中国地质大学（北京）

《编译原理设计报告》

实验报告

学　　院：信息工程学院

专　　业：计算机科学与技术

班　　级：10041811

学　　号：1005183121

姓　　名：周子杰

联系方式：18006163783

邮　　箱：Dodo.ZhouZJ@outlook.com

指导老师：耿明芹

日　　期：2021 年 6 月 2 日

目录

编译程序概述……3
功能……3
结构……3
词法分析器……4
功能……4
实现……4
全局变量……4
单词符号……4
单词符号表……4
功能函数……5
词法分析器……8
测试…… 10
语法分析器…… 11
功能…… 11
实现…… 12
全局变量…… 12
功能函数…… 12
产生式…… 12
测试…… 18
语义分析器和中间代码生成器…… 21
功能…… 21
实现…… 21
全局变量…… 21
文法符号…… 22
功能函数…… 22
产生式…… 25
测试…… 30
总结…… 32
出现的问题…… 32
经验与展望…… 32
附录…… 32
完整代码…… 32

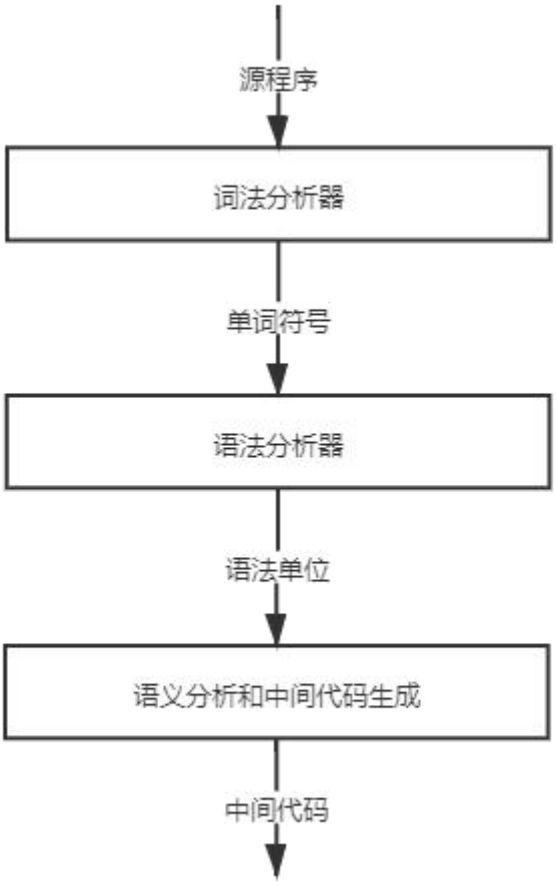
编译程序概述

功能

经过了四周的课程设计，我们完成了一个具有词法分析、语法分析、语义分析和中间代码生成等的编译程序。对于词法分析，其功能是读入输入，识别单词符号和值；对于语法分析，其功能是读入输入，在词法分析的基础上分析其句子组成；而对于语义分析和中间的代码生成，其功能是读入输入，在语法分析的基础上产生中间代码。

完成内容：包括必做和选做的全部内容。

结构



对于完整的编译器，我们实现了其中的三个部分：词法分析，语法分析，语义分析和中间代码生成。其大致结构如上图所示。

词法分析器

功能

通过对输入的源程序进行词法分析，我们能够不断读取其中的字符，并根据单词符号表来识别单词符号，并返回单词种别和属性值，返回值可留给后续的语法分析使用。

实现

全局变量

```
char ch = ' ';    //存放最新读进的源程序字符
string strToken; //存放构成单词符号的字符串
string totalStr; //完整的输入串
int tmpStr = 0;  //当前输入串扫描到的位置
string tmpID;    //暂存标识符，供之后常量返回
```

单词符号

```
//单词符号
struct TOKEN
{
    int kind = 0;           //单词种别-整数编码
    string attribute = "-"; //单词属性
};
```

对于单词符号，我们用一个结构体来表示它，在结构体中拥有两个变量，一个表示单词种别，一个表示单词属性，在词法分析后，返回的也是该单词符号结构体。在后续的内容中，我将用 TOKEN 来代表该结构体。

单词符号表

```
//单词符号表
int rsNum = 0;    //保留字个数
int opNum = 0;    //运算符个数
int boNum = 0;    //界符个数
int idNum = 0;    //标识符个数
int coNum = 0;    //常数个数
TOKEN rsWord[30]; //保留字
```

```
TOKEN opWord[30]; //运算符
TOKEN boWord[30]; //界符
TOKEN idWord[30]; //标识符
TOKEN coWord[30]; //常数
void initword()
{
    //保留字
    rsWord[rsNum].attribute = "int";
    rsWord[rsNum].kind = 5;
    rsNum++;
    rsWord[rsNum].attribute = "else";
    rsWord[rsNum].kind = 15;
    rsNum++;
    rsWord[rsNum].attribute = "if";
    rsWord[rsNum].kind = 17;
    rsNum++;
    rsWord[rsNum].attribute = "while";
    rsWord[rsNum].kind = 20;
    rsNum++;
}
```

我将单词符号表分为了保留字、界符、运算符、标识符和常数五个大类（也可以不分），用 TOKEN 数组来进行保存。这五个大类的实现是类似的，为了不显得非常冗长，这里我们**以保留字为例**展示代码。

功能函数

```
void GetChar()
{
    if (tmpStr < totalStr.length())
        ch = totalStr[tmpStr++];
    else
        ch = '#';
}
```

获取字符串的下一个字符并存入 ch，如果没有则为#。

```
void GetBC()
{
    while (ch == ' ')
        GetChar();
}
```

检查输入是否为空白，是则调用 getChar 直到非空白。

```
int Reserve()
{
    for (int i = 0; i < rsNum; ++i)
        if (strToken == rsWord[i].attribute)
            return rsWord[i].kind;
    return 0;
}
```

对 strToken 查找保留字表，若是保留字则返回编码，否则返回 0 值

```
void Retract()
{
    tmpStr--;
    ch = ' ';
}
```

将搜索指示器回调一个位置，将 ch 置为空白符。

```
bool IsLetter(char ch)
{
    if ((ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z'))
        return true;
    else
        return false;
}
```

判断是否为字母

```
bool IsDigit(char ch)
{
    if (ch >= '0' && ch <= '9')
        return true;
    else
        return false;
}
```

判断是否为数字

```
int IsOp(char ch)
{
    //先检查长度为 2 的
    for (int i = 0; i < opNum; ++i)
        if (opWord[i].attribute.length() == 2 && ch == opWord[i].attribute[0]
 && totalStr[tmpStr] == opWord[i].attribute[1])
        {
            return i;
        }
    //再检查长度为 1 的
    for (int i = 0; i < opNum; ++i)
        if (opWord[i].attribute.length() == 1 && ch == opWord[i].attribute[0]
)
            return i;
    return -1;
}
```

判断是否为运算符,运算符长度为 1~2 位,返回位置。如果是长度为 2 的，我们在检查完第一个数字之后要再看是否有匹配的第二个。

```
TOKEN InsertId(string str)
{
    idWord[idNum].kind = 111;
    idWord[idNum].attribute = str;
    return idWord[idNum++];
}
```

将 strToken 中的标识符插入标识符表，并返回该标识符的 TOKEN。

```
TOKEN InsertConst(string str)
{
    coWord[coNum].kind = 100;
    // coWord[coNum].attribute = str;
    if(tmpID != "")
        coWord[coNum].attribute = tmpID;
    else
        coWord[coNum].attribute = str;
    return coWord[coNum++];
}
```

将 strToken 插入常数表，返回该常数的 TOKEN。

词法分析器

```
TOKEN myLexer()
{
    TOKEN tmp; //返回值
    int code;  //种别编码
    strToken = "";
    GetChar(); //获取下一个字符
    GetBC();   //跳过空格
    //结束，这时候 tmp 中 kind 值为 0
    if (ch == '#')
        return tmp;
    //为标识符或保留字
    else if (IsLetter(ch))
    {
        while (IsLetter(ch) || IsDigit(ch))
        {
            Concat();
            GetChar();
        }
        if (ch != '#')
            Retract();
        code = Reserve();
        //为标识符
        if (code == 0)
        {
            tmp = InsertId(strToken);
            tmpID = strToken;
            strToken.clear();
            return tmp;
        }
        //为保留字
        else
        {
            tmp.kind = code;
            tmp.attribute = "-";
            strToken.clear();
            return tmp;
        }
    }
    //为数字
    else if (IsDigit(ch))
    {
```

```
42.         while (IsDigit(ch))
43.         {
44.             Concat();
45.             GetChar();
46.         }
47.         if (ch != '#')
48.             Retract();
49.         //判断是否是 a = num1 + num2 等等这种情况
50.         GetChar();
51.         GetBC();
52.         if(IsOp(ch) != -1)
53.             tmpID = "";
54.         tmpStr--;   // 回溯
55.         tmp = InsertConst(strToken);
56.         strToken.clear();
57.         // tmp.isNum = 1;
58.         return tmp;
59.     }
60.     //为运算符
61.     else if (IsOp(ch) != -1)
62.     {
63.         tmp = opWord[IsOp(ch)];
64.         if (tmp.attribute.length() == 2)
65.             GetChar();
66.         tmp.attribute = "-";
67.         return tmp;
68.     }
69.     //为界符
70.     else if (IsBo(ch) != -1)
71.     {
72.         tmp = boWord[IsBo(ch)];
73.         tmp.attribute = "-";
74.         return tmp;
75.     }
76.     //其他情况排除，此时返回的 tmp 中 kind 为 0
77.     else
78.     {
79.         tmp.kind = 0;
80.         tmp.attribute = "-";
81.         return tmp;
82.     }
83. }
```

一个一个读入非空格的字符，并判断是哪种类型的单词符号，一旦确认是某个单词符号，

则返回其 TOKEN（种别编码+属性值），如果遇到了其他单词符号，则返回单词种别为 0 的 TOKEN。

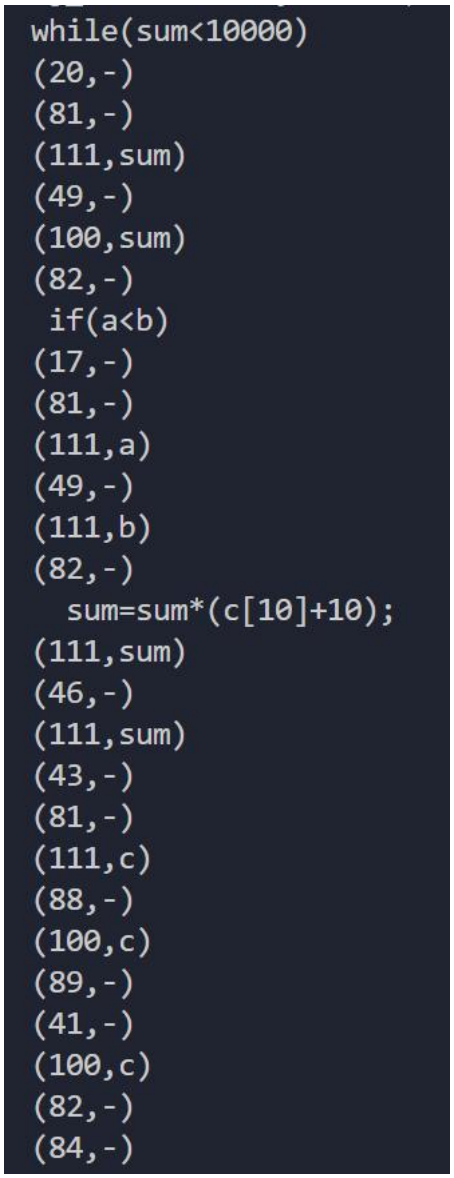
- 第一个字符为'#'：结束；
- 第一个字符为字母：输入直到遇到非数字或字母的符号，判断是否是保留字，是则返回其 TOKRN，不是则为标识符，加入标识符表并返回其 TOKEN；
- 第一个字符为数字：输入直到遇到非数字的符号，加入常数表并返回其 TOKEN；
- 第一个字符为运算符：判断是否是运算符，是则返回其 TOKEN；
- 第一个字符为界符：返回该界符的 TOKEN。

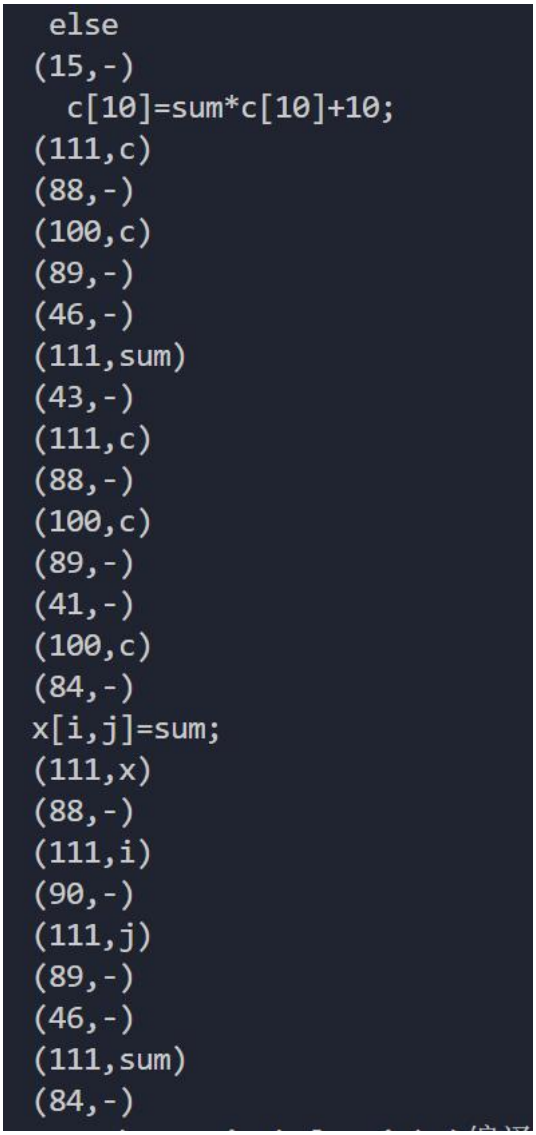
测试

对于输入：

```
while(sum<10000)
 if(a<b)
  sum=sum*(c[10]+10);
 else
  c[10]=sum*c[10]+10;
x[i,j]=sum;
```

我们拥有输出：





测试结束，结果正确。

语法分析器

功能

语法分析器将调用词法分析器返回的单词符号对程序的语法的组成进行分析，返回语法单位。这里我们主要对算数运算、关系运算、数组和各种语句等进行了分析，并输出了各个数据的分析过程。

实现

全局变量

```
int tmpStr = 0;  //当前输入串扫描到的位置
string tmpID;    //暂存标识符，供之后常量返回
int isEnd = 0;  //判断结束
TOKEN SYM; //从词法分析器调用的单词符号
```

功能函数

```
void advance()
{
    SYM = myLexer();
    while(SYM.kind == 0 && ch != '#')
        SYM = myLexer();
}
```

调用词法分析器，如果是单词符号表中不存在的单词符号则继续调用直到不是或结束。

产生式

```
void expr();
void term();
void rest5();
void rest6();
void unary();
void factor();
void bool_();
void equality();
void rest4();
void rel();
void rop_expr();
void stmts();
void rest0();
void stmt();
void loc();
void resta();
void elist();
void rest1();
```

总共有这么多个产生式，由于大多实现的方式都是类似的，在这里我就挑其中一些典型的来进行说明。

```
1.  void term()
2.  {
3.      cout<<"term -> unary rest6"<<endl;
4.      unary();
5.      rest6();
6.      return;
7.  }
```

对于 term()，其功能是 term→unary rest6，所以我们输出并调用 unary() 和 rest6()产生式。

算数运算

```
1.  void rest6()
2.  {
3.      // * 43
4.      if(SYM.kind == 43)
5.      {
6.          advance();
7.          cout<<"rest6 -> * unary rest6"<<endl;
8.          unary();
9.          rest6();
10.         return;
11.     }
12.     // / 44
13.     else if(SYM.kind == 44)
14.     {
15.         advance();
16.         cout<<"rest6 -> / unary rest6"<<endl;
17.         unary();
18.         rest6();
19.         return;
20.     }
21.     else
22.     {
23.         cout<<"rest6 -> ε"<<endl;
24.         return;
25.     }
26. }
```

对于乘除**运算**，先匹配乘除号，如果不匹配则直接输出并返回，匹配则进行下一步操作，调用产生式函数后返回。加减法等也是类似的，就不再一一说明了。

```
void factor()
{
    if(SYM.kind == 100)
    {
        cout<<"factor -> num"<<endl;
        advance();
    }
    // ( 81
    else if(SYM.kind == 81)
    {
        advance();
        expr();
        // ) 82
        if(SYM.kind == 82)
        {
            cout<<"factor -> (expr)"<<endl;
            advance();
        }
        else
        {
            cout<<"ERROR_factor_NO_)"<<endl;
            exit(0);
        }
    }
    else
    {
        cout<<"factor -> loc"<<endl;
        loc();
        return;
    }
}
```

匹配左括号等**界符**的时候，要注意与它对应的符号的匹配，在这里就是右括号，只有当右括号也匹配上了才算完成，否则要报错。这里就是匹配完左括号，执行做好里的产生式函数后再进行右括号的匹配。

布尔表达式及控制语句

```
void stmt()
```

```
2.  {
3.      // if 17
4.      if(SYM.kind == 17)
5.      {
6.          advance();
7.          // ( 81
8.          if(SYM.kind == 81)
9.          {
10.             advance();
11.             bool_();
12.             // ) 82
13.             if(SYM.kind == 82)
14.             {
15.                 advance();
16.                 stmt();
17.                 // ; 84
18.                 if(SYM.kind == 84)
19.                     advance();
20.                 // else 15
21.                 if(SYM.kind == 15)
22.                 {
23.                     cout<<"stmt -> if(bool) stmt else stmt"<<endl;
24.                     advance();
25.                     stmt();
26.                     return;
27.                 }
28.                 else
29.                 {
30.                     cout<<"ERROR_stmt_if_NO_else"<<endl;
31.                     exit(0);
32.                 }
33.             }
34.             else
35.             {
36.                 cout<<"ERROR_stmt_if_NO_)"<<endl;
37.                 exit(0);
38.             }
39.         }
40.         else
41.         {
42.             cout<<"ERROR_stmt_if_NO_("<<endl;
43.             exit(0);
44.         }
45.     }
```

```
    //while 20
    else if(SYM.kind == 20)
    {
        advance();
        // ( 81
        if(SYM.kind == 81)
        {
            advance();
            bool_();
            // ) 82
            if(SYM.kind == 82)
            {
                cout<<"stmt -> while(bool) stmt"<<endl;
                advance();
                stmt();
                return;
            }
            else
            {
                cout<<"ERROR_stmt_while_NO_)"<<endl;
                exit(0);
            }
        }
        else
        {
            cout<<"ERROR_stmt_while_NO_("<<endl;
            exit(0);
        }
    }
    else
    {
        loc();
        // = 46
        if(SYM.kind == 46)
        {
            cout<<"stmt -> loc = expr"<<endl;
            advance();
            expr();
        }
        else
        {
            cout<<"ERROR_stmt_NO_="<<endl;
            exit(0);
        }
```

```
90.     }
91. }
```

对于 if 和 while 这样的**控制语句**，其实本质上和上面的情况是一样的，这里我以 if(bool) stmt else stmt 为例，当匹配完 if 之后我们匹配左括号，之后进行 bool 函数调用再匹配右括号，匹配成功后再进行 stmt 函数调用（也就是调用自己），之后匹配 else，最后再调用 stmt 函数。其中有任意地方不匹配都要进行报错处理。

数组

```
1.  void resta()
2.  {
3.      // [ 88
4.      if(SYM.kind == 88)
5.      {
6.          advance();
7.          elist();
8.          // ] 89
9.          if(SYM.kind == 89)
10.         {
11.             cout<<"resta -> [elist]"<<endl;
12.             advance();
13.             return;
14.         }
15.         else
16.         {
17.             cout<<"ERROR_resta_NO_]"<<endl;
18.             exit(0);
19.         }
20.     }
21.     else
22.     {
23.         cout<<"resta -> ε"<<endl;
24.         return;
25.     }
26. }
```

对于**数组**，操作和左右括号是类似的，先匹配[但后调用中间的 elist()函数，最后匹配右括号。中间的函数会继续扩充里面的内容，其过程较为简单，也不再一一说明了。

测试

对于输入:

```
while(sum<10000)
 if(a<b)
  sum=sum*(c[10]+10);
 else
  c[10]=sum*c[10]+10;
x[i,j]=sum;
```

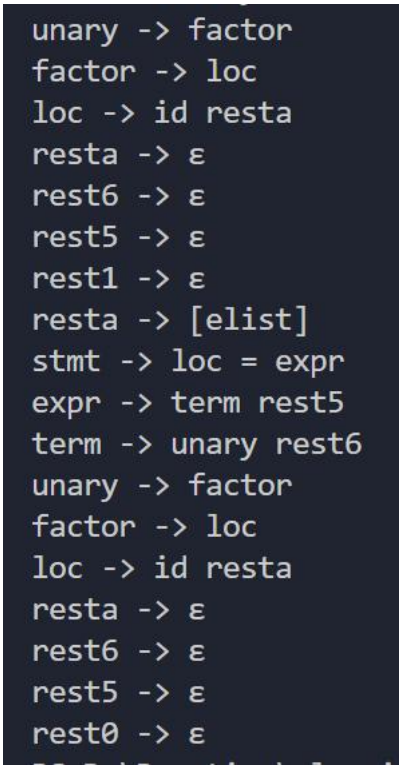
我们拥有输出:

```
while(sum<10000)
 if(a<b)
  sum=sum*(c[10]+10);
 else
  c[10]=sum*c[10]+10;
x[i,j]=sum;
^Z
stmts -> stmt rest0
bool -> quality
equality -> rel rest4
rel -> expr rop_expr
expr -> term rest5
term -> unary rest6
unary -> factor
factor -> loc
loc -> id resta
resta -> ε
rest6 -> ε
rest5 -> ε
rop_expr -> < expr
expr -> term rest5
term -> unary rest6
unary -> factor
factor -> num
rest6 -> ε
rest5 -> ε
rest4 -> ε
stmt -> while(bool) stmt
bool -> quality
equality -> rel rest4
rel -> expr rop_expr
expr -> term rest5
term -> unary rest6
unary -> factor
factor -> loc
```

```
unary -> factor
factor -> loc
loc -> id resta
resta -> ε
rest6 -> ε
rest5 -> ε
rop_expr -> < expr
expr -> term rest5
term -> unary rest6
unary -> factor
factor -> loc
loc -> id resta
resta -> ε
rest6 -> ε
rest5 -> ε
rest4 -> ε
loc -> id resta
resta -> ε
stmt -> loc = expr
expr -> term rest5
term -> unary rest6
unary -> factor
factor -> loc
loc -> id resta
resta -> ε
rest6 -> * unary rest6
unary -> factor
expr -> term rest5
term -> unary rest6
unary -> factor
factor -> loc
loc -> id resta
elist -> expr rest1
expr -> term rest5
term -> unary rest6
```

```
unary -> factor
factor -> num
rest6 -> ε
rest5 -> ε
rest1 -> ε
resta -> [elist]
rest6 -> ε
rest5 -> + term rest5
term -> unary rest6
unary -> factor
factor -> num
rest6 -> ε
rest5 -> ε
factor -> (expr)
rest6 -> ε
rest5 -> ε
stmt -> if(bool) stmt else stmt
loc -> id resta
elist -> expr rest1
expr -> term rest5
term -> unary rest6
unary -> factor
factor -> num
rest6 -> ε
rest5 -> ε
rest1 -> ε
resta -> [elist]
stmt -> loc = expr
expr -> term rest5
term -> unary rest6
unary -> factor
factor -> loc
loc -> id resta
resta -> ε
rest6 -> * unary rest6
```

```
unary -> factor
factor -> loc
loc -> id resta
elist -> expr rest1
expr -> term rest5
term -> unary rest6
unary -> factor
factor -> num
rest6 -> ε
rest5 -> ε
rest1 -> ε
resta -> [elist]
rest6 -> ε
rest5 -> + term rest5
term -> unary rest6
unary -> factor
factor -> num
rest6 -> ε
rest5 -> ε
rest0 -> ε
stmts -> stmt rest0
loc -> id resta
elist -> expr rest1
expr -> term rest5
term -> unary rest6
unary -> factor
factor -> loc
loc -> id resta
resta -> ε
rest6 -> ε
rest5 -> ε
rest1 -> , expr rest1
expr -> term rest5
term -> unary rest6
unary -> factor
```



可以看到，没有报错，结果正确，测试完毕。

语义分析器和中间代码生成器

功能

语义分析器的功能是调用语法分析器得到语法单位，通过填充一些操作来实现对各种语句的翻译，这里用四元式来表示。这里我完成了**赋值表达式、数组、布尔表达式和控制表达式**的语义分析和中间代码生成。

实现

全局变量

```
string out4YS[100][4];  //保存输出结果 四元式
int outList[100];       //保存各个四元式的链首
int nOut = 0;   //当前存放位置
int nextquad = 0;   //指向下一条将要产生的四元式地址
int countEmit = 0;
```

```
int startOut = 0;
bool flag[100] = {0};
int countTemp = 1;
```

文法符号

```
// 文法符号属性
struct X
{
    string place = "";      //变量名字
    string inArray = "";    //数组名字
    int inNdim = 0;         //下标表达式个数及维数
    string inPlace = "";    //由 Elist 中的下标表达式计算出来的值
    string array = "";      //指向符号表中相应数组名字表项的指针
    string offset = "";     //X 为简单名字，为 NULL；X 为数组名字，为数组地址中变量部分
    int truelist = -1;
    int falselist = -1;
    int inTruelist = -1;
    int inFalselist = -1;
    int nextlist = -1;  //指向下一个四元式
    int inNextlist = -1;
};
```

这是文法符号的数组，包括产生式的各种属性。

功能函数

```
void initOutList()
{
    for(int i=0; i<100; ++i)
        outList[i] = -1;
}
```

初始化四元式的链首表

```
void emit(string op, string a, string b, string tmp)
{
    nextquad++;
    // cout<<countEmit++<<": "<<op<<","<<a<<","<<b<<","<<tmp<<endl;
    out4YS[nOut][0] = op;
    out4YS[nOut][1] = a;
```

```
7.      out4YS[nOut][2] = b;
8.      out4YS[nOut][3] = tmp;
9.      outList[nOut] = nOut;
10.     nOut++;
11. }
```

将生成的四元式加入到四元式数组 out4YS 中去，并更新 nOut 和 nextquad。同时，还需要将该四元式的链首初始化为自己本身。

```
1.  void output4YS()
2.  {
3.      for(int i=startOut; i<nOut; ++i)
4.          cout<<countEmit++<<": "<<out4YS[i][0]<<","<<out4YS[i][1]<<","<<out4YS[i][2]<<","<<out4YS[i][3]<<endl;
5.  }
```

输出四元式数组里的四元式。

```
1.  int makelist(int i)
2.  {
3.      return i;
4.  }
```

创建一个仅含 i 的新链表，由于该链表只含一个元素，所以我们直接返回即可，后续我采用数组的形式来模拟链表。

```
1.  int merge(int p1, int p2)
2.  {
3.      out4YS[p1][3] = to_string(p2);
4.      for(int i=0; i<nOut; ++i)
5.      {
6.          if(out4YS[i][3].find('T') == 0)
7.              continue;
8.          if(outList[i] == p1)
9.          {
10.             out4YS[i][3] = to_string(p2);
11.             outList[i] = p2;
12.         }
13.     }
14.     return p2;
15. }
```

Merge 函数的作用是将 p1 和 p2 为链首的两条链合二为一，我这里的实现是将链首为 p1 的链以及 p1 链的链首改为 p2，并返回 p2。

```
void backpatch(int p, int t)
{
    if(p != -1)
        for(int i=0; i<nOut; ++i)
        {
            if(outList[i] == p && flag[i]==0)
            {
                out4YS[i][3] = to_string(t);
                outList[i] = t;
                flag[i] = 1;
            }
        }
}
```

回填函数，把 p 所连接的每个四元式的第四个参数都回填 t。

```
string limit(string array, int j){
    return array + to_string(j);
}
```

返回 array 数组的第 j 维长度，这里就简单返回 arrayj。

```
int M()
{
    return nextquad;
}

int N()
{
    int n = makelist(nextquad);
    emit("j","-","-","0");
    return n;
}
```

M()和 N()函数本来应该算作产生式里的内容，但是介于我的实现方式和和课本略有不同，所以就放在功能函数里了。首先他们的功能都是用来回填的，对于 M，N，后续只需要定义一个 int 类型的变量来保存即可。对于 M 而言，保存到是是当时语句里的下一条，也就是 nextquad，而 N 则会产生一个新的四元式，并返回新生成的那个链的链首。

```
int countTemp = 1;
string newtemp()
{
    return "T" + to_string(countTemp++);
}
```

生成一个新的临时变量名，如 T1。

产生式

```
X expr();
X term();
X rest5(X rest5_);
X rest6(X rest6_);
X unary();
X factor();
X bool_();
X equality();
X rest4(X rest4_);
X rel();
X rop_expr(X rop_expr_);
X stmts();
X rest0(X rest0_);
X stmt();
X loc();
X resta(X resta_);
X elist(X elist_);
X rest1(X rest1_);
```

需要实现的产生式如上，和上面一样，在这里我只选择一部分来进行说明，其余原理类似或者十分简单的就不再赘述了。

赋值表达式

```
X rest5(X rest5_)
{
    // cout<<rest5_.inArray<<endl;
    // + 41
    if(SYM.kind == 41)
    {
        X rest5Tmp, rest51Tmp,termTmp;
```

```
        advance();
        // cout<<"rest5 -> + term rest5"<<endl;
        termTmp = term();
        rest51Tmp.inArray = newtemp();
        emit("+",rest5_.inArray,termTmp.place,rest51Tmp.inArray);
        rest5Tmp = rest5(rest51Tmp);
        rest5Tmp.place = rest51Tmp.place;
        return rest5Tmp;
    }
    // - 42
    else if(SYM.kind == 42)
    {
        X rest5Tmp, rest51Tmp,termTmp;
        advance();
        // cout<<"rest5 -> - term rest5"<<endl;
        termTmp = term();
        rest51Tmp.inArray = newtemp();
        emit("-",rest5_.inArray,termTmp.place,rest51Tmp.inArray);
        rest5Tmp = rest5(rest51Tmp);
        rest5Tmp.place = rest51Tmp.place;
        return rest5Tmp;
    }
    else
    {
        // cout<<"rest5 -> ε"<<endl;
        rest5_.place = rest5_.inArray;
        return rest5_;
    }
}
```

这里选加减为例，首先把原本的输出去掉，改为将结果加入四元式数组，然后将原本的 void 函数改为 X 的函数并调用他们，对得到的结果进行利用。当然，在这里由于需要调用自身，所以需要保存一下前一次这个函数的返回值，我们通过传参传入过来，也就是 rest5_。

数组

```
X resta(X resta_)
{
    // [ 88
    if(SYM.kind == 88)
    {
        X elistTmp, restaTmp;
        // elistTmp.inArray = resta_.inArray;
        advance();
```

```
        elistTmp = elist(elistTmp);
        // ] 89
        if(SYM.kind == 89)
        {
            elistTmp.inArray = resta_.inArray;
            // cout<<"resta -> [elist]"<<endl;
            advance();
            restaTmp.place = newtemp();
            emit("-",elistTmp.inArray,"C",restaTmp.place);
            restaTmp.offset = newtemp();
            emit("*","w",elistTmp.offset,restaTmp.offset);
            return restaTmp;
        }
        else
        {
            cout<<"ERROR_resta_NO_]"<<endl;
            exit(0);
        }
    }
    else
    {
        // cout<<"resta -> ε"<<endl;
        resta_.place = resta_.inArray;
        resta_.offset = "";
        return resta_;
    }
}
```

数组的实现方式和前面的赋值语句大同小异，也是在前面语法分析的基础上去掉输出并改为 emit 语句，匹配单词符号、函数调用等等。

布尔表达式和控制语句

```
X stmt()
{
    // if 17
    if(SYM.kind == 17)
    {
        X boolTmp, stmt1Tmp, stmt2Tmp;
        int m1, m2, n;
        int m0 = M();
        advance();
        // ( 81
        if(SYM.kind == 81)
```

```
        {
            advance();
            boolTmp = bool_();
            // ) 82
            if(SYM.kind == 82)
            {
                advance();
                m1 = M();
                stmt1Tmp = stmt();
                n = N();
                // ; 84
                if(SYM.kind == 84)
                    advance();
                // else 15
                if(SYM.kind == 15)
                {
                    // cout<<"stmt -> if(bool) stmt else stmt"<<endl;
                    advance();
                    m2 = M();
                    stmt2Tmp = stmt();
                    stmt2Tmp.nextlist = merge(merge(stmt1Tmp.nextlist,n),stmt2Tmp.nextlist);
                    backpatch(boolTmp.truelist,m1);
                    backpatch(boolTmp.falselist,m2);
                    return stmt2Tmp;
                }
                else
                {
                    cout<<"ERROR_stmt_if_NO_else"<<endl;
                    exit(0);
                }
            }
            else
            {
                cout<<"ERROR_stmt_if_NO_)"<<endl;
                exit(0);
            }
        }
        else
        {
            cout<<"ERROR_stmt_if_NO_("<<endl;
            exit(0);
        }
    }
```

```
    //while 20
    else if(SYM.kind == 20)
    {
        X boolTmp, stmtTmp;
        int m1,m2;
        advance();
        // ( 81
        if(SYM.kind == 81)
        {
            advance();
            m1 = M();
            // cout<<m1<<endl;
            boolTmp = bool_();
            // ) 82
            if(SYM.kind == 82)
            {
                // cout<<"stmt -> while(bool) stmt"<<endl;
                advance();
                m2 = M();
                stmtTmp = stmt();
                m3 = M()+1;
                boolFALSE = boolTmp.falselist;
                backpatch(stmtTmp.nextlist,m1);
                backpatch(boolTmp.truelist,m2);
                stmtTmp.nextlist = boolTmp.falselist;
                emit("j","-","-",to_string(m1));
                return stmtTmp;
            }
            else
            {
                cout<<"ERROR_stmt_while_NO_)"<<endl;
                exit(0);
            }
        }
        else
        {
            cout<<"ERROR_stmt_while_NO_("<<endl;
            exit(0);
        }
    }
    else
    {
        X exprTmp,locTmp,stmtTmp;
        locTmp = loc();
```

```
        // = 46
         if(SYM.kind == 46)
         {
             // cout<<"stmt -> loc = expr"<<endl;
             advance();
             exprTmp = expr();
             if(locTmp.offset == "")
                 emit("=",exprTmp.place,"-",locTmp.place);
             else
                 emit("[]=",exprTmp.place,"-",locTmp.place+"["+locTmp.offset
   +"]");
             stmtTmp.nextlist = makelist(nextquad);/////
             return stmtTmp;
         }
         else
         {
             cout<<"ERROR_stmt_NO_="<<endl;
             exit(0);
         }
     }
}
```

由于布尔表达式的实现和之前的差不太多，在此就和控制语句一起进行说明。除了基本的函数调用，匹配单词符号等，控制语句最重要的东西就是回填，这里就需要链表的操作。我采用的数组，从某种意义上也可以对链表进行模拟。这里我会对 while 语句和 if 语句分别进行说明。

对于 if 语句，其形式为 stmt⟶if(bool) m1 stmt1 n else m2 stmt2，那么在进行到后续步骤的时候，我们需要对 bool 这个布尔为真和假的情况的跳转进行回填，需要对 stmt1 和是 stmt2 的跳转做出合并，合并方式为将 stmt1 的 nextlist 和 n 都变为 stmt2.nextlist，这里需要调用 merge 函数来进行合并操作。

对于 while 语句 stmt⟶ while(m1 bool) m2 stmt1，我们也需要对 bool 为真的情况下进行普通的回填操作，而对于 bool 为假的情况，则稍有不同。我们在对前一个 stmt 进行回填为 while 的地方后将 stmt 的 nextlist 指向它的 falselist。通俗的来讲，就是首先将 stmt1 的 nextlist 指向 while 循环，保证结束后能继续进入到 bool 判断之后，接着要回填 bool 为真的情况，及继续进行 stmt1 的操作，也就是回填 m2，而对于 stmt 本身，其 nextlist 指向 bool 为假的情况，也就是跳出 while 循环。

测试

对于输入：

```
while(a<b)
 if(c)
  while(x<y)
    y=x;
 else
  if(x>y)
    x=y;
    else
    y=x;
a=y;
```

我们拥有输出：

```
while(a<b)
 if(c)
  while(x<y)
    y=x;
 else
  if(x>y)
    x=y;
    else
    y=x;
a=y;
^Z
0: j<,a,b,2
1: j,-,-,15
2: jnz,c,-,4
3: j,-,-,9
4: j<,x,y,6
5: j,-,-,0
6: =,x,-,y
7: j,-,-,4
8: j,-,-,0
9: j>,x,y,11
10: j,-,-,13
11: =,y,-,x
12: j,-,-,0
13: =,x,-,y
14: j,-,-,0
15: =,y,-,a
```

结果正确，测试完毕。

总结

出现的问题

首先就是对于 while 循环语句，在一开始我选择了用自己方法来实现，也就是将 while 里头的 bool 的 falselist 定为 stmt1 执行完后的那个 nextquad。但其实这样是不对的，当拥有多个 while 嵌套的时候，while 结束后不一定直接走向下一行，而可能会跳转到别的地方上去。所以后续我修改为了课本上的方法。

其次是对于 N（）数组生成的四元式语句，我将其链首定为了-1，意为不可改变。但这个操作并没有实际意义，并且还带来了麻烦。在后续合并的时候因为其 outList 已经改变，导致合并的时候是找不到这个四元式的，这样会带来二次跳转的问题。

经验与展望

这次的实验确实让我收获了很多，它加深了我对编译原理的理解，也大大提升了我的编码能力。纸上得来终觉浅，实践的过程能更好地帮助我消化课上学习的理论知识，而这次实践最大的作用也恰恰在此。

在编码的时候，很多细节都需要格外注意，可能一个小地方的错误就需要付出狠毒偶的努力来 debug。这也增加了我的代码能力，提升了我对代码的理解。

附录

完整代码

考虑到代码有 1061 行，我将其上传到了这个网站：
https://paste.ubuntu.com/p/V4TWv9sdCF/