

# 程 设 计 报 告

设计题目：一个简单文法的解释器的设计与实现

## 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。本课设是词法分析、语法分析、语义分析的综合，外加上扩展任务中间代码的优化和目标代码的生成，主要是锻炼学生的逻辑思维能力，进一步理解编译原理的方法和步骤。

本次课程设计我们小组完成了一个简单文法编译器的设计与实现，其语言要求与 C 语言一致，我们参考了 C99 标准，设计出了类 C 的简单但完整的程序生成文法，以及对应的翻译文法。在此基础上我们还做了处理赋值语句、if-else 语句、while 语句的语法分析和四元式的生成，这样我们就设计完了四元式的生成。然后，我们又设计了符号表，对应四元式的目标代码生成。

其中的特色点有：在词法、语法分析阶段能够检测出错误，并且能指出错误在哪一行，具体为什么错误；表示式的四元式采用了逆波兰式的方法；同时像 if-while，我们的编译器能判断其中的 boolean 表达式的真值，从而能采用正确的逻辑得出正确的结果；

**关键词：**编译原理，完整编译器，LL1 分析法，语法制导，四元式，符号表，目标代码

# 目 录

摘要	I
1 概述	6
2 课程设计任务及要求	8
2.1 设计任务	8
2.2 设计要求	8
3 算法与数据结构	9
3.1 算法的总体思想（流程）	9
3.2 词法扫描模块	9
3.2.1 功能	9
3.2.2 数据结构	9
3.2.3 算法	11
3.3 语法分析模块	12
3.3.1 功能	12
3.3.2 数据结构	12
3.3.3 算法	13
3.3.4 算法流程图	14
3.4 语义分析四元式以及符号表分析模块	20
3.4.1 功能	20
3.4.2 数据结构	20
3.4.3 算法	21
3.5 目标代码生成模块	29
3.5.1 功能	29
3.5.2 数据结构	29
3.5.3 算法	32
4. 程序设计与实现	33
4.1 程序流程图	33
4.1.1 程序流程图	
4.1.2 设计框架	
4.1.3 文法的设计	
4.1.4 翻译文法的设计	
4.2 程序说明	33

4.2.1 程序说明	
4.2.2 软件结构	
4.3 实验结果	37
4.3.1 测试用例	
4.3.2 用例语法正确时结果	
4.3.3 用例语法错误时结果	
5. 结论	55
6. 参考文献。	56
7. 收获、体会和建议。	57

## 1. 概述

编译原理是计算机专业的一门重要专业课，旨在介绍编译程序构造的一般原理和基本方法。内容包括语言和文法、词法分析、语法分析、语法制导翻译、中间代码生成、存储管理、代码优化和目标代码生成。编译原理是计算机专业设置的一门重要的专业课程。虽然只有少数人从事编译方面的工作，但是这门课在理论、技术、方法上都对学生提供了系统而有效的训练，有利于提高软件人员的素质和能力。由于时间和同学们的水平有限，故本次课设内容只涉及到了词法分析，语法分析，及语义分析中的中间代码的四元式生成和符号表以及目标代码的生成（仅在语义分析阶段做了简单的优化）。具体概述介如下：

1. 词法分析：在这个阶段编译器实际阅读源程序（通常以分析程序字符流的形式表示）。扫描程序执行词法分析注释树符号表：它将字符序列收集到称作记号错误处的有意义单元中，记号同自然语言，与源代码处理器语中的字词相似。因此可以认为扫描程序执行与优化程序拼写相似的任务。本实验的词法分析程序用于生成 Token 序列。并设置简单的出错处理，可以指出哪个单词在哪行出错并简单指出出错的原因

2. 语法分析：该程序从扫描程序中获取记号形式的源代码，并完成定义程序结构的语法分析，这与自然语言中句子的语法分析类似。语法分析定义了程序的结构元素及其关系。其任务是识别和处理比单词更大的语法单位。本实验用于指出程序设计语言中的表达式、各种说明和语句乃至全部源程序其中的语法错误；必要时，可生成内部形式，便于下一阶段处理。

3. 语义分析：程序的语义就是它的“意思”，它与语法或结构不同。程序的语义确定程序的运行，但是大多数的程序设计语言都具有在执行之前被确定而不易由语法表示和由分析程序分析的特征。这些特征被称作静态语义，而语义分析程序的任务就是分析这样的语义。一般的程序设计语言的典型静态语义包括声明和类型检查。由于同学们水平有限，我们只做了其中的符号表部分。

4. 中间代码：根据中间代码的类型和优化的类型，该代码可以是文本串的数组、临时文本文件或是结构的连接列表。对于进行复杂优化的编译器。由于同学们水平有限，我们只做了四元式的生成。

5. 目标代码：根据中间代码结合语义分析时得到的符号表再结合特地的目标机器生存目标代码(本次使用的是 8086 处理器的目标代码格式)。经过生成的目标代码由于一定的水平限制还不能在特地机器下跑起来,但是我们整体的方向是对的。

编译原理课程兼有很强的理论性和实践性，是计算机专业的一门非常重要的专业基础课程，在系统软件中占有十分重要的地位。编译原理课程设计是本课程重要的综合实践教学环节，是对平时实验的一个补充。通过编译器相关子系统的设计，使学生能够更好地掌握编译原理的基本理论和编译程序构造的基本方法和技巧，融会贯通本课程所学专业理论知识；培养学生独立分析问题、解决问题的能力，以及系统软件设计的能力；培养学生的创新能力及团队协作精神。

## 2. 课程设计任务及要求

### 2.1 设计任务

在下列内容中任选其一：

- 1、一个简单文法的编译器前端的设计与实现。
- 2、一个简单文法的编译器后端的设计与实现。
- 3、一个简单文法的编译器的设计与实现。
- 4、自选一个感兴趣的与编译原理有关的问题加以实现，要求难度相当。

这次实验我们小组所做的内容涵盖了一个编译器的前端和后端，因此基本实现了一个简单文法的编译器的设计与实现。

主要内容如下：

- 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 语句等)。
- 2、词法扫描设计实现。
- 3、语法分析设计实现。
- 4、中间代码设计实现。
- 5、符号表的生成。
- 6、目标汇编语言的生成。

### 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；

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 算法与数据结构

### 3.1 算法的总体思想

我们的课程设计实验基于自动机的词法扫描，采用 LL1 分析法的语法分析的方法实现编译器前端和对应后端代码的生成(目标机器为 X8086)。整体框架主要部分有词法分析、语法分析、语义分析中四元式生成、符号表建立等以及对程序的目标代码生成(运用单寄存器下目标生成算法，将四元式翻译成为汇编语言。)。在搭建框架的时候我们定义了一套接口，以协同合作。

整体的框架类图关系如下(使用 UML 组织结构, 其中也可以看出我们的接口之间的连接关系):

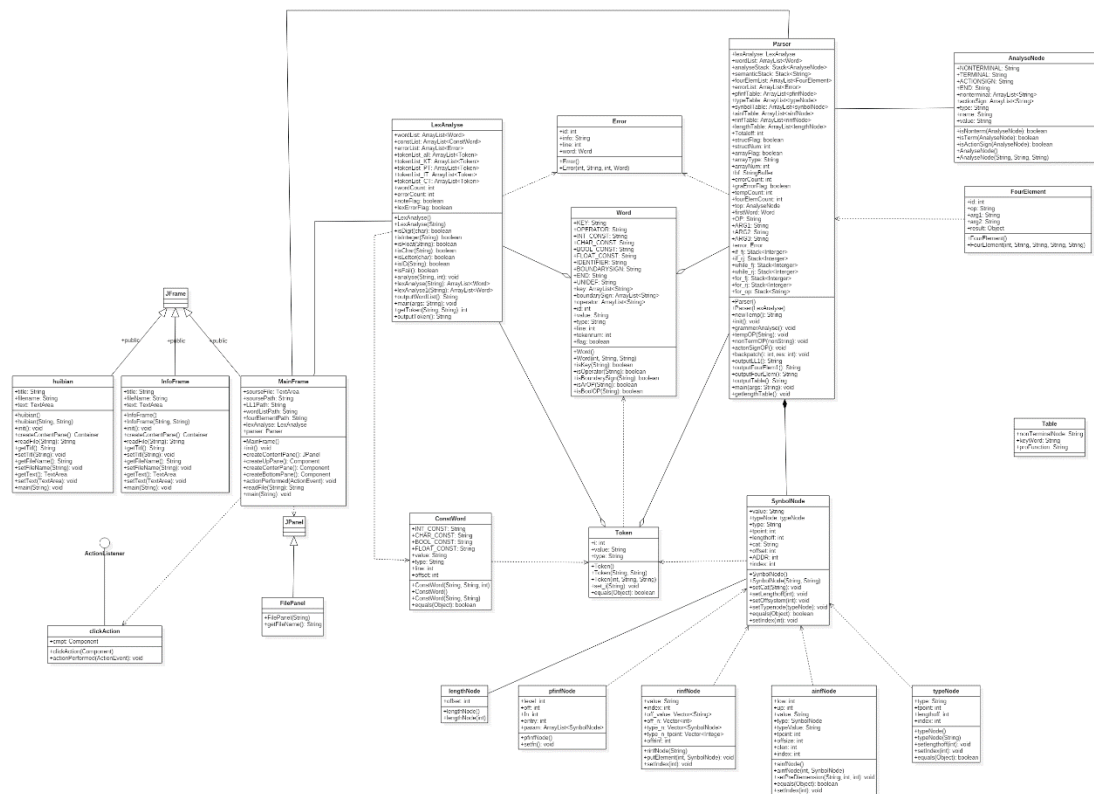


图 3-1 程序框架图

### 3.2 词法分析

#### 3.2.1 功能

词法分析程序又称扫描器，任务有二：

- (1) 识别单词——从用户的源程序中把单词分离出来；
- (2) 翻译单词——把单词转换成机内表示，便于后续处理。

词法分析是编译的第一步,其目的是对程序进行扫描,并生成 token 序列,以便后面进行语法分析。

### 3.2.2 数据结构

词法分析中用到的数据结构如下所示:

LexAnalyse 词法分析器类设计:

```
public class LexAnalyse {  
  
    ArrayList<Word>      wordList      = new ArrayList<Word>(); //单词表  
    public ArrayList<ConstWord> constList = new ArrayList<ConstWord>(); //常数表  
    ArrayList<Error>     errorList     = new ArrayList<Error>(); // 错误信息列表  
  
    ArrayList<Token>     tokenList_all = new ArrayList<Token>(); //总token表不能重复  
    ArrayList<Token>     tokenList_KT = new ArrayList<Token>(); //关键字表  
    ArrayList<Token>     tokenList_PT = new ArrayList<Token>(); //界符表  
    ArrayList<Token>     tokenList_IT = new ArrayList<Token>(); //标识符表  
    ArrayList<Token>     tokenList_CT = new ArrayList<Token>(); //常量表  
  
    int                  wordCount      = 0; // 统计单词个数  
    int                  errorCount     = 0; // 统计错误个数  
    boolean              noteFlag       = false; // 多行注释标志  
    boolean              lexErrorFlag   = false; // 词法分析出错标志  
  
    public LexAnalyse() {}  
  
    public LexAnalyse(String str) {}  
    * 数字字符判断  
    private static boolean isDigit(char ch) {}  
  
    * 判断单词是否为int常量  
    private static boolean isInteger(String word) {}  
  
    * 判断单词是否为float常量  
    private static boolean isFloat(String word) {}  
  
    * 判断字符是否为字母  
    private static boolean isLetter(char ch) {}  
  
    * 判断单词是否为合法标识符  
    private static boolean isID(String word) {}  
  
    * 判断词法分析是否通过  
    public boolean isFail() {}  
}
```

图3-2LexAnalyse词法分析器类结构

在程序中用链表存储总Token表,关键字表,界符表,常数表,标识符表以及错误信息表。对应的结点内容如下所列。

Token 类设计:

```

public class Token {

    int i;
    String value;
    String type;

    Token(){}

    Token(String value,String type){}

    Token(int i,String value,String type){}
    //根据获得的value来设定对应的i
    void set_i(String value){}

    public boolean equals(Object obj){}

}

```

图 3-3Token 类设计

ConstWord 类设计:

```

public class ConstWord
{
    public final static String INT_CONST    = "整形常量";//offset=4B
    public final static String CHAR_CONST   = "字符常量";//offset=1B
    public final static String BOOL_CONST   = "布尔常量";//offset=1B
    public final static String FLOAT_CONST  = "浮点常量";//offset=8B

    String value;    // 单词的值
    String type;     // 单词类型
    int line;        // 单词所在行
    int offset=4;

    ConstWord(String value, String type, int line){}

    ConstWord(){}

    ConstWord(String value, String type){}

    public boolean equals(Object obj){}

}

```

图 3-4 ConstWord 类设计

Word 类设计:

```

public class Word {

    public final static String KEY          = "关键字";
    public final static String OPERATOR     = "运算符";

    public final static String INT_CONST    = "整形常量";
    public final static String CHAR_CONST   = "字符常量";
    public final static String BOOL_CONST   = "布尔常量";
    public final static String FLOAT_CONST  = "浮点常量";//新增

    public final static String IDENTIFIER   = "标志符";
    public final static String BOUNDARYSIGN = "界符";
    public final static String END          = "结束符";
    public final static String UNIDEF       = "未知类型";

    public static ArrayList<String> key      = new ArrayList<String>();//关键字集合
    public static ArrayList<String> boundarySign = new ArrayList<String>();//界符集合
    public static ArrayList<String> operator  = new ArrayList<String>();//运算符集合

    static {

    }

    int id;           // 单词序号
    String value;      // 单词的值
    String type;       // 单词类型
    int line;          // 单词所在行
    int tokennum;      // 单词token值
    boolean flag = true; //单词是否合法

    public Word() {}

    public Word(int id, String value, String type, int line) {}

    public static boolean isKey(String word) {}

    public static boolean isOperator(String word) {}

    public static boolean isBoundarySign(String word) {}

    // 判断单词是否为算术运算符
    public static boolean isArOP(String word) {}
}

```

图 3-5 Word 类设计

Error 类设计:

```

public class Error {

    int id ;//错误序号;
    String info;//错误信息;
    int line ;//错误所在行
    Word word;//错误的单词

    public Error() {}

    public Error(int id,String info,int line,Word word) {}
}

```

图 3-6 Error 类设计:

关键字定义表如下:

```

//-----关键字
public final static int MAIN    =0;
public final static int PRINTF  =1;
public final static int SCANF   =2;
public final static int STRUCT  =3; //struct
public final static int RETURN  =4; //return
public final static int IF      =5; //if
public final static int ELSE    =6; //else
public final static int DO      =7; //do
public final static int WHILE   =8; //while
public final static int FOR     =9; //for
public final static int VOID    =10;
public final static int INT     =11;
public final static int CHAR    =12;
public final static int BOOL    =13;
public final static int FLOAT   =14;

```

表 3-1 关键字定义表

界符表定义如下：

```
//-----界符
public final static int SEMI    =15;///;
public final static int COMMA   =16;///,
public final static int LMB     =17;///[
public final static int RMB     =18;///]
public final static int LBRA    =19;///(
public final static int RBRA    =20;///)
public final static int LBIGBRA =21;///{
public final static int RBIGBRA =22;///}
public final static int CHARDEF =23;///'
public final static int STRDEF  =24;///"

public final static int ASS      =25;///=
public final static int EQ       =26;///==
public final static int UNEQ     =27;///!=
public final static int BIG      =28;///>
public final static int LESS    =29;///<
public final static int BIGEQ    =30;///>=
public final static int LESSEQ   =31;///<=
public final static int ADD      =32;///+
public final static int SUB      =33;///-
public final static int MUL      =34;///*
public final static int DIV      =35;/// /
public final static int AND      =36;///&&
public final static int OR       =37;///||
public final static int NON      =38;///!
public final static int DADD     =39;///++
public final static int DSUB     =40;///--
public final static int QUES     =41;///?
public final static int BOR      =42;///|
public final static int BAND     =43;///&
public final static int END      =44;///#
```

表 3-2 界符表定义

### 3.2.3 算法

词法分析主要是通过自动机来写的，设计如下：

一个简单识别器（有限自动机）的设计，如图 3-2-3-2 所示：

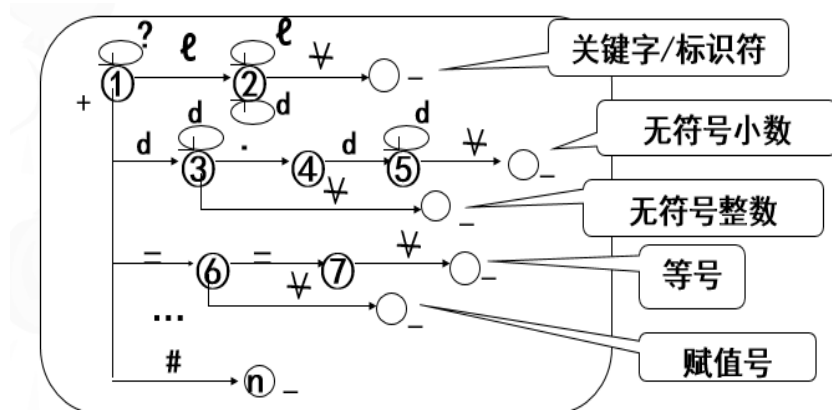


图 3-7

〈字母〉  $\rightarrow A|B|C|\dots|Z|a|b|c|\dots|z$

〈数字〉  $\rightarrow 0|1|2|3|4|5|6|7|8|9$

其中 (1)  $l$  (字母)， $d$  (数字)， $\#$  (源程序结束符)；

(2)  $?$  (空格，回车，换行)，需要过滤掉；

(3)  $\nabla$  (泛指单词的后继符)；

(4)  $\dots$  (表示省略了其他界符的处理)。

一个简单词法分析器设计，如图 3-2-3-2 所示：

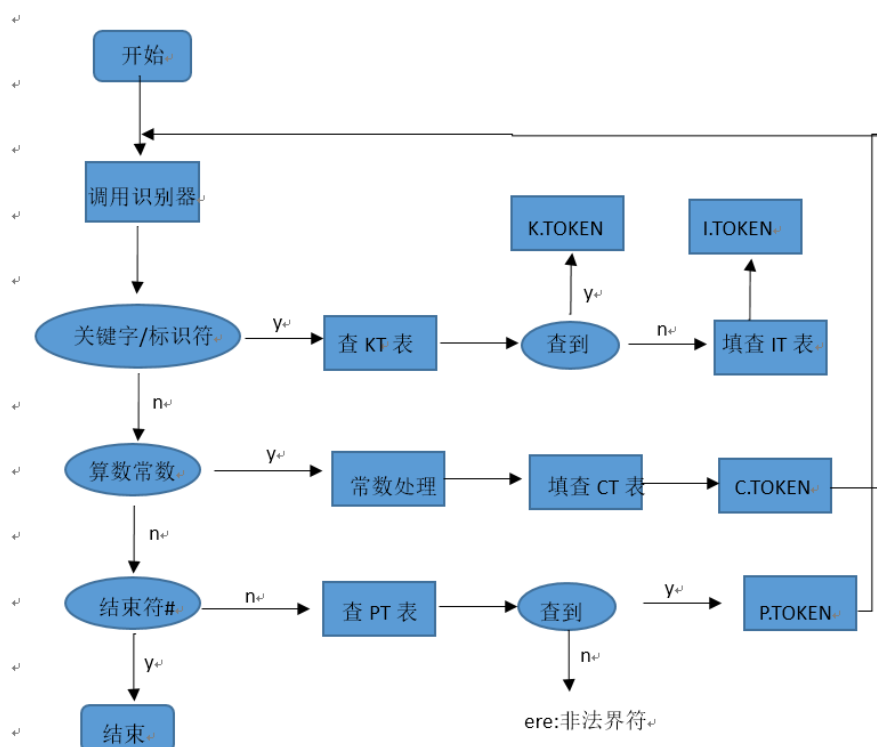


图 3-8

## 3.3LL1 分析法设计语法

### 3.3.1 功能

语法扫描器的功能主要有三：

- (1) 识别一般源程序——检查源程序中是否符合语法规范；
- (2) 实现控制语句的识别以及功能函数等的识别。
- (3) 程序的纠错——对于源程序中出现的语法规范错误进行纠错改正并且提示。

### 3.3.2 数据结构

由于 Parser 类数据结构较为复杂，这里只简单列出成员变量

Parser 类结构设计：

```
public class Parser {  
  
    private LexAnalyse    lexAnalyse;//词法分析器  
    ArrayList<Word>      wordList      =new ArrayList<Word>(); //单词表  
    Stack<AnalyseNode>    analyseStack  =new Stack<AnalyseNode>(); //分析栈  
    Stack<String>          semanticStack =new Stack<String>(); //语义栈  
  
    ArrayList<FourElement> fourElemList =new ArrayList<FourElement>(); //四元式列表  
    ArrayList<Error>       errorList    =new ArrayList<Error>(); //错误信息列表  
  
    ArrayList<pfinfNode>    pfinfTable   =new ArrayList<pfinfNode>(); //函数表  
    ArrayList<typeNode>    typeTable    =new ArrayList<typeNode>(); //类型表  
    ArrayList<SynbolNode>  synbolTable   =new ArrayList<SynbolNode>(); //符号表总表  
    ArrayList<ainfNode>    ainfTable     =new ArrayList<ainfNode>(); //数组表  
    ArrayList<rinfNode>    rinfTable     =new ArrayList<rinfNode>(); //结构体表  
    ArrayList<lengthNode> lengthTable   =new ArrayList<lengthNode>(); //长度表  
    int                    Totaloff      =0; //系统区距  
    boolean                structFlag    =false; //结构体定义@ASS_I=true开始标识符,为真开始定义@TAB_I时=false  
    int                    structNum     =0; //定义的结构体数  
    boolean                arrayFlag     =false; //数组定义@ASS_U'开始定义数组,为真开始定义@TAB_U'时=false结束数组定义  
    String                 arrayType; //数组最小单元的类型//在@ASS_U'时初始化保存下来,在@TAB_U'时复原  
    int                    arrayNum      =0; //定义的数组数  
  
    StringBuffer           bf; //分析栈缓冲流  
    int                    errorCount    =0; //统计错误个数  
    boolean                graErrorFlag  =false; //语法分析出错标志  
    int                    tempCount     =0; //用于生成临时变量  
    int                    fourElemCount =0; //统计四元式个数  
}
```

图 3-9 Parser 类结构设计

其中，除了用到了 Error 类进行错误信息的详细输出以及词法分析阶段用到的 Word 类和 Token 类等之外，因为涉及到语义分析阶段还引入了分析结点类(语义分析时再细说)以及符号表类中的总符号表(标识符结点类)以及函数表(函数结点类)、数组表(数组结点类)、结构体表(结构体结点类)、类型表(类型结点类)等。

AnalyseNode 分析栈节点类结构



```

/**
 * 分析栈节点类
 * String type;//节点类型
 * String name;//节点名
 * Object value;//节点值
 */
public class AnalyseNode {

    public final static String NONTERMINAL = "非终结符";
    public final static String TERMINAL = "终结符";
    public final static String ACTIONSIGN = "动作符";
    public final static String END = "结束符";

    static ArrayList<String> nonterminal = new ArrayList<String>(); //非终结符集合
    static ArrayList<String> actionSign = new ArrayList<String>(); //动作符集合
    static {}

    String type;//节点类型(终结符、非终结符、动作符、结束符)
    String name;//节点名(S、A、B,@开头的动作符号等)//
    String value;//节点值(对应wordList中的单词)

    //非终结符
    public static boolean isNonterm(AnalyseNode node) {}

    //终结符
    public static boolean isTerm(AnalyseNode node) {}

    //语义动作
    public static boolean isActionSign(AnalyseNode node) {}

    public AnalyseNode() {}

    public AnalyseNode(String type, String name, String value) {}
}

```

图 3-10 AnalyseNode 分析栈节点类结构

### SynbolNode 数据结构

```

public class SynbolNode { //标识符总表结点
    String value; //值==>Word.value
    typeNode typenode; //类型结点==>根据type得到对应的类型结点
    String type; //类型==>语义动作时根据语义栈中的声明类型关键字传入整型:int,字符:char,布尔:bool,浮点:float
    int tpoint; //结点对应类型指针,根据类型结点typenode.tpoint得到
    int lengthoff; //所占内存字节数==>typenode.lengthoff得到,数组的是typenode.lengthoff*Clen;结构体的是叠加==>内容填入对应的长度表中
    String cat; //类别(根据位于不同位置的语义动作产生:语句块声明语句一般为变量类型v)
    int offset; //在内存中的偏移量相对于函数基地址(从0开始,通过将长度表中内容相加再加上结点所占字节数lengthoff)得到
    int ADDR; //地址指针====>
    int index=0;
    SynbolNode() {}

    SynbolNode(String value, String type, int offSystem) //传入单词的值,类型,系统数据地址偏移量:0开始{}

    public void setCat(String cat) {} //语义动作时手动设置{}

    public void setLengthoff(int lengthoff) {} //语义动作时手动设置:数组的偏移量,结构体的偏移量{}

    public void setOffsystem(int offSystem) {} //修改标识符的地址{}

    public void setTypeNode(typeNode tn) {} //语义动作时手动设置:数组的偏移量,结构体的偏移量{}

    public boolean equals(Object obj) {}

    public void setIndex(int index) {}
}

```

图 3-11 SynbolNode 数据结构

### 3.3.3 程序生成文法:

这里就不给出在求证是否符合 LL1 文法属性时所求出的 select 集合了，经过验证确实符合文法属性。

```
//-----  
//类C语言简单文法  
//经验证已符合LL1文法属性  
//-----  
//1、主函数语句产生式：  
//S->void main(){A}|int main(){A return 0;}  
//-----  
//2、声明语句产生式：  
//X-> YZ;  
//Y-> int|char|bool|float  
//  
//Z-> UZ'  
//Z' -> ,Z|$|[num]U'  
//U-> idU'  
//U' ->=L|$|[num]  
//  
//I->struct id{A};  
//-----  
//3、赋值语句产生式：  
//R->id=L;  
//-----  
//4、算术运算(逻辑运算)语句产生式：  
//-----  
//L->TL'  
//L' ->+L|-L|$  
//T->FT'  
//T' ->*T|/T|$  
//F->(L)  
//F->id|num  
//Q->idO|$  
//O->++|--  
//-----
```

图 3-12 文法结构

```

//-----
//5、布尔运算语句产生式：
//E->HE'
//E' ->&&E|$
//H->GH'
//H' ->||H
//H' ->$
//G->FDF
//G->(E)
//G->!E
//D-> <|>|==|!=|<=|>=
//-----
//6、控制语句产生式：
//B->if (E) {A} else {A}
//B->while (E) {A}
//B->for (YZ;G;Q) {A}
//-----
//7、功能函数语句产生式：
//B->printf(P);
//B->scanf(id);
//P->id|ch|num|floatum|boolid
//-----
//8、复合语句产生式：
//A->CA
//C->X|B|R|I
//A->$
//-----

```

图 3-13 文法结构

### 3.3.4 算法流程图：

LL(1)分析法是指从左到右扫描、最左推导(LL)和只查看一个当前符号（括号中的 1）之意；LL(1)分析法又称预测分析法，与递归子程序法同属于自顶向下确定性语法分析方法；

LL(1) 分析法的基本要点有三：

- (1) 利用一个分析表，登记如何选择产生式的知识；
- (2) 利用一个分析栈，记录分析过程；
- (3) 此分析法要求文法必须是 LL(1)文法。

这次实验我们用的就是 LL(1)分析法，在程序设计开始前需要求出每个产生式对应的 select 集(篇幅有限，这里不再赘述)，引入栈结构，记录分析过程。一定程度上增加了我们的难度，但我们最终还是实现了。

## 3.4 语义分析四元式以及符号表分析模块

### 3.4.1 功能

语义分析阶段的功能主要有三：

- (1) 通过语法制导在文法中插入语义动作生成中间代码
- (2) 实现生成中间代码以及符号总表
- (3) 程序的纠错——对于源程序中出现的语法规范错误进行纠错改正并且提示。

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，符号表可以存储标识符的各种信息，以便以后做处理。

### 3.4.2 数据结构

由于语义分析是语法制导的，因此所用结构也为 Parser 类，所用的数据结构为栈(语法、语义栈)和链表(用于存储结点信息:符号结点等)。但其中加入了自定义数据结构:分析结点以及符号表类中的总符号表(标识符结点类)以及函数表(函数结点类)、数组表(数组结点类)、结构体表(结构体结点类)、类型表(类型结点类)等。还有用于生成四元式的四元式类。

Parser 类结构设计：

```
public class Parser {  
  
    private LexAnalyse    lexAnalyse; //词法分析器  
    ArrayList<Word>      wordList      =new ArrayList<Word>(); //单词表  
    Stack<AnalyseNode>    analyseStack  =new Stack<AnalyseNode>(); //分析栈  
    Stack<String>         semanticStack =new Stack<String>(); //语义栈  
  
    ArrayList<FourElement> fourElemList =new ArrayList<FourElement>(); //四元式列表  
    ArrayList<Error>       errorList    =new ArrayList<Error>(); //错误信息列表  
  
    ArrayList<pfinfNode>    pfinfTable  =new ArrayList<pfinfNode>(); //函数表  
    ArrayList<typeNode>    typeTable    =new ArrayList<typeNode>(); //类型表  
    ArrayList<SymbolNode>  symbolTable  =new ArrayList<SymbolNode>(); //符号表总表  
    ArrayList<ainfNode>    ainfTable    =new ArrayList<ainfNode>(); //数组表  
    ArrayList<rinfNode>    rinfTable    =new ArrayList<rinfNode>(); //结构体表  
    ArrayList<lengthNode> lengthTable   =new ArrayList<lengthNode>(); //长度表  
    int                    Totaloff      =0; //系统区距  
    boolean                structFlag    =false; //结构体定义@ASS_I=true开始标识符,为真开始定义@TAB_I时=false  
    int                    structNum     =0; //定义的结构体数  
    boolean                arrayFlag     =false; //数组定义@ASS_U*开始定义数组,为真开始定义@TAB_U*时=false结束数组定义  
    String                 arrayType     =0; //数组最小单元的类型//在@ASS_U*时初始化保存下来,在@TAB_U*时复原  
    int                    arrayNum      =0; //定义的数组数  
  
    StringBuffer           bf; //分析栈缓冲流  
    int                    errorCount    =0; //统计错误个数  
    boolean                graErrorFlag  =false; //语法分析出错标志  
    int                    tempCount     =0; //用于生成临时变量  
    int                    fourElemCount =0; //统计四元式个数  
}
```

图 3-14 Parser 类结构设计：

AnalyseNode 分析栈节点类结构

```

/**
 * 分析栈节点类
 * String type;//节点类型
 * String name;//节点名
 * Object value;//节点值
 */
public class AnalyseNode {

    public final static String NONTERMINAL = "非终结符";
    public final static String TERMINAL = "终结符";
    public final static String ACTIONSIGN = "动作符";
    public final static String END = "结束符";

    static ArrayList<String> nonterminal = new ArrayList<String>(); //非终结符集合
    static ArrayList<String> actionSign = new ArrayList<String>(); //动作符集合
    static {}

    String type;//节点类型(终结符、非终结符、动作符、结束符)
    String name;//节点名(S、A、B,@开头的动作符号等)//
    String value;//节点值(对应wordList中的单词)

    //非终结符
    public static boolean isNonterm(AnalyseNode node) {}

    //终结符
    public static boolean isTerm(AnalyseNode node) {}

    //语义动作
    public static boolean isActionSign(AnalyseNode node) {}

    public AnalyseNode() {}

    public AnalyseNode(String type, String name, String value) {}

}

```

图 3-15 AnalyseNode 分析栈节点类结构

## SynbolNode 数据结构

```

public class SynbolNode { //标识符总表结点
    String value; //值==>Word.value
    typeNode typenode; //类型结点==>根据type得到对应的类型结点
    String type; //类型==>语义动作时根据语义栈中的声明类型关键字传入整型:int,字符:char,布尔:bool,浮点:float
    int tpoint; //结点对应类型指针,根据类型结点typenode.tpoint得到
    int lengthoff; //所占内存字节数==>typenode.lengthoff得到,数组的是typenode.lengthoff*Clen;结构体的是叠加==>内容填入对应的长度表中
    String cat; //类别(根据位于不同位置的语义动作产生:语句块声明语句一般为变量类型v)
    int offset; //在内存中的偏移量相对于函数基地址(从0开始,通过将长度表中内容相加再加上结点所占字节数lengthoff)得到
    int ADDR; //地址指针====>
    int index=0;
    SynbolNode() {}

    SynbolNode(String value, String type, int offSystem) //传入单词的值,类型,系统数据地址偏移量:0开始{}

    public void setCat(String cat) {} //语义动作时手动设置{}

    public void setLengthoff(int lengthoff) {} //语义动作时手动设置:数组的偏移量,结构体的偏移量{}

    public void setOffsystem(int offSystem) {} //修改标识符的地址{}

    public void setTypeNode(typeNode tn) {} //语义动作时手动设置:数组的偏移量,结构体的偏移量{}

    public boolean equals(Object obj) {}

    public void setIndex(int index) {}

}

```

图 3-16 SynbolNode 数据结构

## typeNode 数据结构

```

class typeNode{//类型表结点
    String type;          //类型==>语义动作时根据语义栈中的声明类型关键字传入整型:int,字符:char,布尔:bool,浮点:float
    int tpoint;           //类型指针==>根据type信息得到整型:0,字符:0,布尔:0,浮点:0
    int lengthoff;        //所占内存字节数==>根据type信息得到整型:4,字符:1,布尔:1,浮点:8
    int index=0;
    typeNode(){}
    typeNode(String type){}

    public void setlengthoff(int off){}

    public void setIndex(int index){}

    public boolean equals(Object obj){}
}

```

图 3-17 typeNode 数据结构

### ainfNode 数组结点数据结构

```

class ainfNode{//数组表结点
    int low=0;
    int up;//数组长度下标-1
    SynbolNode type;//数组结点(存着数组结点的名字value)
    String value;
    String typeValue;//值单元类型
    int tpoint;//值单元对应的指针下标
    int offsize;//值单元总长度:成分类型

    int clen;//值单元个数

    int index =0;
    ainfNode(){}

    ainfNode(int up,SynbolNode ty){//用上界初始化,以及结点的类型{}

    public void setPreDiemension(String tv,int tp,int off){//修改上一维的成分类型为数组,{}

    public boolean equals(Object obj)//根据名字找到同一个数组{}
    public void setIndex(int index){}
}

```

图 3-19ainfNode 数组结点数据结构

### rinfNode 数据结构

```

class rinfNode{//结构体结点
    String value;//结构体名,填符号表时赋给value
    int index=0;

    Vector<String> off_value;//结构体内变量的名字
    Vector<Integer> off_n;//结构体内变量的偏移值(相对于结构体而言)
    Vector<SynbolNode> type_n;//结构体内变量结点
    Vector<Integer> type_n_tpoint;//结点类型==>根据type_n.typenode.tpoint得到

    int offrinf=0;//结构体所占字节数,填符号表时直接赋给lengthoff

    rinfNode(String v){
        this.value =v;
        this.off_value =new Vector<String>();//结构体内变量的名字
        this.off_n =new Vector<Integer>();//结构体内变量的偏移值(相对于结构体而言)
        this.type_n =new Vector<SynbolNode>();//结构体内变量结点
        this.type_n_tpoint =new Vector<Integer>();//结点类型==>根据type_n.typenode.tpoint得到
    }

    public void putElement(int off,SynbolNode type){//语义动作时往结构体结点里面添加符号结点{}

    public void setIndex(int index){}
}

```

图 3-20rinfNode 数据结构

lengthNode 数据结构

```
//长度表结点
class lengthNode{
    int offset;
    lengthNode() {}
    lengthNode(int offset) {}
}
```

图 3-21lengthNode 数据结构

pfinfNode 数据结构

```
//函数表
class pfinfNode{
    public int level=0;
    public int off=0;
    public int fn;
    public int entry=0;
    public ArrayList<SynbolNode> param=new ArrayList<SynbolNode>();
    public void setfn() {}
    pfinfNode() {}
}
```

图 3-22pfinfNode 数据结构

FourElement 数据结构

```
public class FourElement {

    int id;//四元式序号，为编程方便
    String op;//操作符
    String arg1;//第一个操作数
    String arg2;//第二个操作数
    Object result;//结果

    public FourElement() {}

    public FourElement(int id,String op,String arg1,String arg2,String result) {}

}
```

图 3-23FourElement 数据结构

### 3.4.3 算法

对之前的文法进行语法制导翻译后的翻译文法如下：

```

//构造LL1属性翻译文法
//-----
//构造LL1属性翻译文法即在原有LL1文法基础上加上动作符号
//并给非终结符和终结符加上一定属性，给动作符号加上语义子程序。
//对原有LL1文法改进的地方如下：
//-----

//0、声明语句(初始化)
//产生式          ////语义子程序
//-----
//X-> YZ;@INIT_XOFFSET
//Y-> @ASS_Y int|char|bool|float
////@INIT_XOFFSET{offset+typenode.lengthoff}
////@ASS_Y{type=将firstWord.value压入语义栈,}

//Z-> UZ'
//Z'-> ,@ASS_Y Z|$ |[@ASS_U'@TAB_U num]U'@TAB_U'
////@TAB_U{arrayFlag=true开始定义多维数组}
////@TAB_U' {arrayFlag=false 数组定义完毕}@TAB_U' {根据语义栈中内容生成数组符号表，弹语义栈}
////@ASS_U' {U.VAL=num并压入语义栈:数组的简单声明}

//U->@ASS_U id U'
//U'->=L @EQ|$|[@ASS_U' num]
////@ASS_U {U.VAL=id并压入语义栈}
////@EQ {RES=U.VAL, OP='=', ARG1=L.VAL, new fourElement(OP, ARG1, _, RES)} //如果U'->$则不用执行语义动作

//I->struct @ASS_I id{ A } @TAB_I;
////@ASS_I {填结构体表, I.VAL=id存入语义栈，将总的offset压入offset栈中，置offset=0;}
////@TAB_I {填符号表}
//-----

```

图 3-24 属性文法翻译

```

//1、 赋值：
//产生式          ////语义子程序
//-----
//R->@ASS_R id =L;
////@ASS_R {R.VAL=id并压入语义栈}
////@EQ {RES=R.VAL, OP='=', ARG1=L.VAL, new fourElement(OP, ARG1, _, RES)}
//-----

//2、 算术运算：
//产生式          ////语义子程序
//-----
//L->TL' @ADD_SUB      ////@ADD_SUB {If (OP!=null) RES= NEWTEMP; L.VAL=RES, 并压入语义栈; New fourElement(OP, T.VAL, L'.VAL, RES);}
//L'->+L @ADD          ////@ADD {OP=+, ARG2=L.VAL}
//L'->-L @SUB           ////@SUB {OP=-, ARG2=L.VAL}
//L'->$
//T->FT' @DIV_MUL      ////@DIV_MUL { if (OP !=null) RES= NEWTEMP; T.VAL=RES; new FourElement(OP, F.VAL, ARG2, RES) else ARG1=F.VAL;}
//T'->/T @DIV           ////@DIV {OP=/, ARG2=T.VAL}
//T'->*T @MUL           ////@MUL {OP=*, ARG2=T.VAL}
//T'->$
//F->(L) @TRAN_LF      ////@TRAN_LF {F.VAL->L.VAL}
//F->@ASS_F num|id      ////@ASS_F {F.VAL=num|id}

//Q->id O|$
//O->@SINGLE_OP ++ | -- ////@SINGLE_OP {OP=++|--}
//-----

```

图 3-25 属性文法翻译



```

//3、 布尔运算
//产生式
//-----
//G->FDF@COMPARE
//D->@COMPARE_OP<|>|=|!<|=|>=
////@COMPARE{OP=D.VAL;ARG1=F(1).VAL;ARG2=F(2).VAL;RES=NEWTEMP; New fourElement(OP,F.VAL,ARG2, RES );G.VAL=RES并压入语义栈}
////@COMPARE_OP{D.VAL=<|>|=|!<|=|>=,并压入语义栈}
//-----

//4、 控制语句
//产生式
//-----
//if-else语句产生式
//-----
//B->if(E)@IF_HEAD @IF_FJ {A} @IF_BACKPATCH_FJ B' else@IF_EL@IFEL_FJ {A} @IFEL_BACKPATCH_FJ B' @IF_END
//B'->§|else @IF_EL@IFEL_FJ {A} @IFEL_BACKPATCH_FJ @IF_END

////@IF_HEAD{OP="if";ARG1=G.VAL;NEW fourElement(OP,ARG1,_,_ ),将其插入到四元式列表中第i个, 弹栈}
////@IF_FJ{OP="FJ";RES=if_fj, New fourElement(OP,_,_, RES ),将其插入到四元式列表中第i个}
////顺序执行//文法加推导变换后待验证,if语句不用真跳
////@IF_BACKPATCH_FJ{回填前面假出口跳转四元式的跳转序号, BACKPATCH (i,if_fj)}
////@IF_EL{OP="el";NEW fourElement(OP,_,_,_ ),将其插入到四元式列表中第i个}
////@IFEL_FJ{OP="ELFJ";ARG1=G.VAL;RES=ifel_fj, New fourElement(OP,ARG1,_, RES ),将其插入到四元式列表中第i个,弹栈}
////顺序执行
////@IFEL_BACKPATCH_FJ{回填前面假出口跳转四元式的跳转序号, BACKPATCH (i,ifel_fj)}//可能和if_fj公用
////@IF_END{OP="end";NEW fourElement(OP,_,_,_ ),将其插入到四元式列表中第i个}
//-----

```

图 3-26 属性文法翻译

```

//-----
//while语句产生式
//-----
//B->while @WHILE_HEAD (G) @DO @WHILE_FJ {A}@WHILE_RJ @WHILE_BACKPATCH_FJ @WHILE_END

////@WHILE_HEAD{OP="wh";NEW fourElement(OP,_,_,_ ),将其插入到四元式列表中第i个,将i存入wh_rj, 存好真跳点的值i}
////处理(G)
////@DO{OP="do";ARG1=G.VAL; New fourElement(OP,ARG1,_,_ ),将其插入到四元式列表中第i个,弹栈}
////@WHILE_FJ{OP="wh_fj";RES=wh_fj, New fourElement(OP,_,_, RES ),将其插入到四元式列表中第i个}
////顺序执行
////@WHILE_RJ{OP="Rj";ARG1=G.VAL;RES=wh_rj, New fourElement(OP,ARG1,_, RES ),将其插入到四元式列表中第i个}
////@WHILE_BACKPATCH_FJ{回填前面假出口跳转四元式的跳转序号, BACKPATCH (i,wh_fj)}
////@WHILE_END{OP="we";NEW fourElement(OP,_,_,_ ),将其插入到四元式列表中第i个}
//-----

//-----
//for语句产生式
//-----
//B->for @FOR_HEAD (YZ;@FOR_LINE_RJ G @FOR_FJ;Q){A}@FOR_RJ @SINGLE @FOR_BACKPATCH_FJ @FOR_END

////@FOR_HEAD{OP="for";NEW fourElement(OP,_,_,_ ),将其插入到四元式列表中第i个}
////赋值操作
////存好真跳点@FOR_LINE_RJ{}
////逻辑运算操作
////@FOR_FJ{OP="for_fj";ARG1=G.VAL;RES=for_fj, New fourElement(OP,ARG1,_, RES ),将其插入到四元式列表中第i个}
////顺序操作
////@FOR_RJ
////@SINGLE{ARG1=id;RES=NEWTEMP;New fourElement(OP,ARG1,_,RES)}
////@FOR_END{OP="fe";NEW fourElement(OP,_,_,_ ),将其插入到四元式列表中第i个}
//-----

```

图 3-27 属性文法翻译

```

//-----
//说明:
// (1):R.VAL表示符号R的值, VAL是R的一个属性, 其它类似。
// (2):NEWTEMP() 函数: 每调用一次生成一个临时变量, 依次为T1,T2,...,Tn。
// (3):BACKPATCH (int i,int res):回填函数, 用res回填第i个四元式的跳转地址。
// (4):new fourElement(String OP,String ARG1,String ARG2,String RES):生成一个四元式(OP,ARG1,ARG2,RES)
//-----

```

图 3-28 属性文法翻译

符号表的建立是在程序的声明部分，或者是在函数的声明部分，因此我们只要对声明部分进行处理就可以了。

## 3.5 目标代码生成模块

### 3.5.1 功能

根据语义分析得到的四元式结合符号表生成对应目标机器的目标代码，由于水平限制我们只生成了简单的目标代码并未对其进行优化，不够完美。

### 3.5.2 数据结构

这里我设计了一个单独的类封装目标代码的生成，主要用的数据结构是链表，其中封装了之前所用自定义的四元式以及符号表类数据结构。

```
import java.awt.BorderLayout;

public class huibian extends JFrame {

    private static final long serialVersionUID = 8766059377195109228L;
    private static String title;
    private static String fileName;

    private static TextArea text;

    public huibian() {}

    public huibian(String title,String fileName){

    private void init() {}

    private Container createContentPane() {}

    private String readFile(String filename) {}

    public static String getTitl() {}

    public static void setTitl(String title) {}

    public static String getFileName() {}

    public static void setFileName(String fileName) {}

    public static TextArea getText() {}

    public static void setText(TextArea jText) {}

    public static void main(String[] args) {}

}
```

图 3-29huibian 类结构设计

### 3.5.3 算法

利用 LL(1) 分析法语法制导翻译实现语义分析后，根据对应的目标机器生成对应的机器代码(目标代码)。它是指从左到右扫描、最左推导(LL)和只查看一个当前符号(括号中的 1)之意；LL(1) 分析法又称预测分析法，与递归子程序法同属于自顶向下确定性语法分析方法；

LL(1) 分析法的基本要点有三：

- (1) 利用一个分析表，登记如何选择产生式的知识；
- (2) 利用一个分析栈，记录分析过程；
- (3) 此分析法要求文法必须是 LL(1) 文法。

这次实验我们用的就是 LL(1) 分析法，在程序设计开始前需要求出每个产生式对应的 select 集(篇幅有限，这里不再赘述)，引入栈结构，记录分析过程。一定程度上增加了我们的难度，但我们最终还是实现了。

## 4 程序设计与实现

### 4.1 程序流程图

#### 4.1.1 程序流程图

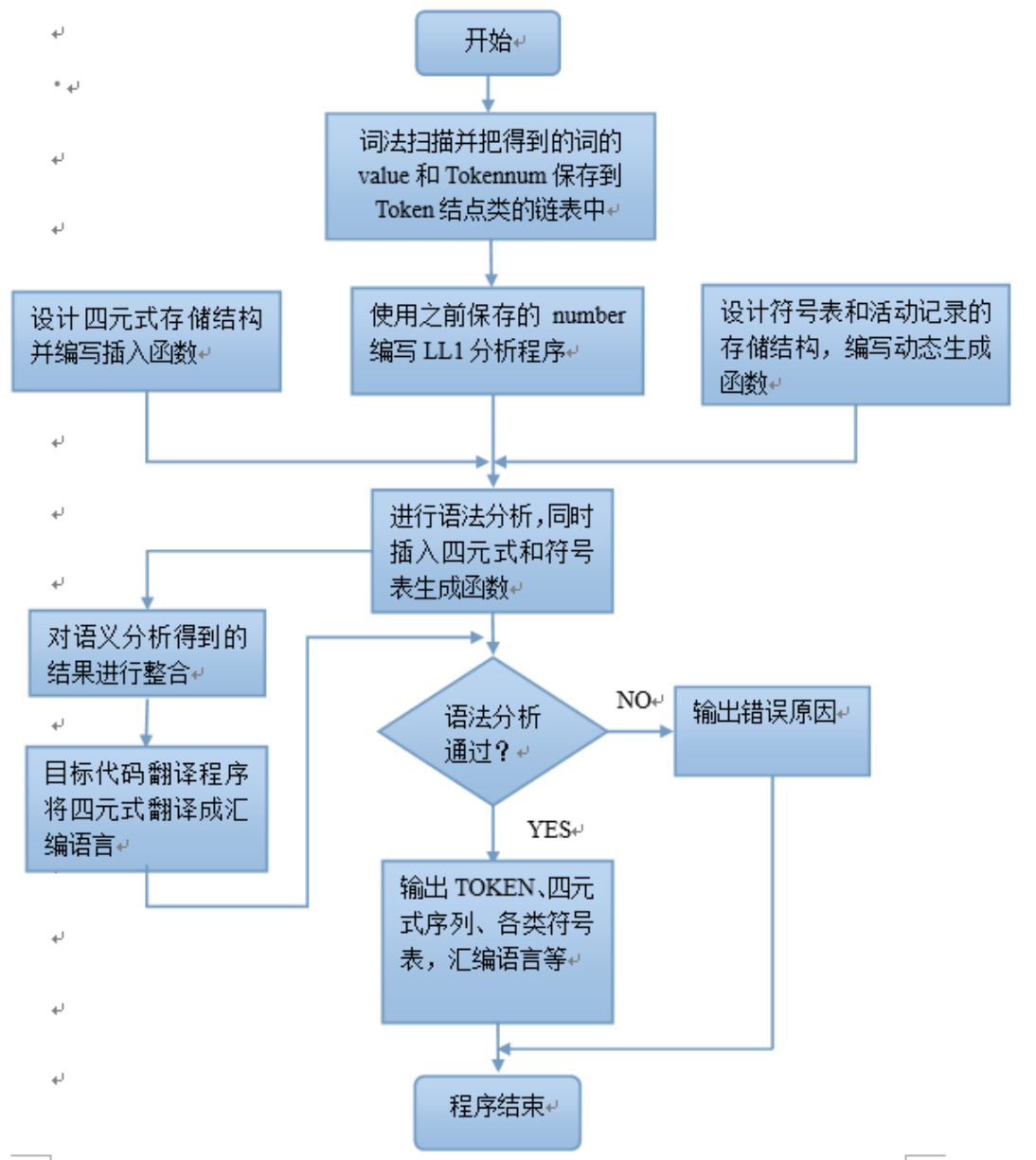


图 4-0 程序流程图

程序设计了可视化界面，按下不同的按钮可以解析输入(可以选择文本输入，也可以直接在文本输入窗口输入)进行输出不同的结果。



```

//-----
//类C语言简单文法
//经验证已符合LL1文法属性
//-----
//1、主函数语句产生式：
//S->void main(){A}|int main(){A return 0;}
//-----
//2、声明语句产生式：
//X-> YZ;
//Y-> int|char|bool|float
//
//Z-> UZ'
//Z' -> ,Z| $\$$ ||[num]U'
//U-> idU'
//U' -> =L| $\$$ |[num]
//
//I->struct id{A};
//-----
//3、赋值语句产生式：
//R->id=L;
//-----
//4、算术运算(逻辑运算)语句产生式：
//-----
//L->TL'
//L' -> +L|-L| $\$$ 
//T->FT'
//T' -> *T|/T| $\$$ 
//F->(L)
//F->id|num
//Q->idO| $\$$ 
//O->++|--
//-----

```

图 4-3 文法的设计

```

//-----
//5、布尔运算语句产生式：
//E->HE'
//E' ->&&E|$
//H->GH'
//H' ->||H
//H' ->$
//G->FDF
//G->(E)
//G->!E
//D-> <|>|==|!=|<=|>=
//-----
//6、控制语句产生式：
//B->if (E) {A} else {A}
//B->while (E) {A}
//B->for (YZ;G;Q) {A}
//-----
//7、功能函数语句产生式：
//B->printf(P);
//B->scanf(id);
//P->id|ch|num|floatum|boolid
//-----
//8、复合语句产生式：
//A->CA
//C->X|B|R|I
//A->$
//-----

```

图 4-4 文法的设计

#### 4.1.4 翻译文法的设计

```

//构造LL1属性翻译文法
//-----
//构造LL1属性翻译文法即在原有LL1文法基础上加上动作符号
//并给非终结符和终结符加上一定属性，给动作符号加上语义子程序。
//对原有LL1文法改进的地方如下：
//-----

//0、声明语句(初始化)
//产生式          ////语义子程序
//-----
//X-> YZ;@INIT_XOFFSET
//Y-> @ASS_Y int|char|bool|float
////@INIT_XOFFSET{offset+typenode.lengthoff}
////@ASS_Y{type=将firstWord.value压入语义栈,}

//Z-> UZ'
//Z'-> ,@ASS_Y Z|$ |[@ASS_U'@TAB_U num]U'@TAB_U'
////@TAB_U{arrayFlag=true开始定义多维数组}
////@TAB_U' {arrayFlag=false 数组定义完毕}@TAB_U' {根据语义栈中内容生成数组符号表，弹语义栈}
////@ASS_U' {U.VAL=num并压入语义栈:数组的简单声明}

//U->@ASS_U id U'
//U'->=L @EQ|$|[@ASS_U' num]
////@ASS_U{U.VAL=id并压入语义栈}
////@EQ{RES=U.VAL,OP='=',ARG1=L.VAL,new fourElement(OP,ARG1,_, RES)}//如果U'->$则不用执行语义动作

//I->struct @ASS_I id{ A } @TAB_I;
////@ASS_I{填结构体表,I.VAL=id存入语义栈，将总的offset压入offset栈中，置offset=0;}
////@TAB_I{填符号表}
//-----

```

图 4-5 翻译文法的设计

```

//1、 赋值：
//产生式          ////语义子程序
//-----
//R->@ASS_R id =L;
////@ASS_R{R.VAL=id并压入语义栈}
////@EQ{RES=R.VAL,OP='=',ARG1=L.VAL,new fourElement(OP,ARG1,_, RES)}
//-----

//2、 算术运算：
//产生式          ////语义子程序
//-----
//L->@L' @ADD_SUB          ////@ADD_SUB{if(OP!=null) RES= NEWTEMP; L.VAL=RES,并压入语义栈;New fourElement(OP, T.VAL, L'VAL, RES);}
//L' ->+L@ADD              ////@ADD{OP=+,ARG2=L.VAL}
//L' ->-L@SUB               ////@SUB{OP=-,ARG2=L.VAL}
//L' ->$
//T->@FT' @DIV_MUL          ////@DIV_MUL{ if (OP !=null) RES= NEWTEMP;T.VAL=RES; new FourElement(OP,F.VAL,ARG2, RES)else ARG1=F.VAL;}
//T' ->/T@DIV               ////@DIV{OP=/,ARG2=T.VAL}
//T' ->*T@MUL               ////@MUL{OP=*,ARG2=T.VAL}
//T' ->$
//F->(L)@TRAN_LF            ////@TRAN_LF{F.VAL->L.VAL}
//F->@ASS_F num|id          ////@ASS_F{F.VAL=num|id}

//Q->id O|$
//O->@SINGLE_OP ++ | -- ////@SINGLE_OP{OP=++|--}
//-----

```

图 4-6 翻译文法的设计



```

//3、 布尔运算
//产生式          ///语义子程序
//-----
//G->FDF@COMPARE
//D->@COMPARE_OP<|>|=|!<|=|>=
////@COMPARE{OP=D.VAL;ARG1=F(1).VAL;ARG2=F(2).VAL;RES=NEWTEMP; New fourElement(OP,F.VAL,ARG2, RES );G.VAL=RES并压入语义栈}
////@COMPARE_OP{D.VAL=<|>|=|!<|=|>=,并压入语栈}
//-----

//4、 控制语句
//产生式          ///语义子程序
//-----
//if-else语句产生式          ///语义子程序
//-----
//B->if(E)@IF_HEAD @IF_FJ {A} @IF_BACKPATCH_FJ B' else@IF_EL@IFEL_FJ {A} @IFEL_BACKPATCH_FJ B' @IF_END
//B'->*>|else @IF_EL@IFEL_FJ {A} @IFEL_BACKPATCH_FJ @IF_END

////@IF_HEAD{OP="if";ARG1=G.VAL;NEW fourElement(OP,ARG1,_,_ ),将其插入到四元式列表中第i个, 弹栈}
////@IF_FJ{OP="FJ";RES=if_fj, New fourElement(OP,_,_, RES ),将其插入到四元式列表中第i个}
////顺序执行//文法加推导变换后待验证,if语句不用真跳
////@IF_BACKPATCH_FJ{回填前面假出口跳转四元式的跳转序号, BACKPATCH (i,if_fj)}
////@IF_EL{OP="el";NEW fourElement(OP,_,_,_ ),将其插入到四元式列表中第i个}
////@IFEL_FJ{OP="ELFJ";ARG1=G.VAL;RES=ifel_fj, New fourElement(OP,ARG1,_, RES ),将其插入到四元式列表中第i个,弹栈}
////顺序执行
////@IFEL_BACKPATCH_FJ{回填前面假出口跳转四元式的跳转序号, BACKPATCH (i,ifel_fj)}//可能和if_fj公用
////@IF_END{OP="end";NEW fourElement(OP,_,_,_ ),将其插入到四元式列表中第i个}
//-----

```

图 4-7 翻译文法的设计

```

//-----
//while语句产生式          ///语义子程序
//-----
//B->while @WHILE_HEAD (G) @DO @WHILE_FJ {A}@WHILE_RJ @WHILE_BACKPATCH_FJ @WHILE_END

////@WHILE_HEAD{OP="wh";NEW fourElement(OP,_,_,_ ),将其插入到四元式列表中第i个,将i存入wh_rj, 存好真跳点的值i}
////处理(G)
////@DO{OP="do";ARG1=G.VAL; New fourElement(OP,ARG1,_,_ ),将其插入到四元式列表中第i个,弹栈}
////@WHILE_FJ{OP="wh_fj";RES=wh_fj, New fourElement(OP,_,_, RES ),将其插入到四元式列表中第i个}
////顺序执行
////@WHILE_RJ{OP="RJ";ARG1=G.VAL;RES=wh_rj, New fourElement(OP,ARG1,_, RES ),将其插入到四元式列表中第i个}
////@WHILE_BACKPATCH_FJ{回填前面假出口跳转四元式的跳转序号, BACKPATCH (i,wh_fj)}
////@WHILE_END{OP="we";NEW fourElement(OP,_,_,_ ),将其插入到四元式列表中第i个}
//-----

//-----
//for语句产生式          ///语义子程序
//-----
//B->for @FOR_HEAD (YZ:@FOR_LINE_RJ G @FOR_FJ;Q){A}@FOR_RJ @SINGLE @FOR_BACKPATCH_FJ @FOR_END

////@FOR_HEAD{OP="for";NEW fourElement(OP,_,_,_ ),将其插入到四元式列表中第i个}
////赋值操作
////存好真跳点@FOR_LINE_RJ{}
////逻辑运算操作
////@FOR_FJ{OP="for_fj";ARG1=G.VAL;RES=for_fj, New fourElement(OP,ARG1,_, RES ),将其插入到四元式列表中第i个}
////顺序操作
////@FOR_RJ
////@SINGLE{ARG1=id;RES=NEWTEMP;New fourElement(OP,ARG1,_,RES)}
////@FOR_END{OP="fe";NEW fourElement(OP,_,_,_ ),将其插入到四元式列表中第i个}
//-----

```

图 4-8 翻译文法的设计

```

//-----
//说明:
// (1):R.VAL表示符号R的值, VAL是R的一个属性, 其它类似。
// (2):NEWTEMP() 函数: 每调用一次生成一个临时变量, 依次为T1,T2,...,Tn。
// (3):BACKPATCH (int i,int res):回填函数, 用res回填第i个四元式的跳转地址。
// (4):new fourElement(String OP,String ARG1,String ARG2,String RES):生成一个四元式(OP,ARG1,ARG2,RES)
//-----

```

图 4-9 翻译文法的设计

## 4.2 程序说明

### 4.2.1 程序说明

说明:

- a. 按照功能，分不同功能，分为 2 个 package, 分别为 com.compiler(编译器包)、com.gui(显示 ui 包)
- b. 每个 package 中可能包含若干个 Class, 实现具体的功能, 组织结构如图 4-2-1 所示。

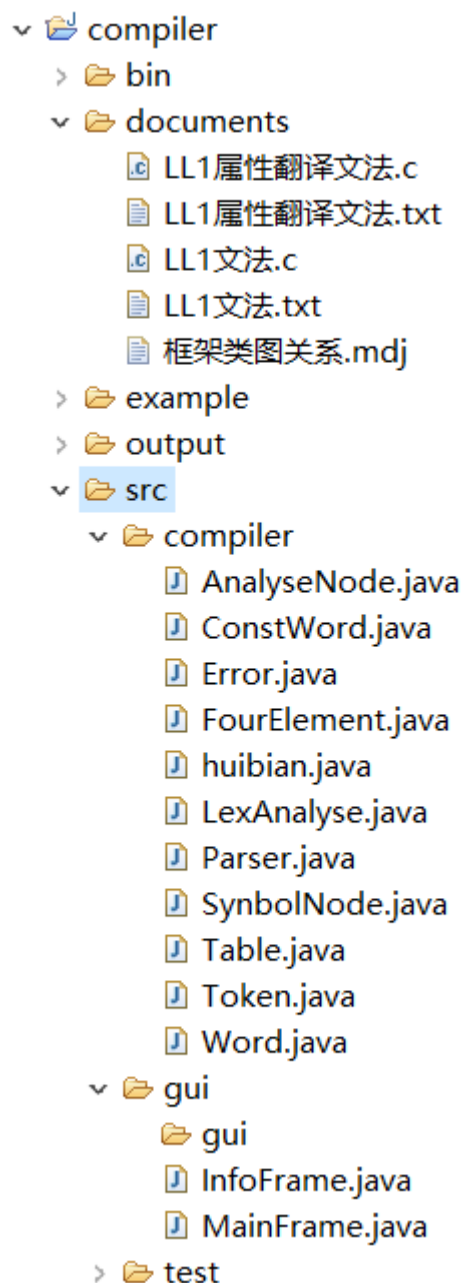


图 4-10

import 包省去，仅显示 package 部分，以显示组织结构

#### 4.2.2 程序源代码

由于篇幅受限，这里暂时不展示出来。

## 4.3 实验结果

### 4.3.1 测试用例

```
1 int main()  
2 {  
3     int a=0,b=1,c=2;  
4     char g[20];  
5     char d='d';  
6     float e=23.1;  
7     struct i{  
8         int j=0;  
9         int k=4;  
10    };  
11    int f[10][20];  
12    struct h{  
13        int x=0;  
14        char y='y';  
15        float z=3.23;  
16    };  
17  
18    if(a>b)  
19    {  
20        a=c;  
21    }  
22    else  
23    {  
24        b=c;  
25    }  
26  
27    for(int i=0;i<c;i++)  
28    {  
29        i=i+1;  
30    }  
31  
32    while(a>b)  
33    {  
34        a=c;  
35    }  
36  
37    return 0;  
38 }  
39 #
```

图 4-11 测试用例

### 4.3.2 用例语法正确时结果

主界面：

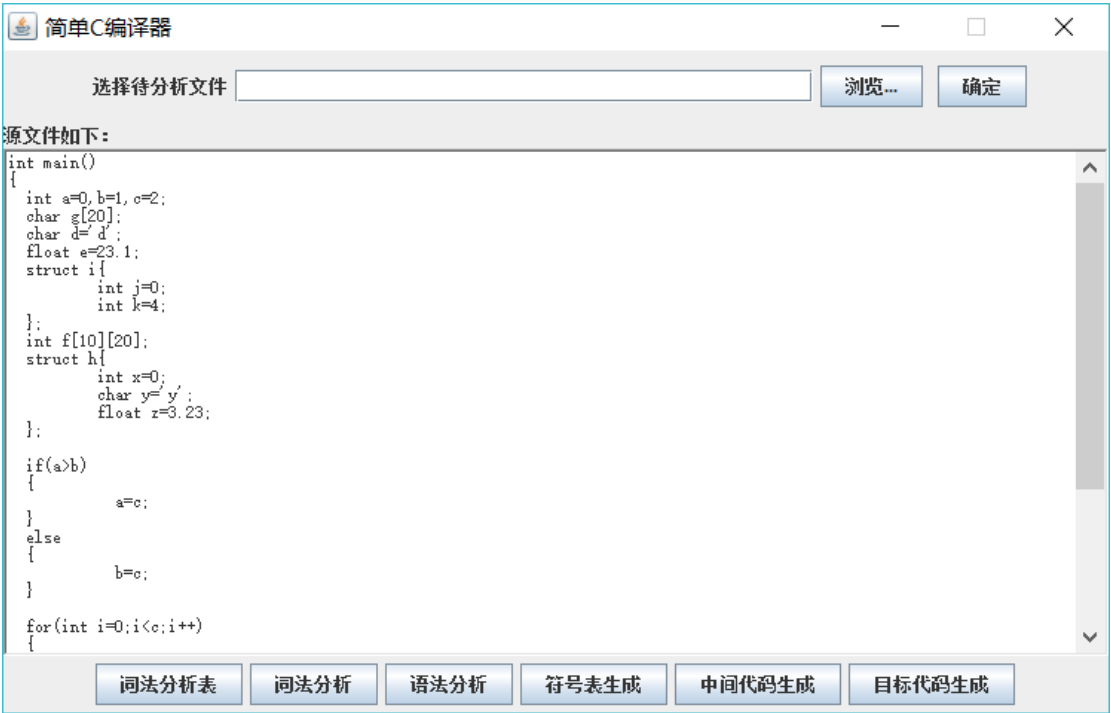


图 4-12 主界面

词法分析表：

词法分析表		
序号	Token 单词	对应码值
1	int	11
2	main	0
3	(	19
4	)	20
5	{	21
6	int	11
7	a	45
8	=	25
9	0	46
10	,	16
11	b	45
12	=	25
13	1	46
14	,	16
15	c	45
16	=	25
17	2	46
18	;	15
19	char	12
20	g	45
21	[	17
22	20	46
23	]	18
24	;	15
25	char	12
26	d	45
27	=	25

图 4-13Token

词法分析表	
KT关键字表	
序号	单词
1	int
2	main
3	char
4	float
5	struct
6	if
7	else
8	for
9	while
10	return
PT界符号表	
序号	单词
1	(
2	)
3	{
4	=
5	,
6	;
7	[
8	]
9	}
10	>
11	<
12	++
13	+
14	#

图 4-14Token

CT常数表	
序号	单词
1	0
2	1
3	2
4	20
5	23.1
6	4
7	10
8	3.23
IT标识符表	
序号	单词
1	a
2	b
3	c
4	g
5	d
6	e
7	i
8	j
9	k
10	f
11	h
12	x
13	y
14	z

图 4-15Token

词法分析：

# 词法分析

单词序号	单词的值	单词类型	单词所在行	单词是否合法
1	int	关键字	1	true
2	main	关键字	1	true
3	(	左括号	1	true
4	)	右括号	1	true
5	{	左大括号	2	true
6	int	关键字	3	true
7	a	标识符	3	true
8	=	赋值符	3	true
9	0	常量	3	true
10	,	逗号	3	true
11	b	标识符	3	true
12	=	赋值符	3	true
13	1	常量	3	true
14	,	逗号	3	true
15	c	标识符	3	true
16	=	赋值符	3	true
17	2	常量	3	true
18	;	分号	3	true
19	char	关键字	4	true
20	g	标识符	4	true
21	[	左方括号	4	true
22	20	常量	4	true
23	]	右方括号	4	true
24	;	分号	4	true
25	char	关键字	5	true
26	d	标识符	5	true
27	=	赋值符	5	true
28	'd'	字符常量	5	true
29	;	分号	5	true
30	float	关键字	6	true
31	e	标识符	6	true
32	=	赋值符	6	true
33	23.1	浮点常量	6	true
34	;	分号	6	true
35	struct	关键字	7	true
36	i	标识符	7	true
37	{	左大括号	7	true
38	int	关键字	8	true
39	j	标识符	8	true
40	=	赋值符	8	true
41	0	常量	8	true
42	;	分号	8	true
43	int	关键字	9	true
44	k	标识符	9	true
45	=	赋值符	9	true
46	4	常量	9	true
47	;	分号	9	true
48	}	右大括号	10	true
49	;	分号	10	true
50	int	关键字	11	true
51	f	标识符	11	true
52	[	左方括号	11	true
53	10	常量	11	true
54	]	右方括号	11	true
55	[	左方括号	11	true
56	20	常量	11	true
57	]	右方括号	11	true
58	;	分号	11	true
59	struct	关键字	12	true
60	h	标识符	12	true
61	{	左大括号	12	true
62	int	关键字	13	true
63	x	标识符	13	true
64	=	赋值符	13	true
65	0	常量	13	true
66	;	分号	13	true
67	char	关键字	14	true
68	y	标识符	14	true
69	=	赋值符	14	true
70	'y'	字符常量	14	true
71	;	分号	14	true
72	float	关键字	15	true
73	z	标识符	15	true
74	=	赋值符	15	true
75	3.23	浮点常量	15	true
76	;	分号	15	true
77	}	右大括号	16	true
78	;	分号	16	true
79	if	关键字	18	true
80	(	左括号	18	true



图 4-16 测试截图

## 词法分析

57	]	单符	11	true
58	:	单符	11	true
59	struct	关键字	12	true
60	{	单符	12	true
61	{	单符	12	true
62	int	关键字	13	true
63	x	变量	13	true
64	=	运算符	13	true
65	0	常量	13	true
66	:	单符	13	true
67	char	关键字	14	true
68	y	变量	14	true
69	=	运算符	14	true
70	'y'	常量	14	true
71	:	单符	14	true
72	float	关键字	15	true
73	z	变量	15	true
74	=	运算符	15	true
75	3.23	常量	15	true
76	:	单符	15	true
77	}	单符	16	true
78	:	单符	16	true
79	if	关键字	18	true
80	(	单符	18	true
81	a	变量	18	true
82	>	运算符	18	true
83	b	变量	18	true
84	)	单符	18	true
85	{	单符	19	true
86	a	变量	20	true
87	=	运算符	20	true
88	c	变量	20	true
89	:	单符	20	true
90	}	单符	21	true
91	else	关键字	22	true
92	{	单符	23	true
93	b	变量	24	true
94	=	运算符	24	true
95	c	变量	24	true
96	:	单符	24	true
97	}	单符	25	true
98	for	关键字	27	true
99	(	单符	27	true
100	int	关键字	27	true
101	i	变量	27	true
102	=	运算符	27	true
103	0	常量	27	true
104	:	单符	27	true
105	i	变量	27	true
106	<	运算符	27	true
107	c	变量	27	true
108	:	单符	27	true
109	i	变量	27	true
110	++	运算符	27	true
111	)	单符	27	true
112	{	单符	28	true
113	i	变量	29	true
114	=	运算符	29	true
115	i	变量	29	true
116	+	运算符	29	true
117	1	常量	29	true
118	:	单符	29	true
119	}	单符	30	true
120	while	关键字	32	true
121	(	单符	32	true
122	a	变量	32	true
123	>	运算符	32	true
124	b	变量	32	true
125	)	单符	32	true
126	{	单符	33	true
127	a	变量	34	true
128	=	运算符	34	true
129	c	变量	34	true
130	:	单符	34	true
131	}	单符	35	true
132	return	关键字	37	true
133	0	常量	37	true
134	:	单符	37	true
135	}	单符	38	true
136	#	结束符	39	true

词法分析通过:



### 语法分析过程截图



符号表输出:



常量表

单词的值	单词类型	所占存储单元
0	整形常里	4
1	整形常里	4
2	整形常里	4
20	整形常里	4
d	字符常里	1
23.1	浮点常里	8
4	整形常里	4
10	整形常里	4
y	字符常里	1
3.23	浮点常里	8

图 4-20 测试截图

类型表

函数表

函数名	层次	区距	参数个数	参数表	入口地址
main	0	0	0	0	0

类型表

序号	类型名	类型指针
1	整型0	0
2	字符0	0
3	数组1	1
4	浮点0	0
5	结构体1	2
6	数组2	1
7	数组3	1
8	结构体2	2

符号表总表

序号	标识符名	类型指针	种类	地址	所占内存单元
1	main	整型0	函数	0	4
2	a	整型0	变量	4	4
3	b	整型0	变量	8	4
4	c	整型0	变量	12	4
5	g	数组0	变量	16	20
6	d	字符0	变量	36	1
7	e	浮点0	变量	37	8
8	i	结构体1	变量	45	8
9	j	整型0	域名	45	4
10	k	整型0	域名	49	4
11	f	数组3	变量	53	800
12	h	结构体2	变量	853	13
13	x	整型0	域名	853	4
14	y	字符0	域名	857	1
15	z	浮点0	域名	858	8

数组表

序号	数组名	下界	上界	成分类型	成分长度
1	g	0	20	字符0	20
2	f	0	10	数组3	800
3	f	0	20	整型0	80

图 4-21 测试截图

类型表

11	f	数组3	变量	53	800
12	h	结构体2	变量	853	13
13	x	整型0	域名	853	4
14	y	字符0	域名	857	1
15	z	浮点0	域名	858	8

数组表					
序号	数组名	下界	上界	成分类型	成分长度
1	g	0	20	字符0	20
2	f	0	10	数组3	800
3	f	0	20	整型0	80

结构体表					
序号	结构体名	域名信息	区距	成分类型	
1	i	j	4	0	
		k	8	0	
2	h	x	4	0	
		y	5	0	
		z	13	0	

长度表	
序号	长度:字节
1	4
2	4
3	4
4	4
5	20
6	1
7	8
8	8
9	4
10	4
11	800
12	13
13	4
14	1
15	8

图 4-22 测试截图

中间代码：

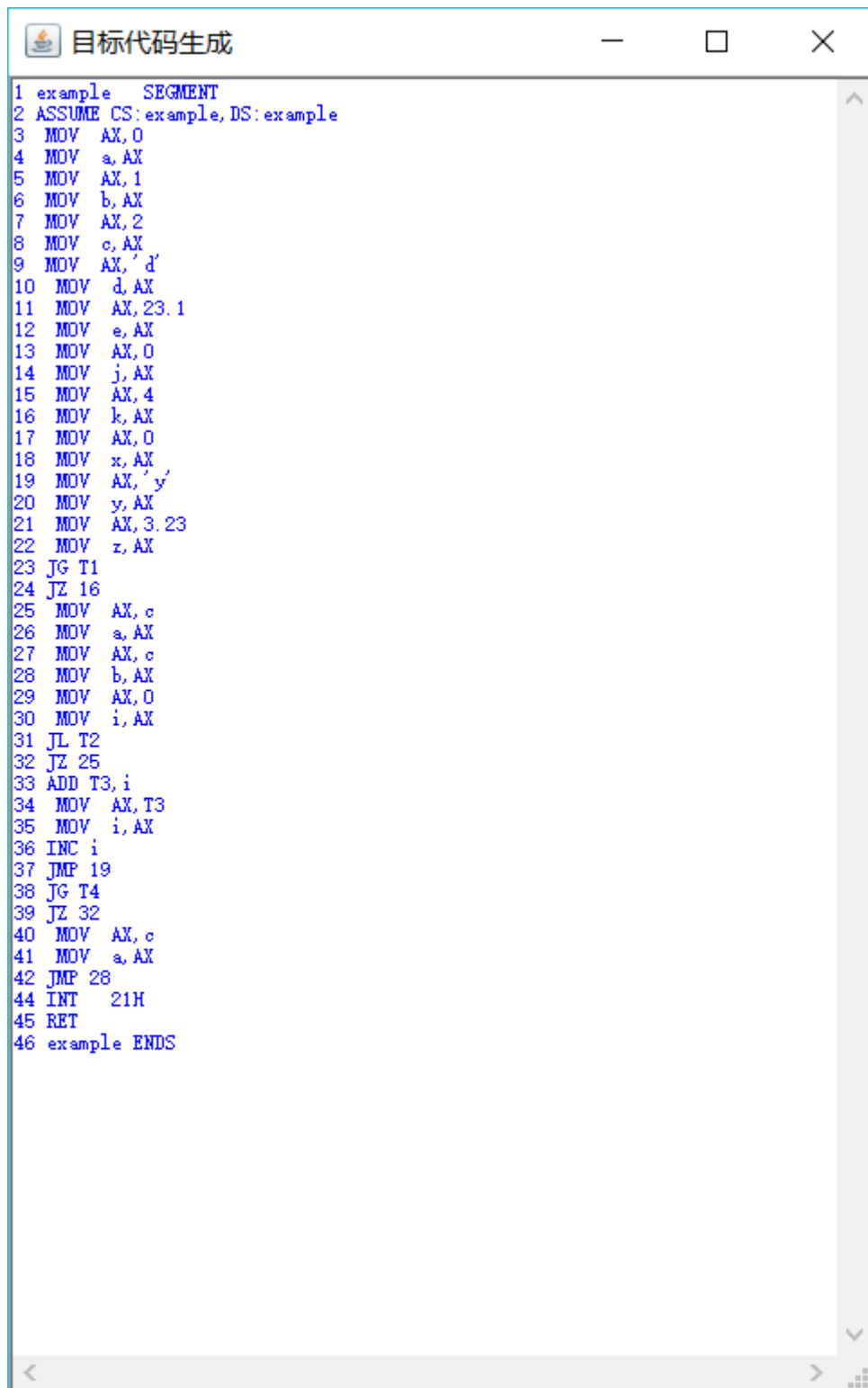


中间代码生成

生成的四元式如下				
序号:	OP	ARG1	ARG2	RESULT
1	=	0		a
2	=	1		b
3	=	2		c
4	=	'd'		d
5	=	23.1		e
6	=	0		j
7	=	4		k
8	=	0		x
9	=	'y'		y
10	=	3.23		z
11	>	a	b	T1
12	if	T1		
13	FJ			16
14	=	c		a
15	el			
16	=	c		b
17	ie			
18	for			
19	=	0		i
20	<	i	c	T2
21	FJ	T2		25
22	+	i	1	T3
23	=	T3		i
24	++	i		i
25	RJ			19
26	fend			
27	wh			
28	>	a	b	T4
29	do	T4		
30	FJ	T4		32
31	=	c		a
32	RJ			28
33	wend			

图 4-23 测试截图

目标代码:



The screenshot shows a window titled "目标代码生成" (Target Code Generation) with a list of assembly instructions. The instructions are as follows:

```
1 example SEGMENT
2 ASSUME CS:example, DS:example
3 MOV AX, 0
4 MOV a, AX
5 MOV AX, 1
6 MOV b, AX
7 MOV AX, 2
8 MOV c, AX
9 MOV AX, 'd'
10 MOV d, AX
11 MOV AX, 23.1
12 MOV e, AX
13 MOV AX, 0
14 MOV j, AX
15 MOV AX, 4
16 MOV k, AX
17 MOV AX, 0
18 MOV x, AX
19 MOV AX, 'y'
20 MOV y, AX
21 MOV AX, 3.23
22 MOV z, AX
23 JG T1
24 JZ 16
25 MOV AX, c
26 MOV a, AX
27 MOV AX, c
28 MOV b, AX
29 MOV AX, 0
30 MOV i, AX
31 JL T2
32 JZ 25
33 ADD T3, i
34 MOV AX, T3
35 MOV i, AX
36 INC i
37 JMP 19
38 JG T4
39 JZ 32
40 MOV AX, c
41 MOV a, AX
42 JMP 28
44 INT 21H
45 RET
46 example ENDS
```

图 4-24 测试截图

#### 4.3.3 用例语法错误时结果



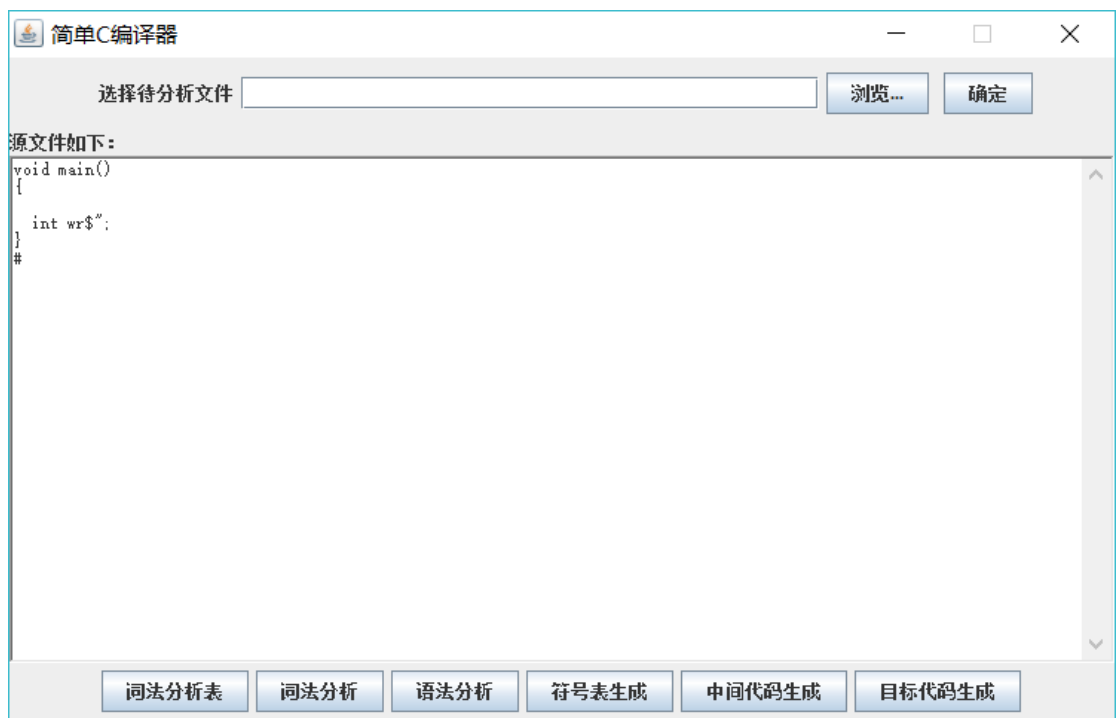


图 4-25 测试截图



图 4-26 测试截图

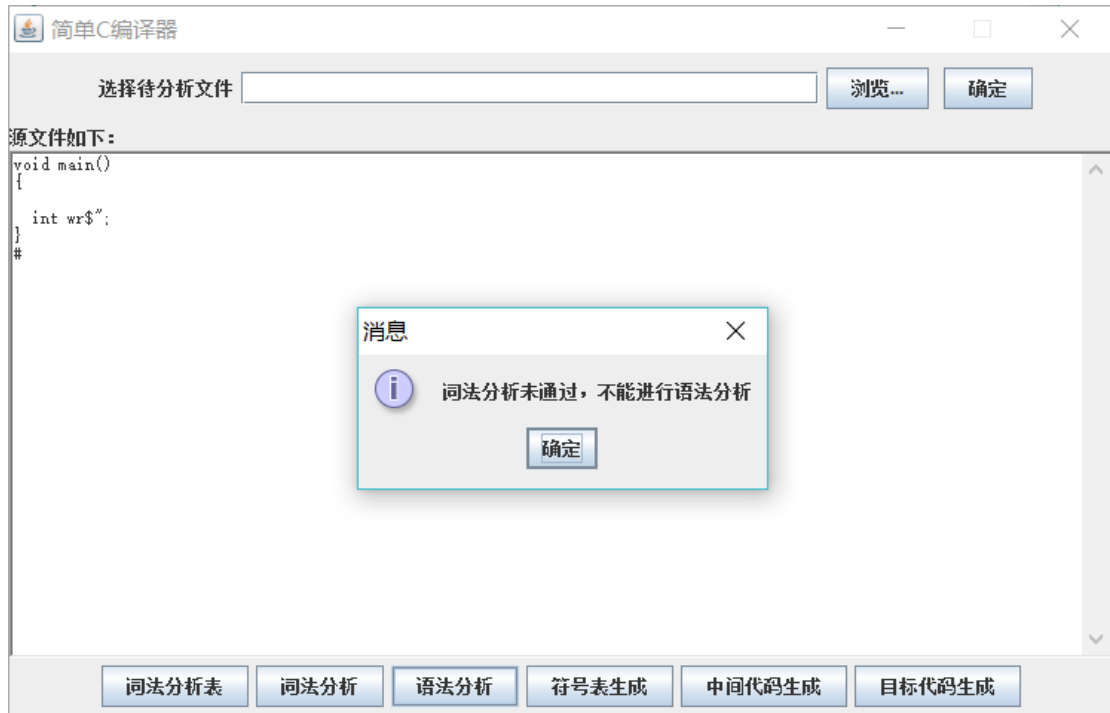


图 4-27 测试截图

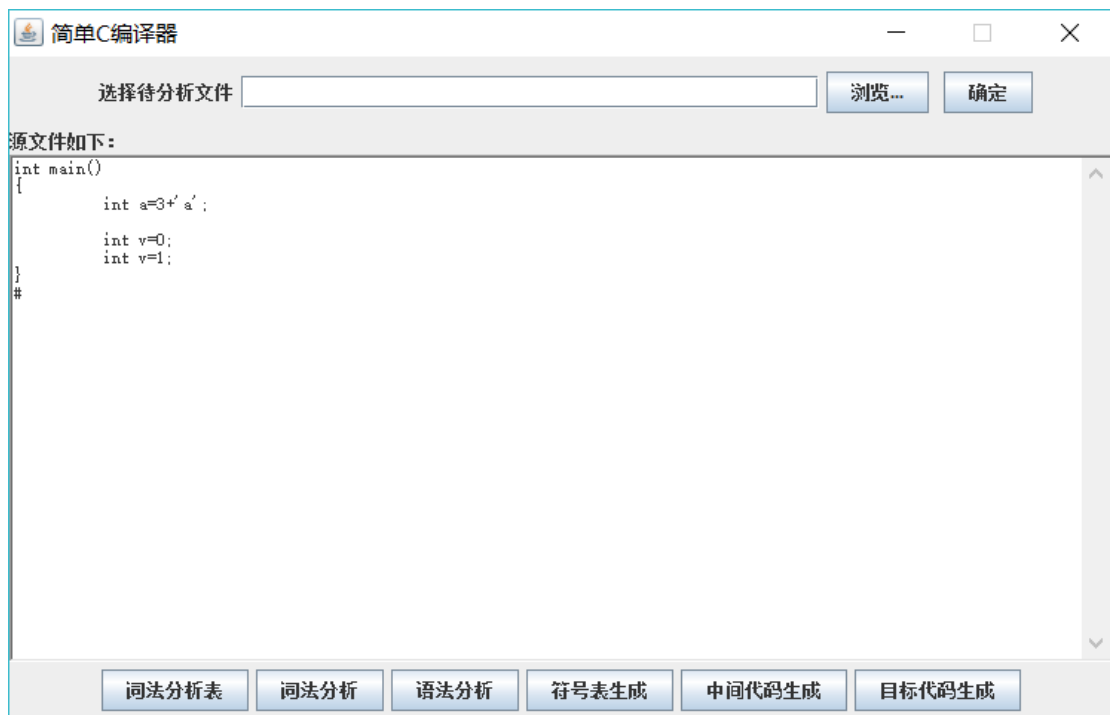


图 4-28 测试截图

词法分析				
单词序号	单词的值	单词类型	单词所在行	单词是否合法
1	int	关键字	1	true
2	main	关键字	1	true
3	(	左圆括号	1	true
4	{	左花括号	1	true
5	int	关键字	2	true
6	a	标识符	3	true
7	=	赋值运算符	3	true
8	3	常量	3	true
9	+	算术运算符	3	true
10	,	逗号	3	true
11	a	标识符	3	true
12	;	分号	3	true
13	int	关键字	5	true
14	v	标识符	5	true
15	=	赋值运算符	5	true
16	0	常量	5	true
17	;	分号	5	true
18	int	关键字	6	true
19	v	标识符	6	true
20	=	赋值运算符	6	true
21	1	常量	6	true
22	;	分号	6	true
23	}	右花括号	7	true
24	#	结束符	8	true

词法分析通过:

图 4-29 测试截图



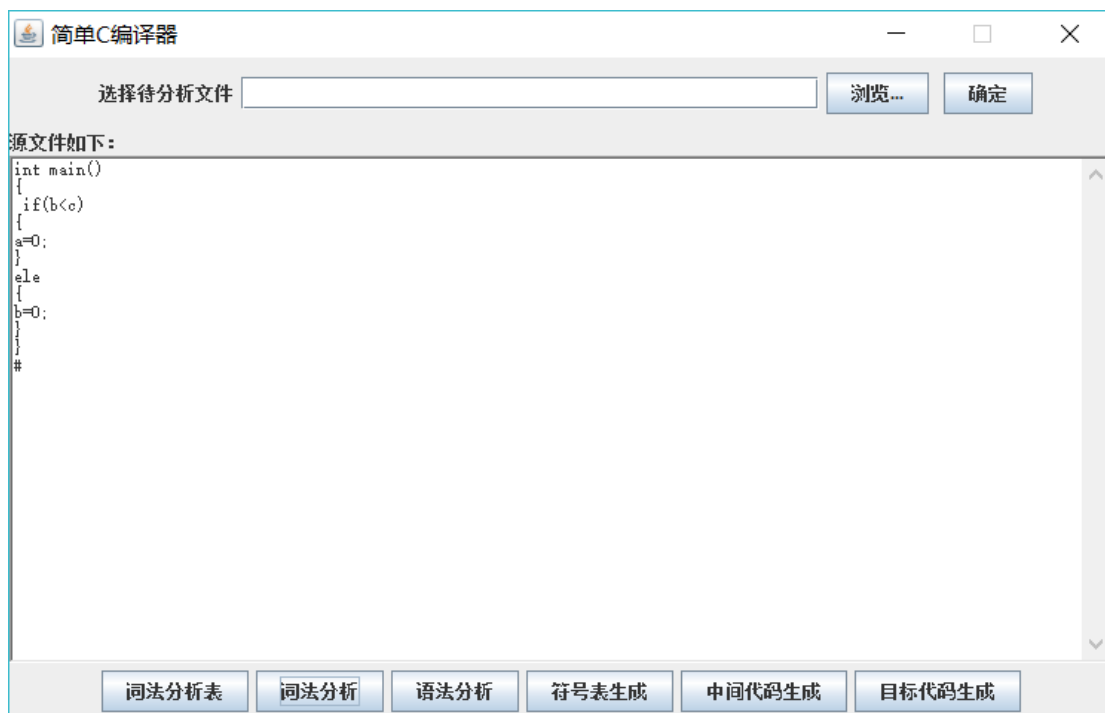


图 4-31 测试截图

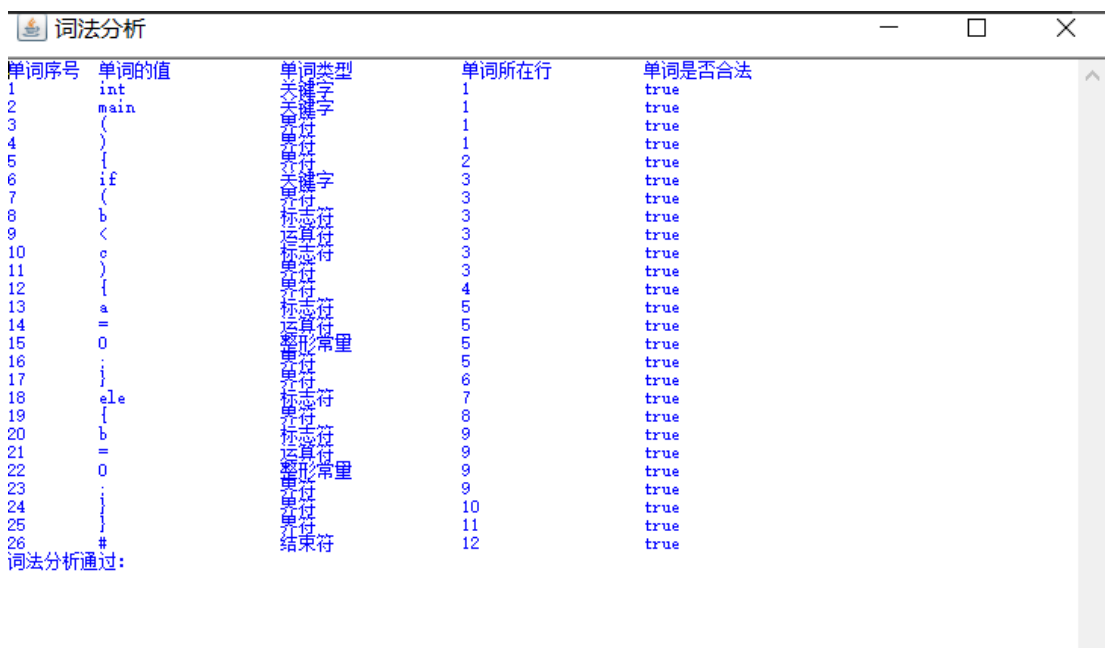


图 4-32 测试截图



图 4-33 测试截图

## 5. 结论

本次实验我们采用了 Java 语言来编写一个类 C 语言的编译器，步骤按照先自动机词法扫描源程序同时生成 Token 序列，然后运用 LL1 分析方法进行语法分析，接着往里面插语义动作语法制导同步生成四元式与符号表构建。最后是对语义分析产生的中间代码结合符号表生成目标代码。

词法扫描是通过读文件模式，将代码和关键字表与界符表从文件读入 Java 的数据结构 HashMap, 采用键值对的形式存储。之后就是运用 Java API 中独有的 String 类的各类函数对代码进行有限自动机处理，最终生成了 Token。

语法分析通过自建文法，采用 LL1 分析法，求出所有产生式的 select 集合然后根据 select 集合进行语法分析，然后一步一步实现在此之中生成了四元式，其中表达式的四元式是采用逆波兰式的方法生成的，于此同时完善符号表。

目标代码的生成是由四元式结合了符号表生成的，这个必须指定目标机器才有意义，我们这次目标代码的指定目标机器是 X8086。

其中的特色点有：各分析阶段能够检测出错误，并且能指出错误在哪一行，具体为什么错误；表示式的四元式采用了逆波兰式的方法；同时控制语句，我们的编译器能判断其中的 boolean 表达式的真值，从而能采用正确的逻辑得出正确的结果；符号表全面完善；目标代码符合目标机器要求。

## 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、建议

### 7.1 收获、体会

本次《编译原理》课程设计我们都学习到了许多，一方面是真正把上课所学的各章理论知识联系起来，完成了一个完整的编译器的设计制作；另一方面我们通过课程设计也提升了自己的自信心。从一开始拿到课题担心最终完不成，到最后我们不仅仅制作了编译器前端，这个过程中我想我们都收获了许多。当然，就像农民伯伯秋天的收获需要大半年的辛勤一样，我们最终目标的达到并不是一蹴而就的，需要在成功前不断地努力。我们所有组员为这个目标的达成付出了许多，尤其是作为组长，两周的时间几乎天天都在写代码和调试互相之间的接口，甚至于需要不断地黑盒测试来发现其他模块的错误。虽然过程很辛苦，但是能够为整个小组最后目标的达成，这种奉献精神是必要的。从这个角度来说，《编译原理》课设不仅仅是一次写代码的过程，也是一次发现自我，增强团结协作意识的机会。在搭建框架的时候使用了 UML 语言进行程序设计，还使用了 Github 的版本控制功能，能够使我能四个人的项目随时同步。当然了，在编译原理知识我也花了不少心思，比如词法分析，语法分析，四元式，符号表的内容我都了解了，在队友们编写程序的时候我也经常一起讨论知识点，这样就进一步巩固了我的知识点。总之这次实验收获还是挺多的。

### 7.2 建议

课设时间提前是非常明智的选择，既给了同学们提前复习的机会，也减轻了最后期末阶段的压力。但是有一点是既然是课程设计，希望老师应该多给我们项目经验，如何开展一个项目，中间得注意些什么，毕竟大家大多数还是得进公司发展，比较贴近实际需求。同时也希望计算机学院方面重视同学们的实战经历，说实话，没有第一次的实战经历，同学们压根没有热情、没有兴趣开展编程、更谈不上学理论知识了。但是若是开展了第一次的实战项目，以后同学们不管学习什么科目，都会有兴趣学习理论知识，从而达到理论与实践相结合的目的。最后感谢老师的大力帮助。还有辛勤的指导。