《编译原理》课程设计
实验报告书

班　　级：_____

学　　号：_____

姓　　名：_____

指导老师：_____

课设地点：_____

课设时间：_____

年　月　日

目　录

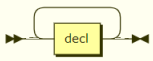
一、C 语言的语法图描述

铁路图，又叫语法图(syntax diagrams)，是一种表示形式语法的方式，是巴科斯范式和扩展巴科斯范式的图形化表示

- 规则：
  - 从左边界开始，沿着轨道到右边界。
  - 沿途，你在圆框中遇到的是字面量，在方块中遇到的是规则或描述。
  - 任何沿着轨道能走通的序列都是合法的。
  - 任何不能沿着轨道走通的序列都是不合法的。
  - 末端只有一个竖条的铁路图，表示允许在任意一对符号中间插入空白。而在末端有两个竖条的铁路图则不允许。

这里，我使用了一个在线铁路图生成网站 https://bottlecaps.de/rr/ui，下面是语法规则对应的语法图。

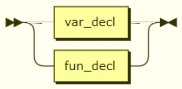
program:




```
program  ::= decl+
```

no references

decl:

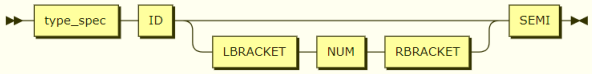



```
decl     ::= var_decl
           | fun_decl
```

referenced by:

- program

var_decl:

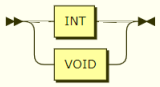



```
var_decl ::= type_spec ID ( LBRACKET NUM RBRACKET )? SEMI
```

referenced by:

- compound_stmt
- decl

type_spec:

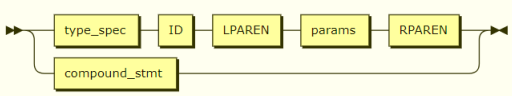



```
type_spec
          ::= INT
            | VOID
```

referenced by:

- fun_decl
- param
- var_decl

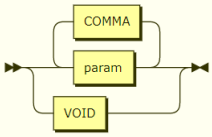
fun_decl:




```
fun_decl ::= type_spec ID LPAREN params RPAREN
           | compound_stmt
```

referenced by:

- decl

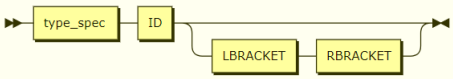



```
params   ::= param ( COMMA param )*
           | VOID
```

referenced by:

- fun_decl

param:




```
param    ::= type_spec ID ( LBRACKET RBRACKET )?
```

referenced by:

- params

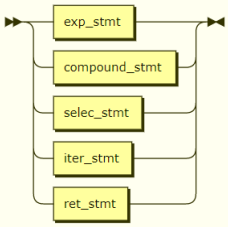
compound_stmt:


```
compound_stmt
         ::= LBRACE var_decl* stmt* RBRACE
```

referenced by:

- fun_decl
- stmt

stmt:

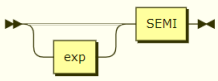



```
stmt     ::= exp_stmt
           | compound_stmt
           | selec_stmt
           | iter_stmt
           | ret_stmt
```

referenced by:

- compound_stmt
- iter_stmt
- selec_stmt

exp_stmt:

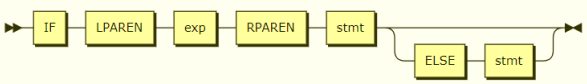



```
exp_stmt ::= exp? SEMI
```

referenced by:

- stmt

selec_stmt:

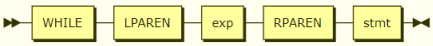



```
selec_stmt
         ::= IF LPAREN exp RPAREN stmt ( ELSE stmt )?
```

referenced by:

- stmt

iter_stmt:

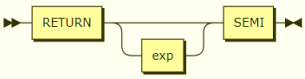



```
iter_stmt
        ::= WHILE LPAREN exp RPAREN stmt
```

referenced by:

- stmt

ret_stmt:

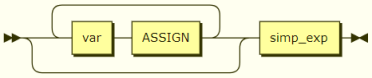



```
ret_stmt ::= RETURN exp? SEMI
```

referenced by:

- stmt

exp:

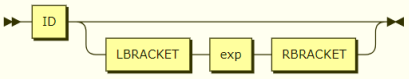



```
exp     ::= ( var ASSIGN )* simp_exp
```

referenced by:

- args
- exp_stmt
- factor
- iter_stmt
- ret_stmt
- selec_stmt
- var

var:

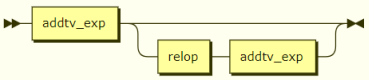



```
var     ::= ID ( LBRACKET exp RBRACKET )?
```

referenced by:

- exp
- factor

simp_exp:

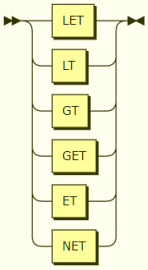



```
simp_exp ::= addtv_exp ( relop addtv_exp )?
```

referenced by:

- exp

relop:

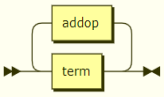



```
relop    ::= LET
           | LT
           | GT
           | GET
           | ET
           | NET
```

referenced by:

- simp_exp

addtv_exp:

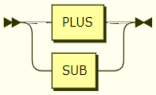



```
addtv_exp
        ::= term ( addop term )*
```

referenced by:

- simp_exp

addop:

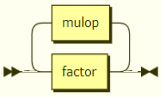



```
addop    ::= PLUS
           | SUB
```

referenced by:

- addtv_exp

term:

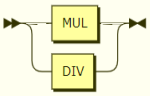



```
term     ::= factor ( mulop factor )*
```

referenced by:

- addtv_exp

mulop:

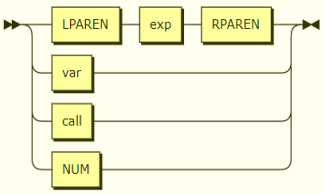



```
mulop    ::= MUL
           | DIV
```

referenced by:

- term

factor:

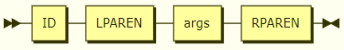



```
factor   ::= LPAREN exp RPAREN
           | var
           | call
           | NUM
```

referenced by:

- term

call:

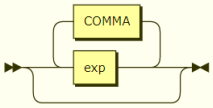



```
call     ::= ID LPAREN args RPAREN
```

referenced by:

- factor

args:




```
args     ::= ( exp ( COMMA exp )* )?
```

referenced by:

- call

综上就是 C-minus 的语法规则对应的语法图。

由于其语法规则比较简单，其与 C 语言有一些需要注意的区别：

1. 函数必须传参，若没有，则填入 void
2. 所有的声明语句必须放在最前面
3. 变量在声明时，不能对其进行赋值

二、系统设计

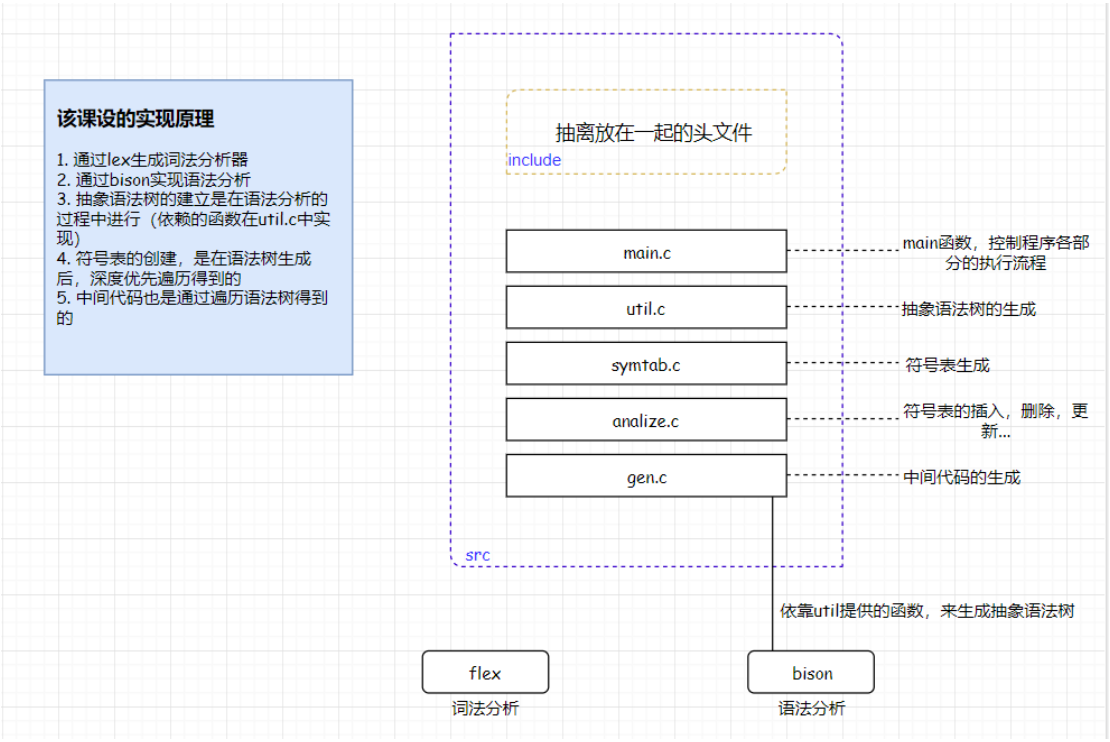
2.1 系统的总体结构

2.1.1 文件结构

```
├── Makefile                #
├── README.md               # 项目介绍及使用
├── lex                     # 生成扫描器
|   └── cm.l
├── yyac                    # 语法解析器生成器
|   └── cm.y
├── test                    # 存放了可执行文件以及测试用例
|   └── ...
└── src
    ├── main.c
    ├── util.c
    ├── symtab.c
    ├── analyze.c
    ├── obj                 # 存放目标文件
    |   └── ...
    └── include
        ├── globals.h
        └── ...             # 一系列头文件
```

2.1.2 实现思路

这里是各主要主要功能的逻辑关系，后面会进行重点介绍



2.2 主要功能模块设计

2.2.1 词法分析器

Flex 是一个快速词法分析生成器，它可以将用户用正则表达式写的分词匹配模式构造成一个有限状态自动机。Flex 使用很简单，只需要对待识别的 token 进行匹配规则的编写，他就可以自动帮忙生成词法分析器。

token 匹配思路：

- 关键字，运算符，结符一符一种
- 标识符： ID {letter}{letter}*
- 常熟： NUM {digit}{digit}*
  - 其中：letter [a-zA-Z] digit 0|1|2|3|4|5|6|7|8|9

2.2.2 语法分析器

依照老师给出的 Cminus 语法，编写语法规则。Bison 的编写依赖抽象语法树的相关函数

- 设置词法分析中使用到的 token
- 规定运算符优先级
- 规定非终结符为 treeNode 类型

2.2.3 抽象语法树生成相关

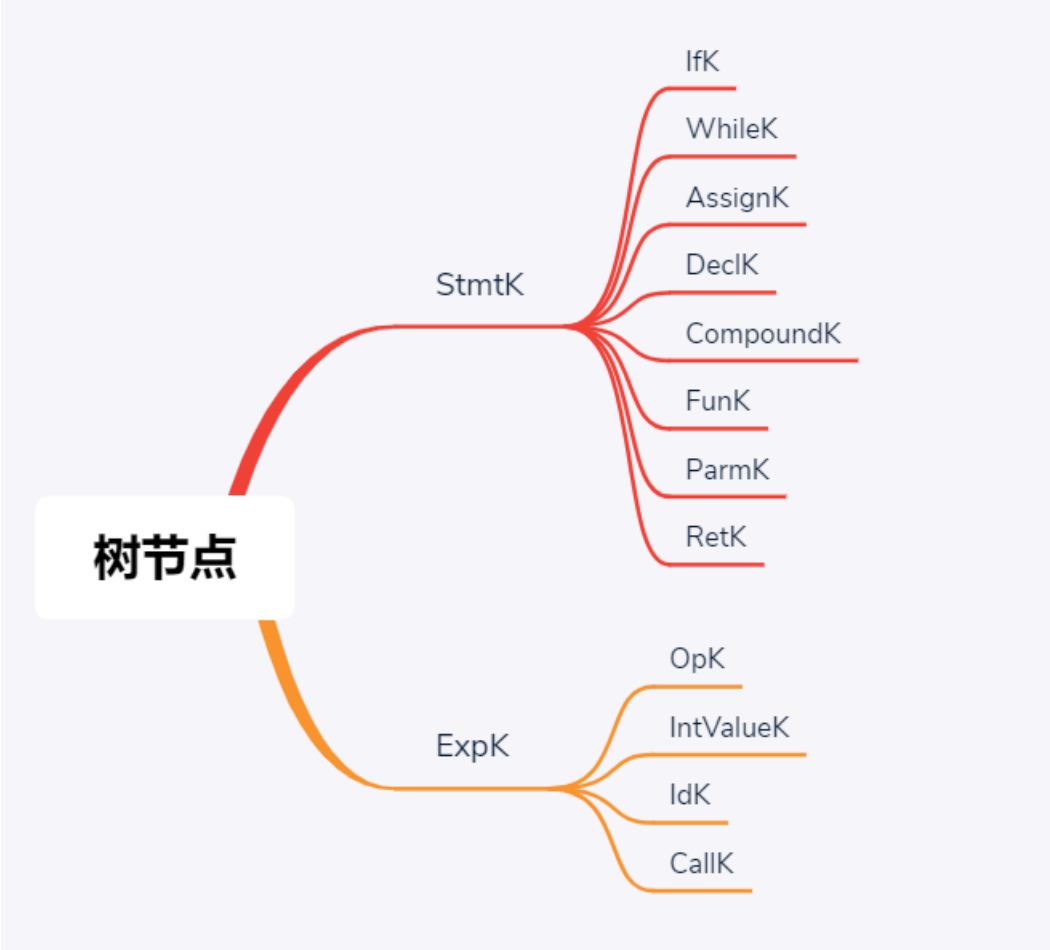
涉及文件：global.h util.c

treeNode 结构体：

```
/* 语法树节点的定义，包括子树的数目，兄弟节点，所在行数，节点类型以及相应的属性 */
typedef struct treeNode
    {   struct treeNode * child[MAXCHILDREN];
        struct treeNode * sibling;
        int lineno;
        NodeKind nodekind;
        /* 联合 kind：语句的种类 或 表达式的种类 */
        union
        {
            StmtKind stmt;
            ExpKind exp;
        } kind;

        /* 联合 attr：操作符 或 变量名 */
        union
        {   int op;
            char * name;
        } attr;


        /* 联合 value：整型或实型 */
        union
        {   int int_val;
        } value;

```

```
        /* treeNode 作为整型与浮点型的数组定义节点，
        使用 treeNode 中 attr 里的 int_val 作为数组的长度。
        */
        union
        {
            int * intArray;
            void * voidArray;
        } array;
        int arrayLength;
        ExpType type;
    } TreeNode;
```

关于**树节点的类型**，我们根据语法规则进行了分级的划分，下面是划分规则：



2.2.4 符号表创建

涉及文件：analyze.h　analyze.c　symtab.h　systab.c

analyze.h 文件实现了遍历抽象语法树，建立符号表的函数

```
#ifndef _ANALYZE_H_
#define _ANALYZE_H_

/*  buildSymtab()先序遍历语法树，构造符号表。
```

```
 */
void buildSymTab(TreeNode *);
/*  checkNode()为类型检查。
    返回值为 type.
    */
int checkNode(TreeNode *);
int execTree( TreeNode * tree );

#endif
```

systab.h 文件实现了符号表的插入，查找，类型检验，更新，输出符号表的函数，

```
/*  searchSymTab()用于查找和插入符号表，并协助类型检查。
参数说明：
    name 是当前的变量名，
    varLineno 是当前行号，用于错误提示，
    type 为变量声明时需要填入符号表的符号类型。
    arrayLength 为数组长度，为－123 代表非数组，其他的小于 0 的值报错。
返回值：
    正确返回时为 arrayLength，遇错为－1；
    */
int insertSymTab( char * name,  int varLineno, int type, int arrayLength );

/*  查找符号表，若找不到则返回－1，否则返回变量的 arrayLength，
    arrayLength 为 0 表示非数组，arrayLength 非 0 表示下标的值。
    */
int lookupSymTab ( char * name, int int_val, double real_val, int varLineno
);

/*  参数 type 为 0 时，直接返回变量在符号表中的类型；
    type 不为 0 时，检查变量名的类型与传入的 type 是否一致，一致返回 0，不一致返回
1。
    */
int checkType(char * name, int type, int varLineno);

/*  printSymTab()打印符号表，验证符号表是否正确构造。
    */
void printSymTab(FILE * listing);

/*  以下为解释执行时所用的函数，分别为：
    得到变量的值，更新变量的值，
    得到数组元素的值，
    更新数组元素的值。
```

```
    */
int getValue(char * name);

void updateValue(char * name, int int_val, double real_val);

int getArray(char * name, int index);

void updateArray(char * name, int index, int int_val, double real_val);
```

2.2.5 中间代码生成

涉及文件： gen.h gen.c

思想：

采用了深度优先遍历，采用了一次遍历的方法产生并回填了中间代码

定义了所有可能出现的情况，在每种情况内，先访问其子节点，然后回调当前空链

主要函数如下：

```
#ifndef _GEN_H_
#define _GEN_H_

typedef struct quater{
    char op[20];
    char arg1[20];
    char arg2[20];
    int flag;          //为 1 则最后填 quad，否则为字符串
    char resultc[20];
    int resulti;
    int order;         //记录此条四元式标号
}Quater;

void getRoot(TreeNode *tree);
Quater visitstmt(TreeNode *tree);
char* newtemp();
Quater unconditionJump();
Quater visitOpK(TreeNode *tree);
Quater visitExpK(TreeNode *tree);
Quater visitAssignK(TreeNode *tree);
Quater visitCallK(TreeNode *tree);
Quater visitRetK(TreeNode *tree);
Quater visitIfK(TreeNode *tree);
Quater visitWhilK(TreeNode *tree);
Quater visitFuncK(TreeNode *tree,TreeNode *argsTree);
void BFSTree( TreeNode * tree );
void printQuad(Quater tempq);
void printQua();
```

```
29.
30.
31. #endif
```

2.3 系统运行流程

运行流程：

- 扫描测试文件
- 语法分析
- 语义分析
- 遍历抽象语法树，生成中间代码
- 遍历抽象语法树，生成符号表

代码展示：

```
1.  main( int argc, char * argv[] )
2.  {
3.      TreeNode * syntaxTree;
4.      char pgm[120]; /* 源文件名 */
5.      if (argc < 2)
6.      {   fprintf(stderr,"usage: %s <filename>\n",argv[0]);   //错误信息立即输
    出
7.          exit(1);
8.      }
9.      listing = stdout;
10.     if (argc > 2)
11.     {
12.         if (strchr (argv[2],".") == NULL)          //添加-s 打印语法分析树
13.             strcat(argv[2],".txt");
14.         listing = fopen(argv[2], "w");
15.     }
16.     strcpy(pgm,argv[1]) ;             //第二个参数为源文件名
17.     if (strchr (pgm, 'cm') == NULL)       //检测是否带文件后缀.cm
18.         strcat(pgm,".cm");            //手动添加
19.     source = fopen(pgm,"r");
20.     if (source==NULL)
21.     {   fprintf(stderr,"File %s not found\n",pgm);     //打开文件错误
22.         exit(1);
23.     }
24.     fprintf(listing,"\nCM Interpretation: %s\n",pgm);
25.
26.     if (Parse)
27.     {
28.         syntaxTree = parse();   // 得到抽象语法树
29.         if (printSyntaxTree)    // 打印语法树
30.         {   fprintf(listing,"Syntax tree:\n");
31.             printTree(syntaxTree);
```

```
32.         }
33.         if(printQUAD)           // 生成中间代码
34.         {
35.
36.             getRoot(syntaxTree);
37.
38.             BFSTree(syntaxTree);
39.
40.             fprintf(listing,"Quaternary postures:\n");
41.             printQua();
42.         }
43.         if (! Error)
44.         {   if (Analyze)   // 生成符号表
45.             {
46.                 buildSymTab(syntaxTree);
47.                 if(! Error)
48.                 {
49.                     checkNode(syntaxTree);
50.                     if(! Error)
51.                         fprintf(listing,"Type check completed.\n");
52.                     else
53.                         fprintf(listing,"\nTypes of variables have got some
    errors.。\n");
54.                 }
55.                 else
56.                     fprintf(listing,"\nResult:Some errors occurred in symbol
     table construction.\n");
57.             }
58.         }
59.     }
60.     fclose(source);
61.     return 0;
62. }
```

三、系统实现

3.1 词法分析

在 2.2.1 部分已经介绍了 flex 的匹配规则的书写
语法规则扫描的输入串，是通过 iniLexer 函数得到的

```
1.  /* 用于语法分析时初始化词法分析接口 */
2.  void iniLexer(void)
3.  {
4.      static int firstTime = TRUE;
5.      lineno = 0;
6.      if (firstTime)
7.      {   firstTime = FALSE;
8.          lineno++;
9.          yyin = source;   /* 获取输入串 */
10.         yyout = listing;
11.     }
12. }
```

3.2 语法分析

yyerror 主要负责错误的处理（没识别的 token，语法错误时错误的处理）

```
1.  int yyerror(char * message)
2.  {
3.      // 输出错误所在的行号及相关信息
4.      fprintf(listing,"Syntax error at line %d: %s\n",lineno,message);
5.      fprintf(listing,"Current token: %s",tokenString);
6.      printToken(yychar);
7.      Error = TRUE;
8.      return 0;
9.  }
```

Parse 函数调用 yyparse，这个函数不仅维持了一个状态栈，还维持了一个属性栈，因此，语义规则中可以使用$$来获得对应的属性值。 并在最终返回生成的抽象语法树的根节点。

```
1.  TreeNode * parse(void)
2.  {   iniLexer();
3.      yyparse();
4.      return savedTree;
5.  }
```

3.3 抽象语法树生成

树节点的建立：

由于树节点类型分为两大类，所以我们定义了两个函数 newstmtNode(StmtKind)和 newExpNode(ExpKind)，通过传入的参数，设置节点的类型。

```
63  /* newStmtNode 创建一个新的语句节点 */
64  TreeNode * newStmtNode(StmtKind kind)
65 >{ TreeNode * t = (TreeNode *) malloc(sizeof(TreeNode));…
79  }
80
81  /* newExpNode 创建一个新的表达式节点 */
82  TreeNode * newExpNode(ExpKind kind)
83 >{ TreeNode * t = (TreeNode *) malloc(sizeof(TreeNode));…
96  }
97
```

抽象语法树的生成：

抽象语法树在语法分析的过程中生成，当遇到可以作为语法树节点的产生式时，就调用 newXxxNode 生成指定类型的节点。示例如下：（注释中进一步进行了解释）

```
66 program   : decl_list
67             { savedTree = $1; /* 将产生的第一个节点赋给全局指定的更节点 */ }
68           ;
```

```
204 selec_stmt  : IF LPAREN exp RPAREN stmt    /*if(exp) +各种类型的动作*/
205             {
206             $$ = newStmtNode(IfK);  /* 指定节点的类型 */
207             $$->child[0] = $3;      /* 对其子节点就行赋值 */
208             $$->child[1] = $5;
209             $$->child[2] = NULL;
210             $$->lineno = lineno;
211             }
212         | IF LPAREN exp RPAREN stmt ELSE stmt    /*if(exp)+动作+else+stmt*/
213             {
214             $$ = newStmtNode(IfK);
215             $$->child[0] = $3;
216             $$->child[1] = $5;
217             $$->child[2] = $7;
218             $$->lineno = lineno;
219             }
220              ;
```

抽象语法树的输出：

功能函数：printTree(TreeNode *tree)

通过中序遍历每一个树节点，通过 switch/if 来判断细化节点类型，并打印节点相应信息：

辅助函数：

```
/* 缩进数 */
static int preIndentCount = -12;
static int indentCount = -12;
/* 改变需要输出的缩进的数目—宏定义 */
#define INDENT   {preIndentCount=indentCount; indentCount+=12;}
#define UNINDENT {preIndentCount=indentCount; indentCount-=12;}

/* printSpaces 通过输出空格实现缩进 */
static void printSpaces(void)
{ int i;
  if (preIndentCount != indentCount)
  {
    for (i=0;i<indentCount;i++)
      fprintf(listing," ");
    fprintf(listing, "|\n");
  }
  for (i=0;i<indentCount;i++)
    fprintf(listing," ");
  fprintf(listing,"|---------- ");
}
```

printTree(TreeNode *tree)

```
145  /* printTree 输出语法树 */
146  void printTree( TreeNode * tree )
147  { int i;      // 子节点计数器
148    INDENT;
149    while (tree != NULL)
150    { printSpaces();
151      if (tree->nodekind==StmtK)
152      {
153        switch (tree->kind.stmt)
154        { case IfK:…
234        }
235      }
236      else if (tree->nodekind==ExpK)
237      { switch (tree->kind.exp)
238        { case OpK:
239            fprintf(listing,"Op: ");
240            switch(tree->attr.op)
241            { case PLUS: fprintf(listing, "+\n"); break;…
252            }
253            break;
254          case IntValueK:…
263          case CallK:…
269        }
270      }
271      else…
273      for (i=0;i<MAXCHILDREN;i++)
274        printTree(tree->child[i]);
275      tree = tree->sibling;
276    }
277    UNINDENT;
278  } //  printTree end
```

INDENT和UNINDENT是两处宏定义，负责更新该节点前需打印的空格个数

辅助函数：打印空格，便于打印出有层次的树结构

对节点的处理，判断出节点类型，打印节点信息

中序遍历语法树

3.4 中间代码生成

中间代码生成思路：

- 深度优先遍历语法树
- 采用一次遍历的方式生成中间代码
- 细化到每一种情况，先生成其子节点，然后进行回调

四元式结构体如下：

```
10  typedef struct quater{
11    char op[20];
12    char arg1[20];
13    char arg2[20];
14    int flag;          //为1 则最后填quad，否则为字符串
15    char resultc[20];
16    int resulti;
17    int order;         //记录此条四元式标号
18  }Quater;
```

对回填的分析（以条件语句为例）：

```
1.  Quater visitIfK(TreeNode *tree)
2.  {
3.      Quater expqua,errorqua,errorqua2;
4.      Quater stmtqua,stmtqua2;
5.      TreeNode *tempTree;
6.
7.      if(tree->child[2]==NULL)      //if (exp) stmt
8.      {
```

```
9.          tempTree=tree->child[0];
10.         expqua=visitOpK(tempTree);       //exp 的最后一个项便于回填
11.         //生成一个错误出口
12.         errorqua=unconditionJump();      //错误出口  ，此项需回填
13.         qua[expqua.order].flag=1;            //表示是 int 型的四元式最后一项
14.         qua[expqua.order].resulti=quad;
15.
16.         tempTree=tree->child[1];
17.         stmtqua=visitstmt(tempTree);    //正确出口
18.
19.         qua[errorqua.order].resulti=quad;     //回填错误出口 即下一条语句
20.     }
21.     else                          //if(exp)stmt1 else stmt2
22.     {
23.         tempTree=tree->child[0];
24.         expqua=visitOpK(tempTree);       //exp 的最后一个项便于回填
25.         errorqua=unconditionJump();      //错误出口  ，此项需回填
26.
27.         qua[expqua.order].flag=1;            //表示是 int 型的四元式最后一项
28.         qua[expqua.order].resulti=quad;      //跳过错误出口，回填 exp 真出口
29.
30.         tempTree=tree->child[1];
31.         stmtqua=visitstmt(tempTree);    //正确出口 执行完了需跳过 stmt2
32.         errorqua2=unconditionJump();
33.
34.         qua[errorqua.order].resulti=quad;  //回填错误出口即 stmt2
35.
36.         tempTree=tree->child[2];
37.         stmtqua2=visitstmt(tempTree);
38.
39.         qua[errorqua2.order].flag=1;
40.         qua[errorqua2.order].resulti=quad;
41.
42.
43.     }
44. }
```

3.5 符号表生成

符号表的设计：

```
7  /* SIZE 为符号表大小，是一个比较合适的素数 */
8  #define SIZE 211    设置表得大小
9
10 /* SHIFT 用于移位运算 */
11 #define SHIFT 4
12
13 /* 哈希函数*/
14 static int hash ( char * key )
15 { int temp = 0;
16   int i = 0;
17   while (key[i] != '\0')
18   { temp = ((temp << SHIFT) + key[i]) % SIZE;
19     ++i;
20   }
21   return temp;
22 }
23
24 /*  符号表表项，每一个表项包含：
25   变量名，类型，声明时的行号，下一个表项的地址
26  */
27 typedef struct SymbolRec
28 { char * name;
29   ExpType type;
30   //LineList lines;
31   int varLineno;
32   union { int int_val;
33       double real_val; } value;
34   union { int * intArray;
35       double * realArray; } array;
36   int arrayLength;
37   struct SymbolRec * next;
38 } * Symbol;
39
40 /* the hash table */
41 static Symbol hashTable[SIZE];
```

对符号名进行操作，返回对应得hash值

符号表实现采用了杂凑技术：

* 符号表是一个数组，大小为211

* 符号表的每一个表项都是一个链表，负责把对应同一hash值得符号链在一起

原理介绍

符号表相关函数：

符号表的查找，出入，更新操作都是链表的操作，这里不做详细的介绍。

```
1.  int insertSymTab( char * name, int varLineno, int type, int arrayLength )
2.  {
3.      /* 生成符号名对应的 hash 值，定位其在符号表中的位置 */
4.      int h = hash(name);
5.      Symbol s =  hashTable[h];
6.
7.      /* 遍历符号表，查找是否有同名项 */
8.      while ((NULL != s)&&(0 != strcmp(name,s->name)))
9.          s = s->next;
10.
11.     if(NULL == s)       /* 该符号不在符号表中，将其加入符号表，插入表项 */
12.     {   s = (Symbol) malloc(sizeof(struct SymbolRec));
13.         s->name = name;
14.         s->varLineno = varLineno;
15.         s->type = (ExpType)type;
```

```
16.         if (0 >= arrayLength)
17.         {   fprintf(listing,
18.                 "Error:line %d:The length of array should be more than zero.\
    n", varLineno);
19.             Error = TRUE;
20.         }
21.         s->arrayLength = arrayLength;
22.
23.             int tempArray[arrayLength];
24.             s->array.intArray = tempArray;
25.
26.         s->next = hashTable[h];
27.         hashTable[h] = s;
28.         return arrayLength;
29.     }
30.     else    /* 若该符号在符号表中，报错 */
31.     {   fprintf(listing,
32.             "Error:line %d:Variable %s with the same name has been declared a
    t line %d.\n",
33.                 varLineno, name, s->varLineno);
34.         Error = TRUE;
35.     }
36.     return -1;  //返回值为－1 表示插入过程中出错。
37. }
```

符号表的建立：

```
76 /*  buildSymtab()构造符号表
77    使用先序遍历法
78  */
79 void buildSymTab(TreeNode * syntaxTree)
80 {
81   traverse(syntaxTree,insertNode,nullProc);
82   fprintf(listing,"\nSymbol table:\n");
83   printSymTab(listing);
84 }
```

buildSymtab 函数调用 traverse()函数对语法树进行先序遍历，**traverse()函数如下：**

```
15 static void traverse( TreeNode * tree,
16                        void (* preProc) (TreeNode *),
17                        void (* postProc) (TreeNode *) )
18 { if (tree != NULL)
19   { preProc(tree);
20     { int i;
21       for (i=0; i < MAXCHILDREN; i++)
22         traverse(tree->child[i],preProc,postProc);
23     }
24     postProc(tree);
25     traverse(tree->sibling,preProc,postProc);
26   }
27 }
```

由上述分析可知，上述遍历对每一个树节点，调用了 **insertSymtab 函数**

```
1.  int insertSymTab( char * name, int varLineno, int type, int arrayLength )
2.  {
3.      /* 生成符号名对应的 hash 值，定位其在符号表中的位置 */
4.      int h = hash(name);
5.      Symbol s =  hashTable[h];
6.
7.      /* 遍历符号表，查找是否有同名项 */
8.      while ((NULL != s)&&(0 != strcmp(name,s->name)))
9.          s = s->next;
10.
11.     if(NULL == s)       /* 该符号不在符号表中，将其加入符号表，插入表项 */
12.     {   s = (Symbol) malloc(sizeof(struct SymbolRec));
13.         s->name = name;
14.         s->varLineno = varLineno;
15.         s->type = (ExpType)type;
16.         if (0 >= arrayLength)
17.         {   fprintf(listing,
18.                 "Error:line %d:The length of array should be more than zero.
    \n", varLineno);
19.             Error = TRUE;
20.         }
21.         s->arrayLength = arrayLength;
22.
23.             int tempArray[arrayLength];
24.             s->array.intArray = tempArray;
25.
26.         s->next = hashTable[h];
27.         hashTable[h] = s;
28.         return arrayLength;
29.     }
30.     else    /* 若该符号在符号表中，报错 */
31.     {   fprintf(listing,
32.             "Error:line %d:Variable %s with the same name has been declared
    at line %d.\n",
33.                 varLineno, name, s->varLineno);
34.         Error = TRUE;
35.     }
36.     return -1;  //返回值为－1 表示插入过程中出错。
37. }
```

四、 课程设计心得

编译原理的课程设计，刚接触时不知道下手的地方，但熟悉了之后就发现词法分析、语法分析部分都不需要我们自己实现，需要我们去写的部分还是关于链表，树的知识。做课设的过程中进一步意识到数据结构的重要性。

1. 实践是认识的目的，只有自己动手去做了，才会发现用到的东西并不能。
2. 查资料的能力很重要，百度搜索到的关于 flex 和 bison 的知识，都是片面，混乱的。但 Google 搜索就可以找到一些系统性的教程，它不仅注重理论解释，而且还注重由浅入深的讲解 flex 和 bison 的使用。
3. 不能固化自己的思维，刚写好词法分析和语法分析时，自己进行样例的测试，发现有些地方莫名就报错停止。后面发现，是由于我们的语法规则是 Cminus 的，与 C 语言还是有较大区别，所以用 C 语言习惯写出的测试程序就会出错。这也反应出自己不求甚解的一面，有了正确语法规则，自己仅限于将其填入.y，却懒于去推导一遍。

五、 建议和意见

在做课程设计的过程中，查到了许多别人的教程，也参考了其他人的优秀代码，如一些数据类型的结构体等。

这里有一篇自己觉得很好的 flex 和 bison 入门的教程 https://pandolia.net/tinyc/ch1_overview.html，我觉得老师可以将其推荐给之后的学弟学妹。

这里面除了解释 flex 和 bison 的使用，还有一些语法分析的概念知识，我觉得如果在授课过程中穿插着这些练习，大家学习效果能得到一定程度的提高。

六、 参考资料

- 自己动手写编译器 https://pandolia.net/tinyc/ch1_overview.html
- 老师提供的示例代码