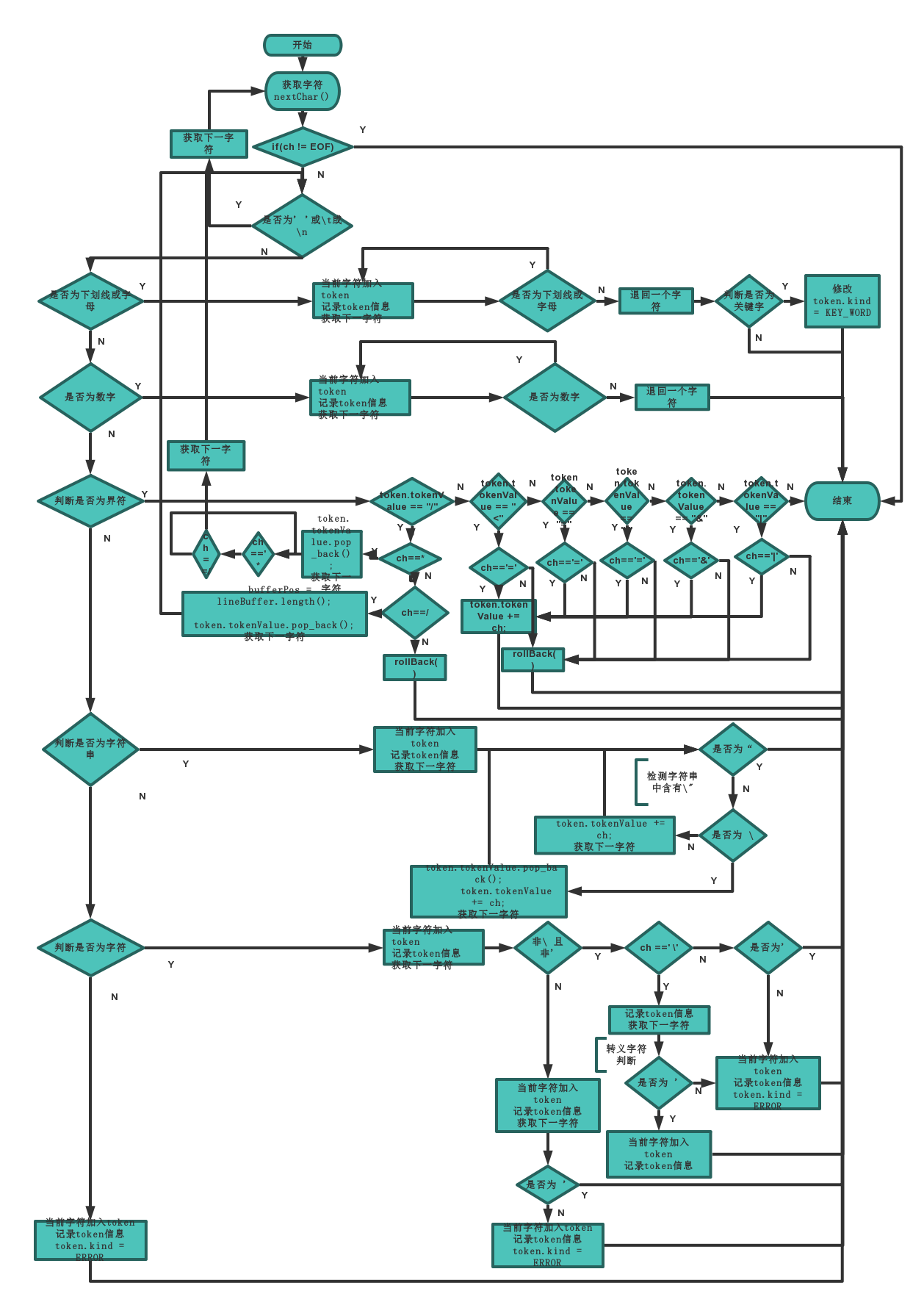
P\_INCOMMENT\_STATE, // 快要结束注释状态

DONE\_STATE, // 结束状态

ERROR\_STATE // 错误状态

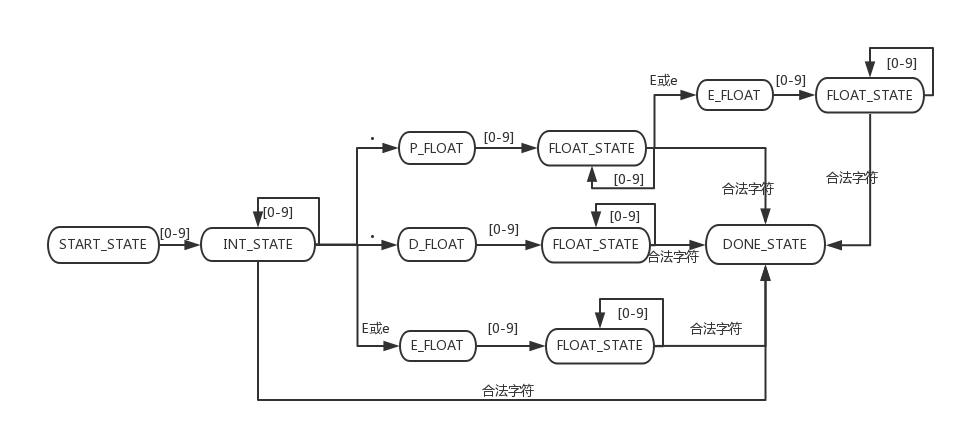
};

2、词法分析器



词法分析器

3、常数处理机



常数处理机

在此部分，在程序中并未实现，仅仅设计了常数处理机状态转换图。

## 3.3 语法分析模块

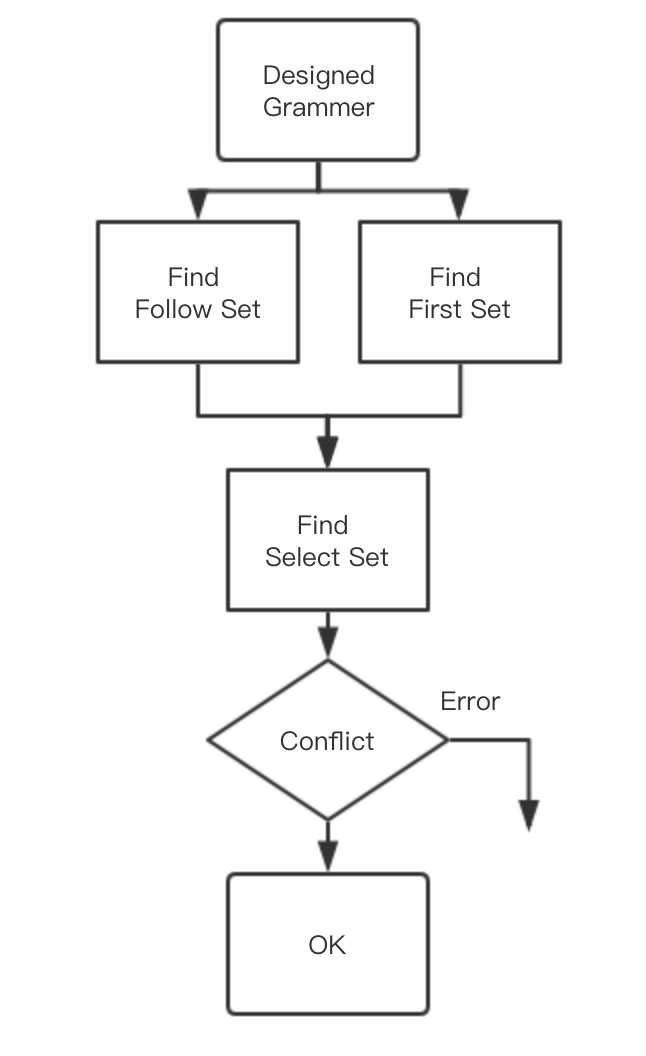
### 递归下降语法分析

### 3.3.1.1功能

语法分析是编译过程的核心部分。它的任务是在词法分析识别出单词符号串的基础上，分析并判定程序的语法结构是否符合语法规则。语法分析器在编译程序中的地位也是非常重要。语言的语法结构是用上下文无关文法描述的。因此，语法分析器的工作本质上就是按照文法的产生式，识别输入符号串是否为一个句子。按照语法分析树的建立方法，可以粗略的把语法分析方法分成两类，一类是自上而下的分析方法，另一类是自下而上的分析方法。在本次的课程设计中我们组使用的是自上而下的分析方法中的递归下降分析法。

其中递归下降语法分析的好处是，直观易懂，便于表示做递归和因子提取，对文法的要求比较低，可以通过文法变换的方法将其转换为递归下降的方式。通过递归下降的方法将文法翻译成语法制导的形式，将接收到的token进行扫描分析。根据我们组设计文法的特性（类java），并且在网上了解到java语言的编译过程，选择了构建语法树的方式进行语义分析，所以在语法分析的过程中即生成语法分析树，记录词的信息。我们实现了数组，类的定义，函数定义及调用，头文件等功能。同时在语法中出现的错误会报告出来，并返回空的节点。例如类名与函数名不匹配，缺少分号，大括号等问题。

其中这是我们语法分析的主干，我们做的额外工作是用LL(1)分析法来检验是否可以用递归子程序法来描述我们所设计的文法,以此来保证我们整个过程中的适用性和稳定性，保证了在文法层面我们不会导致程序出现过大的问题，从而出现莫名其妙的错误，流程图如下：



### 3.3.1.2数据结构

语法分析与语义分析为parser文件，通过读取待编译文件以及头文件来构建语法树，语法树的节点为TreeNode，限制节点类型NodeKind。

并且由于课内缺少了对于AST的讲述，因此我们在数据结构介绍之后会增加我们语法树的设计解析：

Private中为内部调用的递归子程序嵌套函数。Pubilc中为之后需要用到的接口，以及显示语法树。

class Parser {

public:

enum NodeKind {

None,

CLASS\_K,

CLASS\_VAR\_DEC\_K,

SUBROUTINE\_DEC\_K,

BASIC\_TYPE\_K,

CLASS\_TYPE\_K,

NULL\_K,

PARAM\_K,

VAR\_DEC\_K,

ARRAY\_K,

VAR\_K,

IF\_STATEMENT\_K,

WHILE\_STATEMENT\_K,

CALL\_EXPRESSION\_K,

RETURN\_STATEMENT\_K,

CALL\_STATEMENT\_K,

BOOL\_EXPRESSION\_K,

COMPARE\_K,

OPERATION\_K,

BOOL\_K,

ASSIGN\_K,

SUBROUTINE\_BODY\_K,

BOOL\_CONST\_K,

NEGATIVE\_K,

INT\_CONST\_K,

CHAR\_CONST\_K,

STRING\_CONST\_K,

THIS\_K,

FUNCTION\_CALL\_K,

CONSTRUCTOR\_CALL\_K,

METHOD\_CALL\_K,

KEY\_WORD\_CONST

};

class TreeNode {

public:

Scanner::Token token;

TreeNode \*child[5];

TreeNode \*next;

NodeKind nodeKind;

TreeNode() {

nodeKind = None;

child[0] = child[1] = child[2] = child[3] = child[4] = nullptr;

next = nullptr;

}

};

private:

vector<string> filenames;

string currentParserFilename;

TreeNode \*syntaxTree;

Scanner scanner;

bool haveReturnStatement; // 要保证每个函数都有return语句, 即使返回值为void

Scanner::Token getToken(); // 从缓冲区中取出一个token

Scanner::Token ungetToken(); // 把上一次取出的token放入到缓冲区中

deque<Scanner::Token> tokenBuffer1; // 左缓冲区

deque<Scanner::Token> tokenBuffer2; // 右缓冲区

string getFullName(string name); // 返回

TreeNode \*parseClassList();//类表

TreeNode \*parseClass();//类

TreeNode \*parseClassVarDecList();//类变量表

TreeNode \*parseClassVarDec();//类变量记录

TreeNode \*parseVarNameList();//变量表

TreeNode \*parseType();//类型判断

TreeNode \*parseSubroutineDecList();//子程序表

TreeNode \*parseSubroutinDec();//子程序

TreeNode \*parseParams();//参数识别

TreeNode \*parseParamList();//参数表

TreeNode \*parseParam();//参数记录

TreeNode \*parseSubroutineBody();//子程序主体

TreeNode \*parseVarDecList();//子程序变量表

TreeNode \*parseVarDec();//变量

TreeNode \*parseStatements();//状态判断

TreeNode \*parseStatement();//状态转换

TreeNode \*parseAssignStatement();//

TreeNode \*parseLeftValue();//等号左边

TreeNode \*parseIfStatement();//if

TreeNode \*parseWhileStatement();//while

TreeNode \*parseReturnStatement();//return

TreeNode \*parseCallStatement();//.()

TreeNode \*parseExpressions();//表达式识别

TreeNode \*parseExpressionList();//表达式表

TreeNode \*parseExpression();//表达式

TreeNode \*parseBoolExpression();//bool判断

TreeNode \*parseAdditiveExpression();//+ -

TreeNode \*parseTerm();//\* /

TreeNode \*parseFactor();// 负号

TreeNode \*parsePositiveFactor(); //~

TreeNode \*parseNotFactor(); //()等

TreeNode \*parseCallExpression();//.()

void printSyntaxTree(TreeNode \*tree);

public:

Parser(vector<string> &filenames);

bool hasError();

TreeNode \*getSyntaxTree();

void print();

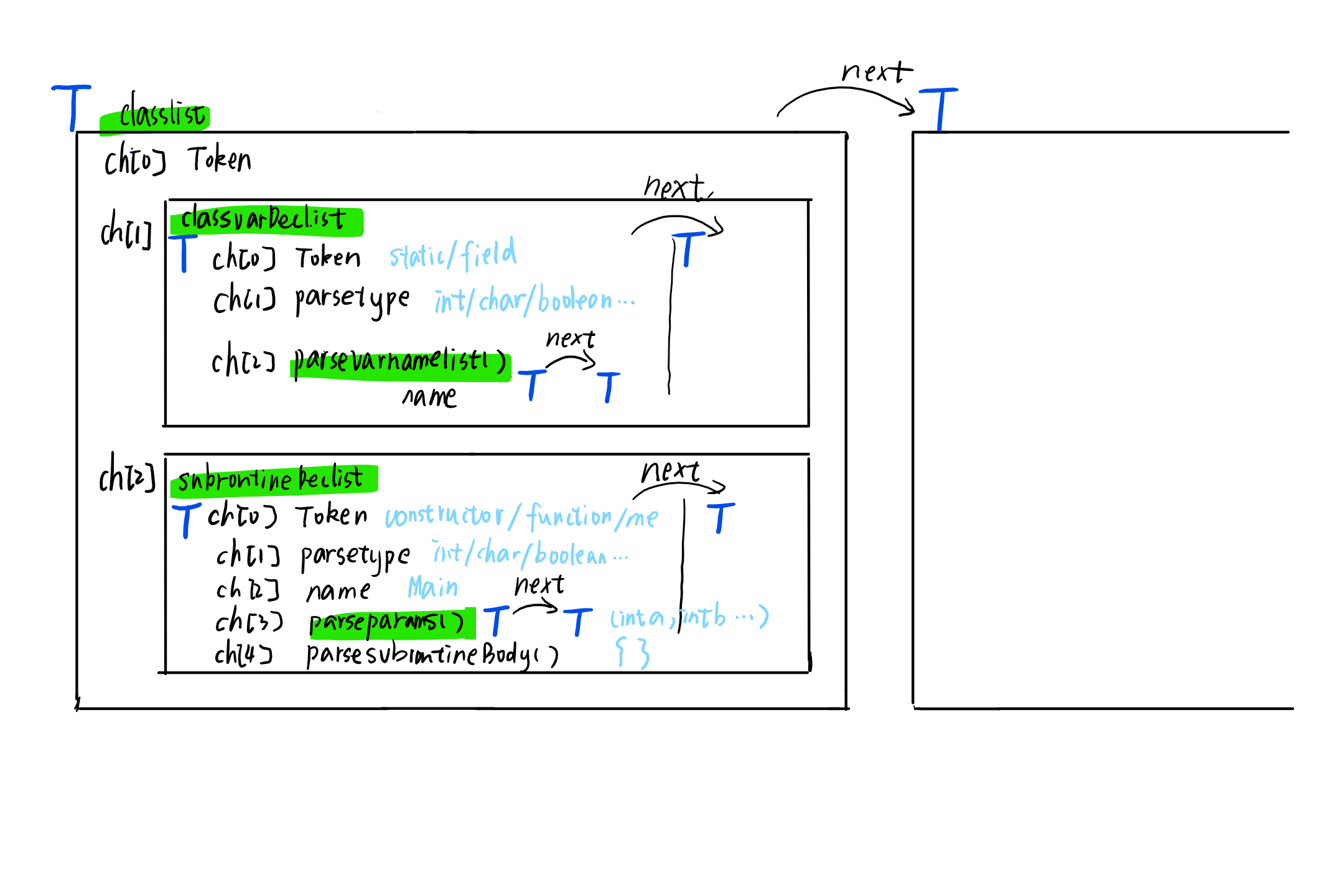
void parse\_program();

static string getCallerName(string const& fullName);

static string getFunctionName(string const &fullName);

};

**语法树设计详解（图示）**



注：兄弟节点用next相连接，便于访问，并减少了遍历的节点数。

### 3.3.1.3算法

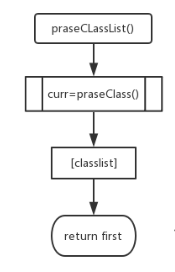
以下为本次课程设计中递归下降语法分析的程序流程图，其中包含语义动作，相关含义结合语义分析部分进行理解：

以下的描述结构为：先是文法产生式---然后是算法描述流程图

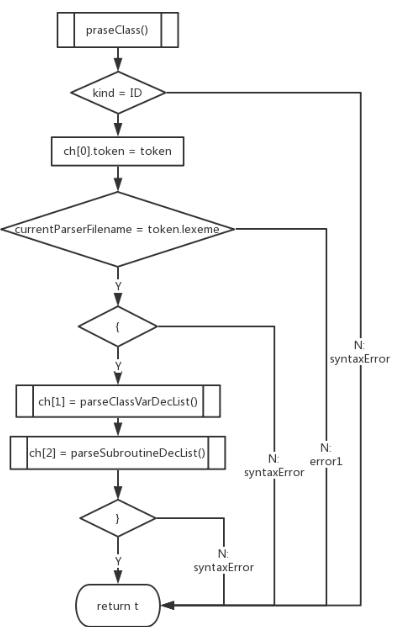
program -> classlist

classlist -> classlist class

| class

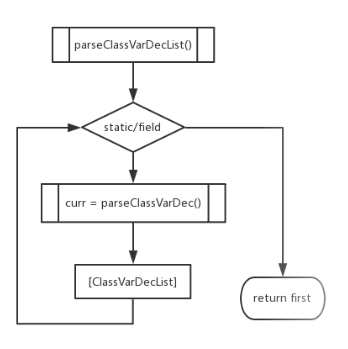


class -> class ID { classVarDecList subroutineDecList }



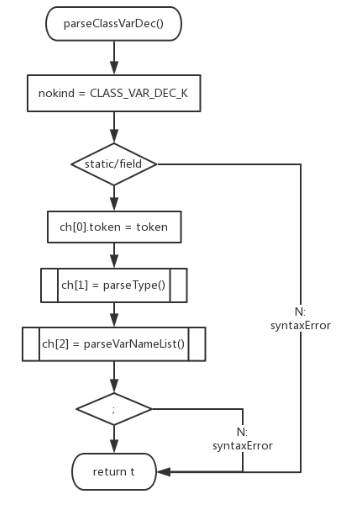
classVarDecList -> classVarDecList classVarDec

|



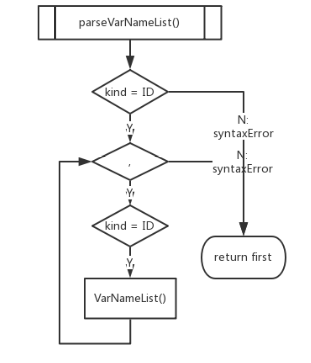
classVarDec -> static type varNameList ;

| field type varNameList ;



varNameList -> varNameList , ID

| ID



type -> int

| float

| char

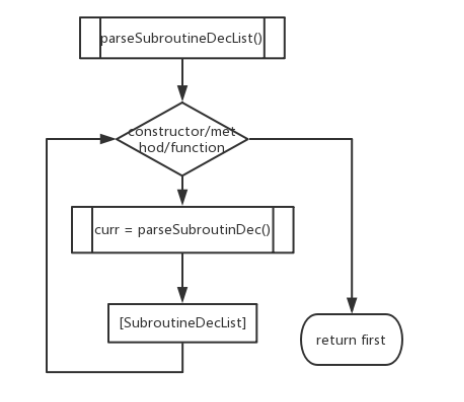
| boolean

| void

| ID

subroutineDecList -> subroutineDecList subroutineDec

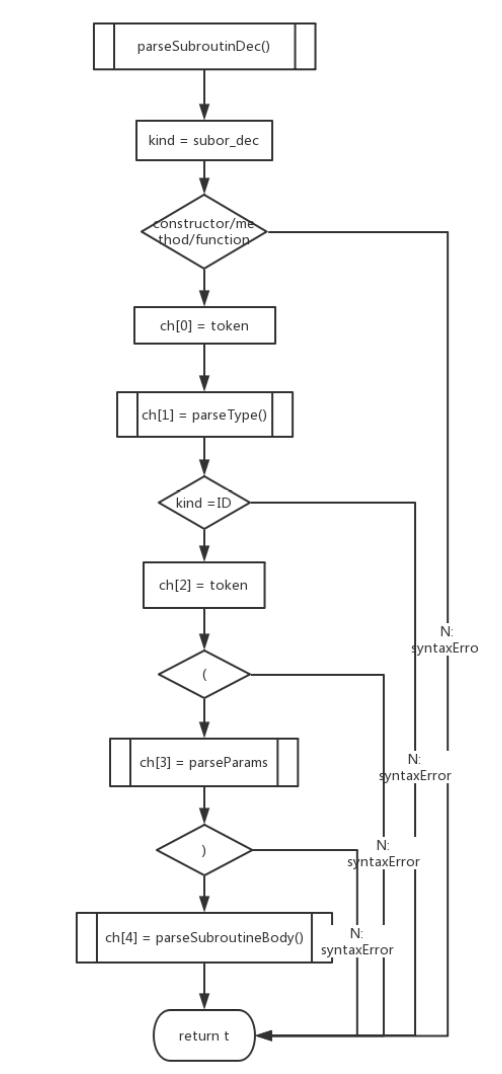
|



subroutineDec -> constructor type ID ( params ) subroutineBody

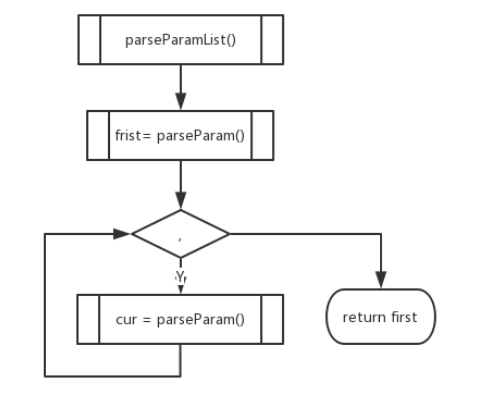
| function type ID ( params ) subroutineBody

| method type ID (params ) subroutineBody



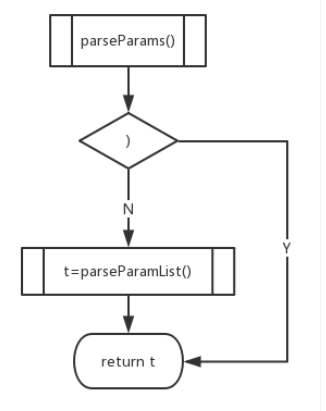
params -> paramList

|

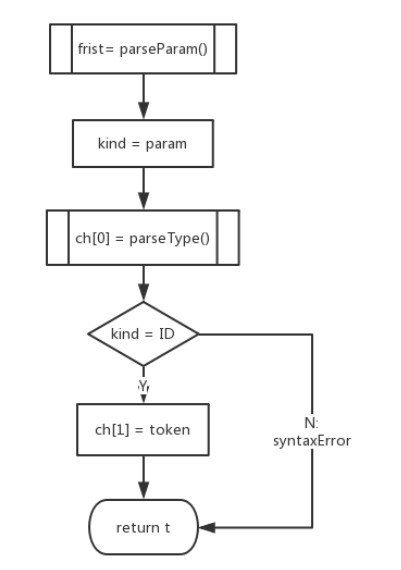


paramList -> paramList , param

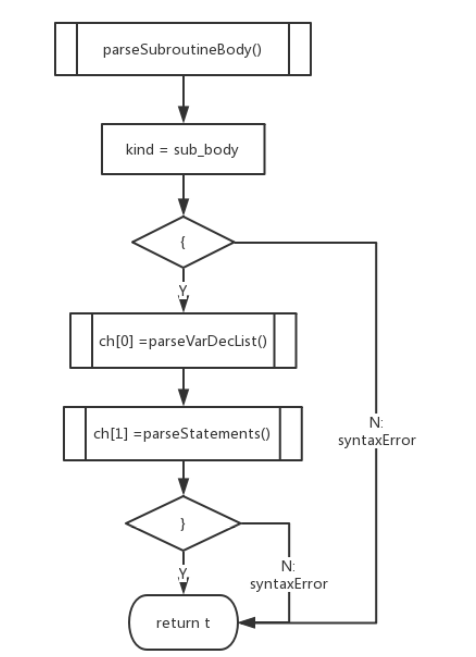
| param



param -> type ID

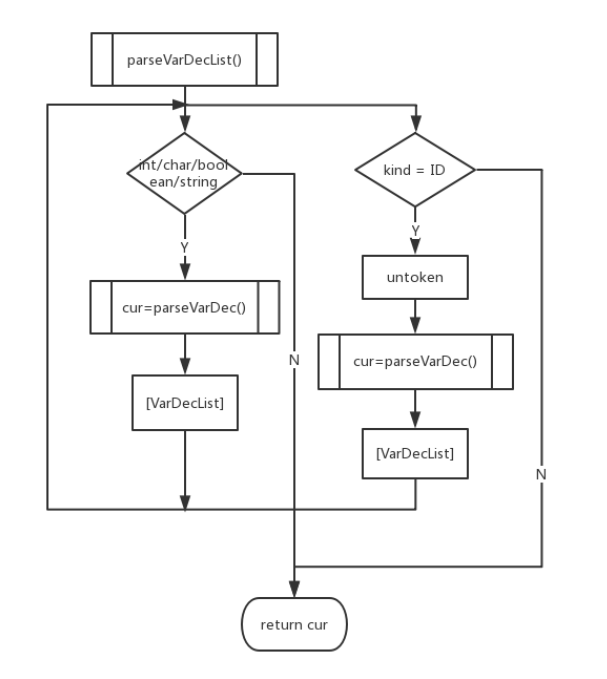


subroutineBody -> { varDecList statements }

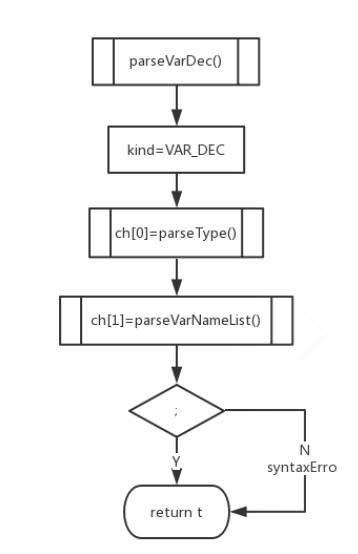


varDecList -> varDecList varDec

|

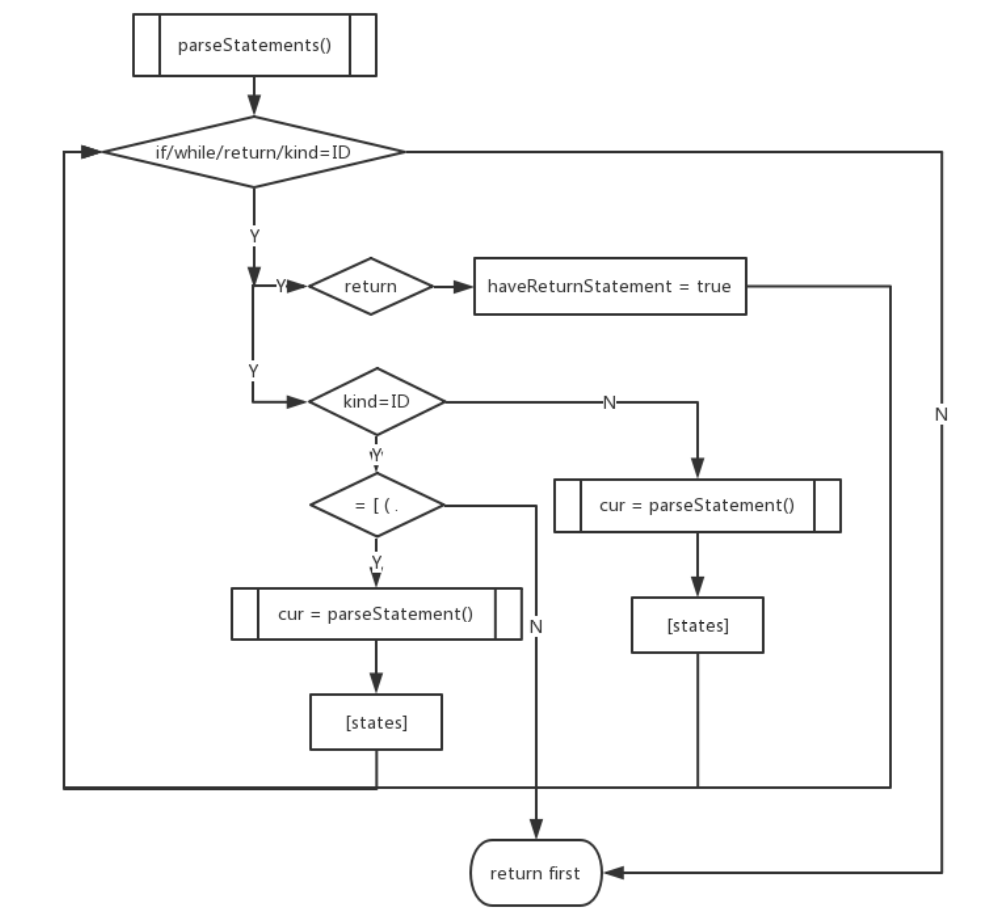


varDec -> type varNameList ;



statements -> statements statement

|



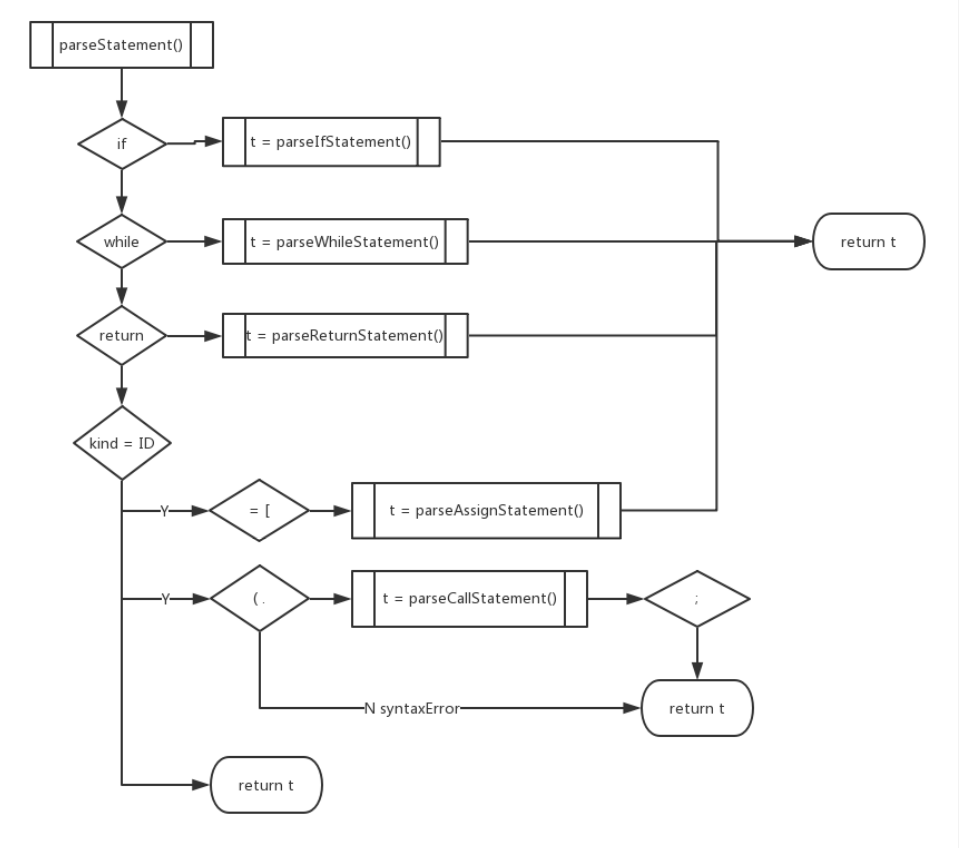
statement -> assign\_statement

| if\_statement

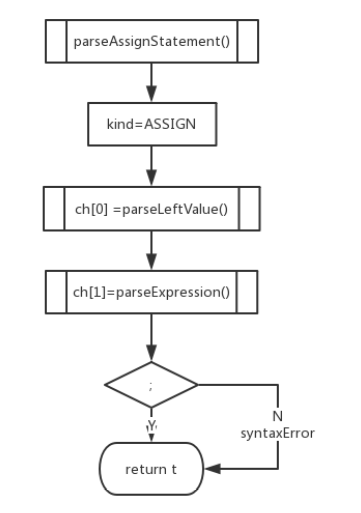
| while\_statement

| return\_statement

| call\_statement ;

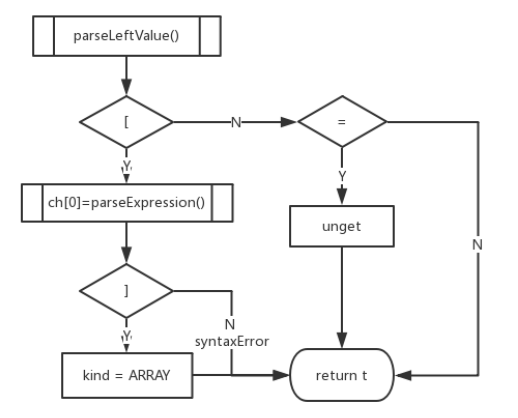


assign\_statement -> leftValue = expression ;



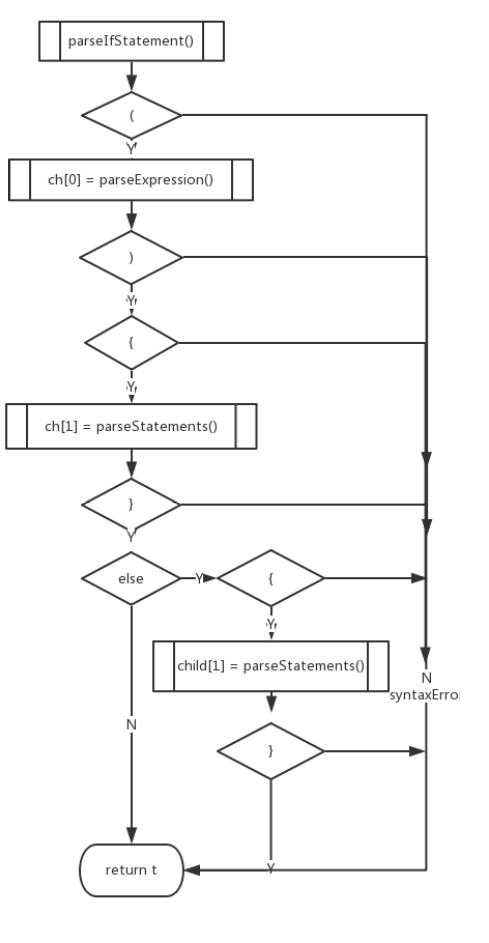
leftValue -> ID

| ID [ expression ]



if\_statement -> if ( expression ) statement

| if ( expression ) statement else statement



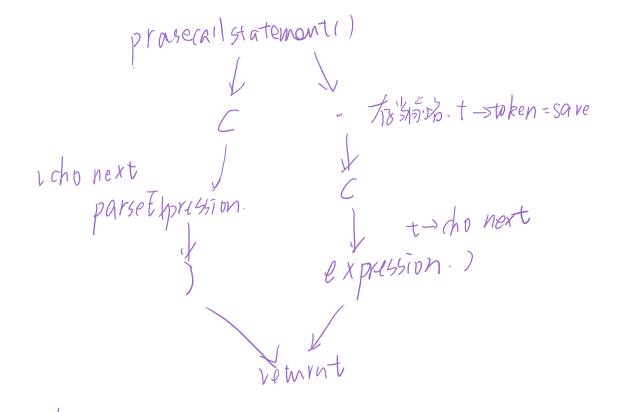
while\_statement -> while ( expression ) { statement }

return\_statement -> return ;

| return expression ;

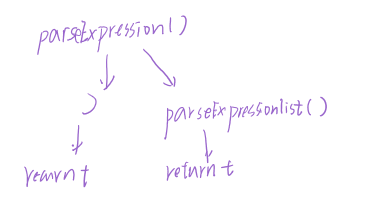
call\_statement -> ID ( expressions )

| ID . ID ( expressions )



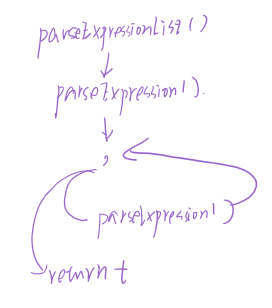
expressions -> expression\_list

|



expression\_list -> expression\_list , expression

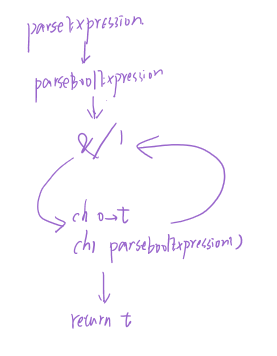
| expression



expression -> expression & boolExpression

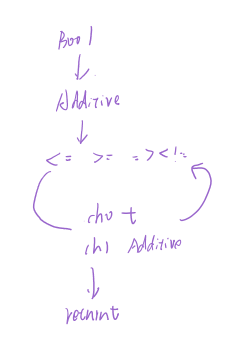
| expression | boolExpression

| boolExpression



boolExpression -> additive\_expression relational\_operator additive\_expression

| additive\_expression



relational\_operator -> <=

| >=

| ==

| <

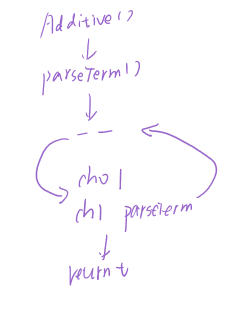
| >

| !=

additive\_expression -> additive\_expression + term

| additive\_expression – term

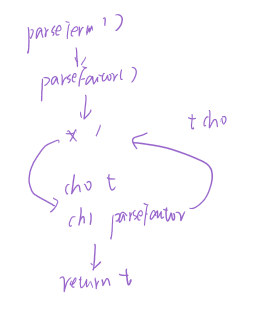
| term



term -> term \* factor

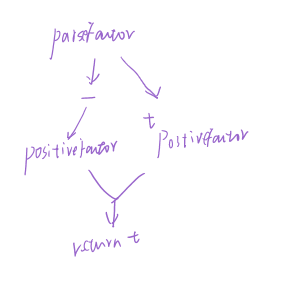
| term / factor

| factor



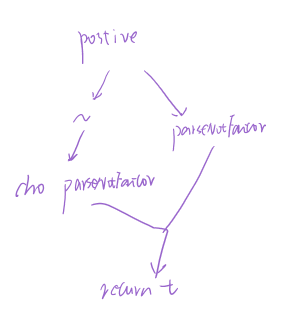
factor -> - positive\_factor

| positive\_factor



positive\_factor -> ~ not\_factor

| not\_factor



not\_factor -> INT\_CONST

| CHAR\_CONST

| STRING\_CONST

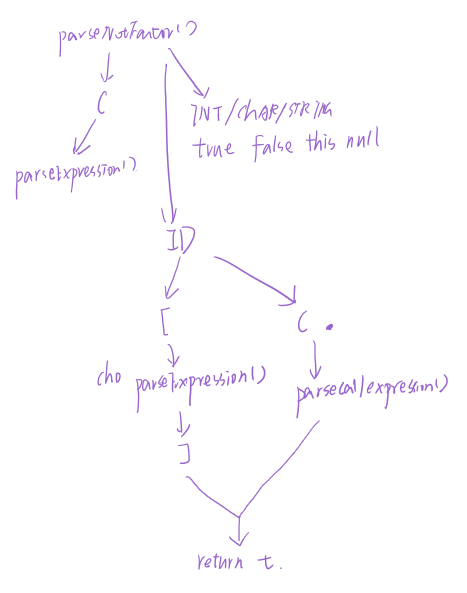
| keywordConstant

| ID

| ID [ expression ]

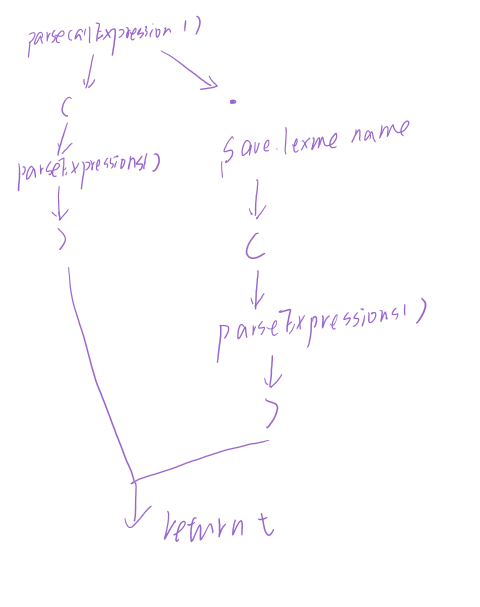
| call\_expression

| ( expression )



call\_expression -> ID ( expression )

| ID . ID ( expression )



### 2）LL1语法分析

### 3.3.2.1 功能

语法分析是编译过程的核心部分。它的任务是在词法分析识别出单词符号串的基础上，分析并判定程序的语法结构是否符合语法规则。语法分析器在编译程序中的地位也是非常重要。

语言的语法结构是用上下文无关文法描述的。因此，语法分析器的工作本质上就是按照文法的产生式，识别输入符号串是否为一个句子。

按照语法分析树的建立方法，可以粗略的把语法分析方法分成两类，一类是自上而下的分析方法，另一类是自下而上的分析方法。在本次的课程设计中我们采用了两种语法分析方法完成了两个语法分析器，我这里采用的是LL1分析方法。

我们的LL(1)分析只是用来检验文法，因为此方法在实际编程中没有很好的直观性，在充分听取老师意见以后，决定采用递归下降法作为骨架，而LL（1）分析方法仅作为检验。

### 3.3.2.2 数据结构

public class LL1 {

public static void main(String[] args) throws Exception {

// // LL（1）文法产生集合

ArrayList<String> gsArray = new ArrayList<String>();

// // Vn非终结符集合

// TreeSet<Character> nvSet = new TreeSet<Character>();

// // Vt终结符集合

// TreeSet<Character> ntSet = new TreeSet<Character>();

Gs gs = new Gs();

initGs(gsArray);

gs.setGsArray(gsArray);

// getNvNt(gsArray, gs.getNvSet(), gs.getNtSet());

gs.getNvNt();

gs.initExpressionMaps();

gs.getFirst();

// 设置开始符

gs.setS('E');

gs.getFollow();

gs.getSelect();

// 创建一个分析器

Analyzer analyzer = new Analyzer();

analyzer.setStartChar('E');

analyzer.setLl1Gs(gs);

analyzer.setStr("i+i\*i#");

analyzer.analyze();

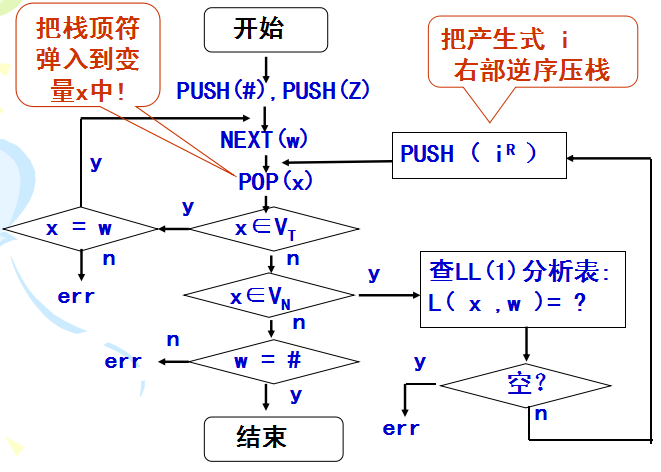
gs.genAnalyzeTable();

System.out.println("");

}

}

### 3.3.2.3 算法



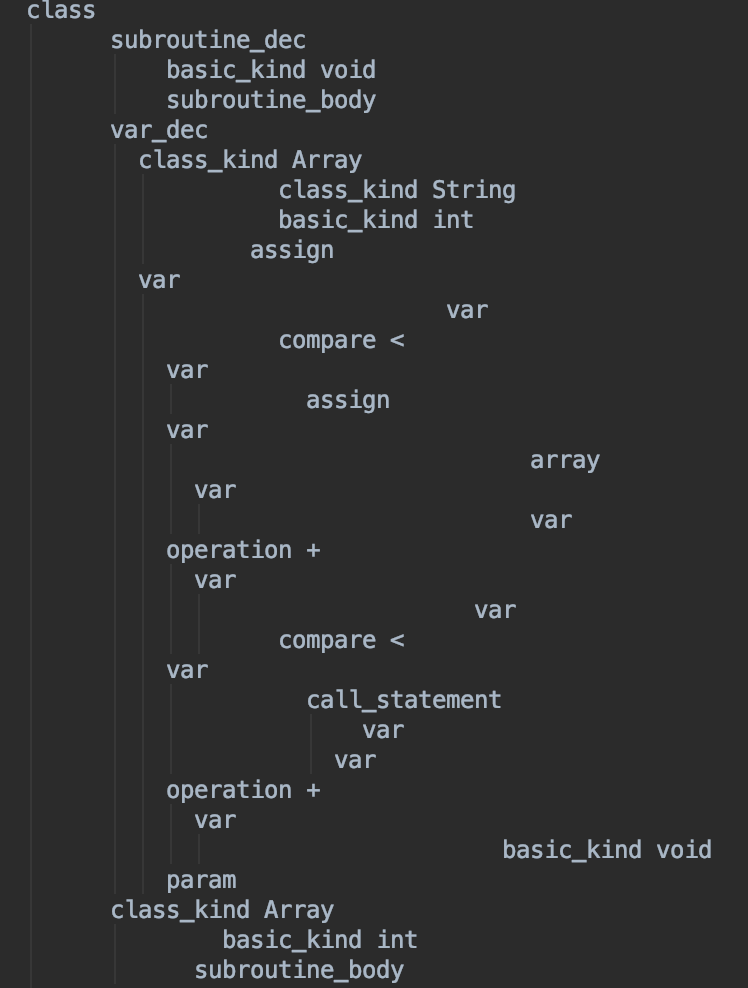
## 3.4 中间代码产生器及错误判断模块

### 3.4.1 功能

中间代码是高级程序语言中，各种语法成分的语义结构表示；它介于源语言和目标语言之间，是一种折中的体现形式，并且在实际生产中有着非常广泛的应用。

虽然源程序可以直接翻译为目标语言代码，但是许多编译程序却采用了独立于机器的复杂性介于源语言和机器语言之间的中间语言。这样做的好处是：便于进行与机器无关的代码优化工作；使编译程序改变目标机更容易；使编译程序的结构在逻辑上更为简单明确，以中间语言为界面，编译前端和后端的接口更清晰。

中间代码的形式有多种，本实验中采用的是语法树形式。因为语法树有着直观的特点，并且在计算机科学中有着广泛的应用，最重要的是，在优化中和数据结构中，甚至是活动记录和很多方有着很多革新，并且具有挑战性，对界面的输出也有很强的友好性。而我们最终输出的是一个语法树，按照树的深度缩进进行展示树。例子如下：即类中包含了函数、变量等。



### 3.4.2 数据结构

节点类型

enum NodeKind {

None,

CLASS\_K,

CLASS\_VAR\_DEC\_K,

SUBROUTINE\_DEC\_K,

BASIC\_TYPE\_K,

CLASS\_TYPE\_K,

NULL\_K,

PARAM\_K,

VAR\_DEC\_K,

ARRAY\_K,

VAR\_K,

IF\_STATEMENT\_K,

WHILE\_STATEMENT\_K,

CALL\_EXPRESSION\_K,

RETURN\_STATEMENT\_K,

CALL\_STATEMENT\_K,

BOOL\_EXPRESSION\_K,

FUNCTION\_CALL\_K,

CONSTRUCTOR\_CALL\_K,

COMPARE\_K,

OPERATION\_K,

BOOL\_K,

ASSIGN\_K,

SUBROUTINE\_BODY\_K,

BOOL\_CONST\_K,

NEGATIVE\_K,

METHOD\_CALL\_K,

INT\_CONST\_K,

CHAR\_CONST\_K,

STRING\_CONST\_K,

KEY\_WORD\_CONST,

THIS\_K

};

节点定义

class TreeNode {

public:

Scanner::Token token;

TreeNode \*child[5];

TreeNode \*next;

NodeKind nodeKind;

TreeNode() {

nodeKind = None;

child[0] = child[1] = child[2] = child[3] = child[4] = nullptr;

next = nullptr;

}

};

### 3.4.3 算法

在输出中间代码—语法树的时候，我们实际上可以采用对树遍历的形式，对树进行输出。而我们用缩进的方式来表示树的结构，相邻的父子节点缩进相差两个空格，并且是按照最左孩子先遍历，并且在设计语法树的时候，我们消除了二义性，可以得到一颗唯一的语法树。

**判错部分：**

输出语法树的时候，实际上是对语法树的遍历，因此可以访问到节点的信息-树的信息，也就是说可以检验目标是否有语法错，可以在访问每个节点中插入错误处理机制，从而来报得错误的信息和行号，从而实现错误中的语法错判定，构成错误处理中的一部分。

例如，声明语句声明的变量若已被填写了种类V，则此变量已定义过，则该变量为重定义；变量使用时若其种类未填写，则该变量未定义；赋值语句时若被赋值标识符类型与表达式运算结果类型不一致，则赋值语句类型不匹配；数组引用时将一数组整体赋值给另一数组时若被赋值数组长度小于另一数组则被赋值数组大小太小。以上各种错误情况输出相应错误信息，错误处理见error的接口。

错误类型：

extern int errorNum;

bool hasError();

void syntaxError(string curParserFile, string expected, Scanner::Token token);

/\*The types of Error\*/

void error1(string curParserFile);//class name isn't the same as function name

void error2(string curClass, int row, string type, string name);//Var re-definition

void error3(string curCLass,int row,string type,string name);//func re-definition

void error4(string curClassName,int row,string type);//type-non-definition

void error5(string curClassName,int row,string varName);//Var-non-definition

void error6(string curClassName,int row,string type);//no-matching for type

//non-definition for function

void error7(string curClassName,string callerName,int row,string functionName);

void error8(string curClassName,int row,string functionName);//not the same with function type

void error9(string curClassName,string callerName,int row,string functionName);//error-function type

void error10(string curClassName, string callerName, int row, string functionName);//error-function type

void error11(string curClassName, string type, int row);//error-return value

void error12(string curClassName, int row);//error-return value

void error13(string currentClassName, int row);//error-return value

void error14(string currentClassName, string functionName, int row);//too few args

void error15(string currentClassName, string functionName, int row);//too many args

void error16();//below are to be extended

void error17();

void error18();

void error19();

void error20();

## 3.5 符号表模块

### 3.5.1 功能

符号表是标识符的动态语义词典，属于编译中语义分析的知识库。

主要内容：

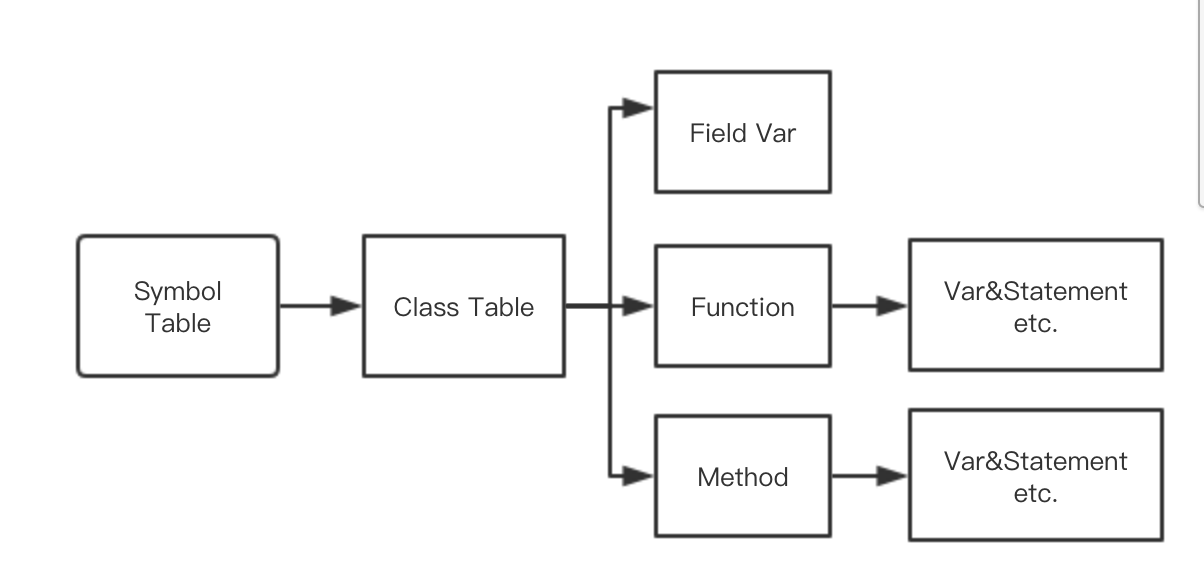
⑴ 名字 — 标识符源码，用作查询关键字；

⑵ 类型 -- 该标识符的数据类型及其相关信息；

⑶ 种类 -- 该标识符在源程序中的语义角色；

⑷ 信息 -- 与值单元相关的一些信息

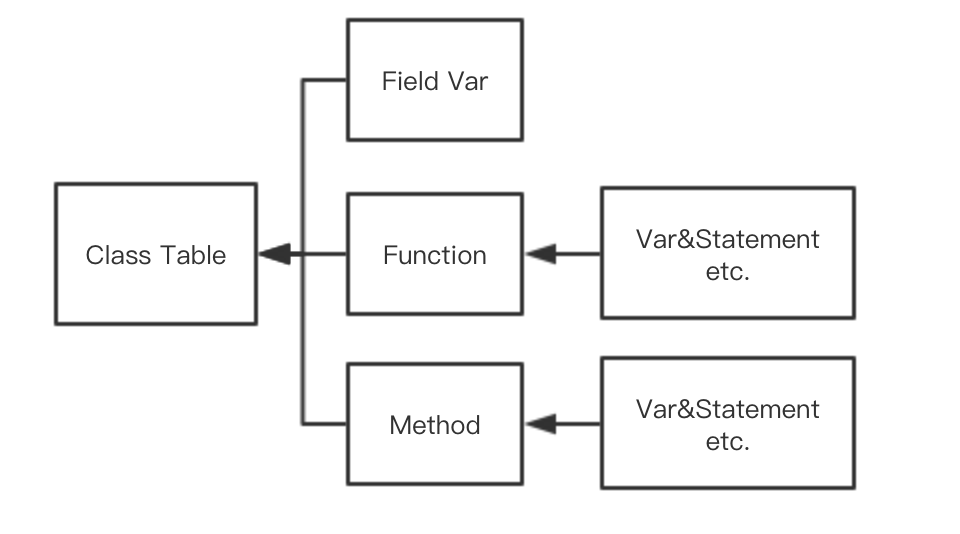
符号表的创新结构：



由于本项目使用了面向对象的语言，因此符号表的设计和课内讲的面向过程的符号表有些不同。此时的箭头代表了包含关系

我们的总表设计为了类表，存放出现在源代码中的所有的类，而在类表中由指针指出来其包含的成员变量，函数，类方法等。而类方法和函数中含有其自身的变量和需要用到的数据类型。

为了实现活动记录，我们全盘采用了指针的形式，即为每个变量设置当当前的functionName和ClassName，因此可以方便地在符号表中进行查询、更改、插入等操作。同时也增加一个nField变量，来记录每个块的深度，从而实现了每个块的基本区分。见下图，此时的回边代表的是指针的指向：



其功能包括：

① 定义和重定义检查；

② 类型匹配校验；

③ 数据的越界和溢出检查；

④ 值单元存储分配信息；

⑤ 函数、过程的参数传递与校验等

查表、插入等

### 3.5.2 数据结构

class SymbolTable {

public:

enum Kind { STATIC, FIELD, ARG, VAR, FUNCTION, METHOD, CONSTRUCTOR, NONE };

class Info {

public:

string type; // int, float, char, string

Kind kind; // kind : static, field, var, argument

int index;

vector<string> args;

Info() {

type = "0";

kind = NONE;

index = 0;

}

friend bool operator==(Info info1, Info info2) {

if (info1.type == info2.type && info1.kind == info2.kind &&

info1.args == info2.args)

return true;

else

return false;

}

};

static Info None;

private:

int static\_index;

int field\_index;

int arg\_index;

int var\_index;

int errorNum;

map<string, int> classIndex; // 从类名到数组索引

vector<map<string, Info>>

classesTable; // 类符号表数组, 将一直保留着不会被销毁

map<string, Info> subroutineTable; // 函数符号表

int currentClassNumber; // 遍历语法树的时候, 保存当前类符号表数组索引

string currentClass; // 遍历语法树的时候, 保存当前类名称

void initialSubroutineTable(); // 销毁函数符号表

SymbolTable();

static SymbolTable \*instance; // 指向符号表单实例对象

public:

static SymbolTable \*getInstance(); // 返回符号表单实例对象

void classesTableInsert(Parser::TreeNode \*t); // 类符号表的插入操作

void subroutineTableInsert(Parser::TreeNode \*t); // 函数符号表的插入操作

Info subroutineTableFind(string const& name); // 函数符号表的查找操作

Info classesTableFind(string const& className,

string const& functionName); // 类符号表的查找操作

bool classIndexFind(string const& className); // 判断className是不是合法的类名

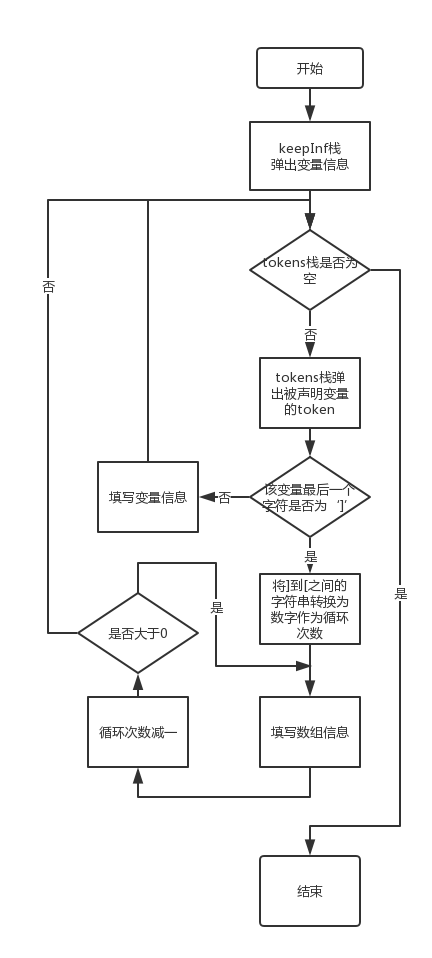
int getFieldNumber(string const& className);

void printClassesTable(); // 测试程序, 打印类符号表

};

### 3.5.3 算法

**1）填写变量：**



填写变量信息中，要时刻记得深度的嵌套信息，是类变量还是函数变了，和填写函数的思想是一样的。

**2）填写函数信息**

填写时，根据AST的遍历情况，不断对节点的类型进行判断，从函数名、返回值、参数列表等多方面进行判断，根据实际情况看情况是否type符合类型，如果符合类型，并且没有语法错，就填到函数表中

**3）填写类信息**

作为总的符号表，有着组织全局的重要作用，并且作为最大的识别单元，主要从修饰符，类名等方面进行不断遍历，然后进行递归地填表。直到当前类填写完毕，然后通过访问节点的next进行下一个类的填写，直到把源代码中所有的类信息填入表中。

## 3.6优化器模块

#### 3.6.1功能：

优化处理是指产生更高效的目标代码所做的工作。

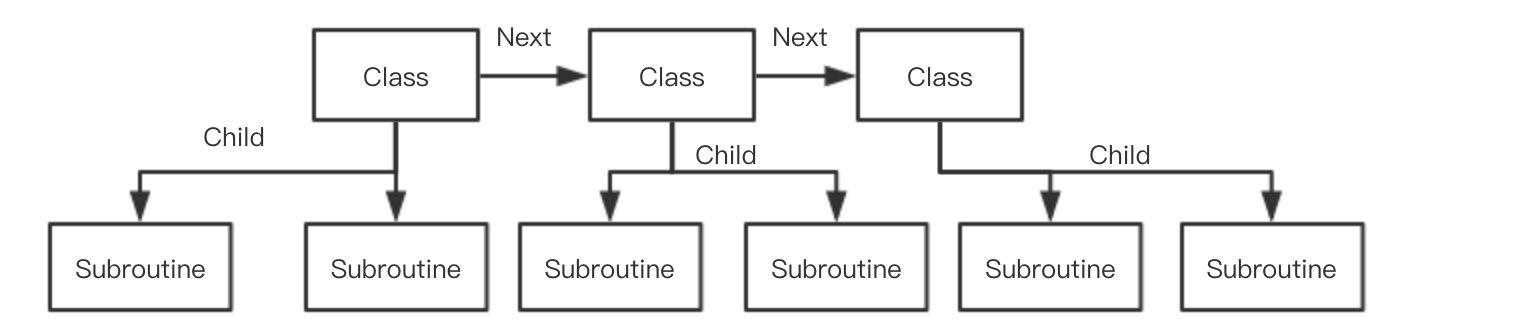
1. **API的优化**

我们进行了条件编译，用自身携带的条件判断，如果使用了目标的API的话

就进行编译，实现了动态的编译，因此可以大幅减少了目标代码的数量，实现了效果很卓越的优化

1. **数据结构的优化**

在树中增添一份next结构，由此可以指向树的平行结构，从而避免了回溯带来的效率低下的问题。



### 3.6.2数据结构：

class TreeNode {

public:

Scanner::Token token;

TreeNode \*child[5];

TreeNode \*next;

NodeKind nodeKind;

TreeNode() {

nodeKind = None;

child[0] = child[1] = child[2] = child[3] = child[4] = nullptr;

next = nullptr;

}

};

### 3.6.3算法：

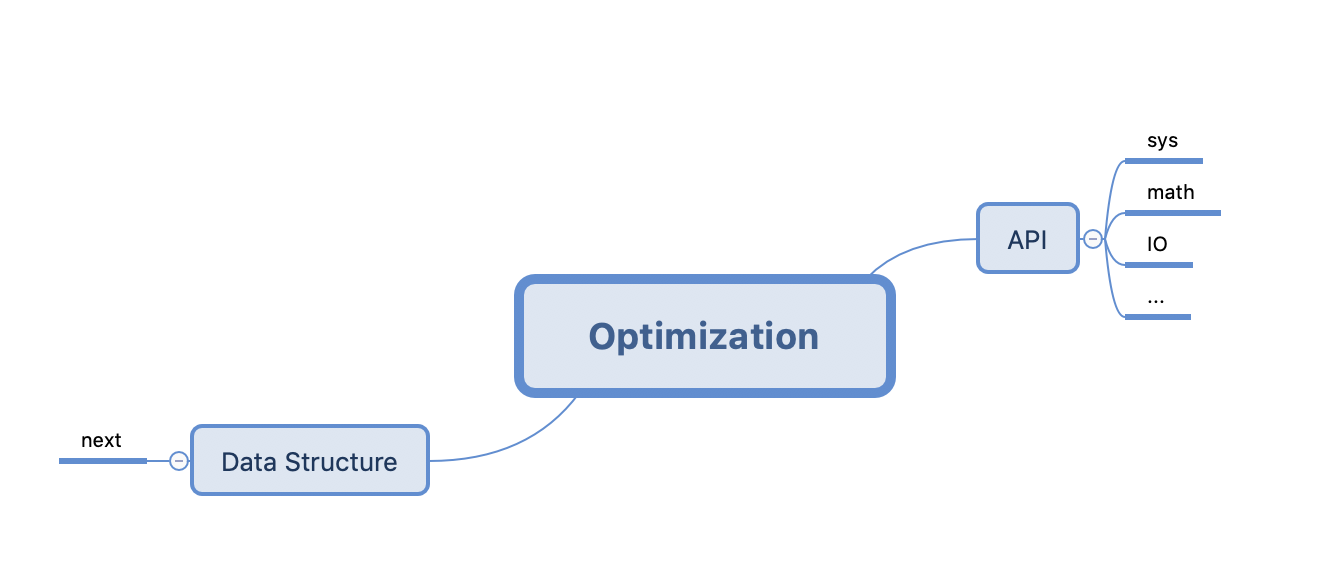
**（1）基本块划分算法：**

找出基本块的入口语句，它们是：程序的第一个语句或转向语句转移到的语句；紧跟在转向语句后面的语句。

对每一入口语句，构造其所属的基本块：从该入口语句到另一入口语句之间的语句序列；从该入口语句到另一转移语句（或停止语句）之间的语句序列

**（2）优化算法：**

优化算法主要体现在语法分析阶段，在语法分析的阶段可以进行条件编译，并且可以通过next可以进行高效访问节点，并且不需要单独再对树进行分析，减少了扫描的次数，效率更加高，可以减少程序的运行时间，并且可以减少目标代码的生成。并且应用了简单的常量合并等算法



**常量节省：**

一个基本块内，如果标识符为加减乘除，将左右节点进行相应运算，判断得出的值是否存在于符号表中，存在则查表，不存在则添加，将该值转换回string，存入子节点中。

**常值表达式节省：**

一个基本块中，判断左右节点和子节点的字符如果出现过，且该节点的标识符为等号，说明该字符被赋值过，查图，用它的值替换它本身，直到它再次被定义，将赋值语句注释掉。

**公共子表达式节省：**

一个基本块中，如果两次出现标识符、左右节点或子节点均相同的节点，证明两节点为有公共子表达式的节点，查图，在后者被再次定义前，用前者将其替换，并将后者节点注释掉。

**无用赋值节省：**

在一个基本块中，查图，是否存在对节点一个重新定义，且重新定义前从未使用过的节点，如果有，该节点即为无用赋值，将其注释掉。

**生成新的语法树：**

优化完成后，实际上在图形化界面中产生的也就是已经优化过的语法树，我们进行优化的结果在本文程序结果中可以看到目标代码量大大缩减，效果良好，并且我们的优化工作不是在课内学到的，而是在对AST充分理解后对其数据结构的优化，和对工作方式的优化，没有采取新的算法，但是也起到了非常不错的效果，正所谓“殊途同归”，虽然想法简单，但是有了实际的应用，让我们也十分地有成就感，

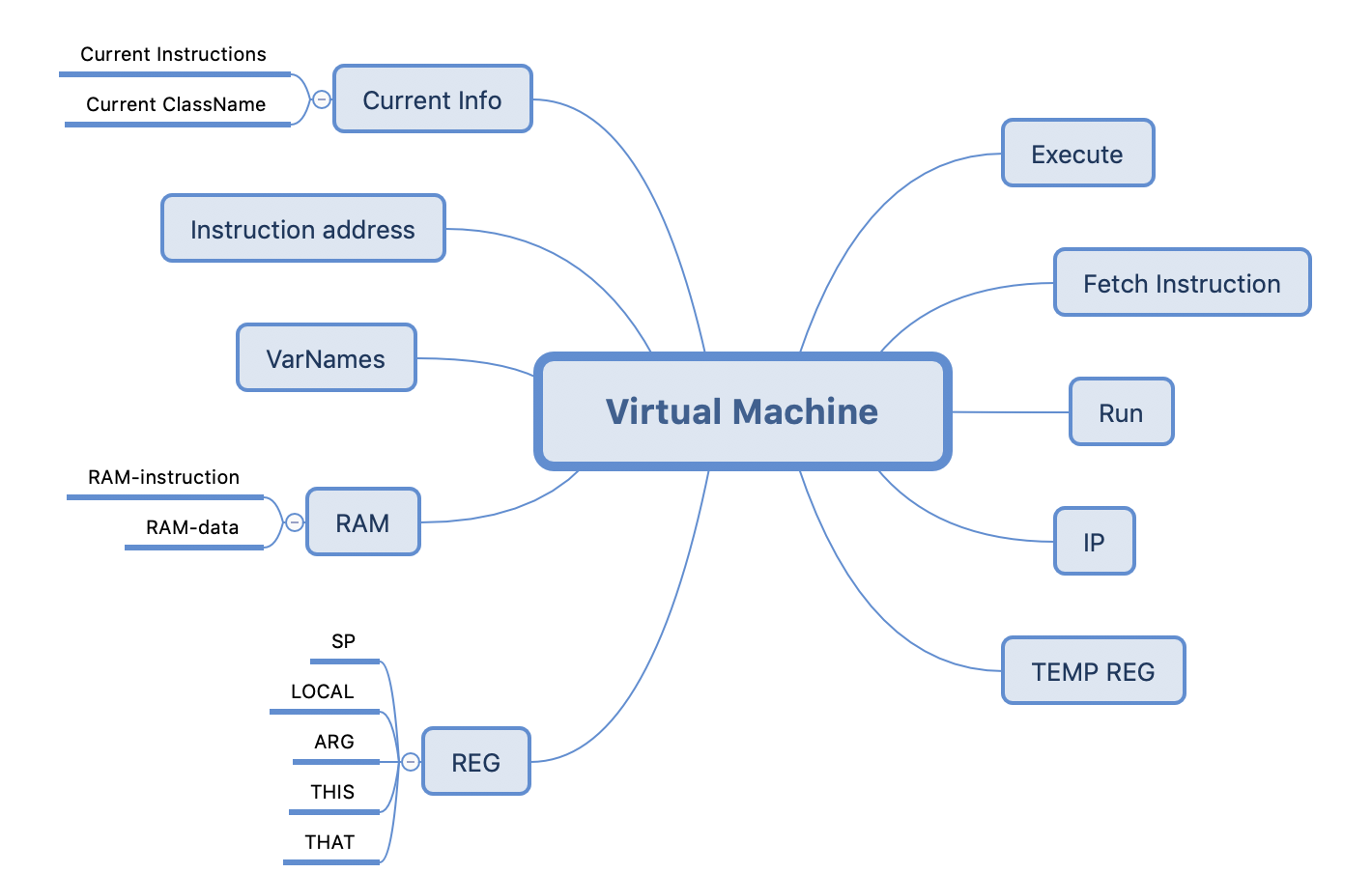
## 3.7目标代码生成器模块

### 3.7.1功能

编译模型的最后一个阶段是代码生成。它以源程序的中间代码作为输入，并产生等价的目标程序作为输出。代码生成器的输入包括中间代码和符号表的信息。代码生成是把语义分析后或优化后的中间代码换成目标代码。

在本次课程设计中，我们实现了自己的虚拟机，因此也需要设计新的指令集，然后根据这个为导向来考虑目标代码的生成。并且由于我们的虚拟机虚拟出来的是一段RAM作为存储空间，所以push和pop指令必须重点实现。

为短目标代码长度，减少访存次数，我们在虚拟机上使用目标代码时对寄存器分配做了优化。因此我们首先介绍我们的虚拟机设计：主要有取指令，执行指令。还有一系列的寄存器、一系列的RAM分配，和一系列需要用到的相关信息，如下图所示：



### 3.7.2数据结构

虚拟机部分：

static int sp; // REG 栈指针: 指向栈中下一个最顶的基址

static int local; // REG 指向当前VM函数local的基址

static int argument; // REG 指向当前VM函数argument段的基址

static int \_this; // REG 指向当前this段(在堆中)的基址

static int that; // REG 指向当前that段(在堆中)的基址

static int ip; // REG 指向下一条要执行的指令

static int temp[7]; // REG 存储临时值

/\*

\* RAM ADRRESS FUNCTION

\* 0~15 No use，to be uesd

\* 16~155 vm's static value

\* 256~2047 stack

\* 2048~16383 heap to store obj & array

\*\*/

static short ram[266385]; // 数据存储器

static vector<vector<string>> instructions\_ram; // 指令存储器

static int staticCount; // 记录静态变量已经分配的数量

static unordered\_map<string, int> staticVarNames; // 记录静态变量在内存中的位置 POINTER

static unordered\_map<string, int>

instruction\_address; // 保存label和function指令在指令存储器中的地址

static vector<string> currentInstruction; // 保存当前正在执行的指令

static string currentClassName; // 保存当前正在执行的指令所在的类的名字

static bool arriveEnd = false; // 标记是否到达程序结尾

生成目标代码部分：

class CodeGen {

private:

enum Segment { CONST, ARG, LOCAL, STATIC, THIS, THAT, POINTER, TEMP };

enum Command { ADD, SUB, NEG, EQ, GT, LT, AND, OR, NOT };

SymbolTable \*symbolTable;

Parser::TreeNode \*tree;

string currentClassName;

string currentFunctionName;

int ifLabelCount;

int whileLabelCount;

ofstream fout;

bool isMethod;

void writePush(Segment segment, int index);

void writePop(Segment segment, int index);

void writeArithmetic(Command command);

void writeLabel(string const& label);

void writeGoto(string const& label);

void writeIf(string const& label);

void writeCall(string const& name, int nArgs);

void writeFunction(string const& namFe, int nArgs);

void writeReturn();

void writeExpression(Parser::TreeNode \*t);

void writeCallExpression(Parser::TreeNode \*t);

void translateCall(Parser::TreeNode \*t);

public:

CodeGen();

void translate(Parser::TreeNode \*t);

void write(Parser::TreeNode \*t);

};

### 3.7.3算法

**活跃信息标注：**

首先取到一个基本块，将活跃信息表赋初值，用户定义变量的活跃信息为y，临时变量的活跃信息为n；之后从下至上地标注活跃信息。将活跃信息表中某个变量的活跃信息填到符号表对应变量的活跃信息之中，然后再更新活跃信息表中的信息，则活跃信息表中的活跃信息更新为n，AST中的子树中该变量参与运算，则活跃信息表中的用户变量更新为y。

**寄存器优化算法：**

1.主动释放：如果第一运算对象A已经在寄存器Ri中，则选择Ri；

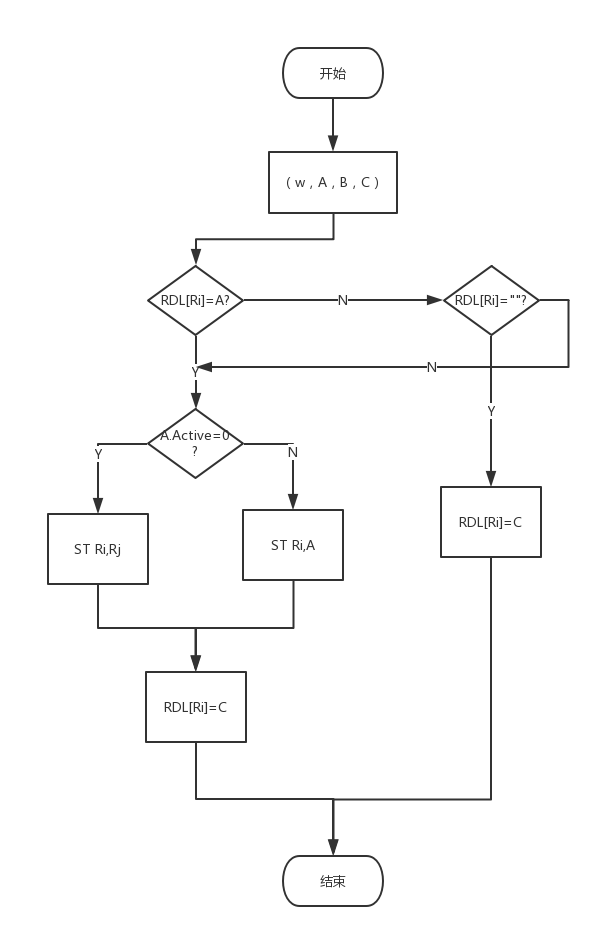
若A的Active=0，有空闲寄存器Rj时，则生成指令ST Ri，Rj，否则生成指令ST Ri，A。最后改写寄存器描述表；

2.选空闲者：从空闲的寄存器中选一Ri

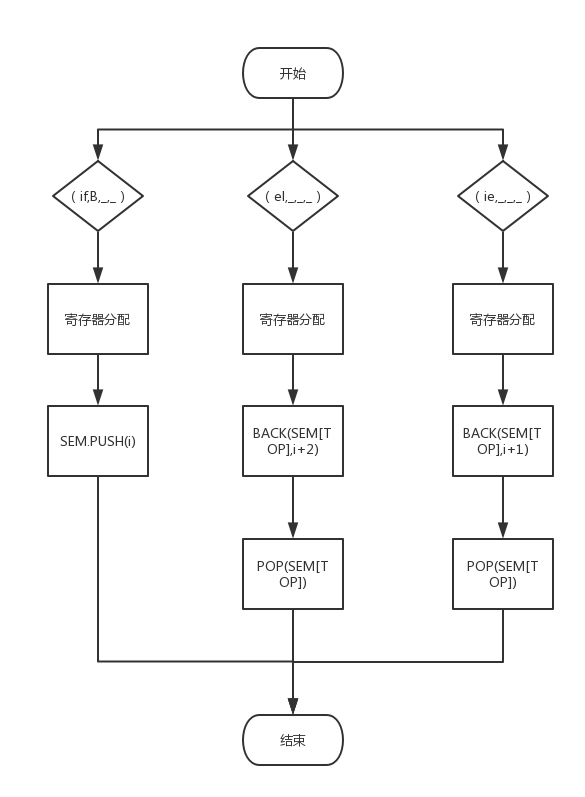
把A填入寄存器表；

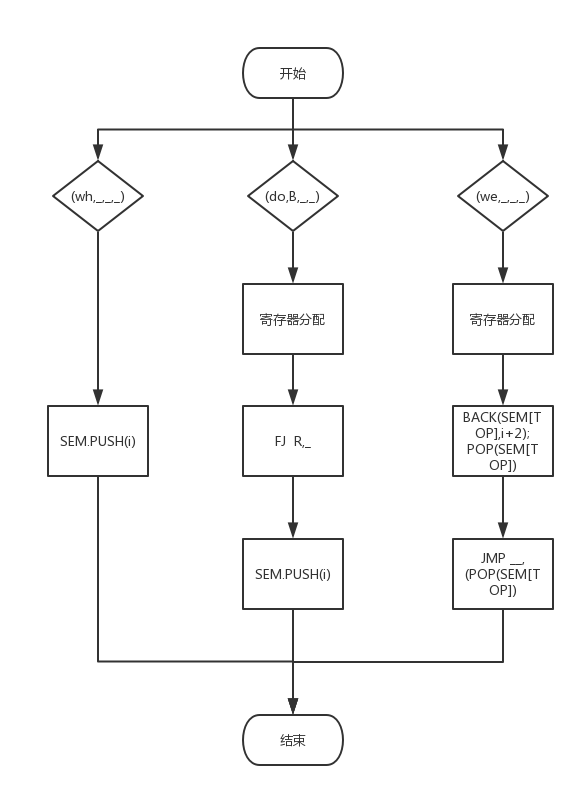
3.强迫释放：剥夺一Ri

处理办法同1.



跳转语句生成算法1：

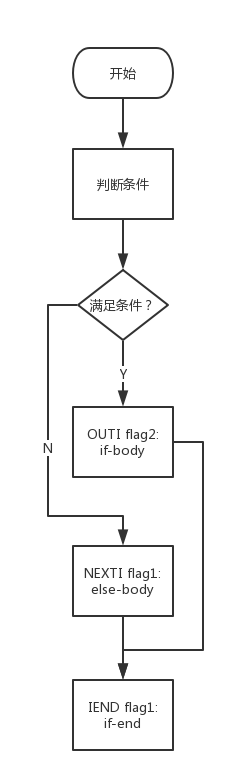




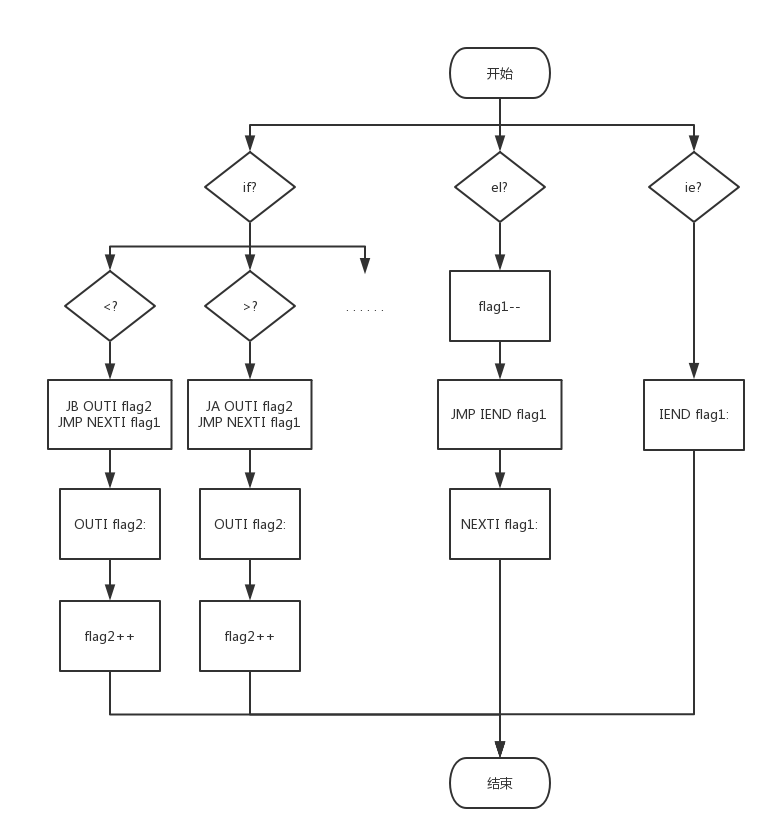
**跳转语句生成算法2：**

第一种算法利用一个语义栈实现了跳转地址的获取，而第二种算法则利用了汇编语言中的标号功能，定义一组全局变量， flag1、flag2、tot1、tot2，作为跳转标号的编号，无需使用语义栈保存要跳转的地址信息，并且扫描完当前语句就可以立即生成目标指令，简化了逻辑。

以if语句为例，将高级语言中的if-else结构用汇编语言表达，流程图为：



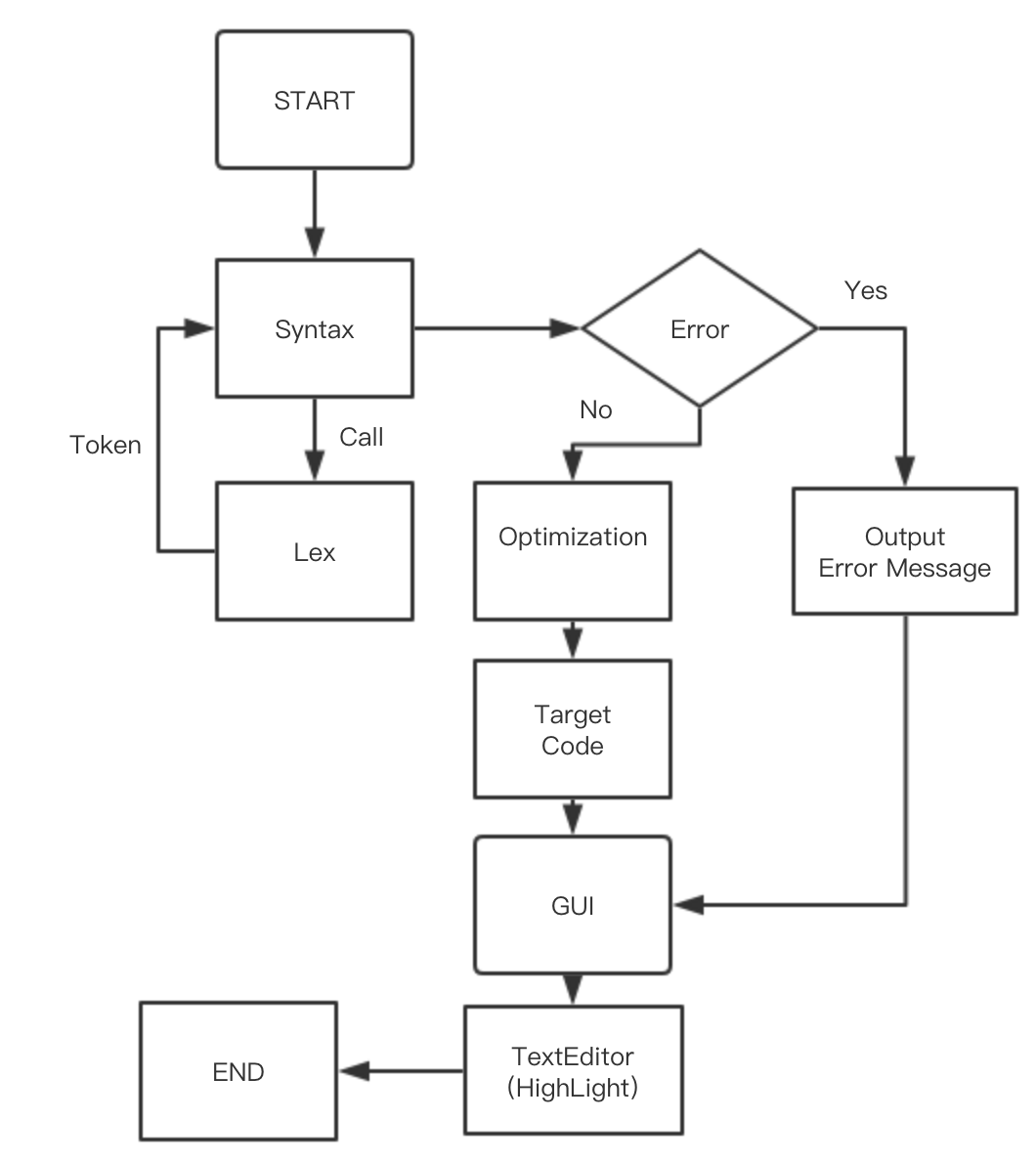
图中OUTI flag2是if-body段的标号，NEXTI flag是else-body段的标号，IEND flag1是if结构结束的标志。根据以上标号，我们可以进一步画出此算法的流程图：



处理while的过程与if相似，不再赘述。

# 4. 程序设计与实现

## 4.1 程序流程图



模块具体流程图请参照第3部分中的内容

## 4.2 程序说明

1.程序运行环境：CLion

2.入口：jackc.cpp

3.头文件及数据结构：

class Main

{

function void main()

{

Array arr;

String s;

int i;

arr = Array.new(5); // 创建一个大小为5的数组

i = 0;

while (i < 5)

{

s = Input.readLine();

arr[i] = s.intValue();

i = i + 1;

}

Main.bubble\_sort(arr, 5);

i = 0;

while (i < 5)

{

Output.printInt(arr[i]);

Output.println();

i = i + 1;

}

Output.println();

return;

}

/\* 冒泡排序 \*/

function void bubble\_sort(Array arr, int n)

{

int i, j, tmp;

i = n - 1;

while (i > 0 | i == 0) // 由于还没有加上 >= 运算符, 所以暂时用这个代替

{

j = 0;

while (j < i)

{

if (arr[j] > arr[j + 1])

{

tmp = arr[j];

arr[j] = arr[j + 1];

arr[j + 1] = tmp;

}

j = j + 1;

}

i = i - 1;

}

return;

}

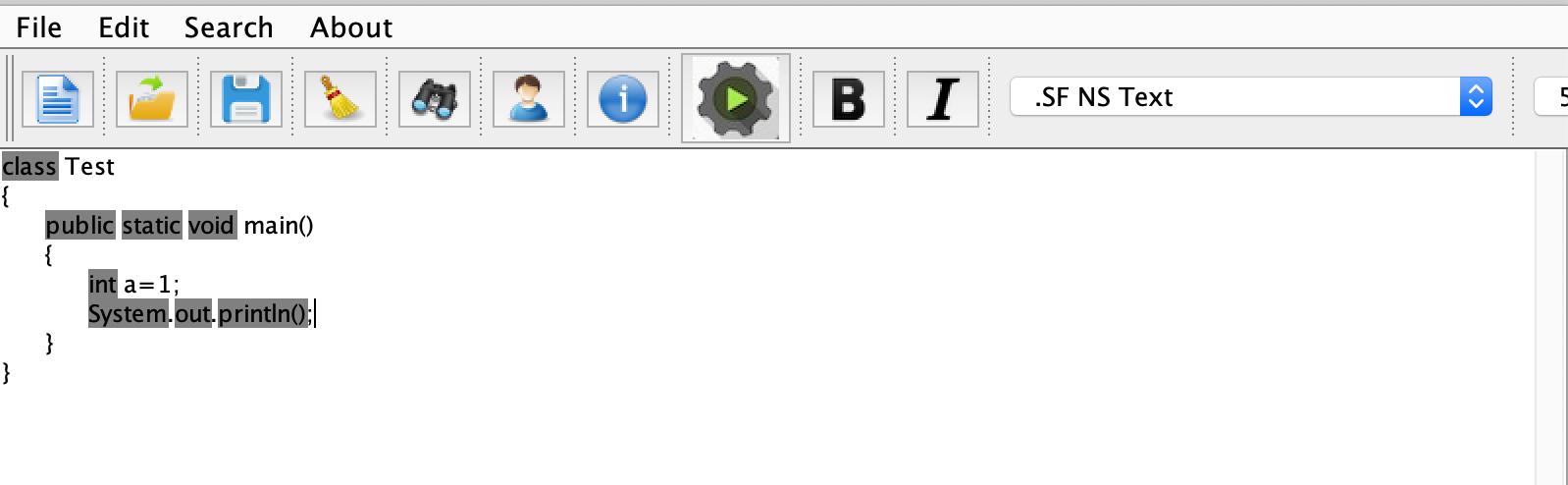
}

## 4.3 实验结果

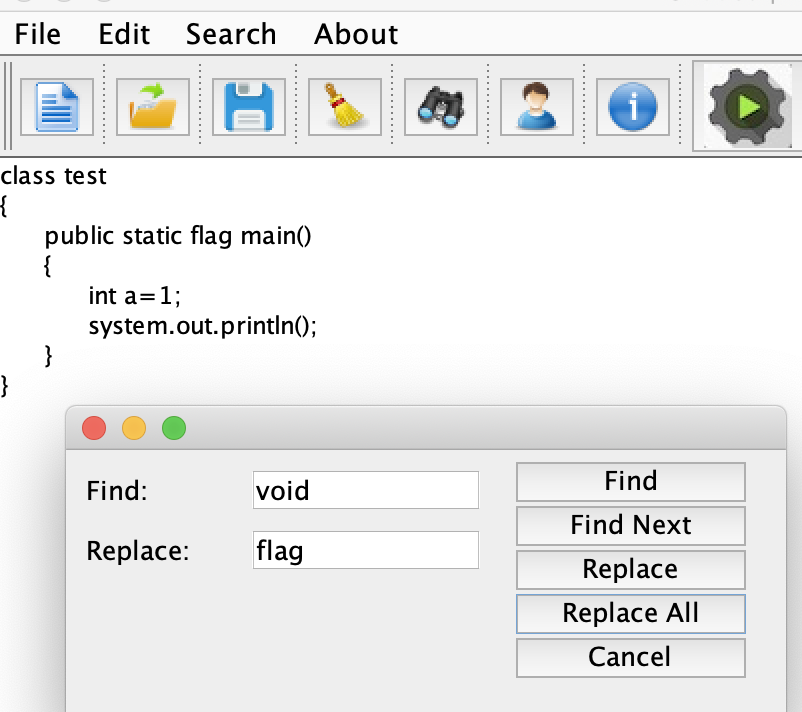
输入如下一段代码，包含课程设计的基本要求：声明、赋值、分支、循环，还包含数组的声明、赋值和引用。其中的write语句是我们自己定义的、为了方便演示，让汇编程序输出的语句。

测试程序：

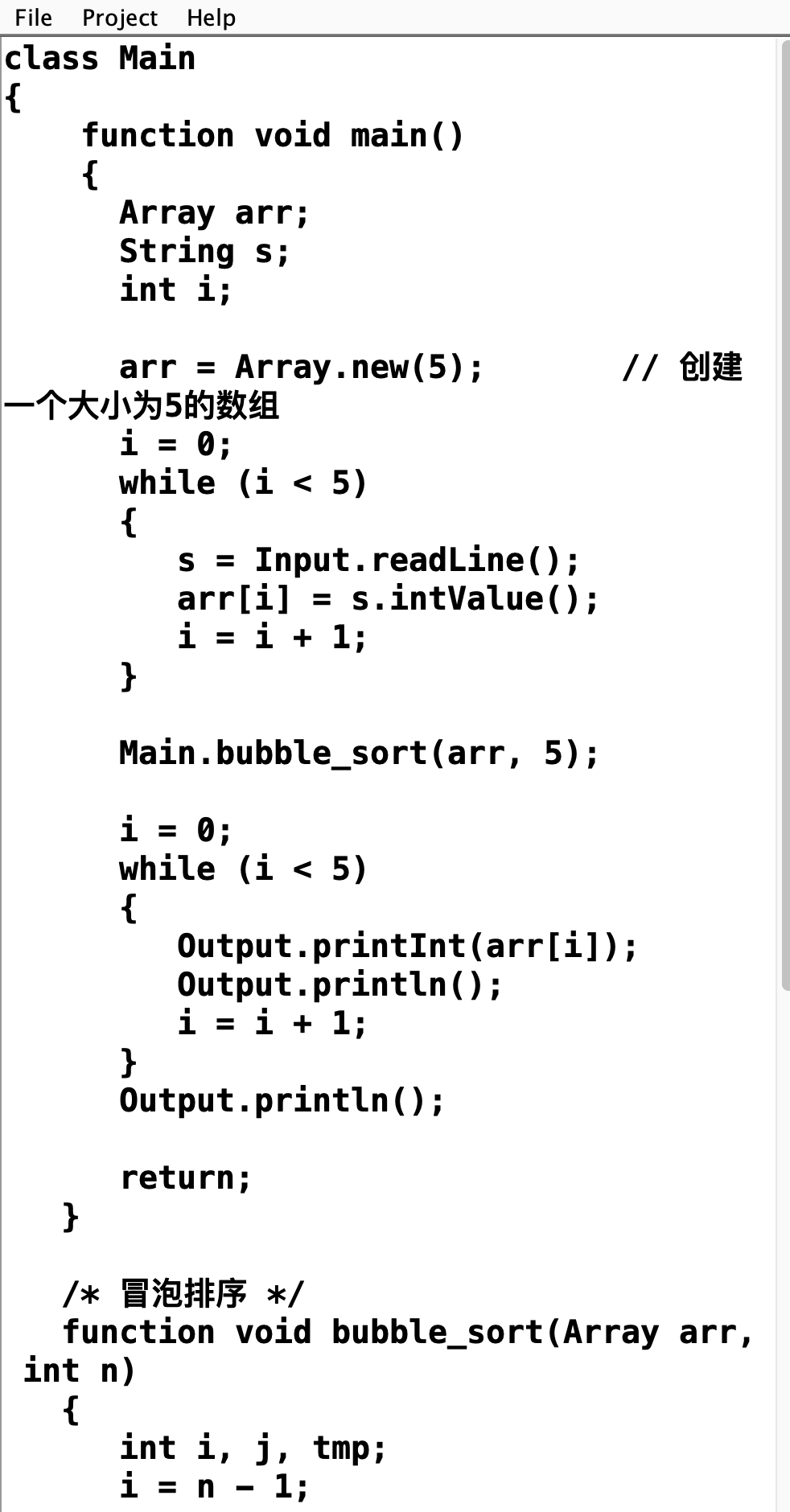
文本编辑器界面：（高亮显示）



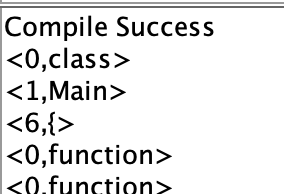
文本编辑器：（查找替换）



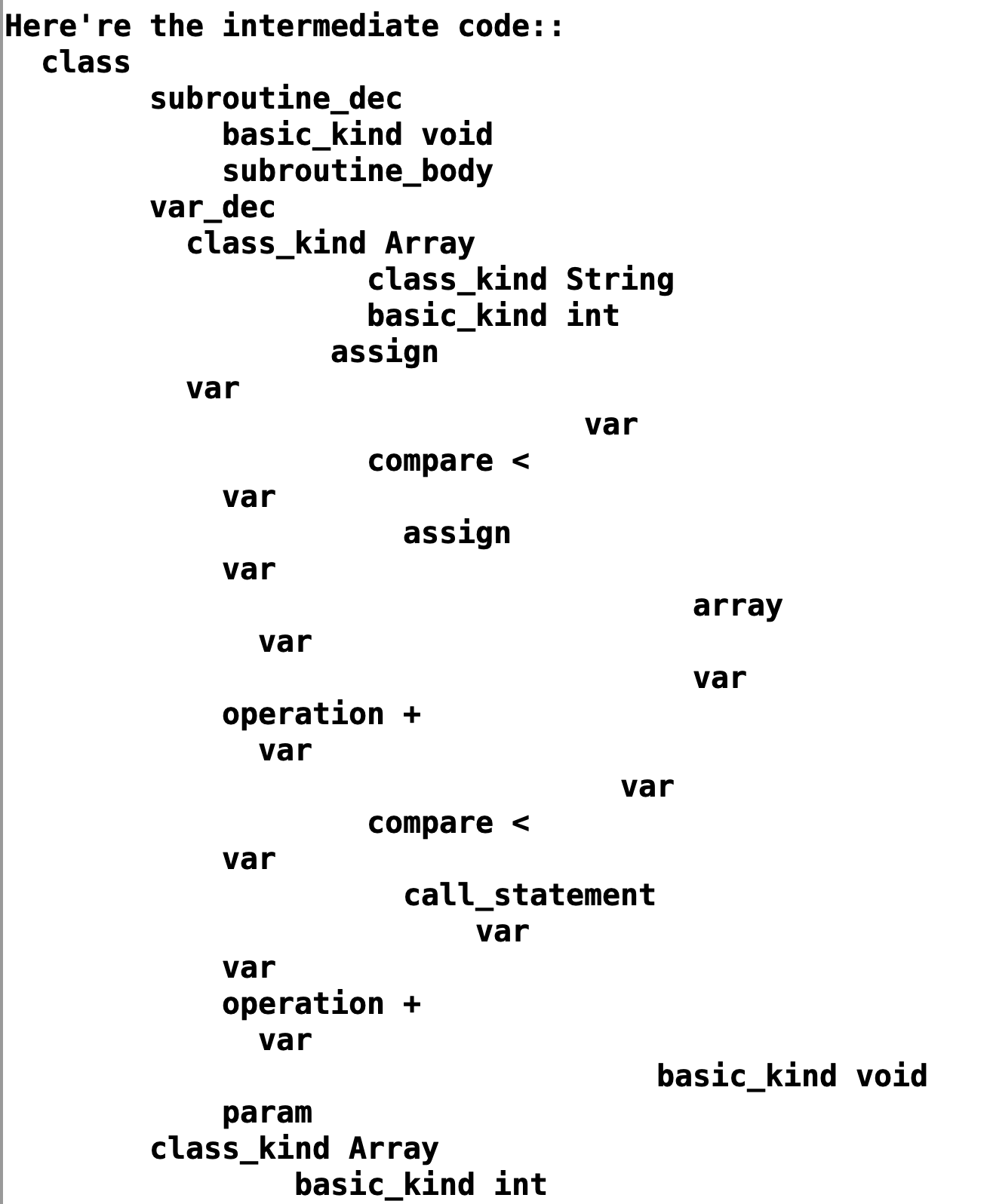
首先输入程序并点击编译按钮：



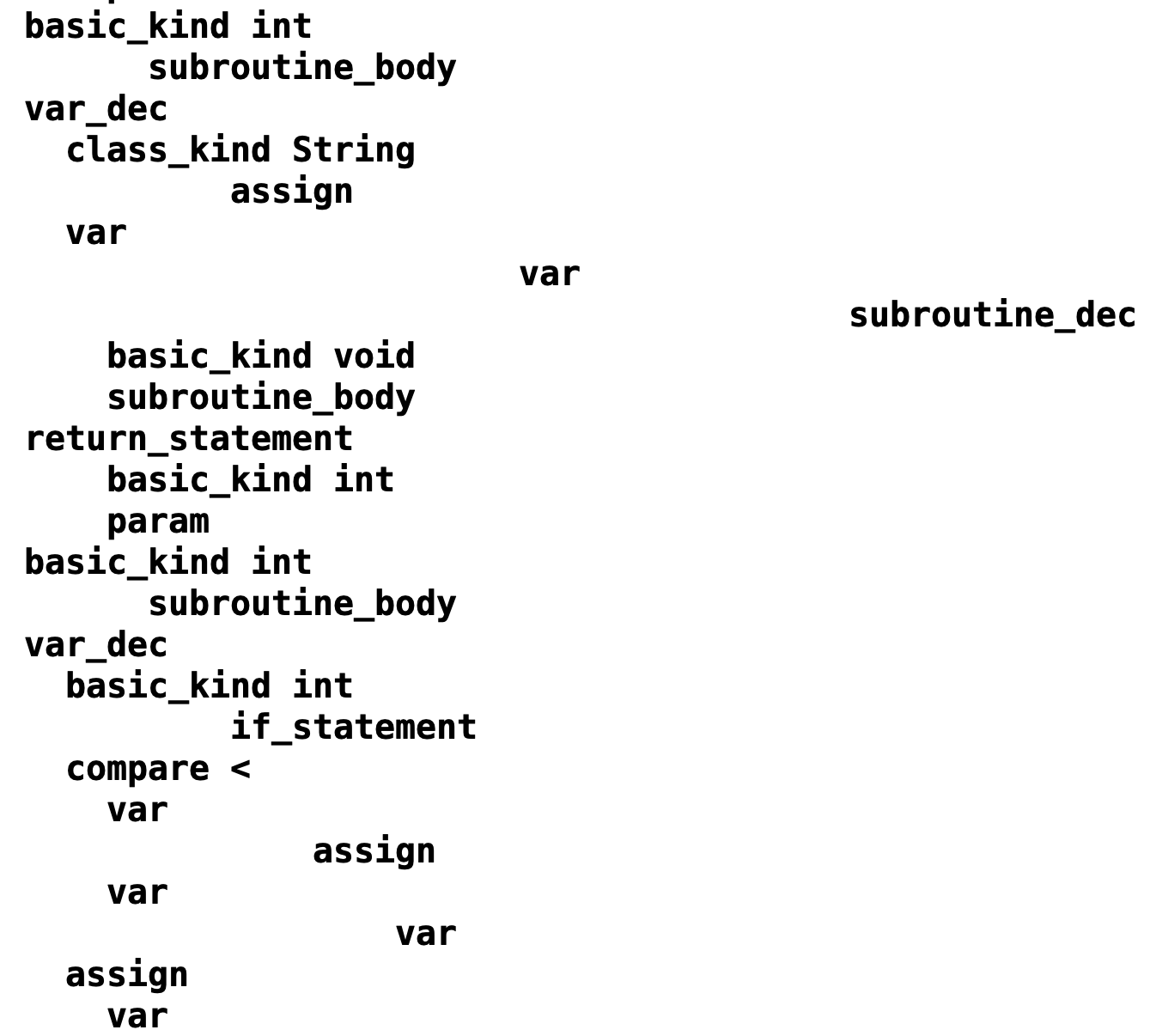
词法分析扫描结果（部分显示）：



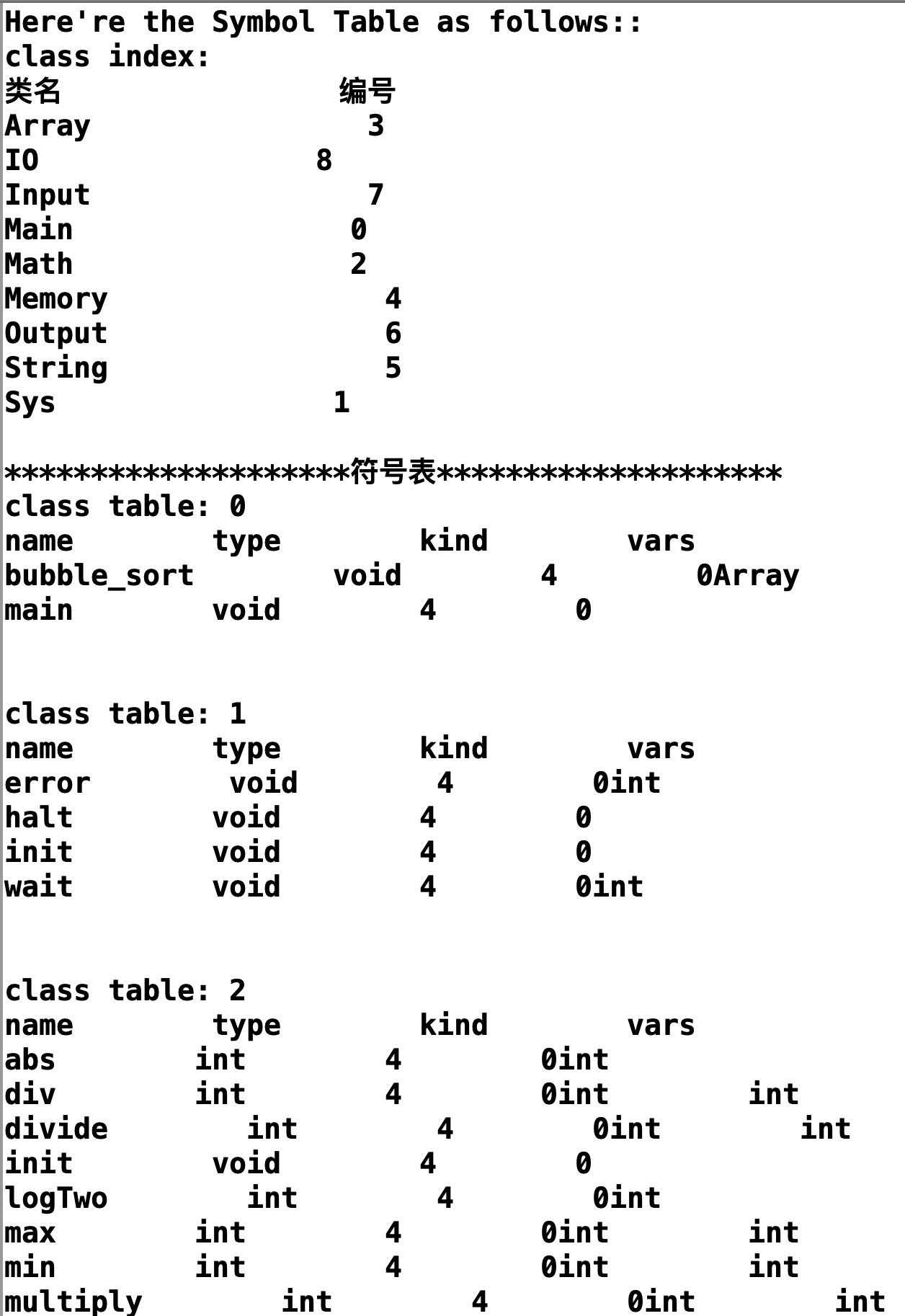
语法分析结果：



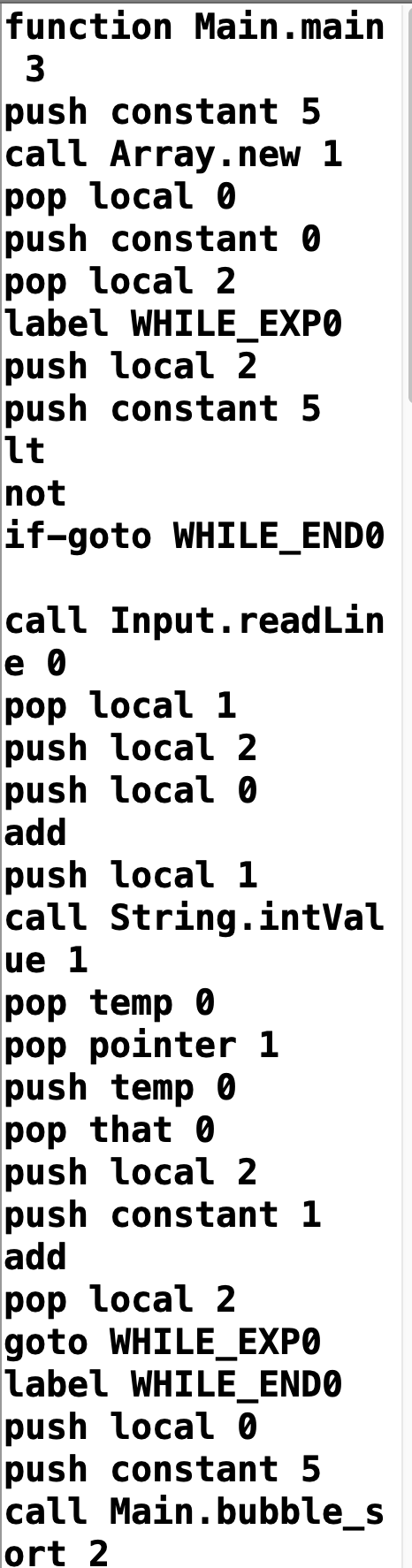
语法树的优化（部分显示）：



符号表的显示（部分显示）：

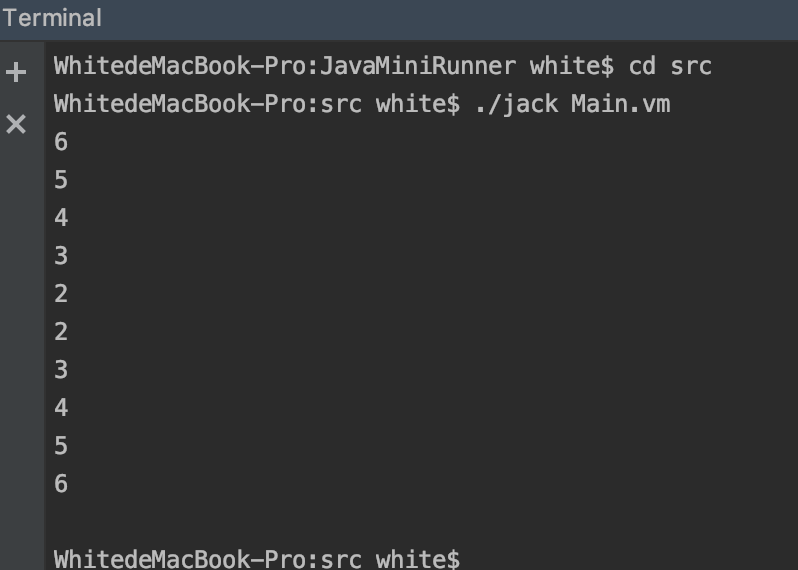


目标代码的生成（部分显示）：



生成的目标代码在虚拟机上运行的结果为：

注：测试样例为冒泡排序数组中的5个元素，并输出

****

运行结果正确

# 5. 结论

我们所设计的类java语言编译系统可以根据自己所定义的文法成功的进行词法分析，生成相应的Token序列、成功地生成静态符号表，并能对静态符号表随时进行查看。我们所设计的java语言编译系统也可以成功地对文档中的内容采用LL1分析法（验证文法是否为为合格的文法）和递归下降法（实际上的语法分析）两种方法进行语法分析，并且可以检查出错误类型和错误位置，并提示出错。另外，在进行语法分析的同时，我们通过插入语义动作可以同时生成AST。通过测试，我们可以成功的生成所需的AST。我们也对AST的优化进行了测试，我们可以成功地对原始的AST进行部分的优化，对API的重复编译和AST的数据结构进行了优化从而可以避免遍历时不必要的回溯。我们还成功地通过语法树生成了机器代码，并且使我们的机器代码能够正确地运行。此外，我们还成功地在我们设计的java语言编译系统中添加数组的相关功能，实现了数组的声明、引用和整体调用，也实现了函数、类等结构相关的功能，和类函数、类方法等面向对相关的功能，从而可以更结合实际进行面向对象程序的开发。

因此，整体来说，我们所设计的java语言编译系统是成功的。它拥有完整的前后端、能够对语法错误进行定位和分类，并且能够输出可以正确运行的汇编代码。并且可以完美地在图形界面中展示出所有的中间结果，并且额外实现了一个带高亮功能的文本编辑器。

编译器还有很多类型和值得创新的地方，正所谓学无止境，永远会有更加高效的解决方式尚未发现，在日后的编程生涯中，还需要多加注重效率的问题，使得程序运行的复杂度得到最大程度的优化，尽力编写出优质的程序。

# 6. 参考文献

1、陈火旺.《程序设计语言编译原理》（第3版）. 北京：国防工业出版社.2000.

2、美 Alfred V.Aho Ravi Sethi Jeffrey D. Ullman著.李建中，姜守旭译.《编译原理》.北京：机械工业出版社.2003.

3、美 Kenneth C.Louden著.冯博琴等译.《编译原理及实践》.北京：机械工业出版社.2002.

4、金成植著.《编译程序构造原理和实现技术》. 北京：高等教育出版社. 2002.

# 7. 收获、体会和建议

## 窦洹彰心得：

这次编译原理的课程设计是我在大学以来课内接触到的规模最大，难度最高的工程，半个多月的过程，完整的经历下来，这次课设给我带来的感受很多，也让我成长了许多。总结起来可以用以下的几个关键词：创新、学习、挑战、坚持。

编译原理号称是计算机科学与技术专业最难的课程，因为其连贯性强，难度大著称。因此在课程之前，我们还熟悉了许多额外的东西，比如看mooc，看龙叔，看博客来增强自己对编译过程的理解能力。事实证明这些事前工作是非常有用的，提前扫清了我们许多的疑惑，比如语法树遍历过程的细节，和面向对象符号表的设计构造等。

首先，课程设计的书面时间是半个月，但我们的准备时间却远不止半个月。因为在课上学到的知识知识理论上的，而且都是分块的，所以我们对一个编译器实际上如何进行设计、组织毫无头绪，只对编译实验课上的词法分析、语法分析等有一些简要的了解，所以我们还是有许多工作去做的。

在设计层面上，我们三个人达成一致想要做一些不一样的产品出来，总体想设计一个带查找、替换、高亮功能的文本编辑器和一个可以用文本编辑器驱动的编译器。在文法上则参照了java来设计一种面向对象的语言，这与我们在课上学到的知识有很大的出入，因此我们面临在设计上的设计的创新与困难还是很具有挑战性的

虽然我们个人的能力都不出众，但是因为提前准备和之前三次实验的基础，我们的初期进度还是很快的：第一次检查进度时，我们的前端已经能够把实验要求的基本功能大致实现，但如果实现后端的话，符号表和目标代码即后端的实现便有了很大的难点，毕竟面向对象和面向过程的文法还是有很大的不同，比如成员变量、成员函数等实现在课程上是无法体现的。我们查阅了龙书和一系列的博客，不断学习，才有了初步的设计思路。

在编译器的重要骨架语法分析方面，我们放弃了使用原本的四元式的方法，而是转而使用了语法树。初期考虑的是语法树（AST）具有挑战性，并且在优化的时候可能会出现较好的效果，但与此同时带来的我们的符号表、活动记录，活跃信息，基本块划分等课上的知识全部失效，我们不得不去寻找资料，参考外文教材去学习全新的知识来匹配语法树的结构，当然事实证明我们这样的选择不但可以起到了提升自己的效果，并且可以在后期的实现中起到特别好的效果。

此次作为组长，我主要负责编译器的后端实现、虚拟机的编写、文本编辑器的实现和项目的整体拼接。我的核心任务还是整体项目的拼接，因为这个项目从头到尾有着非常大的连贯性，从语法分析到虚拟机的实现，前后有着紧密的联系，所以对于自己的和队员的工作必须有很强的全局观念，整体把握程序的每一个逻辑环节，每一处出错都可能会导致整体项目的无法运行，这是我认为作为组长最具有挑战性的地方。在项目的整体拼接之前，后端的进展其实是十分顺利的，我们可以根据语法树来产生目标代码（虚拟机依赖），并且经过优化可以实现很大程度地减少目标代码的数量。主要是在实现虚拟机的时候，需要考虑到目标代码指令集，对整体难度有着极大的提升。并且我们敢于创新，更新了数据结构，全盘学习新的知识，新的符号表组织，新的实现形式，新的优化算法，因此创新性也是一个巨大的难点。

平心而论，后端虽然可借鉴的资料很少，许多细节性的问题在书中和网上很难找到成熟可用的答案，但是在算法上相比于前端来说无疑是相对简单的。但是我们等到实现后端的时候，实际上已经过去了很长时间，因为前期主要的工作是学习和搜集资料，找齐了相关知识和博客后，再码代码，速度也就快了许多，但是时间的紧迫还是十分挑战性，到最后几天和组员通力合作、工作了几个晚上才将后端实现完整。

十分有趣味性的是，在最后两天时，我突然想到文本编辑器中的高亮功能，与课程中学习到的词法分析有很大的关系，于是心血来潮想做一个具有高亮功能的编辑器。一开始只能实现静态的高亮（保存文本后关键字显示不同的颜色），之后想到了可以用java的事件机制，将键盘的触发作为驱动词法分析的条件，可以实现动态的高亮，说干就干，在验收报告的前一天，终于实现了动态的高亮和一个简单的功能齐全的文本编辑器，虽然过程艰辛，但是成就感满满，实现了具有应用功能的东西。如下图



总地来说，我们的工作中存在着许多沟通上的问题，但是结果已经出乎我的意料了：我们三个同学，做出了一个功能完善，前后端齐备，封装了可视化界面的编译器，并且实现了功能完备具有高亮功能的文本比那机器，这是我在课设之前从预料到的事情。在这里真的要感谢我们负责前端的队友做出功能强大且完善的前端，包括词法分析，错误处理，AST的编写，几天不厌其烦的改BUG，一次又一次的重写算法，适应新的数据结构。还有负责课设的老师，每次遇到无法解决的问题时，都会耐心的给我们讲解，解决了我们很多理论上较不准的细节。也感谢许多网络资源给我们的疑问一步步解决，学会到了具体问题解决过程中查阅资料的重要性。

这次课程设计，不但让我把课上的理论和实际结合到了一起，让我更加深刻地了解了编译原理的相关知识，更多地，这个过程为我编写大型工程积累了经验，让我知道了程序员之间怎样才能更加高效的合作。不止让我更加了解编译的整个流程，知晓我们平时编写的程序是如何变成最后的机器语言，更深入了解计算机执行的过程，更加理解理论课上的知识，激发了我对学习的积极性，同时也带给了我一次很难忘的经历。

实际上也对我们以后的团队编程开发提供了很好的经验，体会到了编程的简单流程，在今后的编程生活中也会时刻记起这次课设的过程，总结教训，积累经验，做到出现的错误不再犯，出现的长处继续发扬，争取在今后的生活中有着更强的代码风格和团队协作能力。

## 张明阳心得：

本次课程设计是我初次接触较大规模的实验，也是我收获最大的一次实验。虽然前期的编译原理课程的学习为此次课设打下了一定的理论基础，但只有亲身实践之后，才会发现知识仅仅停留在理论层面是远远不够的，还需要将理论结合实践，才能对知识有更深入的理解。

由于本人的基础较为薄弱，故和团队一起在课程设计开始前一周就着手准备。团队先以最简单的算术表达式入手，而本人主要负责词法分析以及语法分析模块，发现该部分内容与之前实验内容相差不大，便对之前的内容加以修改，然则发现其与课程设计的最终目的相差甚远。经过讨论发现，团队存在的问题在于没有一套成体系的文法，于是一起设计了仿C语言的简单文法，而后又根据进度对其修改添加。有了文法后，队友之间的沟通交流顺畅了许多，同时对自己的工作也更加明晰，而我在拿到文法后就先分析所用到的关键字表、界符表以及词法分析和语法分析的流程图，有了这些准备工作后，代码的实现就容易了许多，为后期功能的添加留下了时间。

在之前的三次实验中已经对编译的各个步骤有了初步的了解，在课程设计之前，组内讨论决定选择一个简易的java的语言，这样就产生了一些问题，在课堂上学习的东西是远远不够的。如何记录类的层次，以及嵌套关系，并进行语义分析，在小组讨论时，我们上网搜索，发现java的编译一般是采用语法树这样的形式，我们讨论后也觉得语法树虽然难构造一些，数据结构嵌套更复杂一些，但是还是选择了这样更直观，更容易理解的方法，采用了最基础的递归子程序的语法分析方法。由于这个工作更加细致，所以分给了我这个女孩子来负责。

由于有了文法参考，我们对java的语言进行了删减，形成了在我们能力范围内能够实现的文法，商讨好了接口，就开始着手做了。实验中也面临了许多问题，为了减少出错，我先画出流程图，有了明确的思路，再着手写代码，由于对c++掌握的不够扎实，所以写起代码来比较慢。

通过该课程设计，全面系统的理解了编译原理程序构造的一般原理和基本实现方法。把死板的课本知识变得生动有趣，激发了学习的积极性。把学过的计算机编译原理的知识强化，能够把课堂上学的知识通过自己设计的程序表示出来，加深了对理论知识的理解。以前对于计算机操作系统的认识是模糊的，概念上的，现在通过自己动手做实验，从实践上认识了操作系统是如何处理命令的，如何协调计算机内部各个部件运行，对计算机编译原理的认识更加深刻。课程设计中程序比较复杂，在调试时应该仔细，在程序调试时，注意指针，将不必要的命令去除。

这次试验让我对编译原理的理论有了很深刻的认识，能够学习更多新的知识，能够不停思考去解决问题，非常锻炼自己思维逻辑；而让我学到更多的是团队合作，在面临困难的时候不是孤军奋战的感觉真的是太好了。

## 刘滋雨心得：

这次课程设计是我收获最大的一次实验。前期的编译原理课程的学习和之前的三次编译原理实验为此次课设打下了一定的理论基础，理论与实践结合，使我对知识有更深入的理解。本次编译原理课程设计历时两周，在老师的指导与队友的合作下，我们顺利完成本次课设。

由于我的基础较为薄弱，因此在课程设计开始前就着手准备了。我主要负责词法分析以及报错模块，发现该部分内容与之前实验内容相差不大，便对之前的内容加以修改，然则发现其与课程设计的最终目的相差甚远。经过讨论，对自己的工作更加明确，我在拿到文法后就先分析所用到的关键字表、界符表以及词法分析的流程图，有了这些准备工作后，代码的实现就容易了许多，为后期功能的添加留下了时间。在报错模块中，队友在实现模块功能是就有判断错误，我仅仅需要输出显示什么错误，结合token序列中的row行实现错误的定位。

这次课程设计，花费了两周的时间，不仅收获满满，也让我成长了许多。首先，通过实践加深了对编译器理论知识的理解，尤其是对前端部分模块的功能实现有了清楚的认识。其次，通过对词法分析以及报错模块的编码，提高了本人编写代码的能力，在实践过程中，由于之前基础不扎实，许多功能的实现都需要重新查阅相关资料加以学习了解，比如set的相关使用是我第一次深入接触。此次课程设计的成功进行离不开队友的帮助与鼓励，当遇到问题时，团队之间总会及时进行沟通，积极寻找解决问题的方法，比如实现含转义字符串识别时，团队之间积极进行讨论，最终找到一种正确的方式即在词法分析中加入相关状态转换完成功能，这锻炼了我们团队协作能力，为以后团队合作打下了基础。最后，特别感谢几位老师对我们团队的帮助与鼓励，当我们遇到问题向老师求助之时，老师都会及时解决我们的困惑，为我们指出方向，帮助我们快速找到不足，加以改进。