编译原理课程设计报告

课题名称：构造 C- 语言的词法分析器和语法分析器

提交文档学生姓名：郭安洁

提交文档学生学号：2017141461297

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 6 月 26 日

1. 课程设计目标

构造了 c-语言的编译器的词法分析部分和语法分析部分。

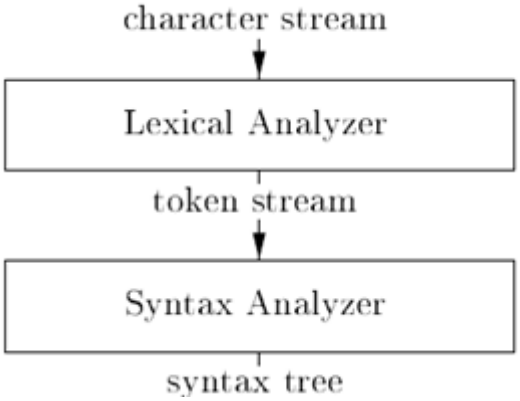
2. 分析与设计

词法分析部分运用了三种方法：手工实现：From Diagram to Lexical Analyser；手工实现：表驱动方法；lex；

语法分析部分运用了两种方法：递归向下；LL(1)。

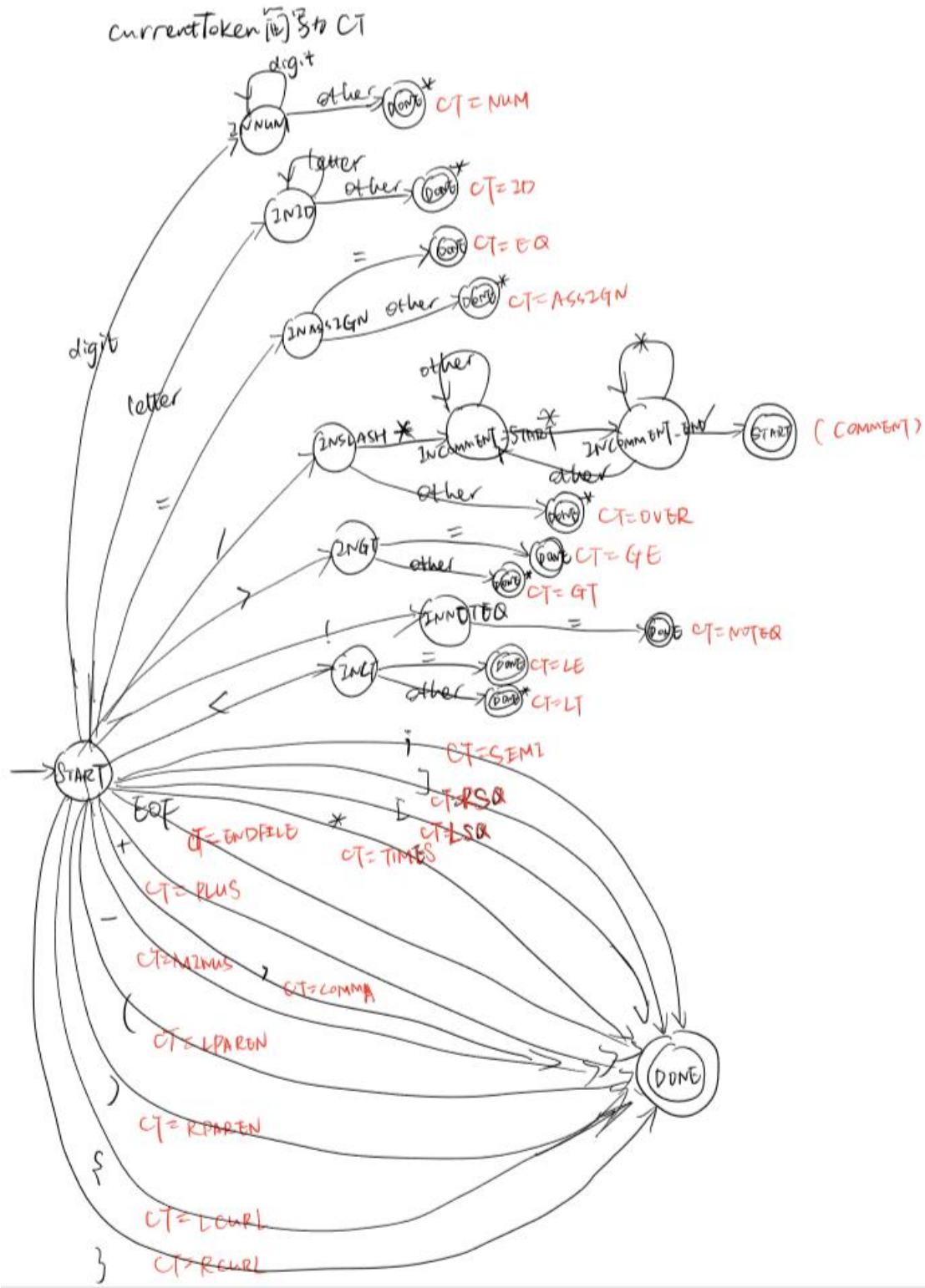
所用编程语言：c++

系统总图:



一. 扫描器:

C- DFA:



由词法规则确定正则表达式，再由正则表达式确定对应的 NFA，进而化为 DFA。

手工实现中均所用到的符号表示：以枚举的方式保存 token 类型和状态类型

```
typedef enum
    {ENDFILE,ERROR,
     IF,ELSE,INT,RETURN,WHILE,VOID,     //保留字
     ID,
     ASSIGN,EQ,GT,LT,GE,LE,NOTEQ,
     PLUS,MINUS,TIMES,OVER,LPAREN,RPAREN,SEMI,
     LCURL,RCURL,LSQ,RSQ,COMMA,NUM,NULL_TK
    } TokenType;

    typedef enum
    { START,INASSIGN,INSLASH,INCOMMENT_START,INCOMMENT_END,INNUM,
    INGT,INLT,INGE,INLE,INNOTEQ,INID,DONE }
    StateType;
     //DFA 的状态名字
```

1. **手工实现：From Diagram to Lexical Analyser**

每读到一个字符，就根据当前状态和字符类型确定下一状态，起始状态为 START，终止状态为 DONE。在循环中，用 switch-case 语句来控制状态的进入，然后用 if-else 或 switch-case 语句进行后续状态的选择，每调用一次 getToken 函数便得到一个 token 类型的返回。

(1)主要函数伪代码实例：

```
TokenType getToken:
  state=START;
  while(state!=DONE){
     c=getNextChar();
     switch(state){
          case START:
               if(isdigit(c))    state=INNUM;
               else if(c == '=')   state=INASSIGN;
               …
          case INNUM:
               …
          case …
```

```
    }
    If(save)    do save staff
    If(state==DONE)      …
}
return currentToken;
```

（2）辅助函数

```
1.  int getNextChar(void);  //读取下一个字符
2.  TokenType reservedLookup (char * s); //若当前 token 为 ID 时，判断其是否属于保留字
3.  void ungetNextChar(void);  //用于回退一个字符
```

（3）遇到的问题和解决方法：

1. 对 C-的 token 的正则表达式形式的转换，以及对 DFA 图的考虑。

2. 有的正则表达式较复杂，如注释，在代码实现的过程中也迭代地对其实现进行了修改，。

3. 考虑一些状态变量应如何设置，如 save、state、currentToken 等，分别用于记录当前 token 是否保存、当前状态、当前 token，这些变量是贯穿整个程序的关键的变量。

4. 考虑状态 state 的值用什么形式保存，要具有易于快速且方便地访问某个值；由于有很多判断语句，如果能用 switch 代替 if-else 也会更加便于写代码。最后决定用 enum 来保存。

2.手工实现：Table-driven

用表来保存状态的转移，首先初始化表的内容，然后每读入一个字符，根据当前状态和字符类型来确定要进入的状态和当前 token 的类型，以及是否需要保存当前字符（是否是有效的字符）、是否回退一个字符。初始状态是 START，接受状态是 DONE。

（1）所用数据结构：map<State_Input,TableVal> M;

说明：map 的 State_Input（key）部分：

```
struct State_Input{
 StateType st;     //当前所在 DFA 的状态
 int c;            //当前所读到的字符（ASCII 码）
};
```

Map 的 TableVal（value）部分：

```
struct TableVal{
```

```
    StateType st;          //由当前状态进入到下一个 DFA 的状态
    TokenType tk;          //判断当前 token 的类型
    int save;              //是否保存当前字符
    int unget;             //是否要回退一个字符
};
```

（2）M 的构造如下：

注：表中的 other 是相对于每种情况下的其他字符，如当前 state=INID 时，若下一字符非字母，则相当于下一字符为 other。

	letter	digit	+	-	!	white space	\t	\n	other
START	INID, ID, 1, 0	INNUM, NUM, 1, 0	DONE, PLUS, 1, 0	DONE, MINUS, 1, 0	INNOTEQ, NULL_TK, 1, 0	START, NULL_TK, 0, 0	START, NULL_TK, 0, 0	START, NULL_TK, 0, 0	
INID	INID, ID, 1, 0								DONE, ID, 0, 1
INNUM		INNUM, NUM, 1, 0							DONE, NUM, 0, 1
INASSIGN									DONE, ASSIGN, 0, 1
INSLASH									
INCOMMENT_START									
INCOMMENT_END									
INGT									
INNOTEQ									
INLT									
DONE									

	/	*	-1	=	>	<	other
START	INSLASH, NULL_TK, 1, 0	DONE, TIMES, 1, 0		INASSIGN, NULL_TK, 1, 0	INGT, NULL_TK, 1, 0	INLT, NULL_TK, 1, 0	
INID							
INNUM							
INASSIGN				DONE, EQ, 1, 0			
INSLASH		INCOMMENT_START, NULL_TK, 0, 0					DONE, OVER, 1, 1
INCOMMENT_START		INCOMMENT_END, NULL_TK, 0, 0	DONE, ENDFILE, 0, 0				INCOMMENT_START, NULL_TK, 0, 0
INCOMMENT_END	START, NULL_TK, 0, 0	INCOMMENT_END, NULL_TK, 0, 0					INCOMMENT_START, NULL_TK, 0, 0
INGT				DONE, GE, 1, 0			DONE, GT, 0, 1
INNOTEQ				DONE, NOTEQ, 1, 0			DONE, LT, 0, 1
INLT				DONE, LE, 1, 0			
DONE							

	;	,	{	}	[	]	(	)
START	DONE, SEMI, 1, 0	DONE, COMMA, 1, 0	DONE, LCURL, 1, 0	DONE, RCURL, 1, 0	DONE, LSQ, 1, 0	DONE, RSQ, 1, 0	DONE, LPAREN, 1, 0	DONE, RPAREN, 1, 0
INID								
INNUM								
INASSIGN								
INSLASH								
INCOMMENT_START								
INCOMMENT_END								
INGT								
INNOTEQ								
INLT								
DONE								

（3）主要函数原型

TokenType getToken_t(map<State_Input,TableVal> &M); //返回 token 的类型

（4）伪代码

```
getToken_t:
    state=START;
    while(state!=DONE){
        c = getNextChar();
        tbVal = M[{state,c }];
        state=tbVal.st;
        currentToken = tbVal.tk;
```

```
        if (tbVal.save)   do save staff;
        if(tbVal.unget)   do unget staff;
        if (state == DONE){
            if (currentToken == ID && currentToken is reserved word)
              currentToken = reservedLookup(tokenString_t);
         }

    }
```

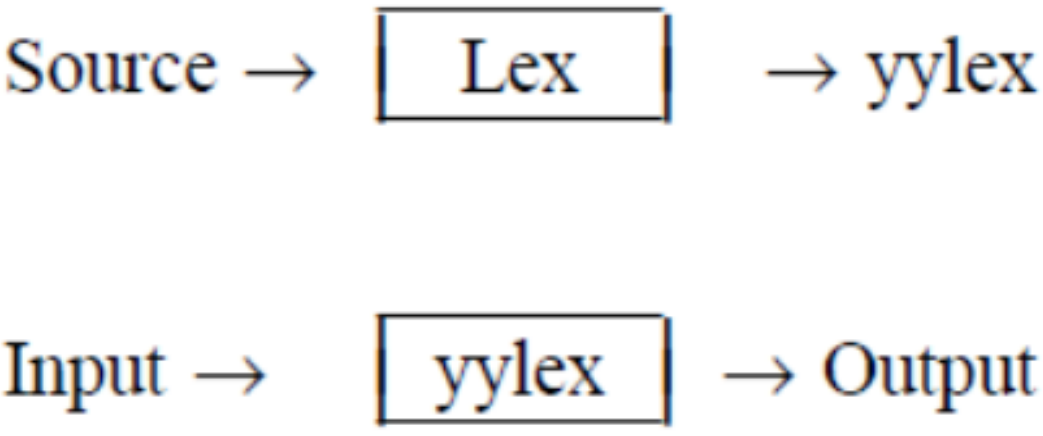
（5）遇到的问题和解决方法：

1.对数据结构的考虑：由于要建立一张表来存放状态的转移，要方便保存和查阅，并且保存和查阅都是没有顺序的。因此用 Map 来保存比较合适，key 即是当前状态的各种信息，value 是下一个状态的信息。然而考虑细节时发现，由于 key 和 value 都不止一个值，因此需要用结构体保存，并且由 DFA 的特性，每一步状态完全确定，因此是否保存当前 token、是否回退也需要记录。建立表花了较长时间，也修改了很多次，最终得到一个较为合理的版本。

2.如要读取一个 NUM 类型的 token，其间有“若遇到 digit 则继续，遇到 other 则 accept”的要求。而从编码上来讲，因为在 scan 的过程中，不再分别考虑 token 的类型和当前状态，因此表中必须对每一种情况有确定的记录，因此需要对每个[0-9]遇到任意一个[0-9]或非[0-9]都进行记录，其余的 token 类似。一开始纠结于这样做对内存的占用，但由于每个 token 的 DFA 中的 other 都是相对而言的，因此这样处理算是较为简单也便于理解的。

3.由于没有一步得到最终版的表结构，编码的前期做了一些写代码、再推翻的无用功，当然，有些问题也只有实践的时候才能暴露出来。

2. Lex



(1) flex 的说明：Lex 是 Unix 下的工

具，windows 下用 flex 代替

先定义.l 文件。flex 的输入文件由三段组成，用一行中只有%%来分隔。

```
定义;definition
%%
规则;rules
%%
用户代码;code
```

使用 flex *.l 命令生成 lex.yy.c 文件，编译并运行该程序即可。

（2）输入文件 flexInput.l

```
%{
#include<math.h>
#include<stdlib.h>
#include<stdio.h>
%}

DIGIT [0-9]
ID [a-z][a-z0-9]*
%%
{DIGIT}+                    {printf("整数:   %s(%d)\n",yytext,atoi(yytext));}
{DIGIT}+"."{DIGIT}+             {printf("实数:   %s(%g)\n",yytext,atof(yytext));}
if|else|int|return|void|while       {printf("关键字: %s\n",yytext);}
{ID}                        {printf("标识符: %s\n",yytext);}
"+"|"-"|"*"|"/"|"="                 {printf("运算符: %s\n",yytext);}
"=="                            {printf("判断相等: %s\n",yytext);}
"!="                            {printf("判断不相等: %s\n",yytext);}
">"                          {printf("greater than: %s\n",yytext);}
"<"                          {printf("less than: %s\n",yytext);}
">="                         {printf("greater/equal: %s\n",yytext);}
"<="                         {printf("less/equal: %s\n",yytext);}
"("|")"|"["|"]"|"{"|"}"|","|";"      {printf("%s\n",yytext);}
```

```
"/*"([^*]|[\r\n]|("*"+([^*/]|[\r\n])))*"*"+"/"          {printf("");}
" "+|[\t\n\x20]+;           {printf("");}
.                       {printf("不能识别的字符:%s\n",yytext);}
%%
int main(int argc,char **argv)
{
    ++argv;
    --argc;
    if(argc>0) yyin=fopen(argv[0],"r");
    else yyin=stdin;
    yylex();
    return 0;
}
int yywrap()
{
    return 1;
}
```

在控制台使用 flex flexInput.l 命令生成 lex.yy.c 文件，编译并运行该 c 文件即可。

（3）遇到的问题和解决方法：

1.由于对 lex 的机制没有了解，因此上网查阅了许多相关资料，也参考了不同的解决方法，对 flex 大体上有了认识。

2.对于 flexInput.l 文件的编写：查阅了资料后发现，这个文件的编写只需要修改对 token 的定义的部分，类似于正则表达式，但个别的正则表达式较复杂，如注释，则查阅了其构造方法，得到其正则表达式。

3.对于如何运行 flex，参考了网上的经验，进行试验后运行成功。

二. parser

1. 递归向下

(1) 数据结构（语法树）

```
typedef enum {StmtK,ExpK} NodeKind;
typedef enum {IfK,WhileK,ReturnK,AssignK,DeVarK,DeFunK,CompoundK} StmtKind;
typedef enum {OpK,ConstK,IdK,TypeK,SubscriptK,ParamK,CallK} ExpKind;
typedef enum {Void,Integer,Boolean} ExpType;
typedef struct treeNode
   { struct treeNode * child[MAXCHILDREN];  //子节点
     struct treeNode * sibling;    //兄弟节点
     int lineno;  //当前在多少行
     NodeKind nodekind;    //节点的种类
     union { StmtKind stmt; ExpKind exp;} kind;
     union { TokenType op;
             int val;
             char * name; } attr;
     ExpType type; /* for type checking of exps */
   } TreeNode;
```

(2) 主要函数的声明

```
TreeNode * declaration_list(void);
TreeNode * declaration(void);
TreeNode * compound_stmt();
TreeNode * type_specifier();
TreeNode * subscript_exp();
TreeNode * params_exp();
TreeNode * param_exp();
TreeNode * if_stmt();
TreeNode * while_stmt();
```

```
10. TreeNode * assign_stmt(void);
11. TreeNode * return_stmt();
12. TreeNode * exp(void);
13. TreeNode * simple_exp(void);
14. TreeNode * term(void);
15. TreeNode * factor(void);
16. TreeNode * id();
17. TreeNode * var_declaration();
18. TreeNode * localDeclarations();
19. TreeNode * statement_list();
20. TreeNode * statement();
21. TreeNode * args();
22. void match(TokenType expected);
23. TreeNode * parse();
```

(3) 将 BNF 文法改写为 EBNF 文法

program → declaration-list

declaration-list → declaration{declaration}

type-specifier → **int**

type-specifier →void

declaration-> type-specifier **ID [[NUM]];**

declaration-> type-specifier **ID (**params**)** compound-stmt

params → param-list

params → void

param-list→ param {,param}

param → type-specifier **ID {[]}**

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

local-declarations → empty{var-declaration}

compound-stmt → **{** empty{var-declaration} statement-list **}**

statement-list → empty{statement}

statement → expression-stmt | compound-stmt| selection-stmt

| iteration-stmt | return-stmt

expression-stmt → expression **;** | **;**

selection-stmt → **if (**expression**)** statement [**else** statement]

iteration-stmt → **while (** expression **)** statement

return-stmt → **return [expression];**

expression → var = expression | simple-expression

var →**ID [[** expression **]]**

simple-expression → additive-expression [relop additive-expression]

relop → **<=** | **<** | **>** | **>=** | **==** | **!=**

additive-expression →term {addop term}

addop → **+** | **-**

term →factor {mulop factor}

mulop → ***** | **/**

factor → **(** expression **)** | var | call | **NUM**

call → **ID (** args **)**

args -> empty{, expression}

（4）实现思路

用递归调用的方式解决问题，语法分析的入口是 parse 函数，由于要建立抽象语法树，每个种类的节点为一次函数调用，返回值是树的节点的指针，在“遇到非终结符调用，遇到终结符 match”的原则下，在函数调用的过程中将语法树建成。

最后输出语法树也是递归打印输出，根据当前节点的类型确定输出的内容，对当前节点的兄弟节点循环打印即可，对当前节点的子节点则要递归调用打印函数进行打印。

（5）遇到的问题和解决方法

1.在编写函数时，发现部分改写后的 ebnf 文法显得冗余，而其实可以在编程过程中简化，如 addop → + | -这条规则，可以单独写成一个函数，也可以直接用 if 语句代替。

2.在考虑抽象语法树的结构时，如对于规则 declaration-> type-specifier ID [[NUM]];，在逻辑上先确定 type-specifier，ID，[[NUM]]的关系，但实际编码中由编码的难易以及处理方法会造成一定偏差，因此在对语法树应该如何构成进行了较多的思考，相应的在对数据结构的考虑上也花费了较多时间，调整不同类型的节点以及节点的分类。

2. LL(1)

(1) 数据结构

1.用于存放 C-语法的产生式、first 集合、follow 集合

```
multimap<string,vector<string> > production;
//保存产生式的左边非终结符（以字符串形式）和右边每个元素按顺序的字符串形式

map<VN,set<TableItem> > first;
map<VN,set<TableItem> > follow;
struct TableItem{
    TokenType t;
    string p;    //t 对应的 production
    vector<P_Token> p_tk;
};
/*
保存 first 和 follow 集合，非终结符以 VN 这一数据结构保存，对应的集合元素为 TableItem，保存了终结符的类型 TokenType 和当前终结符对应的产生式（string）、将产生式各个元素打断后的 vector<P_Token>。每个非终结符的集合元素保存在 set<TableItem>中。
*/
```

2.和 parsing table 相关的数据结构

```
map<TableKey,TableVal> pTable;    //parsing table
struct TableKey{
    VN vn;
    TokenType t;
};
struct  TableVal{
    string p_str;
    vector<P_Token> p_tk;
};
```

```
/*
TableVal 需要保存两种信息，一个是对应的产生式，用于打印语法树，一个是将产生式右边的每个元素打断，然后保存到 vector<P_Token> p_tk;，用于语法分析时的压栈。
*/
```

3.进行具体的 parse 时的 stack 的数据结构以及语法树的节点的数据结构

```
struct P_Token{  //一个“字母”对应的是 vn/vt 以及值
union{ VN vn; TokenType t; } attr;
bool isVn; //是否是非终结符
};
stack<P_Token> s;    //用于 parse 的最根本的栈
stack<TreeNode*> treeStack;     //parse 时构造语法树

typedef struct treeNode{
      struct treeNode * child;
      struct treeNode * sibling;
      string pro;     //输出的产生式
      TreeNode(){
         child=NULL;
         sibling=NULL;
         }
   } TreeNode;
```

3. 根据语法规则定义的 TokenType（vn）和 VT

```
      typedef enum {
      ENDFILE,ERROR,
      /* reserved words */
      IF,ELSE,INT,RETURN,WHILE,VOID,
      ID,
```

```
    ASSIGN,EQ,GT,LT,GE,LE,NOTEQ,
        PLUS,MINUS,TIMES,OVER,LPAREN,RPAREN,SEMI,
        LCURL,RCURL,LSQ,RSQ,COMMA,NUM,NULL_TK,PARSER_START
   } TokenType;

typedef enum {Program,DeList,DeList1,De,De1,VarDe,FunDe,Type,
Type1,Params,ParamList,ParamList1,Param,Param1,CmpdStmt,LocalDes,   //15
LocalDes1,StmtList,StmtList1,Stmt,Stmt2,ExpStmt,SelecStmt,IterStmt,ReturnStmt,
Exp,Exp1,SelecStmt1,ReturnStmt1,SimpleExp,SimpleExp1,Relop,AddExp,AddExp1,Addop,
Mulop,Term,Term1,Factor,Factor1,Var,Call,Args,ArgList,ArgList1
 } VN;
```

(2) 主要函数的原型

```
void getFirst();
void getFollow();
void initParsingTable();
TreeNode * parse();
```

(3) 改写的语法

program - declarationList

program - EOF

declarationList - declaration declarationList1

declarationList1 - declaration declarationList1

declarationList1 - ep

declarationList1 - EOF

declaration - typeSpecifier ID declaration1

declaration1 – varDeclaration

declaration1 – funDeclaration

varDeclaration - type1

funDeclaration - (params) compoundStmt

type1 - ;

type1 - [NUM] ;

typeSpecifier - int

typeSpecifier - void

params - paramList

params - void

paramList - param paramList1

paramList1 - , param paramList1

paramList1 - ep

param - typeSpecifier ID param1

param1 - [ ]

param1 - ep

compoundStmt - { localDeclarations statementList }

localDeclarations - localDeclarations1

localDeclarations1 - declaration localDeclarations1

localDeclarations1 – ep

statementList - statementList1

statementList1 - statement statementList1

statementList1 - ep

statement - expressionStmt

statement - selectionStmt

statement - iterationStmt

statement - compoundStmt

statement - returnStmt

expressionStmt - expression ;

expressionStmt - ;

selectionStmt - if (simpleExpression) statement2 selectionStmt1

statement2 - expressionStmt

statement2 - iterationStmt

statement2 - compoundStmt

statement2 - returnStmt

selectionStmt1 - else statement2

selectionStmt1 - ep

iterationStmt - while (simpleExpression) statement

returnStmt - return returnStmt1

returnStmt1 - ;

returnStmt1 - simpleExpression ;

expression - factor expression1

expression1 - = simpleExpression

expression1 - ep

simpleExpression - additiveExpression simpleExpression1

simpleExpression1 - relop additiveExpression

simpleExpression1 - ep

relop - <=

relop - <

relop - >=

relop - >

relop - ==

relop - !=

addop - +

addop - -

mulop - *

mulop - /

additiveExpression - term additiveExpression1

additiveExpression1 - addop term additiveExpression1

additiveExpression1 - ep

term - factor term1

term1 - mulop factor term1

term1 - ep

factor - (simpleExpression)

factor - ID factor1

factor1 - var

factor1 - call

factor - NUM

call - (args)

var - [simpleExpression]

var - ep

args - argList

args - ep

argList - simpleExpression argList1

argList1 - , simpleExpression argList1

argList1 - ep

(4) 实现思路

1. 将 C-语法规则的左递归去除（以及间接的左递归），提取左因子。
2. 计算出每个 VN 的 first、follow 集合
3. 通过 first、follow 集合构造出 parsing table
4. 通过 paring table 来进行 parse

(5) 遇到的问题和解决方法：

1.改写语法时，一开始没有意识到有的间接左递归的存在，因此调试时出错但找不到问题；修改间接左递归时由于不能直接用套路修改，需要结合对文法的理解进行适当变化来修改，而修改的文法有时十分片面，不能彻底解决问题，只有在调试的时候才会发现问题，因此修改文法这个过程几乎贯穿了开发的过程。

2.求 first 和 follow 集合的时候，首先是数据结构的选取，由产生式和 first、follow 集合的特性，用 map 存放 first 和 follow 集合，key 即是非终结符，而 value 的数据结构则经过了多次考虑才得出最终版本。首先 value 是个集合 set，而 set 中每个元素的类型不好确定，若只保存 TokenType，后期语法分析时则不能满足要求。数据结构的修改进行了多次，而每次修改都有一定的工作量，因此在这一部分耗费了较多时间。由于最终版本的数据结构较为复杂，一些基本操作如 copy、erase 等要重新编写，并且要切合需要，操作上有一定难度。程序没有理想的调试方法，所以编写求 first 和 follow 集合的程序要考虑很多方面，容易出错。

3.构建语法分析表时，由数据结构的选择，有较多指针操作，并且难以调试，需要仔细考虑。

4.语法分析时，即在调用函数 parse 时，要建语法树，也涉及大量的对数据结构的应用，还要定义语法树的节点 TreeNode，分析时指针操作建树。此阶段容易出现死循环、指针赋值不当等问题。编写时，有一步在不应该给指针赋值时将指针设置为 null，由于调试困难，在不明白具体原因时我将重点放在另外一部分，忽视了真正的错误位置，在这个问题上花费了大量时间。因此在解决程序问题时不能想当然的认为错误的位置，要顾及各个部分。

3.程序代码实现

（1）词法分析器手工实现：From Diagram to Lexical Analyser

部分代码

```
1.  TokenType getToken(){    //调用此函数，返回当前token的类型
2.     int tokenStringIndex = 0;
3.
4.     TokenType currentToken;
5.     StateType state = START; //初始化状态为开始状态
6.     int save;
7.     while (state != DONE){  //如果没有到接受状态
8.       int c = getNextChar(); //c：下一个字符
9.       //printf("%c",c);
10.      save = TRUE;
11.      switch (state) //根据当前状态和输入的符号的类型判断下一个状态
12.      { case START:
13.          if (isdigit(c))
14.            state = INNUM;
15.          else if (isalpha(c))
16.            state = INID;
17.          else if (c == '=')
18.            state = INASSIGN;
19.          else if (c == '/')
20.            state = INSLASH;
21.          else if ((c == ' ') || (c == '\t') || (c == '\n'))
22.            save = FALSE;
23.          else if (c == '>'){
24.             state = INGT;
25.          }
26.          else if (c == '<'){
27.             state = INLT;
28.          }
```

```
29.         else if (c == '!'){
30.            state = INNOTEQ;
31.         }
32.         else   //不是以上符号，则表示是以下可以确定的终结符
33.         { state = DONE;
34.           switch (c)
35.           { case EOF:
36.               save = FALSE;
37.               currentToken = ENDFILE;
38.               break;
39.             case '+':
40.               currentToken = PLUS;
41.               break;
42.             case '-':
43.               currentToken = MINUS;
44.               break;
45.             case '*':
46.               currentToken = TIMES;
47.               break;
48.             case '(':
49.               currentToken = LPAREN;
50.               break;
51.             case ')':
52.               currentToken = RPAREN;
53.               break;
54.             case '{':
55.               currentToken = LCURL;
56.               break;
57.             case '}':
58.               currentToken = RCURL;
59.               break;
60.             case '[':
```

```
61.                 currentToken = LSQ;
62.                 break;
63.               case ']':
64.                 currentToken = RSQ;
65.                 break;
66.               case ',':
67.                  currentToken = COMMA;
68.                  break;
69.               case ';':
70.                 currentToken = SEMI;
71.                 break;
72.               default:
73.                 currentToken = ERROR;
74.                 break;
75.             }
76.           }
77.           break;
78.           //如果是其他状态，如下
79.         case INSLASH:
80.             if(c == '*'){
81.                 state = INCOMMENT_START;
82.                 save = FALSE;
83.             }
84.             else{
85.                 state = DONE;
86.                 currentToken = OVER;
87.                 ungetNextChar();
88.             }
89.             break;
90.         case INCOMMENT_START:
91.             save = FALSE;
92.             tokenStringIndex=0;
```

```
93.             if(c=='*'){
94.                 state=INCOMMENT_END;
95.             }
96.             break;
97.         case INCOMMENT_END:
98.             save = FALSE;
99.             if(c=='*'){
100.                state=INCOMMENT_END;
101.            }
102.            else if(c=='/'){
103.                state = START;
104.            }
105.            else{
106.                state = INCOMMENT_START;
107.            }
108.            break;
109.        case INGT:
110.            if(c=='='){
111.                state = INGE;
112.            }
113.            else{
114.                currentToken = GT;
115.                save=FALSE;
116.                ungetNextChar();
117.                state = DONE;
118.            }
119.            break;
120.         case INGE:
121.            currentToken = GE;
122.            save=FALSE;
123.            ungetNextChar();
124.            state = DONE;
```

```
125.             break;
126.         case INLT:
127.             if(c=='='){
128.                 state = INLE;
129.             }
130.             else{
131.                 currentToken = LT;
132.                 save=FALSE;
133.                 ungetNextChar();
134.                 state = DONE;
135.             }
136.             break;
137.         case INLE:
138.             currentToken = LE;
139.             save=FALSE;
140.             ungetNextChar();
141.             state = DONE;
142.             break;
143.         case INNOTEQ:
144.             state = DONE;
145.             if(c == '='){
146.                 currentToken = NOTEQ;
147.             }
148.             else{
149.                 currentToken = ERROR;
150.             }
151.             break;
152.        case INASSIGN:
153.          state = DONE;
154.          if (c == '=')
155.            currentToken = EQ;    // ==
156.          else
```

```
157.         { /* backup in the input */
158.           ungetNextChar();
159.           save = FALSE;
160.           currentToken = ASSIGN;
161.         }
162.         break;
163.       case INNUM:
164.         if (!isdigit(c))
165.         { /* backup in the input */
166.           ungetNextChar();
167.           save = FALSE;
168.           state = DONE;
169.           currentToken = NUM;
170.         }
171.         break;
172.       case INID:
173.         if (!isalpha(c))
174.         { /* backup in the input */
175.           ungetNextChar();
176.           save = FALSE;
177.           state = DONE;
178.           currentToken = ID;
179.         }
180.         break;
181.       case DONE:
182.       default: /* should never happen */
183.         fprintf(listing,"Scanner Bug: state= %d\n",state);
184.         state = DONE;
185.         currentToken = ERROR;
186.         break;
187.     }
188.     if ((save) && (tokenStringIndex <= MAXTOKENLEN))    //如果要保存
```

```
189.       tokenString[tokenStringIndex++] = (char) c;
190.     if (state == DONE)     //如果已经是接受状态
191.     { tokenString[tokenStringIndex] = '\0';
192.       if (currentToken == ID)  //如果是 ID 类型，再看是不是保留字
193.         currentToken = reservedLookup(tokenString);
194.     }
195.   }
196.   if (TraceScan) {
197.     fprintf(listing,"\t%d: ",lineno);
198.     TokenType tt =  currentToken;
199.     //test(currentToken);
200.     printToken(currentToken,tokenString);  //打印出当前的 token
201.     //test(currentToken);
202.   }
203.   return currentToken;
204.}
```

（2）词法分析器手工实现：Table-driven

建立语法分析表的函数代码

```
1.  map<State_Input,TableVal> M;
2.  void initTable(){
3.      //将各种终结符的项填入表中，不需读取下一个
4.      M[{START,' '}]={START,NULL_TK,0,0}; M[{START,'\n'}]={START,NULL_TK,0,0};
5.      M[{START,'\t'}]={START,NULL_TK,0,0}; M[{START,-1}]={DONE,ENDFILE,0,0};
6.      M[{START,'+'}]={DONE,PLUS,1,0};
7.      M[{START,'-'}]={DONE,MINUS,1,0}; M[{START,'*'}]={DONE,TIMES,1,0};
8.      M[{START,';'}]={DONE,SEMI,1,0}; M[{START,'('}]={DONE,LPAREN,1,0};
9.      M[{START,','}]={DONE,COMMA,1,0}; M[{START,')'}]={DONE,RPAREN,1,0};
10.     M[{START,'{'}]={DONE,LCURL,1,0}; M[{START,'['}]={DONE,LSQ,1,0};
11.     M[{START,'}'}]={DONE,RCURL,1,0}; M[{START,']'}]={DONE,RSQ,1,0};
12.
```

```
13.     //ID
14.     M[{START,'a'}]={INID,ID,1,0};
15.     M[{INID,'a'}]={INID,ID,1,0};
16.     for(int i=-1;i<128;i++){ //'other' for ID
17.         if(!isalpha(i)){
18.             M[{INID,i}]={DONE,ID,0,1};
19.         }
20.     }
21.     //number
22.     M[{START,'0'}]={INNUM,NUM,1,0};
23.     M[{INNUM,'0'}]={INNUM,NUM,1,0};
24.     for(int i=-1;i<128;i++){ //'other' for NUM
25.         if(!isdigit(i)){
26.             M[{INNUM,i}]={DONE,NUM,0,1};
27.         }
28.     }
29.     //遇到等号，分为赋值和判断相等
30.     M[{START,'='}]={INASSIGN,NULL_TK,1,0};
31.     M[{INASSIGN,'='}]={DONE,EQ,1,0};
32.     for(int i=-1;i<128;i++){ //'other' for '='
33.         if(i!='='){
34.             M[{INASSIGN,i}]={DONE,ASSIGN,0,1};
35.         }
36.     }
37.      //遇到/，分为除法和注释
38.     M[{START,'/'}]={INSLASH,NULL_TK,1,0};
39.     M[{INSLASH,'*'}]={INCOMMENT_START,NULL_TK,0,0};
40.     for(int i=-1;i<128;i++){ //'other' for '/'
41.         if(i!='*'){
42.             M[{INSLASH,i}]={DONE,OVER,1,1};
43.             M[{INCOMMENT_START,i}]={INCOMMENT_START,NULL_TK,0,0};
44.         }
```

```
    }
    M[{INCOMMENT_START,-1}]={DONE,ENDFILE,0,0};
    M[{INCOMMENT_START,'*'}]={INCOMMENT_END,NULL_TK,0,0};
    M[{INCOMMENT_END,'/'}]={START,NULL_TK,0,0};
    M[{INCOMMENT_END,'*'}]={INCOMMENT_END,NULL_TK,0,0};
    for(int i=-1;i<128;i++){ //'other' for '/','*'
        if(i!='*' && i!='/'){
            M[{INCOMMENT_END,i}]={INCOMMENT_START,NULL_TK,0,0};
        }
    }
     //用于比较的符号
    M[{START,'>'}]={INGT,NULL_TK,1,0};
    M[{START,'<'}]={INLT,NULL_TK,1,0};
    M[{START,'!'}]={INNOTEQ,NULL_TK,1,0};
    M[{INGT,'='}]={DONE,GE,1,0}; M[{INLT,'='}]={DONE,LE,1,0}; M[{INNOTEQ,'='}]={DONE,NOTEQ
,1,0};
    for(int i=-1;i<128;i++){ //'other' for '/','*'
        if(i!='='){
            M[{INGT,i}]={DONE,GT,0,1};
            M[{INLT,i}]={DONE,LT,0,1};
            M[{INNOTEQ,i}]={DONE,ERROR,0,1};
        }
    }
}
```

（3）递归向下

程序构成为以非终结符为函数名的函数以及相应的一些辅助函数，在此仅举一例

```
TreeNode * var_declaration(){  //used param / local declaration
    TreeNode * t = newStmtNode(DeVarK);  //变量声明类型的节点
    if(t!=NULL){
        t->child[0] = type_specifier();  //第一个子节点为type
        t->child[1] = id();   //第二个子节点为id
    }
```

```
    if(token==LSQ){  //if array
        if(t!=NULL) t->child[2] = subscript_exp();  //调用函数
    }
    match(SEMI);  //遇到终结符则match
    return t;  //返回构造的函数声明节点
}
```

match（）函数：

```
static void match(TokenType expected){
    if (token == expected) token = getToken();
    else {
        switch(expected){  //以下是错误处理，处理不匹配问题
            case RPAREN: syntaxError("expected -> )"); break;
            case LPAREN: syntaxError("expected -> ("); break;
            case RCURL: syntaxError("expected -> }"); break;
            case LCURL: syntaxError("expected -> {"); break;
            case RSQ: syntaxError("expected -> ["); break;
            case LSQ: syntaxError("expected -> ]"); break;
            case SEMI: syntaxError("expected -> ;"); break;
            case COMMA: syntaxError("expected -> ,"); break;
            case ID: syntaxError("expected -> ID"); break;
            case NUM: syntaxError("expected -> NUM"); break;
            default: syntaxError("unexpected token -> ");
                     printToken(token,tokenString);
                     fprintf(listing,"      "); break;
        }
    }
}
```

（4）Ⅱ（1）

部分代码

```
void getFirst(){  //求first集合
    bool isChanged = true;
    multimap<string,vector<string> >::iterator it;
    while(isChanged){
        isChanged=false; //看一轮下来有没有变化
        for(it=production.begin(); it!=production.end(); it++){ //for every production
            bool insertE=true;
            vector<string> tmpStrSet=it->second; //right side of a production
```

```
9.
10.             VN itFirst;
11.             getVn(it->first,itFirst); //产生式左边的VN，由string形式->VN形式
12.
13.             set<TableItem> tmpSet=(first.find(itFirst)->second); //current first set
14.             int beforeLen=tmpSet.size(),afterLen;
15.             for(int i=0;i<tmpStrSet.size();i++){
16.                 TokenType t;
17.                 VN v; //v:产生式右边
18.                 TableItem tbItem;
19.                 if(getVt(tmpStrSet[i],t)){ //terminal
20.                     tbItem.p=getProStr(it->first,tmpStrSet); //产生式输出形式
21.                     tbItem.t=t;
22.                     getProToken(tbItem.p_tk,tmpStrSet);
23.                     tmpSet.insert(tbItem);  //加入新的元素
24.                     insertE = false;
25.                     break;
26.                 }
27.                 else if(getVn(tmpStrSet[i],v)){  //non-terminal
28.                     set<TableItem> vnSet=(first.find(v)->second);  //该vn的first
29.                     TableItemErase(vnSet,NULL_TK);  //去掉ep
30.
31.                     set<TableItem>::iterator v_it;
32.                     for(v_it=vnSet.begin();v_it!=vnSet.end();v_it++){
33.                         tbItem.p=getProStr(it->first,tmpStrSet);
34.                         tbItem.t=(*v_it).t;
35.                         getProToken(tbItem.p_tk,tmpStrSet);
36.                         tmpSet.insert(tbItem);
37.                     }
38.
39.                     if(TableItemHasNull(first.find(v)->second)==false){  //v的first不含
    ep
```

```
                        insertE = false;
                        break;
                    }
                }
            }
            if(insertE==true){//first 中的 ep:建表时只用于判断是否需要考虑 follow 集合
                TableItem ttbItem;
                ttbItem.p="";
                ttbItem.t=NULL_TK;
                tmpSet.insert(ttbItem);  //插入空
            }
            afterLen=tmpSet.size();  //更新集合的长度
            if(afterLen>beforeLen){  //若长度有变化，即有新加入的元素
                isChanged=true;
                (first.find(itFirst)->second)=tmpSet;
            }
        }
    }
}
```

```
void getFollow(){  //求 follow 集合
    //follow
    bool isChanged = true;
    multimap<string,vector<string> >::iterator it;

    while(isChanged){
        isChanged=false;
        for(it=production.begin();it!=production.end();it++){
            vector<string> tmpStrSet=it->second;
            VN itFirst;
            getVn(it->first,itFirst);    //产生式左边的 VN，由 string 形式->VN 形式
```

```
            // cout<<"tmpStrSet = "<<tmpStrSet<<endl;
            for(int i=0;i<tmpStrSet.size();i++){ //分别处理产生式右边的从左到右的非终结符
                bool hasE=false;
                TokenType t;
                VN v;
                if(getVn(tmpStrSet[i],v)){ //non-terminal, v:current VN,  以下处理vn：v
                    set<TableItem> tmpSet=(follow.find(v)->second);
                    int beforeLen=tmpSet.size(),afterLen;
                    if(i!=tmpStrSet.size()-1){  //current not the last non-terminal
                        if(getVt(tmpStrSet[i+1],t)){    //if the next is a terminal
                            TableItem x;
                            x.t=t;
                            x.p=getProEp(tmpStrSet[i]);     //当前处理的是tmpStrSet[i]的follow集合
                            getProEpToken(x.p_tk);
                            tmpSet.insert(x);
                        }
                        else{  //the next is non-terminal
                            hasE=true;
                            for(int j=i+1;j<tmpStrSet.size();j++){
                                if(getVt(tmpStrSet[j],t)){ //the next is terminal
                                    // set<TableItem> tmpSet1=follow.find(itFirst)->second; //????
                                    TableItem x;
                                    x.t=t;
                                    x.p=getProEp(tmpStrSet[i]);
                                    getProEpToken(x.p_tk);
                                    tmpSet.insert(x);
                                    hasE=false;
                                    break;
                                }
                                //the next is non-terminal
```

```
                                getVn(tmpStrSet[j],v);
                                set<TableItem> tmpSet1=first.find(v)->second; //加入该vn的first集合
                                if(TableItemHasNull(tmpSet1)==false){  //j的first不含ep
                                    hasE=false;
                                }
                                TableItemErase(tmpSet1,NULL_TK); //删除

                                set<TableItem> x;
                                x=copyTableItem(tmpSet1,tmpStrSet[i]);
                                tmpSet.insert(x.begin(),x.end());
                                if(hasE==false){  //如果没有空，表示到头了
                                    break;
                                }
                            }
                        }
                    }
                    if(i==tmpStrSet.size()-1 || hasE){ //if the last non-terminal || 该vn右边全为vn并且first都含ep
                        set<TableItem> tmpSet1=follow.find(itFirst)->second; //加入产生式左边的VN的follow
                        set<TableItem> x;
                        x=copyTableItem(tmpSet1,tmpStrSet[i]);
                        tmpSet.insert(x.begin(),x.end());
                    }
                    afterLen=tmpSet.size();
                    getVn(tmpStrSet[i],v);
                    if(afterLen>beforeLen){  //若有变化，更新集合
                        isChanged=true;
                        (follow.find(v)->second)=tmpSet;
                    }
                }
```

```
            }
        }
    }
}
```

```
void initParsingTable(){  //初始化语法分析表
    map<VN,set<TableItem> >::iterator firstIt,followIt;
    set<TableItem>::iterator tbItemSetIt;
    for(firstIt=first.begin();firstIt!=first.end();firstIt++){ //遍历每个非终结符
        set<TableItem> tmpTbSet=(*firstIt).second;  //first集合的元素
        for(tbItemSetIt=tmpTbSet.begin();tbItemSetIt!=tmpTbSet.end();tbItemSetIt++){    //遍历每个非终结符的first集合
            pTable[{(*firstIt).first,(*tbItemSetIt).t}]={(*tbItemSetIt).p,(*tbItemSetIt).p_tk};
        }
        if(TableItemHasNull(tmpTbSet)){ //若first含ep，加入follow
            set<TableItem> followTbSet;
            set<TableItem>::iterator followTbSetIt;  //用于遍历follow集合
            followTbSet=follow.find((*firstIt).first)->second;   //该vn的follow
            for(followTbSetIt=followTbSet.begin();followTbSetIt!=followTbSet.end();followTbSetIt++){

                pTable[{(*firstIt).first,(*followTbSetIt).t}]={(*followTbSetIt).p,(*followTbSetIt).p_tk};  //给table的项赋值

            }
        }
    }
}
```

```
1.  TreeNode * parse(){  //语法分析
2.      P_Token start,program;
3.      start.attr.t=PARSER_START;
4.      start.isVn=false;
5.      program.attr.vn=Program;
6.      program.isVn=true;
7.      s.push(start); //符号栈压入$
8.      s.push(program); //符号栈压入开始符号
9.
10.     t=new TreeNode();
11.     treeStack.push(t); //分析树的根节点
12.
13.     bool isGetTk=true;  //判断是否需要getToken
14.     while(s.size()>1){
15.         //fprintf(listing,"----------------------\n");
16.         P_Token s_top=s.top(); //top of stack
17.
18.         TreeNode *t_top=treeStack.top();  //保存栈顶元素
19.
20.         if(isGetTk){
21.             token = getToken();
22.         }
23.         if(s_top.isVn==false){  //match
24.             s.pop(); //符号栈弹栈
25.
26.             treeStack.pop(); //对应的树的节点栈弹栈
27.
28.             t_top->child = NULL;
29.
30.             if( s_top.attr.t==token){
31.                 isGetTk = true;
32.                 t_top->pro="MATCH: "+vt(s_top.attr.t);
```

```
            }
            else{ //错误处理
                isGetTk = false;
                //cout<<"doesn't match error  "<<endl;
                t_top->pro="----------ERROR: Expected -> "+vt(s_top.attr.t);
            }
        }
        else{
            isGetTk = false;
            s.pop();
            TableVal tbVal=pTable[{s_top.attr.vn,token}];  //找到当前对应的元素
            vector<P_Token> pToken=tbVal.p_tk;  //打断的产生式的右边
            if(pToken.size() == 0){
                //cout<<"error"<<endl;
            }
            else{
                //treeStack 弹栈、压入新指针为产生式左边元素的 child
                t_top->pro = tbVal.p_str.c_str();
                treeStack.pop();

                TreeNode *sib = new TreeNode();
                TreeNode *sib1,*tmp=sib;
                if(pToken[0].isVn==false && pToken[0].attr.t==NULL_TK){ //if vn -> ep

                }
                else{
                    for(int i=0;i<pToken.size();i++){
                        //当前产生式右边后面的元素为产生式右边第一个元素的 sibling
                        treeStack.push(sib);
                        sib1 = new TreeNode();
                        sib1->sibling = sib;
                        tmp = sib;
```

```
                    sib=sib1;
                    s.push(pToken[i]);  //建立兄弟节点，并且压栈
                }
                t_top->child = tmp;  //建立子节点
            }
        }
    }
    return t;
}
```

4.测试结果

（1）测试程序 1（正确的）：

```
int x[10];
int minloc (int a[], int low, int high){
   int i;
   int x;
   int k;
   k = low;
   x = a[low];
   i = low + 1;
   while(i < high){
      if(a[i] < x){
         x = a[i];
         k = i;
      }
      i = i + 1;
   }
   return k;
}
void main(void){
      output(minloc(x, 4, 9));
}
```

词法分析：

-手工实现：From Diagram to Lexical Analyser

```
1: reserved word: int
```

```
1: ID, name= x
1: [
1: NUM, val= 10
1: ]
1: ;
2: reserved word: int
2: ID, name= minloc
2: (
2: reserved word: int
2: ID, name= a
2: [
2: ]
2: ,
2: reserved word: int
2: ID, name= low
2: ,
2: reserved word: int
2: ID, name= high
2: )
2: {
3: reserved word: int
3: ID, name= i
3: ;
4: reserved word: int
4: ID, name= x
4: ;
5: reserved word: int
5: ID, name= k
5: ;
6: ID, name= k
6: =
6: ID, name= low
```

```
6: ;
7: ID, name= x
7: =
7: ID, name= a
7: [
7: ID, name= low
7: ]
7: ;
8: ID, name= i
8: =
8: ID, name= low
8: +
8: NUM, val= 1
8: ;
9: reserved word: while
9: (
9: ID, name= i
9: <
9: ID, name= high
9: )
9: {
10: reserved word: if
10: (
10: ID, name= a
10: [
10: ID, name= i
10: ]
10: <
10: ID, name= x
10: )
10: {
11: ID, name= x
```

```
        11: =
        11: ID, name= a
        11: [
        11: ID, name= i
        11: ]
        11: ;
        12: ID, name= k
        12: =
        12: ID, name= i
        12: ;
        13: }
        14: ID, name= i
        14: =
        14: ID, name= i
        14: +
        14: NUM, val= 1
        14: ;
        15: }
        16: reserved word: return
        16: ID, name= k
        16: ;
        17: }
        18: reserved word: void
        18: ID, name= main
        18: (
        18: reserved word: void
        18: )
        18: {
        19: ID, name= output
        19: (
        19: ID, name= minloc
        19: (
```

```
    19: ID, name= x
    19: ,
    19: NUM, val= 4
    19: ,
    19: NUM, val= 9
    19: )
    19: )
    19: ;
    20: }
    21: EOF
```

-手工实现：表驱动方法

```
    1: reserved word: int
    1: ID, name= x
    1: [
    1: NUM, val= 10
    1: ]
    1: ;
    2: reserved word: int
    2: ID, name= minloc
    2: (
    2: reserved word: int
    2: ID, name= a
    2: [
    2: ]
    2: ,
    2: reserved word: int
    2: ID, name= low
    2: ,
    2: reserved word: int
    2: ID, name= high
```

```
    2: )
    2: {
    3: reserved word: int
    3: ID, name= i
    3: ;
    4: reserved word: int
    4: ID, name= x
    4: ;
    5: reserved word: int
    5: ID, name= k
    5: ;
    6: ID, name= k
    6: =
    6: ID, name= low
    6: ;
    7: ID, name= x
    7: =
    7: ID, name= a
    7: [
    7: ID, name= low
    7: ]
    7: ;
    8: ID, name= i
    8: =
    8: ID, name= low
    8: +
    8: NUM, val= 1
    8: ;
    9: reserved word: while
    9: (
    9: ID, name= i
    9: <
```

```
        9: ID, name= high
        9: )
        9: {
        10: reserved word: if
        10: (
        10: ID, name= a
        10: [
        10: ID, name= i
        10: ]
        10: <
        10: ID, name= x
        10: )
        10: {
        11: ID, name= x
        11: =
        11: ID, name= a
        11: [
        11: ID, name= i
        11: ]
        11: ;
        12: ID, name= k
        12: =
        12: ID, name= i
        12: ;
        13: }
        14: ID, name= i
        14: =
        14: ID, name= i
        14: +
        14: NUM, val= 1
        14: ;
        15: }
```

```
        16: reserved word: return
        16: ID, name= k
        16: ;
        17: }
        18: reserved word: void
        18: ID, name= main
        18: (
        18: reserved word: void
        18: )
        18: {
        19: ID, name= output
        19: (
        19: ID, name= minloc
        19: (
        19: ID, name= x
        19: ,
        19: NUM, val= 4
        19: ,
        19: NUM, val= 9
        19: )
        19: )
        19: ;
        20: }
        21: EOF
```

-lex

```
int x[10];
关键字: int
标识符: x
[
整数:   10(10)
```

```
]
;
int minloc (int a[], int low, int high){

关键字: int
标识符: minloc
(
关键字: int
标识符: a
[
]
,
关键字: int
标识符: low
,
关键字: int
标识符: high
)
{
  int i;

关键字: int
标识符: i
;
  int x;

关键字: int
标识符: x
;
  int k;

关键字: int
```

标识符: k

;

  k = low;

标识符: k

运算符: =

标识符: low

;

  x = a[low];

标识符: x

运算符: =

标识符: a

[

标识符: low

]

;

  i = low + 1;

标识符: i

运算符: =

标识符: low

运算符: +

整数: 1(1)

;

  while(i < high){

关键字: while

(

标识符: i

less than: <

标识符: high

)

{

    if(a[i] < x){

关键字: if

(

标识符: a

[

标识符: i

]

less than: <

标识符: x

)

{

      x = a[i];

标识符: x

运算符: =

标识符: a

[

标识符: i

]

;

      k = i;

标识符: k

运算符: =

标识符: i

;

    }

}

```
    i = i + 1;

标识符: i
运算符: =
标识符: i
运算符: +
整数:   1(1)
;
  }

}
  return k;

关键字: return
标识符: k
;
}

}
void main(void){

关键字: void
标识符: main
(
关键字: void
)
{
output(minloc(x, 4, 9));

标识符: output
(
标识符: minloc
```

```
(
标识符: x
,
整数:   4(4)
,
整数:   9(9)
)
)
;
}


}
```

语法分析：

-递归向下

```
Syntax tree:
  Var declaration
    Type: int
    Id: x
    Subscript:
      Const: 10
  Function declaration
    Type: int
    Id: minloc
    Param
      Type: int
      Id: a
      Subscript:
    Param
      Type: int
      Id: low
```

```
  Param
    Type: int
    Id: high
  Compound
    Var declaration
      Type: int
      Id: i
    Var declaration
      Type: int
      Id: x
    Var declaration
      Type: int
      Id: k
    Assign
      Id: k
      Id: low
    Assign
      Id: x
      Id: a
      Subscript:
        Id: low
    Assign
      Id: i
      Op: +
        Id: low
        Const: 1
    While
      Op: <
        Id: i
        Id: high
      Compound
        If
```

```
            Op: <
              Id: a
              Subscript:
                Id: i
              Id: x
            Compound
              Assign
                Id: x
                Id: a
                Subscript:
                  Id: i
              Assign
                Id: k
                Id: i
          Assign
            Id: i
            Op: +
              Id: i
              Const: 1
      Return
        Id: k
Function declaration
  Type: void
  Id: main
  Param
    Type: void
  Compound
    Call:
      Id: output
      Call: minloc
        Id: x
        Const: 4
```

```
        Const: 9
```

-ll（1）

```
（由于篇幅太大，只选取部分展示）
 program -> declarationList
   declarationList -> declaration declarationList1
     declaration -> typeSpecifier ID declaration1
       typeSpecifier -> int
         MATCH: int
       MATCH: ID
       declaration1 -> varDeclaration
         varDeclaration -> type1
           type1 -> [ NUM ] ;
             MATCH: [
             MATCH: NUM
             MATCH: ]
             MATCH: ;
     declarationList1 -> declaration declarationList1
       declaration -> typeSpecifier ID declaration1
         typeSpecifier -> int
           MATCH: int
         MATCH: ID
         declaration1 -> funDeclaration
           funDeclaration -> ( params ) compoundStmt
             MATCH: (
             params -> paramList
               paramList -> param paramList1
                 param -> typeSpecifier ID param1
                   typeSpecifier -> int
                     MATCH: int
                   MATCH: ID
                   param1 -> [ ]
```

```
                    MATCH: [
                    MATCH: ]
                paramList1 -> , param paramList1
                  MATCH: ,
                  param -> typeSpecifier ID param1
                    typeSpecifier -> int
                      MATCH: int
                    MATCH: ID
                    param1 -> ep
                  paramList1 -> , param paramList1
                    MATCH: ,
                    param -> typeSpecifier ID param1
                      typeSpecifier -> int
                        MATCH: int
                      MATCH: ID
                      param1 -> ep
                    paramList1 -> ep
            MATCH: )
            compoundStmt -> { localDeclarations statementList }
……
                                          term1 -> ep
                                        additiveExpression1 -> ep
                                      simpleExpression1 -> ep
                                    argList1 -> ep
                                  MATCH: )
                              expression1 -> ep
                                MATCH: ;
                              statementList1 -> ep
                        MATCH: }
                declarationList1 -> EOF
              MATCH: EOF
```

（2）测试程序（错误的）

```
int x[10];
int minloc ( int a[], int low, int high )
{
   int i; int x; int k;
   k = low;
   x = a[low];
   i = low + 1;
   while (i < high)
   {
      if (a[i] < x)
      {
         x = a[i];
         k = i;
      }
      i = i + 1;
   }
   return k;
}
void sort ( int a[], int low, int high )
{
   int i; int k;
   i = low;
   while (i < high-1)
   {
       int t;
       k = minloc (a,i,high);
      t =a[k];
      a[k] = a[i];
      a[i] = t;
      i = i + 1;
   }
}
void main (void){
      int i;
      i = 0;
      while (i < 10){
            x[i] = input;
            i = i + 1;
            sort (x,0,10);
            i = 0;
            while (i < 10){
                  output(x[i]);
                  i = i + 1;
            }
```

词法分析：

-手工实现：From Diagram to Lexical Analyser

（篇幅有限，只展示部分）

```
    1: reserved word: int
    1: ID, name= x
    1: [
    1: NUM, val= 10
    1: ]
    1: ;
    2: reserved word: int
    2: ID, name= minloc
    2: (
    2: reserved word: int
    2: ID, name= a
    2: [
    2: ]
    2: ,
    2: reserved word: int
    2: ID, name= low
    2: ,
    2: reserved word: int
    2: ID, name= high
    2: )
    3: {
    4: reserved word: int
    4: ID, name= i
    4: ;
    4: reserved word: int
    4: ID, name= x
    4: ;
    4: reserved word: int
    4: ID, name= k
```

```
4: ;
5: ID, name= k
5: =
5: ID, name= low
5: ;
6: ID, name= x
6: =
6: ID, name= a
6: [
6: ID, name= low
6: ]
6: ;
7: ID, name= i
7: =
7: ID, name= low
7: +
7: NUM, val= 1
7: ;
8: reserved word: while
8: (
8: ID, name= i
8: <
8: ID, name= high
8: )
9: {
10: reserved word: if
10: (
10: ID, name= a
10: [
10: ID, name= i
10: ]
10: <
```

```
        10: ID, name= x
        10: )
        11: {
```

-手工实现：表驱动方法

```
（篇幅有限，只展示部分）
1: reserved word: int
        1: ID, name= x
        1: [
        1: NUM, val= 10
        1: ]
        1: ;
        2: reserved word: int
        2: ID, name= minloc
        2: (
        2: reserved word: int
        2: ID, name= a
        2: [
        2: ]
        2: ,
        2: reserved word: int
        2: ID, name= low
        2: ,
        2: reserved word: int
        2: ID, name= high
        2: )
        3: {
        4: reserved word: int
        4: ID, name= i
        4: ;
        4: reserved word: int
        4: ID, name= x
```

```
        4: ;
        4: reserved word: int
        4: ID, name= k
        4: ;
        5: ID, name= k
        5: =
        5: ID, name= low
        5: ;
        6: ID, name= x
        6: =
        6: ID, name= a
        6: [
        6: ID, name= low
        6: ]
        6: ;
        7: ID, name= i
        7: =
        7: ID, name= low
        7: +
        7: NUM, val= 1
        7: ;
        8: reserved word: while
        8: (
        8: ID, name= i
        8: <
        8: ID, name= high
        8: )
        9: {
        10: reserved word: if
        10: (
        10: ID, name= a
        10: [
```

```
        10: ID, name= i
        10: ]
        10: <
        10: ID, name= x
        10: )
```

-lex

```
（篇幅有限，只展示部分）
int x[10];
关键字: int
标识符: x
[
整数:   10(10)
]
;
int minloc ( int a[], int low, int high )

关键字: int
标识符: minloc
(
关键字: int
标识符: a
[
]
,
关键字: int
标识符: low
,
关键字: int
标识符: high
)
{
```

```
{
int i; int x; int k;

关键字: int
标识符: i
;
关键字: int
标识符: x
;
关键字: int
标识符: k
;
k = low;
```

语法分析：(提示错误信息)

-递归向下，提示了在代码 45 行缺少了一个}

```
（篇幅有限，只展示部分）
        …(前面同词法分析)
        43: ID, name= i
        43: =
        43: ID, name= i
        43: +
        43: NUM, val= 1
        43: ;
        44: }
        45: EOF

-------- Syntax error at line 45: expected '}'

Syntax tree:
```

```
  Var declaration
    Type: int
    Id: x
    Subscript:
      Const: 10
  Function declaration
    Type: int
    Id: minloc
    Param
      Type: int
      Id: a
      Subscript:
    Param
      Type: int
      Id: low
    Param
      Type: int
      Id: high
    Compound
……
            Compound
              Call:
                Id: output
                Id: x
                Subscript:
                  Id: i
              Assign
                Id: i
                Op: +
                  Id: i
                  Const: 1
```

-ll（1）,提示缺少}的错误

```
（篇幅有限，只展示部分）
program -> declarationList
    declarationList -> declaration declarationList1
      declaration -> typeSpecifier ID declaration1
        typeSpecifier -> int
          MATCH: int
        MATCH: ID
        declaration1 -> varDeclaration
          varDeclaration -> type1
            type1 -> [ NUM ] ;
              MATCH: [
              MATCH: NUM
              MATCH: ]
              MATCH: ;
      declarationList1 -> declaration declarationList1
……
                                                                    addop -> +
                                                                      MATCH: +
                                                                    term -> factor term1
                                                                      factor -> NUM
                                                                        MATCH: NUM
                                                                      term1 -> ep

additiveExpression1 -> ep
                                                                          simpleExpression1 -> ep
                                                                  MATCH: ;
                                                              statementList1 -> ep
                                                          MATCH: }
                                                  ----------ERROR: Expected -> }
                                          ----------ERROR: Expected -> }
                              declarationList1 -> EOF
```

MATCH: EOF

5. 总结

- 收获：

  本学期的实验中，我对编译原理理论课中的知识有了更加深刻的理解和认识，如词法分析和语法分析中的各种算法，DFA 中的状态转换在编程中如何体现，first 和 follow 集合的内涵及作用等；

  同时，在算法实现的过程中，遇到了很多困难，这也锻炼了我的编程能力；

  实现程序时会用到各种数据结构以及对数据结构的选择、调整，这也使我对数据结构的知识有了进一步的巩固；

  由于用 c++语言编写，也用到了以往不会用的，如 enum 类型，union 类型和 static 关键字，在指针的操作上也得到了练习，使我对 c++语言的掌握得到了提升。

  总之，本次实验使我的编程能力和对编译原理这门课的理解都得到了较大提升，收获颇丰。

- 不足：

  编写代码时，由于有的细节没有考虑清楚就开始编码，如数据结构的选择、文法改写、语法树的结构的确定等，导致反复修改程序，编程的效率不高；

  对于数据结构的选择，采用的数据结构较为复杂，编码时容易出错；

  程序中有的语句冗余，还有简化的余地；

  在程序的效率上也有改进的余地，如产生词法分析表、生成 first 和 follow 集合的过程等，有一定的时间和空间上的浪费，还可以改进。