编译原理课程设计报告

**课题名称： 词法扫描器和语法分析器的实现**

提交文档学生姓名： 何长鸿

提交文档学生学号： 2016141482154

同组 成 员 名 单： 无

指导 教 师 姓 名：

指导教师评阅成绩：

指导教师评阅意见：

.

.

提交报告时间： 2019 年 06 月 06 日

目录

目录

[1. 课程设计目标 3](#_Toc10693600)

[1.1. 词法扫描器 3](#_Toc10693601)

[1.2. 语法分析器 3](#_Toc10693602)

[2. 分析与设计 3](#_Toc10693603)

[2.1. 实现方案说明 3](#_Toc10693605)

[2.2. 系统模块说明 4](#_Toc10693606)

[2.3. 词法扫描器 5](#_Toc10693607)

[2.3.1. 手工实现的词法扫描器算法流程 5](#_Toc10693608)

[2.3.2. DFA 5](#_Toc10693609)

[2.3.3. Lex工具实现 6](#_Toc10693610)

[2.4. 语法分析器算法流程 8](#_Toc10693611)

[2.4.1. 整体流程说明 8](#_Toc10693612)

[2.4.2. 文法规则整理 9](#_Toc10693613)

[2.4.3. LL（1）分析表的建立及数据结构 10](#_Toc10693614)

[2.4.4. YACC的输入、输出文件 11](#_Toc10693615)

[2.5. 代码设计说明 13](#_Toc10693616)

[2.5.1. 程序框架 13](#_Toc10693617)

[2.5.2. 文件解耦合设计说明 14](#_Toc10693618)

[2.5.3. 关键数据结构 15](#_Toc10693619)

[3. 程序代码实现 16](#_Toc10693620)

[3.1. 关键函数说明 16](#_Toc10693622)

[3.1.1. FIRST以及FOLLOW集计算 16](#_Toc10693623)

[3.1.2. 递归下降语法树输出 18](#_Toc10693624)

[3.1.3. LL(1)预测分析表的建立 19](#_Toc10693625)

[3.1.4. LL(1)分析过程 19](#_Toc10693626)

[3.1.5. 词法扫描自动机实现 20](#_Toc10693627)

[3.1.6. 递归下降分析过程 22](#_Toc10693628)

[3.2. 文件及主要代码 23](#_Toc10693629)

[3.2.1. run\_lex.cpp 23](#_Toc10693630)

[3.2.2. run\_syn.cpp 23](#_Toc10693631)

[3.2.3. LexParser.hpp 24](#_Toc10693632)

[3.2.4. SynParserRec.hpp 25](#_Toc10693633)

[3.2.5. SynParserLL1.hpp 27](#_Toc10693634)

[3.2.6. Utils.hpp 27](#_Toc10693635)

[3.2.7. C-grammar.y 28](#_Toc10693636)

[3.2.8. C-grammar.l 28](#_Toc10693637)

[4. 测试结果 28](#_Toc10693638)

[4.1. 测试结果信息简表 28](#_Toc10693640)

[4.2. 输出结果 29](#_Toc10693641)

[4.2.1. 词法扫描器（lex） 29](#_Toc10693642)

[4.2.2. 词法扫描器（手动实现） 29](#_Toc10693643)

[4.2.3. 语法分析器（递归下降） 30](#_Toc10693644)

[4.2.4. 语法分析器（LL(1)） 31](#_Toc10693645)

[4.2.5. 语法分析器（YACC） 31](#_Toc10693646)

[4.2.6. 错误处理 32](#_Toc10693647)

[5. 总结 32](#_Toc10693648)

[5.1. 收获 32](#_Toc10693650)

[5.2. 不足 32](#_Toc10693651)

图表

[图 1 系统模块图 5](#_Toc10693709)

[图 2 词法扫描处理流程图 6](#_Toc10693710)

[图 3 DFA 7](#_Toc10693711)

[图 4 语法分析流程 10](#_Toc10693712)

[图 5 程序框架图 15](#_Toc10693713)

[表 1程序输出格式 6](#_Toc10693727)

[表 2 LEX工具输入文件 9](#_Toc10693728)

[表 3 yacc输入文件 14](#_Toc10693729)

词法扫描器、语法分析器的设计与实现

# 课程设计目标

本课程设计中，我结合《编译原理》课程理论知识，使用C/C++语言编写构造词法扫描器和语法分析器两个模块。

## 词法扫描器

1. 读入待分析源程序文本。
2. 对源程序进行预处理，去掉注释。
3. 利用有限自动机构建词法扫描过程，从目标程序中识别词法单元，并记录此法单元位置。
4. 提供“获取下一个词法单元”接口，供后续语法分析器调用。
5. 回退词法单元，用于语法分析时试错及错误恢复。
6. 格式化输出词法单元。
7. 识别词法错误并报错

## 语法分析器

1. 调用词法扫描器，获取词法单元。
2. 根据c-的文法，将词法扫描器提供的此法单元转化为语法分析树。
3. 使用缩进方式打印语法分析树或输出分析过程中的产生式。
4. 错误处理

# 分析与设计



## 实现方案说明

词法扫描器：（1）lex工具自动生成；（2）手工实现。

语法分析器：（1）递归下降；（2）LL(1)；（3）BISON(YACC)工具自动生成

## 系统模块说明

如图所示，系统分为run、词法扫描、语法分析三个模块。在run函数中启动程序，如果要测试词法扫描器，则main函数会调用词法扫描器，并进入循环等待输入命令环节，从而格式化输出扫描到的Token。如果要直接进行语法分析，则Main函数会调用语法分析器，语法分析器完成初始化操作后，会一边调用词法扫描器获取词法单元（Token），一边进行语法分析。分析完成后会将语法分析树按照缩进方式打印或按照分析过程输出产生式。如表分别为词法扫描器和语法分析器的部分输出结果示例。

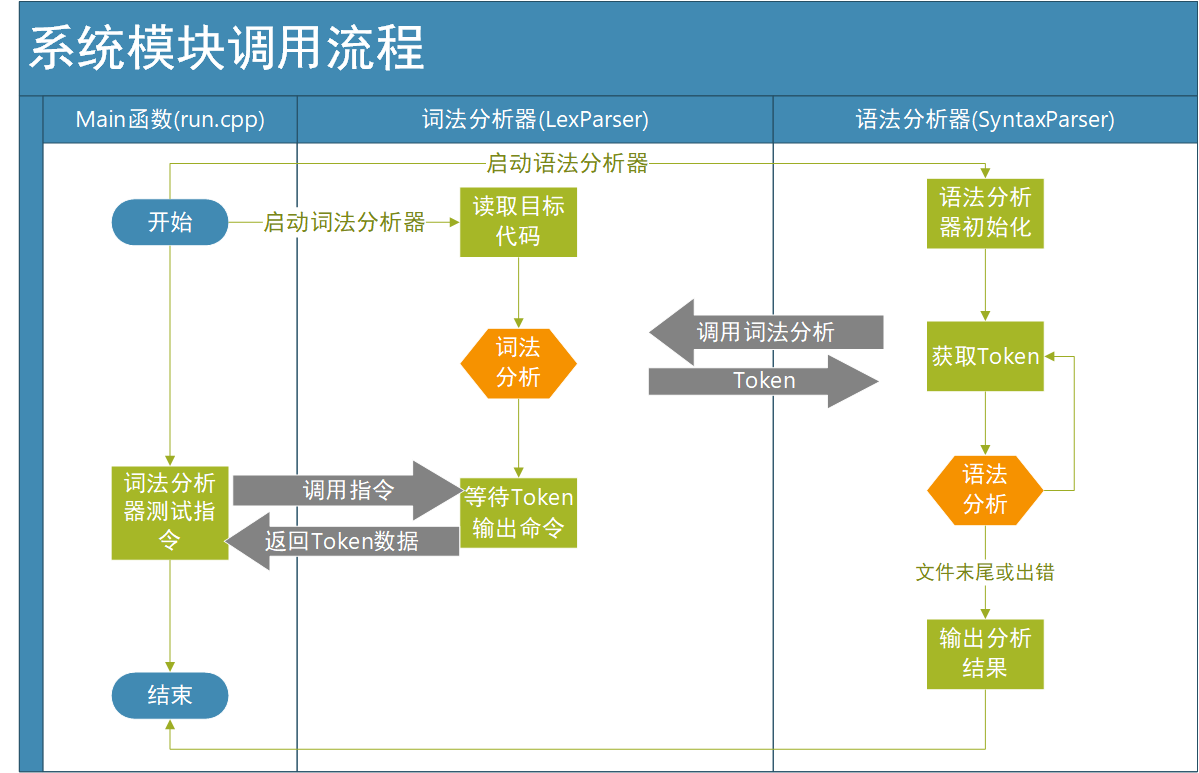


图 1 系统模块图

|  |  |
| --- | --- |
| 词法扫描输出 | 语法分析树（params节点）输出 |
| < KEY int 1>  < ID gcd 1>  < OP ( 1>  < KEY int 1>  < ID u 1>  < OP , 1>  < KEY int 1>  < ID v 1>  < OP ) 1> | params  param\_list  param  type\_specifier  KEY: int  ID: u  OP: ,  param  type\_specifier  KEY: int  ID: v |

表 1程序输出格式

## 词法扫描器

### 手工实现的词法扫描器算法流程

词法扫描器开始运行后，立即逐个字符读取目标源程序，并保存到缓存区，该缓存区由一个字符数组实现，在此过程中对目标程序进行预处理，即删除注释。第二遍从缓存区逐字读取代码并根据有穷自动机提取词法单元，然后以Token形式缓存到vector<Token>数组中。之后等待上级程序调用以完成词法单元的格式化输出、缓存数据同步等操作。

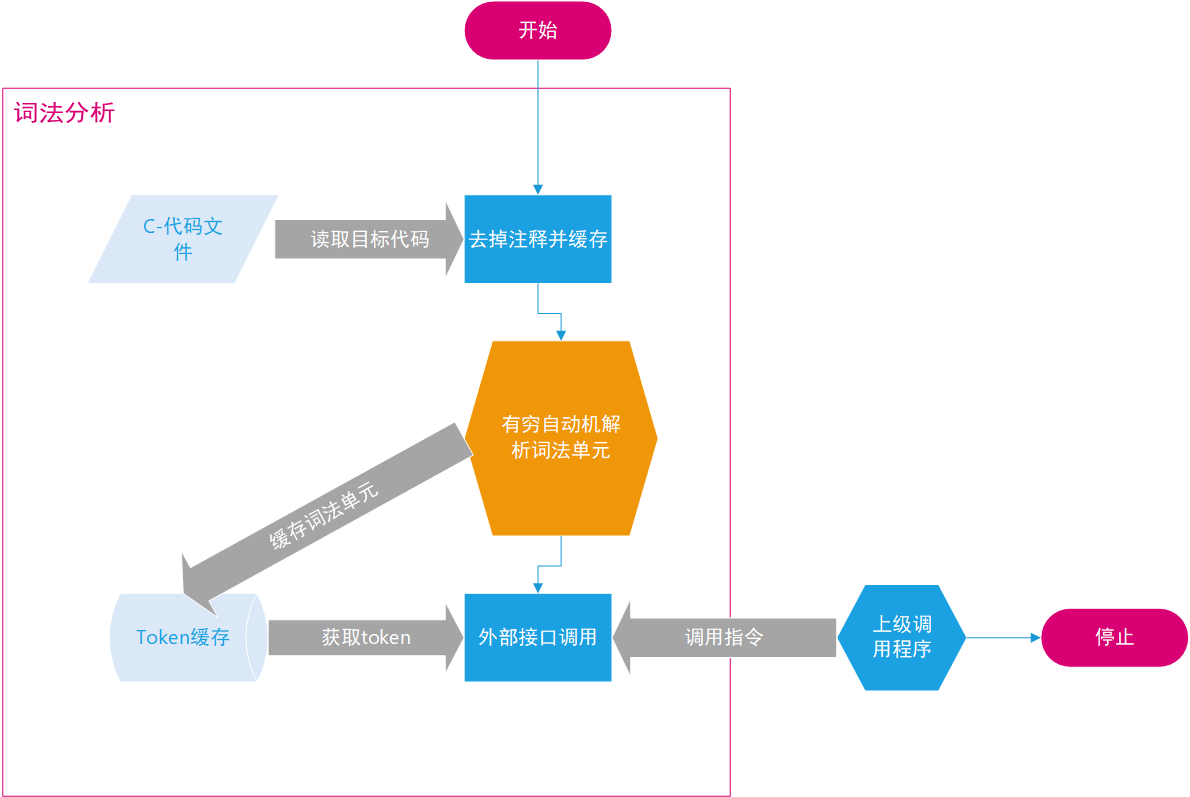


图 2 词法扫描处理流程图

### DFA

c-给定词法规定如下：

|  |
| --- |
| 1、关键字：  else if int return void while  所有的关键字均为保留字，并且全小写。  2、专用符号：  + - \* / < <= > >= == != = ; , ( ) [ ] { } /\* \*/  3、ID和NUM，通过下列正规表达式定义：  ID = letter letter\*  NUM = digit digit\*  letter = a|…|z|A|...|Z  digit = 0|…|9 |

根据C-词法规则，可以构造如下图所示的DFA。

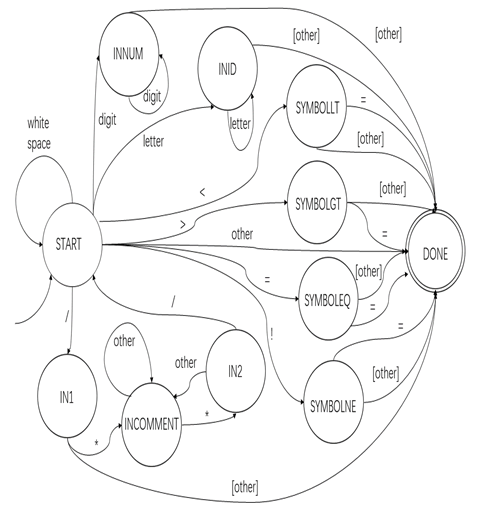


图 3 DFA

由于c-此法规则十分简单，可以根据又穷自动机将此法单元分为关键字、整数、标识符、操作符几个类型。代码中使用如下定义关键字、操作符和词素类型枚举变量：

|  |
| --- |
| const string KEYW[] = {"else", "if", "int", "return", "void", "while"};  const char OPS[] = {'+', '-', '\*', '/', '<', '=', '>', '!', ';', ',', '(',  ')', '[', ']', '{', '}'};  const string TYPE\_NAME[] = {"KEY", "INT", "ID", "OP", "PUNCT"};  enum TYPE{KEY, INT, ID, OP, PUNCT}; |

有穷自动机使用直接模拟方式实现。

### Lex工具实现

输入文件如下表所示，该文件中定义了main()函数，因此lex生成的lex.yy.c文件(文件内容太多无法在此展示)经过编译后可以直接运行并输出此法单元。

|  |
| --- |
| %{  #define KEY 1 //不能从零开始  #define INT 2  #define ID 3  #define OP 4  int yylval;  %}  delim [" "\n\t]  ws {delim}+  digit [0-9]  num {digit}+  letter [A-Za-z]  key (if)|(else)|(int)|(return)|(void)|(while)  id {letter}+  %%  {ws} {}  {num} {yylval = 99; return(INT);}  {key} {yylval = 6;return(KEY);}  {id} {yylval = 100; return(ID);}  "<" {yylval = 11; return(OP);}  ">" {yylval = 12; return(OP);}  "=" {yylval = 13;return(OP);}  "<=" {yylval = 14;return(OP);}  ">=" {yylval = 15;return(OP);}  "!=" {yylval = 16;return(OP);}  "==" {yylval = 17;return(OP);}  "+" {yylval = 18;return(OP);}  "-" {yylval = 19;return(OP);}  "\*" {yylval = 20;return(OP);}  "/" {yylval = 21;return(OP);}  "!" {yylval = 22;return(OP);}  ";" {yylval = 23;return(OP);}  "," {yylval = 24;return(OP);}  "(" {yylval = 25;return(OP);}  ")" {yylval = 26;return(OP);}  "[" {yylval = 27;return(OP);}  "]" {yylval = 28;return(OP);}  "{" {yylval = 29;return(OP);}  "}" {yylval = 30;return(OP);}  %%  int yywrap (){  return 1;  }  int main(int argc,char \* argv[])  {  yyin = fopen(argv[1],"r");  int c=0,i = 0;  char ss[5][5] = {"","KEY","INT","ID","OP"};  while(c=yylex()){  printf("%d %s %s\n",++i,ss[c],yytext);  }  fclose(yyin);  return 0;  } |

表 2 LEX工具输入文件

## 语法分析器算法流程

### 整体流程说明

由于递归下降流程比较简单，仅需要根据各个修改后的产生式非终结符构造递归下降函数即可，因此此处仅给出LL(1)语法分析器的算法流程。如图所示，语法分析器启动后，会读取C-的语法文档（该文档内容应该满足LL(1)文法），然后根据该文法求每个非终结符的FIRST集、FOLLOW集，进而构造预测分析表，然后开始分析过程。分析过程中会调用词法扫描器获取Token, 并且边分析变输出产生式，最后用于构造语法分析树。其中，分析中间结果如下表所示，根据该结果即可构造表1中同结构的语法分析树。

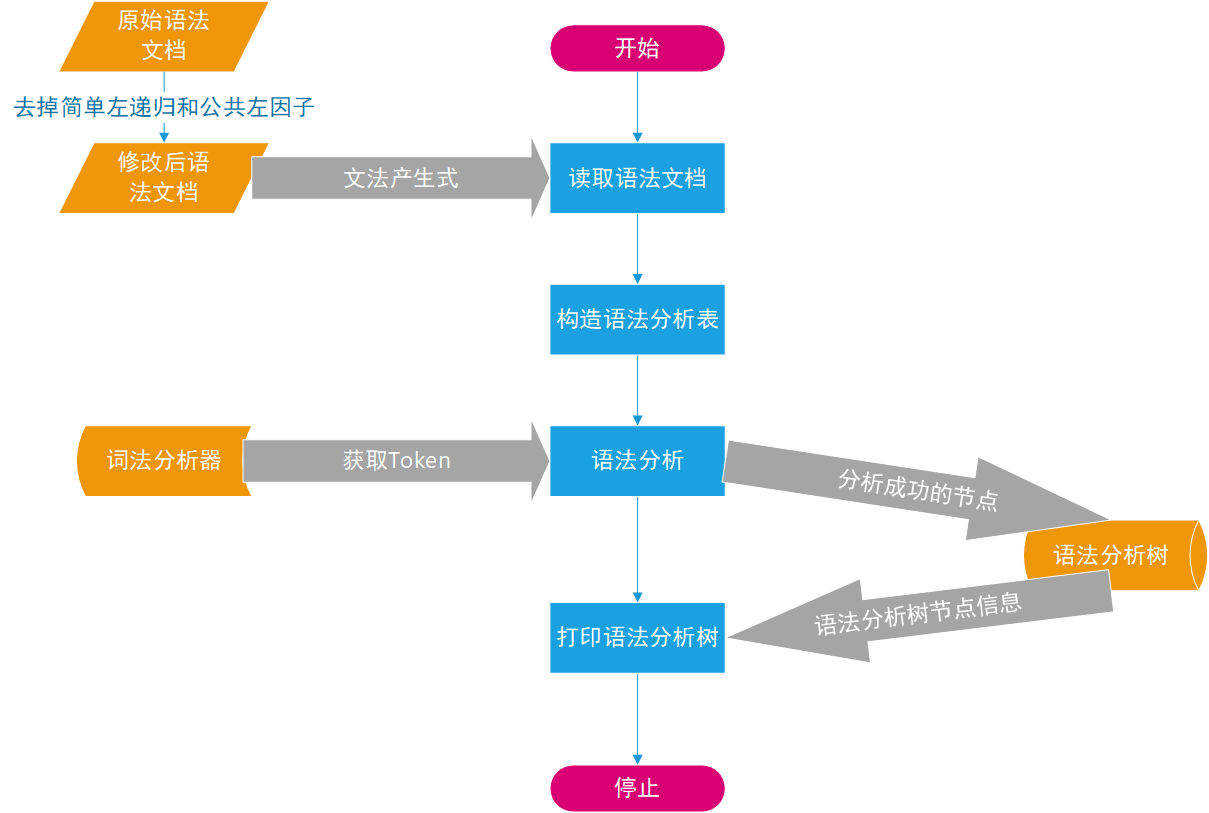


图 4 语法分析流程

|  |
| --- |
| program declaration-list  declaration-list declaration help1  declaration fun-declaration  fun-declaration type-specifier ID ( params )  type-specifier int  < KEY int 1>  < ID gcd 1>  < OP ( 1>  params param-list  param-list param help2  param type-specifier ID help\_lf2  type-specifier int  < KEY int 1>  …… |

### 文法规则整理

递归下降以及LL（1），其中递归下降文法修改比较简单，利用EBNF在编写代码中临时修改即可。文法需要先对c-语法进行整理，LL（1）文法去掉公共左因子和左递归后的文法如下表：

|  |  |
| --- | --- |
| program declaration-list  declaration-list declaration help1  help1 declaration help1  help1 empty  declaration var-declaration  declaration fun-declaration  var-declaration type-specifier ID help\_lf1  help\_lf1 ;  help\_lf1 [ NUM ] ;  type-specifier int  type-specifier void  fun-declaration type-specifier ID ( params )  fun-declaration compound-stmt  params param-list  params void  param-list param help2  help2 , param help2  help2 empty  param type-specifier ID help\_lf2  help\_lf2 empty  help\_lf2 [ ]  compound-stmt { local-declarations statement-list }  local-declarations var-declaration local-declarations  local-declarations empty  statement-list statement statement-list  statement-list empty  statement expression-stmt  statement compound-stmt  statement selection-stmt  statement iteration-stmt  statement return-stmt  expression-stmt expression ;  expression-stmt ;  selection-stmt if ( simple-expression ) statement help\_lf3  help\_lf3 empty  help\_lf3 else statement | iteration-stmt while ( expression ) statement  return-stmt return help\_lf4  help\_lf4 ;  help\_lf4 expression ;  expression simple-expression  expression var = expression  var ID help\_lf5  help\_lf5 empty  help\_lf5 [ expression ]  simple-expression additive-expression help\_lf6  help\_lf6 empty  help\_lf6 relop additive-expression  relop <=  relop <  relop >  relop >=  relop ==  relop !=  additive-expression term help3  help3 addop term help3  help3 empty  addop +  addop -  term factor help4  help4 mulop factor help4  help4 empty  mulop \*  mulop /  factor ( expression )  factor var  factor call  factor NUM  call ID ( args )  args arg-list  args empty  arg-list expression help5  help5 , expression help5  help5 empty |

### LL（1）分析表的建立及数据结构

由于c-minus文法2产生式较多，因此我采用编程实现预测分析表的建立。主要流程如下四个函数：

1. loadGrammar()从保存的文法文件加载产生式到内存中，保产生式的数据结构为vector<vector<ProUnit>> 的二维数组，第一维度的下表表示产生式的编号。
2. getFIRST(m\_grammar)、getFOLLOW(m\_grammar, FIRST) 分别计算first集、follow集。
3. genParseTable() 利用以上两个集合得出分析表。

loadGrammar();

FIRST = getFIRST(m\_grammar);

FOLLOW = getFOLLOW(m\_grammar, FIRST);

genParseTable();

保存分析表的数据结构为map<string, map<string, int>> table; 即一张索引关键字分别为非终结符-终结符，表项为产生式编号的二维表，部分内容如下。例如前两行表示行additive-expression对应列‘（’、‘ID’、‘NUM’的产生式为第54条。

|  |
| --- |
| additive-expression  ( 54 ID 54 NUM 54  addop  + 57 - 58  arg-list  ( 71 ID 71 NUM 71  args  ( 69 ) 70 ID 69 NUM 69  call  ID 68  compound-stmt  { 21 |

### YACC的输入、输出文件

输入文件c-grammar.y部分内容如下表（由于每个非终结符都有对应的匹配模式，内容较多，下表只给出部分非终结符匹配模式）：

|  |
| --- |
| //声明部分  %{  #include <stdio.h>  #include <stdlib.h>  #include <stdarg.h>  extern FILE \*yyin;  int yylexeNode(void);  void yyerror(char \*s);  %}  %union {  char\* strval;  };  %token <strval> NUM IDENTIFIER  %token LE\_OP GE\_OP EQ\_OP NE\_OP  %token IF ELSE WHILE CONTINUE BREAK RETURN VOID INT  %nonassoc LOWER\_THAN\_ELSE  %nonassoc ELSE  %start program  %%  program  : declaration\_list { printf("program declaration\_list\n"); } //输出当前匹配的产生式  ;  var\_declaration  : type\_specifier IDENTIFIER ';'  {  printf("var\_declaration type\_specifier ID:%s ;\n",$2);  }  | type\_specifier IDENTIFIER '[' NUM ']' ';'  {  printf("var\_declaration type\_specifier ID:%s '[' NUM:%s ']' ';'\n",$2, $4);  }  ;  type\_specifier  : INT { printf("type\_specifier INT\n"); }  | VOID { printf("type\_specifier VOID\n"); }  ;  return\_stmt  : RETURN ';' {printf("return-stmt return ;\n"); }  | RETURN expression ';' {printf("return-stmt return expression ;\n"); }  ;  additive\_expression  : additive\_expression addop term  {  printf("additive-expression additive-expression addop term\n");  }  | term  {  printf("additive-expression term\n");  }  ;  %%  void yyerror(char \*s)  {  printf("%s\n", s);  }  //主函数  int main(int argc,char \* argv[])  {  yyin = fopen(argv[1],"r");  yyparse();  fclose(yyin);  return 0;  } |

表 3 yacc输入文件

使用bison c-grammar.y 命令，自动生成c-grammar.tab.c和c-grammar.tab.h文件，由于文件内容较长，此处不再写出。

## 代码设计说明

### 程序框架

如图所示，项目内容有两个主函数程序run\_lex.cpp和run\_syn.cpp，分别用于测试词法扫描器和语法分析器。此外，使用lex工具以及bison工具自动生成的词法扫描和语法分析器单独进行测试。

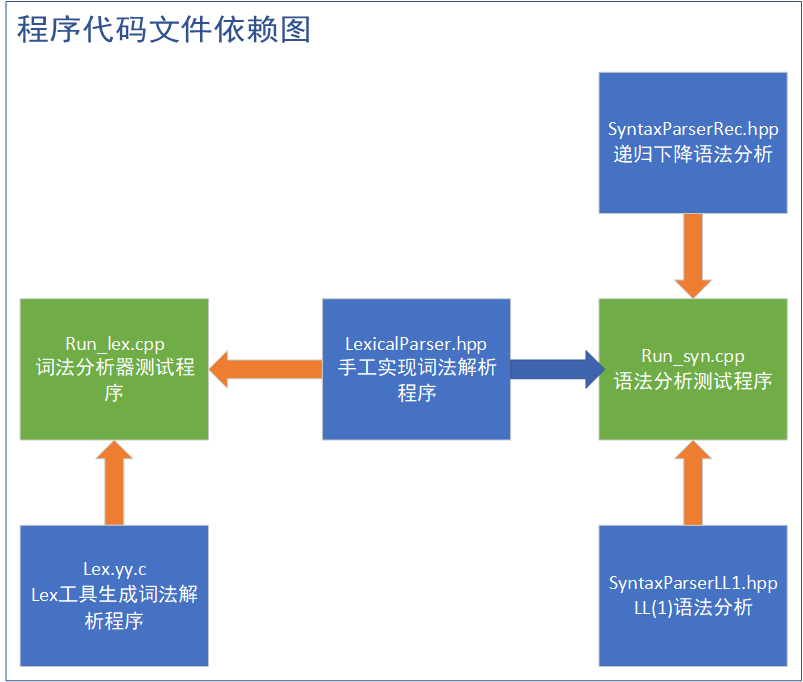
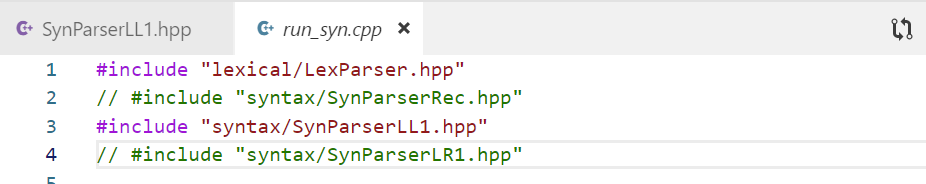


图 5 程序框架图

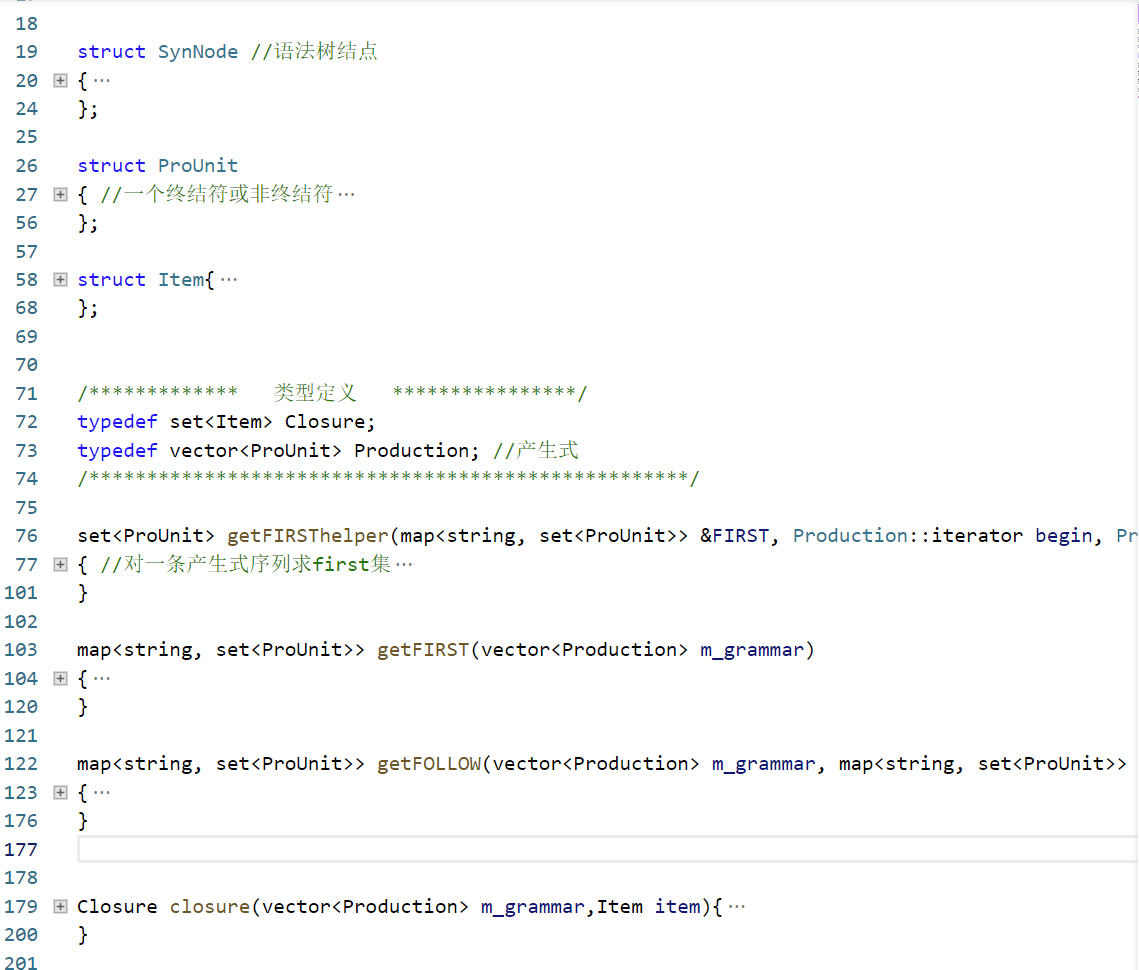
### 文件解耦合设计说明

由于采用多种实现方案，本程序通过以下几点降低程序耦合度，方便测试及维护：

1. 在run\_syn.hpp文件中，通过引用不同的语法分析实现.hpp文件来调用对应的分析方法。如图所示，使用时只需要简单注释掉其他文件即可。



1. 将公共方法以及数据结构抽象出来，统一编写在utils.hpp文件中，例如LL(1), SLR(1)（虽然SLR(1)没有完全实现）等方案中都用到了FIRST以及FOLLOW集的计算，因此将这些计算函数纳入utils文件中。文件主要内容如下图：



### 关键数据结构

各个类名称如下表所示,其中，SynNode、ProUnit、Item等类型声明见3.2.6。

|  |  |
| --- | --- |
| **名称** | **数据结构** |
| 一个终结符或非终结符 | struct ProUnit |
| 语法树结点 | struct SynNode |
| SLR(1)的一个项 | struct Item |
| 产生式 | typedef vector<ProUnit> Production; |
| 闭包 | typedef set<Item> Closure |
| FIRST FOLLOW集 | map<string, set<ProUnit>> FIRST, FOLLOW |
| LL(1)预测分析表 | map<string, map<string, int>> table; |
| 文法规则产生式 | vector<vector<ProUnit>> m\_grammar; |

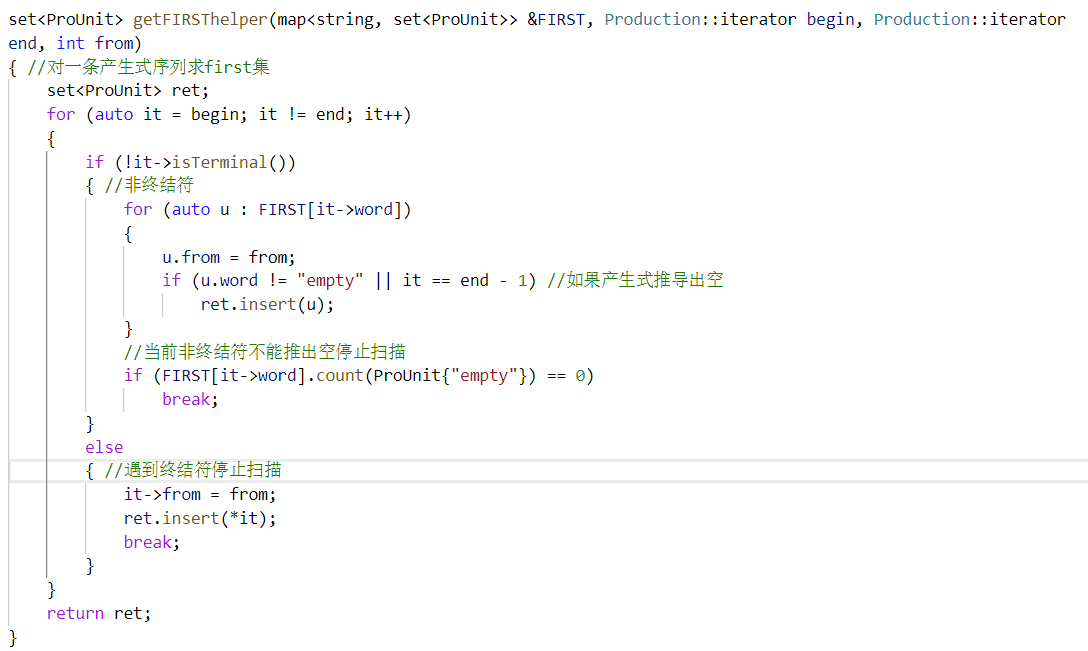
# 程序代码实现



## 关键函数说明

### FIRST以及FOLLOW集计算

由于这两个集合计算都涉及到求一个符号串的FIRST集，将该功能独立出来，设计成getFIRSThelper函数，该函数输入为一条产生式以及当前文法的全部FIRST集（不一定计算完），输出为对应产生式的FIRST集。



在此基础上，只需要输入已经保存在内存中的产生式表，即可计算对应FIRST集和FOLLOW集，代码如下所示；

|  |
| --- |
| map<string, set<ProUnit>> getFIRST(vector<Production> m\_grammar)  {  map<string, set<ProUnit>> FIRST;  //规则1，每条产生式首个终结符直接加入first  bool more = true;  while (more)  {  more = false;  for (int i = 0; i < m\_grammar.size(); ++i)  {  auto has\_new = FIRST[m\_grammar[i][0].word].size(); //初始大小  auto ret = getFIRSThelper(FIRST, m\_grammar[i].begin() + 1, m\_grammar[i].end(), i);  FIRST[m\_grammar[i][0].word].insert(ret.begin(), ret.end());  more |= (has\_new != FIRST[m\_grammar[i][0].word].size());  }  }  return FIRST;  }  map<string, set<ProUnit>> getFOLLOW(vector<Production> m\_grammar, map<string, set<ProUnit>> FIRST)  {  map<string, set<ProUnit>> FOLLOW;  //规则1：开始符号加入$  FOLLOW["program"].insert(ProUnit{"$"});  bool more = true;  while (more)  {  more = false;  for (int idx = 0; idx < m\_grammar.size(); ++idx)  {  auto pd = m\_grammar[idx];  //规则2 对一个产生式中的所有非终结符，将后面的first集加入其follow集  for (int i = 1; i < pd.size() - 1; ++i)  { //从前往后  if (!pd[i].isTerminal())  { // 找到一个非终结符pd[i],将后面的first集并入  string noTerminal = pd[i].word;  auto hasMore = FOLLOW[noTerminal].size();  auto ret = getFIRSThelper(FIRST, pd.begin() + i + 1, pd.end(), idx); //idx记录对应产生式  if (ret.count(ProUnit{"empty"}) > 0)  ret.erase(ProUnit{"empty"});  FOLLOW[noTerminal].insert(ret.begin(), ret.end());  more |= (hasMore != FOLLOW[noTerminal].size());  }  }  //规则3 以非终结符结尾的，将左部的follow集并入末尾非终结符  for (int j = pd.size() - 1; j > 0; j--)  { //从后往前  if (pd[j].isTerminal())  break;  auto hasMore = FOLLOW[pd[j].word].size();  for (auto it = FOLLOW[pd[0].word].begin(); it != FOLLOW[pd[0].word].end(); ++it)  {  FOLLOW[pd[j].word].insert(ProUnit{it->word, idx}); //记录对应产生式  }  more |= (hasMore != FOLLOW[pd[j].word].size());  if (FIRST[pd[j].word].count(ProUnit{"empty"}) == 0)  break; //不含空集，不再往前推导  }  }  }  //去掉空符号  for (auto it = FOLLOW.begin(); it != FOLLOW.end(); ++it)  {  it->second.erase(ProUnit{"empty"});  }  return FOLLOW;  } |

### 递归下降语法树输出

如下表，输入为一颗语法树根节点以及根节点缩进量（默认为0），输出得到对应语法树。

|  |
| --- |
| void SynParser::printTree(SynNode \*root, int indent = 0)  {  for (int i = 0; i < indent; ++i) //缩进量  cout << ' ';  if(root->type!=-1)  cout << TYPE\_NAME[root->type] << ": ";  cout << root->word << endl;  for (auto cld : root->children)  {  printTree(cld, indent + 2);//递归实现孩子结点缩进量增加  }  } |

### LL(1)预测分析表的建立

利用已经计算得到的FIRST集和FOLLOW集，生成预测分析表并保存在类成员变量table中。

|  |
| --- |
| void SynParser::genParseTable()  {  for(auto it=FIRST.begin(); it!=FIRST.end(); ++it){  //搜索FIRST集，填写预测分析表  for(ProUnit fst: it->second){  if(fst.word!="empty"){  if(table[it->first].count(fst.word)>0)  throw runtime\_error("预测分析表产生式冲突！");  table[it->first][fst.word] = fst.from;  }else{//有空符号，则操作follow集  for(ProUnit flw:FOLLOW[it->first]){  if(table[it->first].count(fst.word)>0)  throw runtime\_error("预测分析表产生式冲突！");  table[it->first][flw.word] = fst.from;  }  }  }  }  } |

### LL(1)分析过程

利用预测分析表，以及词法扫描器提供的Token，开始分析过程。

|  |
| --- |
| void SynParser::parse(){  stack<ProUnit> S; //符号栈  S.push(ProUnit{"$"});  S.push(ProUnit{"program"});  ProUnit u = S.top(); //栈顶符号  while(u.word != "$"){  auto mk = lex.getMarker();  Token tk = lex.nextToken();  if(u.isTerminal()){  if((u.word=="NUM" && tk.type==INT) || (u.word=="ID" && tk.type==ID) || (u.word==tk.s\_value)){  cout << tk.strfToken() << endl;  S.pop();  }else{  cout << "tk - u:(" << TYPE\_NAME[tk.type]<<")"<<tk.s\_value<<" - "<<u.word<<endl;  throw runtime\_error("terminal match failed!");  }  }else{//非终结符  lex.toMarker(mk);  int idx = -1;  if(tk.type==INT)  idx = table[u.word]["NUM"];  else if(tk.type==ID)  idx = table[u.word]["ID"];  else  idx = table[u.word][tk.s\_value];  vector<ProUnit> pd = m\_grammar[idx];  //输出产生式和栈的维护  S.pop();  for(auto it=pd.begin(); it!=pd.end();++it){  cout << it->word << ' ';  }  for(auto it=pd.rbegin(); it!=pd.rend()-1;++it){//产生式末尾先入栈  if(it->word=="empty")  continue;  S.push(\*it);  }  cout << endl;  }  u = S.top();  }  cout << "#####"<<endl; //分析完成  } |

### 词法扫描自动机实现

该函数利用LexParser类成员变量中的缓存代码串，根据自动机状态转换关系生成Token, 并保存到成员变量vector<Token> tokens中。

|  |
| --- |
| void LexParser::parseToken()  {  char TokenBuffer[60];  int line = 1;  for(int i=0; i<strlen(s\_code); i++)  {  //跳过空白符  if(s\_code[i]=='\n') {++line;continue;}  if(isblank(s\_code[i])) continue;    //进入int类型状态转换图  if(isdigit(s\_code[i])){  int j=0;  TokenBuffer[j++] = s\_code[i];  while(i<strlen(s\_code)-1 && isdigit(s\_code[i+1])) {TokenBuffer[j++] = s\_code[i+1]; i++;}; //预读一个字符,并判断是否到达流尾  if(isalpha(s\_code[i+1])){  TokenBuffer[j++] = s\_code[i+1];  TokenBuffer[j++] = '\0';  char msg[100];  sprintf(msg,"line %d: illegal lexical: %s",line, TokenBuffer);  throw runtime\_error(msg);  }  TokenBuffer[j] = '\0';  addTokenInt(TokenBuffer, line);  continue;  }  //进入标识符和关键字状态转换图  if(isalpha(s\_code[i])){  int j=0;  TokenBuffer[j++] = s\_code[i];  while(i<strlen(s\_code)-1 && isalpha(s\_code[i+1])) {TokenBuffer[j++] = s\_code[i+1]; i++;}; //预读一个字符,并判断是否到达流尾  TokenBuffer[j] = '\0';  addTokenWord(string(TokenBuffer), line);  continue;  }  //操作符识别  if(isOperator(s\_code[i])){  int j=0;  TokenBuffer[j++] = s\_code[i];  if(s\_code[i]=='<' || s\_code[i]=='>' || s\_code[i]=='!' || s\_code[i]=='='){  if(i<strlen(s\_code)-1 && s\_code[i+1]=='=') {TokenBuffer[j++] = s\_code[i+1]; i++;};  }  TokenBuffer[j] = '\0';  addTokenOp(string(TokenBuffer), line);  continue;  }  //异常  char msg[100];  sprintf(msg,"line %d: thers is an illegal charactor:%c",line, s\_code[i]);  throw runtime\_error(msg);  }  } |

### 递归下降分析过程

如下表，对于产生式declaration → var-declaration|fun-declaration，实现函数如下，函数参数为当前状态的父节点，如果分析成功，则插入一个名称为“declaration”的子节点。

|  |
| --- |
| void SynParser::declaration(SynNode \*parent)  {  uint32\_t mk = lex.getMarker(); //标记当前词法扫描器状态  SynNode \*cur = new SynNode{"declaration"};  try  { //是否为变量声明  var\_declaration(cur);  }  catch (runtime\_error e1)  { //不是变量声明则回到词法扫描器之前状态  cur->children.clear();  lex.toMarker(mk);  try  {//是否为函数声明  fun\_declaration(cur);  }  catch (runtime\_error e2)  {//语法错误  delete cur;  cout <<"near line "<<exception\_line<<": "<< e1.what() << " or " << e2.what() << endl;  throw runtime\_error("illegel declaration");  // throw e;  }  }  //找到申明则将当前结点插入语法树  parent->children.push\_back(cur);  } |

## 文件及主要代码

### run\_lex.cpp

|  |
| --- |
| int main(int argv, char\* args[])  {  if(argv<2){  printf("use: %s filename",args[0]);  exit(0);  }  //call lex parser  LexParser parser(args[1]);  //input commands  showMenu();  stars();  char cmd;  while(cin>>cmd,cmd!='q')  {  switch(cmd)  {  case 'p': parser.printCode(); break;  case 'h': showMenu(); break;  case 't': parser.printTokenList();break;  case 'n': cout<<parser.nextToken().strfToken()<<endl;break;  }  stars();  }  return 0;  } |

### run\_syn.cpp

|  |
| --- |
| #include "lexical/LexParser.hpp"  // #include "syntax/SynParserRec.hpp"  #include "syntax/SynParserLL1.hpp"  // #include "syntax/SynParserLR1.hpp"  void stars();  void showMenu();  int main(int argv, char\* args[])  {  if(argv<2){  printf("use: %s filename",args[0]);  exit(0);  }  //call lex parser  LexParser lex\_parser(args[1]);  SynParser xyn\_parser(lex\_parser);  return 0;  } |

### LexParser.hpp

|  |
| --- |
| class Token  {  public:  TYPE type;  long i\_value;  uint16\_t line;  string s\_value;  Token(TYPE type, const long value, int line);  Token(TYPE type, const string &value, int line);  string strfToken();  };  /\*  \* LexParser每个对象解析一个c-文档，转换为tokens  \*  \*/  class LexParser  {  private:  vector<Token> tokens;  ifstream f\_code;  void preParse();  char s\_code[BUFFER\_SIZE];  uint32\_t cursor = 0;  //词法分析核心函数自动机解析Token  void parseToken();  //判断是否为C-的合法操作符  bool isOperator(char);  //将Token加入到队列  void addTokenInt(const char \*, int);  void addTokenWord(const string &, int);  void addTokenOp(const string &, int);    public:  LexParser(const string&);  ~LexParser();  void printCode() const;  void printTokenList() const;  Token& nextToken();  void putBackToken();  //用于Token队列的保存和恢复  uint32\_t getMarker(){return cursor;}  void toMarker(uint32\_t marker){cursor = marker;}  bool isEnd(){return cursor>=tokens.size();}  }; |

### SynParserRec.hpp

此处只列出一个非终结符的匹配函数。

|  |
| --- |
| class SynParser  {  public:  SynParser(LexParser &lex\_) : lex(lex\_)  {  root = new SynNode{"program"};  curNode = root;  declaration\_list(root);  exception\_line = 0;  step = 1;  }  ~SynParser()  {  printTree(root, 0);  }  private:  SynNode \*root;  SynNode \*curNode;  LexParser &lex;  int exception\_line;  int step;  bool match(string terminal, SynNode \*parent); //匹配终结符  void freeTree(SynNode \*);  void freeChildren(SynNode \*);  void printTree(SynNode \*root, int indent);  //非终结符匹配  void declaration\_list(SynNode \*parent);  void declaration(SynNode \*parent);  }；  void SynParser::declaration(SynNode \*parent)  {  uint32\_t mk = lex.getMarker();  SynNode \*cur = new SynNode{"declaration"};  try  {  var\_declaration(cur);  }  catch (runtime\_error e1)  {  cur->children.clear();  lex.toMarker(mk);  try  {  fun\_declaration(cur);  }  catch (runtime\_error e2)  {  delete cur;  cout <<"near line "<<exception\_line<<": "<< e1.what() << " or " << e2.what() << endl;  throw runtime\_error("illegel declaration");  // throw e;  }  }  //找到申明则将当前结点插入  parent->children.push\_back(cur);  } |

### SynParserLL1.hpp

主要函数代码已在3.1中给出。

### Utils.hpp

|  |
| --- |
| struct SynNode //语法树结点  {  string word; //结点内容  int type = -1; //用于输出语法树时标注终结符词素类型（id key op...）  vector<SynNode \*> children; //子节点(为空时表示根节点，即终结符)  };  struct ProUnit  { //一个终结符或非终结符  string word;  int from = -1;  bool isTerminal()  {  if (word == "NUM" || word == "ID" || word == "empty" || word == "$")  return true;  for (auto &s : KEYW)  {  if (s == word)  return true;  }  for (auto &c : OPS)  {  if (c == word[0])  return true;  }  return false;  }  //重载比较函数，与字符串比较  bool operator==(const ProUnit o) const  {  return (this->word) == o.word;  }  bool operator<(const ProUnit o) const  {  return (this->word) < o.word;  }  };  struct Item{  int pd\_idx;  int dot\_idx;  bool operator<(const Item o) const  {  if(pd\_idx<o.pd\_idx) return true;  if(pd\_idx==o.pd\_idx && dot\_idx<o.dot\_idx) return true;  return false;  }  }; |

### C-grammar.y

见2.4.4

### C-grammar.l

见2.3.3

# 测试结果



## 测试结果信息简表

|  |  |  |
| --- | --- | --- |
| **测试项目** | **Test1.c** | **Test2.c** |
| 词法扫描器（lex） | 成功 | 成功 |
| 词法扫描器（手动实现） | 成功 | 成功 |
| 语法分析器（递归下降） | 成功 | 成功 |
| 语法分析器（LL(1)） | 语法错误（非LL(1)文法） | 语法错误（非LL(1)文法） |
| 语法分析器（YACC） | 成功 | 成功 |

## 输出结果

如上表所示，由于test1.c与test2.c输出结果相似，此处仅列出各项程序在test1.c上的输出结果。测试文件内容如下：

|  |
| --- |
| int gcd (int u, int v)  {  if (v == 0)  return u ;  else  return gcd(v,u-u/v\*v);  /\* u-u/v\*v == u mod v \*/  }  void main(void)  {  int x;  int y;  x = input();  y = input();  output(gcd(x,y));  } |

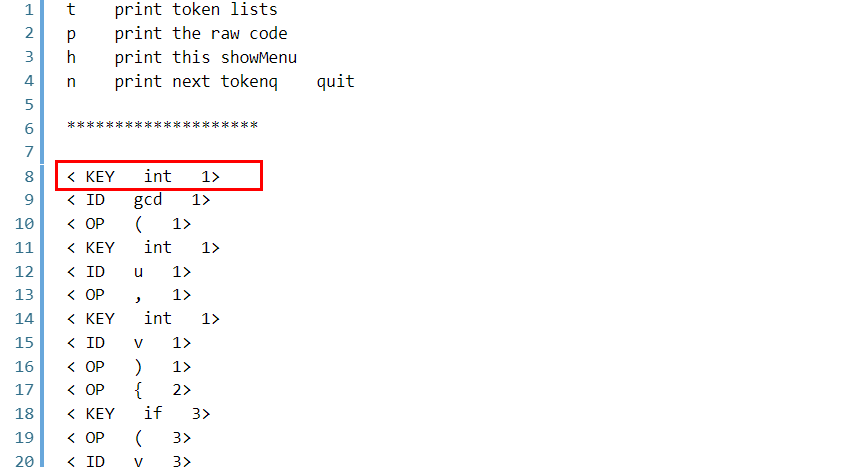
### 词法扫描器（lex）

扫描出的词法单元与测试文件一致。



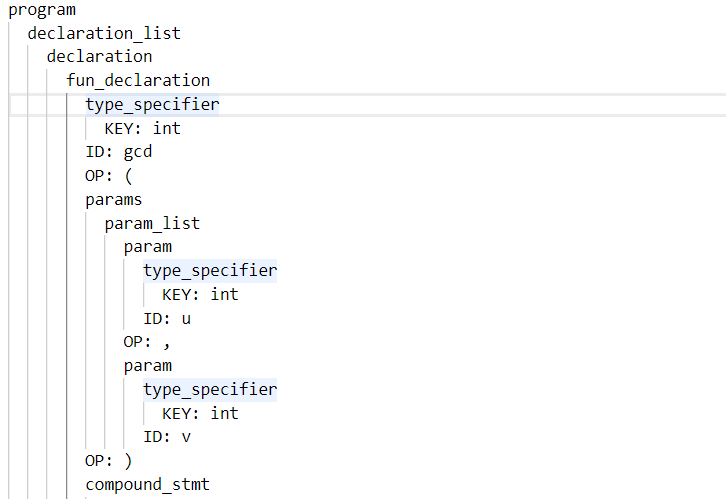
### 词法扫描器（手动实现）

如图，输出Token分别为类型、对应代码内容以及所在代码行组成。



### 语法分析器（递归下降）

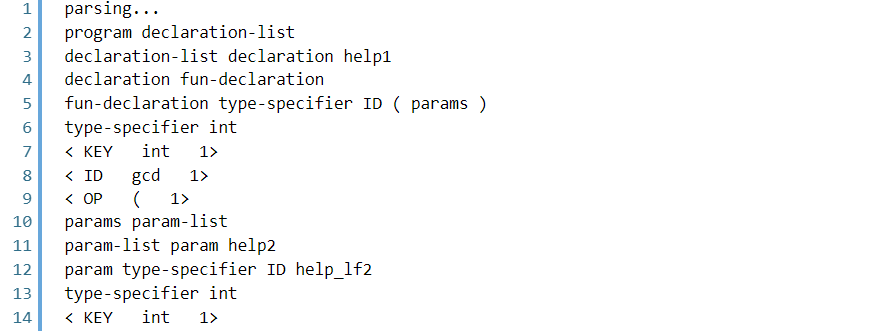
以fun\_declaration为例，下面包含子节点：type\_specifier、ID、（、params、）、compound\_stmt。与所给产生式一致，可见程序运行正确。



### 语法分析器（LL(1)）

比较测试代码与一下产生式，可以知道该分析过程基本正确，但在涉及additive-expression → additive-expression addop term|term

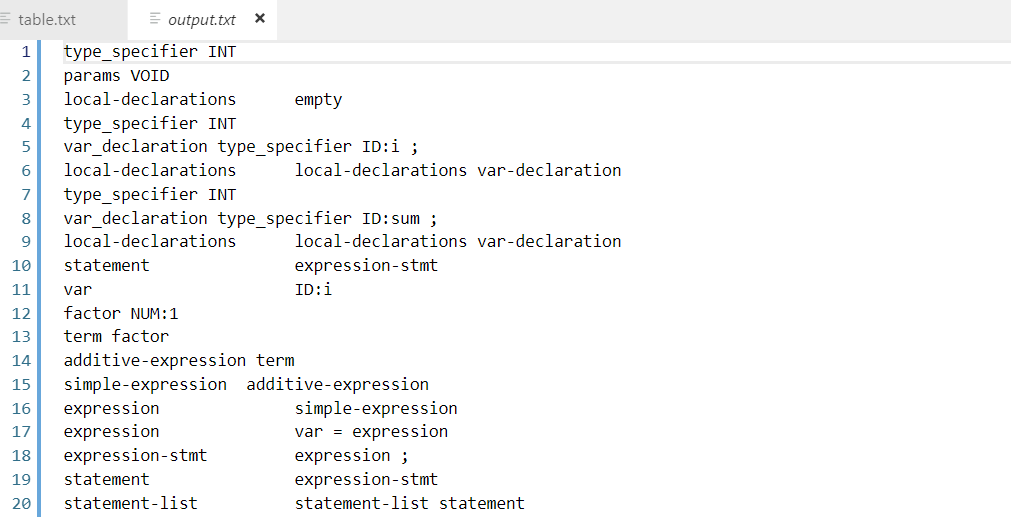
这条产生式存在深层嵌套的间接左递归，因此输出末尾tk - u:(OP); - =表示分析失败，这是因为所给文法不是LL(1)文法，且很难修改。





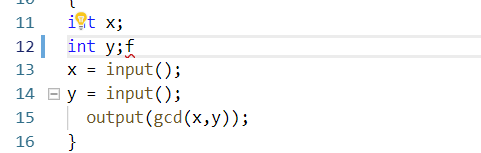
### 语法分析器（YACC）

YACC语法输出为自底向上结构，经过验证，该输出为正确的语法分析过程，可以从中建立语法分析树。

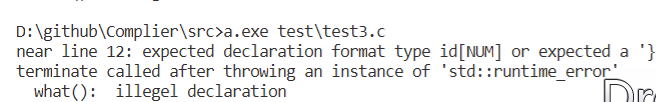


### 错误处理

修改测试文件，在第12行添加字符f使其存在语法错误



运行递归下降，可以看到程序准确判断出错误位置，但错误原因不够准确。



# 总结



## 收获

在编译原理课程设计中，我将编译原理课程所学知识进行了深入实践、这大大加深了我对相关理论知识的理解与记忆。此外，也锻炼了我的编程能力、数据结构与算法设计的能力。实验中，我一步步完成了词法扫描器、递归下降语法分析、LL(1)语法分析，还完成了部分SLR（1）语法分析，这使我深刻理解了有限状态机、FIRST\FOLLOW集的计算，预测分析表的建立、闭包算法等知识。学会了如何在较大的程序中合理的组织代码文件。此外，还利用LEX\BISON等工具实现了词法、语法分析，熟悉了相关工具的使用，体会到前辈们带来的便利。

## 不足

在课程设计中，虽然我用多种方案实现了相关目标，但分析器的错误处理我却很少考虑，因此对于递归下降方法仅能找出错误出现的大概位置和错误原因，判断仍然不够准确需要进一步优化改善。而对于LL（1）分析方法，由于没有事判断文法是否满足LL（1）文法要求，导致实现后发现无法顺利完成语法分析，付出了较大的代价。