**编译原理课程设计报告**

**课题名称： 编译器中的词法分析器及语法分析器的**

**设计与实现**

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： 金军

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 日

词法分析器、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++语言编写构造词法分析器和语法分析器两个模块。

## 词法分析器

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分析过程，从目标程序中识别词法单元。
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。

## 语法分析器

1. 调用词法分析器，获取词法单元。
2. 根据c-的文法，将词法分析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。

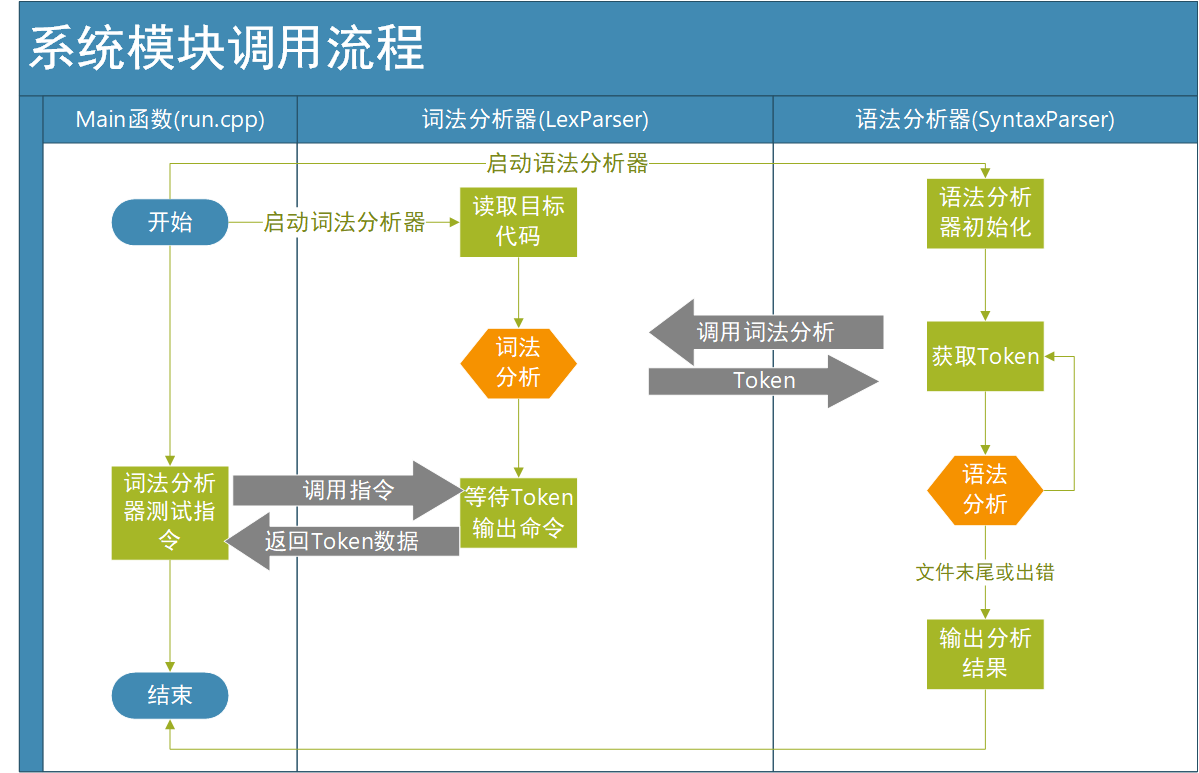
# 分析与设计

## 实现方案说明

词法分析器和采用lex工具自动生成以及手动实现有限自动机两种方案，语法分析器采用递归下降法以及尝试使用LL(1)两种方案实现。但由于所给的语法中含有复杂的嵌套左公因子，难以修改成LL(1)文法，所以改方法在运行到有冲突的非终结符产生式时可能会发生异常而终止。

## 系统模块说明

如图所示，系统分为run、词法分析、语法分析三个模块。在run函数中启动程序，如果要测试词法分析器，则main函数会调用词法分析器，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，会一边调用词法分析器获取词法单元（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。如表分别为词法分析器和语法分析器的部分输出结果示例。

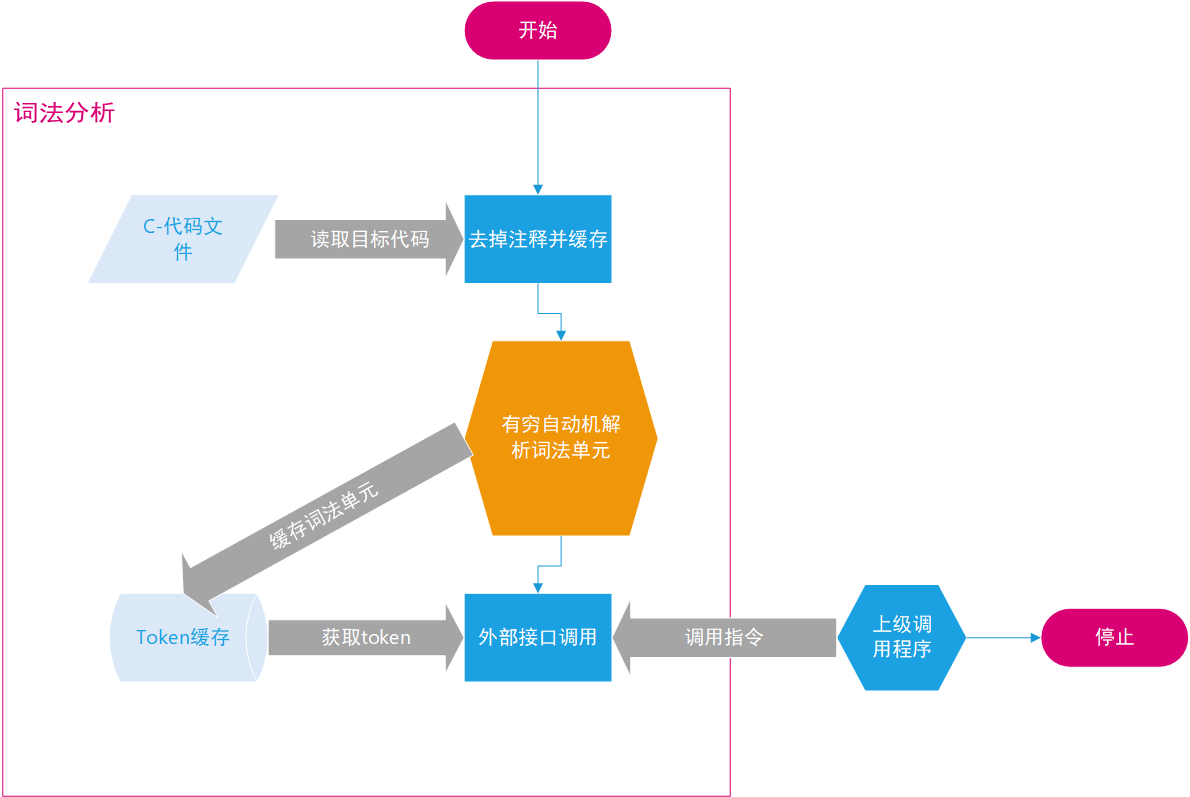


|  |  |
| --- | --- |
| 词法分析输出 | 语法分析树（params节点）输出 |
| < KEY int 1>  < ID gcd 1>  < OP ( 1>  < KEY int 1>  < ID u 1>  < OP , 1>  < KEY int 1>  < ID v 1>  < OP ) 1> | params  param\_list  param  type\_specifier  KEY: int  ID: u  OP: ,  param  type\_specifier  KEY: int  ID: v |

## 词法分析器

### 手工实现的词法分析器算法流程

词法分析器开始运行后，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。

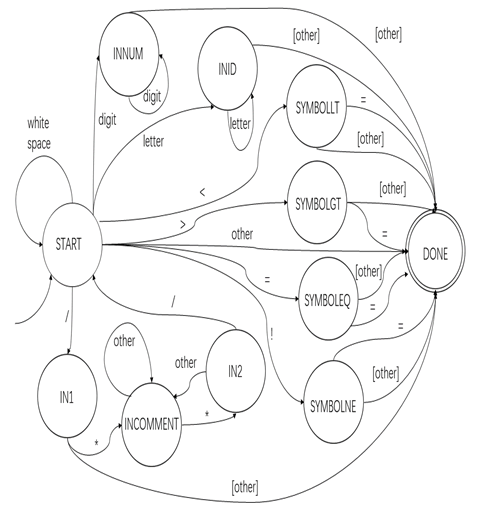


### DFA

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。



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

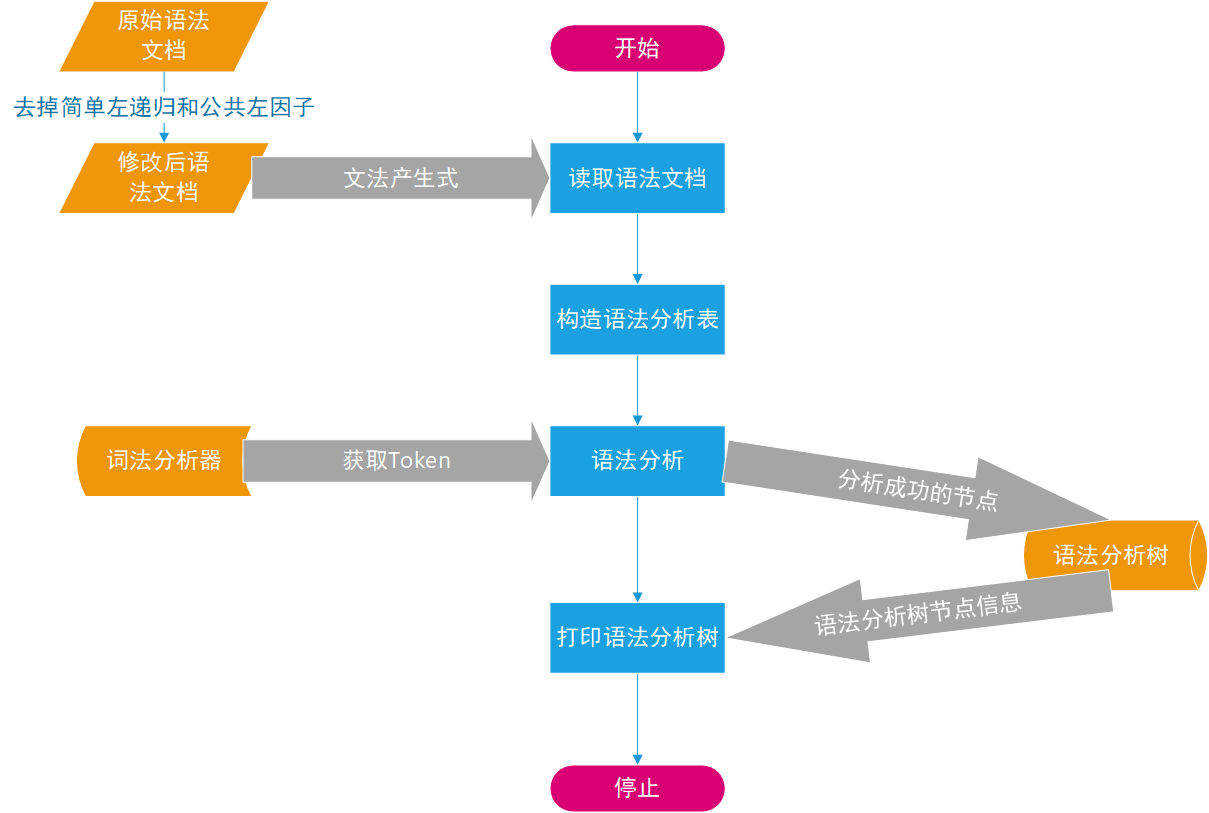
输入文件如下表所示，该文件中定义了main()函数，因此lex生成的lex.yy.c文件(文件内容太多无法在此展示)经过编译后可以直接运行并输出此法单元。

|  |
| --- |
| %{  #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;  } |

## 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分析器获取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。



|  |
| --- |
| 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>  …… |

### 语法分析器编程方法说明

先对c-语法进行整理，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 |

# 程序代码实现

# 测试结果

# 总结