《编译技术》
课程设计 文档

学号：_____12061154_____

姓名：_____冯飘飘_____

2015 年 1 月 10 日

目录

一．需求说明

1．文法说明

获取高级文法，包含 switch 和 while 循环，以下是对文法的解读：

＜加法运算符＞ ::= +|-
分析：表示加减运算符

＜乘法运算符＞ ::= *|/
分析：表示乘除运算符

＜关系运算符＞ ::= <|<=|>|>=|!=|==
分析：表示两个表达式之间的关系

＜字母＞ ::= _|a|...|z|A|...|Z
分析：表示字母是下划线和 26 个字母的大小写
示例：_下划线、 小写 a、 大写 A 等等

＜数字＞ ::= 0|＜非零数字＞
分析：表示单个数字是 0 和非零数字
示例：0、1、2、3、4……

＜非零数字＞ ::= 1|...|9
分析：表示非零单个数字
示例：1、2、3、4……

＜字符＞ ::= '＜加法运算符＞'|'＜乘法运算符＞'|'＜字母＞'|'＜数字＞'
分析：表示单个字符是单引号括起来的加法运算符、乘法运算符、字母或数字
示例：'+'、'-'、'*'、'/'、'_'、'a'、'A'、'4'、'6'等

＜字符串＞ ::= "{十进制编码为 32,33,35-126 的 ASCII 字符}"
分析：表示字符串是由双引号括起来的空白符、感叹号和十进制编码 35-126 的 ASCII 字符组合起来的
示例：""、"sdfnjs"、"jfe384jdf#$Fs"、"8(*&ew8df"

＜程序＞ ::= [＜常量说明＞][＜变量说明＞]{＜有返回值函数定义＞|＜无返回值函数定义＞}＜主函数＞
分析：表示程序的结构：至多有一个的常量说明和变量说明，可有 0 到多个的有返回值函数定义和无返回值函数定义，接着是主函数.若有常量说明则它一定在最前面；若有变量说明则一定紧接着常量说明(没有常量说明则在最前面)，若有有返回值函数定义或无返回值函数定义则一定在主函数之前，顺序不能乱，程序里一定有主函数。
示例：

```
const int global=0;//常量说明
```

```
int a,b; //变量说明
int foo(){return 1;}//有返回值函数定义
void foo2(){}//无返回值函数定义
//主函数
void main()
{
    cons tint c=0;//常量说明
    char d;//变量说明
    a=0;b=1;//语句列
}
```

＜常量说明＞ ::= const＜常量定义＞;{ const＜常量定义＞;}
分析：表示<常量说明>是一个或多个 const 开头分号结尾的常量定义
示例：const int a=0,b=1; const char abc=’a’;

＜常量定义＞ ::= int＜标识符＞＝＜整数＞{,＜标识符＞＝＜整数＞}
| char＜标识符＞＝＜字符＞{,＜标识符＞＝＜字符＞}
分析：以 int 或 char 开头的常量定义，可连续赋值多个标识符
示例：int a=0,b=1 、 char s1=’c’,s2=’d’

＜无符号整数＞ ::= ＜非零数字＞{＜数字＞}
分析：表示无符号整数是由非零数字开头，其后有 0 到多个数字
限定条件：非零数字开头，如 011 不是<无符号整数>
示例：123、245、1、4

＜整数＞ ::= [＋|－]＜无符号整数＞| 0
分析：表示整数是正负号或无符号开头，后接一个无符号整数，或者是 0
示例：-123、0、+6352、234

＜标识符＞ ::= ＜字母＞{＜字母＞|＜数字＞}
分析：表示标识符是由单个字母开头，后跟 0 个或若干字母或数字
限定条件：必须<字母>开头
示例：_abd32d、jdshf、adg2

＜声明头部＞ ::= int＜标识符＞ | char＜标识符＞
分析：表示声明头部是由 int 或 char 开头，后接一个标识符
示例：int abc 、 char cd

＜变量说明＞ ::= ＜变量定义＞;{＜变量定义＞;}
分析：表示变量说明由至少一个<变量定义>;组成
限定条件：至少一个<变量定义>，分号结尾
示例：int a,b; char s[10],c,d,f[3]; 、 int a,c,d;

＜变量定义＞ ::= ＜类型标识符＞(＜标识符＞|＜标识符＞‘[’＜无符号整数＞‘]’){,(＜标识符

＞|＜标识符＞‘[’＜无符号整数＞‘]’)}
分析：表示变量定义由 int 或 char 开头，后接一个或若干个标识符或数组定义
限定条件：int 或 char 开头
示例：int a,b,c　　、　char a,b,s[18]

＜常量＞ ::= ＜整数＞| ＜字符＞
分析：表示常量是整数或者字符
示例： 123 、 ’a’ 、 ‘9’

＜类型标识符＞ ::= int | char
分析：表示类型标识符是 int 或 char

＜有返回值函数定义＞ ::= ＜声明头部＞‘(’＜参数＞‘)’ ‘{’＜复合语句＞‘}’
分析：表示有返回值函数定义的语法：<声明头部>(<参数>){<复合语句>}
限定条件：声明头部是 int 或 char 开头
示例：int max(int a,int b){if (a>b) return a; else return b}//有参数
int foo(){return 1;}//无参数

＜无返回值函数定义＞ ::= void＜标识符＞‘(’＜参数＞‘)’ ‘{’＜复合语句＞‘}’
分析：表示无返回值函数定义的语法：void<标识符>(<参数>){<复合语句>}
限定条件：以 void 开头
示例：

```
void foo()
{
    int a,b;//变量说明
    //语句列
    a=1;
    b=a*2;
}
```

＜复合语句＞ ::= ［＜常量说明＞］［＜变量说明＞］＜语句列＞
分析：表示复合语句是至多一个常量说明，至多一个变量说明，语句列组成
限定条件：必有<语句列>,若常量说明和变量说明都有则常量说明一定在变量说明前面
示例：

```
const int a=0,b=1;//常量说明
char c,d;//变量说明
//语句列
if(a<b)
    b=a;
```

＜参数＞ ::= ＜参数表＞
分析：表示参数是参数表
示例：int a,char b,char c

＜参数表＞ ::= ＜类型标识符＞＜标识符＞{,＜类型标识符＞＜标识符＞}| ＜空＞
分析：表示参数表是空或者至少一个类型标识符加上标识符,以逗号分隔
限定条件：要么为空，要么 int 或 char 开头
示例：int a,int b 、 char c

＜主函数＞ ::= void main‘(’ ‘)’ ‘{’＜复合语句＞‘}’
分析：表示主函数的形式是：void main(){<复合语句>}
限定条件：void main()开头，复合语句用大括号括起来
示例：

```
void main()
{
    const int a=1;//常量说明
    char b;//变量说明
    //语句列
    b=’c’;//语句
    a=a+1;//语句
}
```

＜表达式＞ ::= ［＋|－］＜项＞{＜加法运算符＞＜项＞}
分析：表示表达式是正负号开头或没有正负号，后面跟一个<项>或以<加法运算符>连接起来的若干个<项>
限制条件：至少有一个<项>
示例：a+f、-123、234、c*d+a*b

＜项＞ ::= ＜因子＞{＜乘法运算符＞＜因子＞}
分析：表示项是由一个<因子>或以<乘法运算符>连接起来的若干个<因子>组成的
限制条件：至少有一个<因子>
示例：abc//因子
c*d、e/f*g//<因子>{<乘法运算符><因子>}

＜因子＞ ::= ＜标识符＞|＜标识符＞‘[’＜表达式＞‘]’|＜整数＞|＜字符＞|＜有返回值函数调用语句＞|‘(’＜表达式＞‘)’
分析：表示因子的语法，因子可以是标识符、标识符[表达式]、整数或字符或有返回值函数调用语句或(表达式)
示例：

```
abc//标识符
a[2]//标识符[表达式]
123//整数
’d’//字符
function()//有返回值函数调用语句
(a+b)//(表达式)
```

＜语句＞ ::= ＜条件语句＞|＜循环语句＞| ‘{’＜语句列＞‘}’|＜有返回值函数调用语句＞;|＜无返回值函数调用语句＞;|＜赋值语句＞;|＜读语句＞;|＜写语句＞;|＜空＞;|＜情况语句＞|＜返回语句＞;

分析：表示语句的文法，语句可以使条件语句、循环语句、{语句列}、有返回值函数调用语句;、无返回值函数调用语句;、赋值语句;、读语句;、写语句;、空;、情况语句、返回语句;
示例：

```
if (a==b) b=b+1;//条件语句
while(a<10){a=a+1;}//循环语句
{a=0;b=1;}//{语句列}
max(a,b);// 有返回值函数调用语句;
foo();//无返回值函数调用语句;
a=1;// 赋值语句;
scanf();//读语句;
printf(a); //写语句;
switch(a){case0:b=1;}// 情况语句
return a;// 返回语句;
```

＜赋值语句＞ ::= ＜标识符＞＝＜表达式＞|＜标识符＞‘[’＜表达式＞‘]’=＜表达式＞
分析：表示赋值语句的文法，标识符=表达式，或标识符[表达式]=表达式
限制条件：必须是标识符开头
示例：abc=(a+b)*(a-b)//标识符=表达式
s[0]=c+d//标识符[表达式]=表达式

＜条件语句＞ ::= if ‘(’＜条件＞‘)’＜语句＞[else＜语句＞]
分析：表示条件语句的文法，if(条件)语句或 if(条件)语句 else 语句。if 开头，可以没有 else 语句
示例：

```
if(a==1) b=1;//if(条件)语句
if(a==1) b=1; else b=0;// if(条件)语句 else 语句
```

＜条件＞ ::= ＜表达式＞＜关系运算符＞＜表达式＞|＜表达式＞ //表达式为 0 条件为假，否则为真
分析：表示条件的组成，表达式或表达式之间的关系
示例：

```
a < b 、a==1 //<表达式><关系运算符><表达式>
true//表达式
```

＜循环语句＞ ::= while ‘(’＜条件＞‘)’＜语句＞
分析：表示 while 循环语句
示例：

```
While(i<10)
{
    i=i+1;//语句
}
```

＜情况语句＞ ::= switch ‘(’＜表达式＞‘)’ ‘{’＜情况表＞ ‘}’
分析：switch case 语句，对于“表达式”分情况讨论
示例：

```
wwitch (a)
```

```
{
    //情况表
    case0:b=1;
    case1:b=4;
}
```

＜情况表＞　::=　＜情况子语句＞{＜情况子语句＞}
分析：情况语句里的情况表
限制条件：至少有一个情况子语句
示例：case0:a=1;　//情况子语句
//＜情况子语句＞{＜情况子语句＞}
case1:{b=0;}
case2:{b=3;}

＜情况子语句＞　::=　case＜常量＞：＜语句＞
分析：表示情况子语句
限制条件：case 开头
示例：case0:b=3;

＜有返回值函数调用语句＞ ::= ＜标识符＞‘(’＜值参数表＞‘)’
分析：调用有返回值函数
示例：max(a,b) 、 min(a,b)

＜无返回值函数调用语句＞ ::= ＜标识符＞‘(’＜值参数表＞‘)’
分析：调用无返回值函数
示例：fool(a) 、 foo2(a,b)

＜值参数表＞　::= ＜表达式＞{,＜表达式＞}｜＜空＞
分析：表示函数的参数表
示例：〈空〉、a,b、abd

＜语句列＞　::= {＜语句＞}
分析：表示 0 到多个语句
示例：a=1;b=0;

＜读语句＞　::=　scanf ‘(’＜标识符＞{,＜标识符＞}‘)’
分析：读入语句
示例：scanf(a,c,d)

＜写语句＞　::= printf ‘(’ ＜字符串＞,＜表达式＞ ‘)’｜printf ‘(’＜字符串＞ ‘)’｜
printf ‘(’＜表达式＞‘)’
分析：输出语句
示例：printf(“a=”,a)//printf(字符串，表达式)
printf(abc)//printf(字符串)

printf(a+b)//printf(表达式)

＜返回语句＞ ::= return[‘(’＜表达式＞‘)’]
分析：返回一个值
示例：return
return (a)
return (a+b)

2. 目标代码说明

目标代码是 mips 汇编

3. 优化方案*

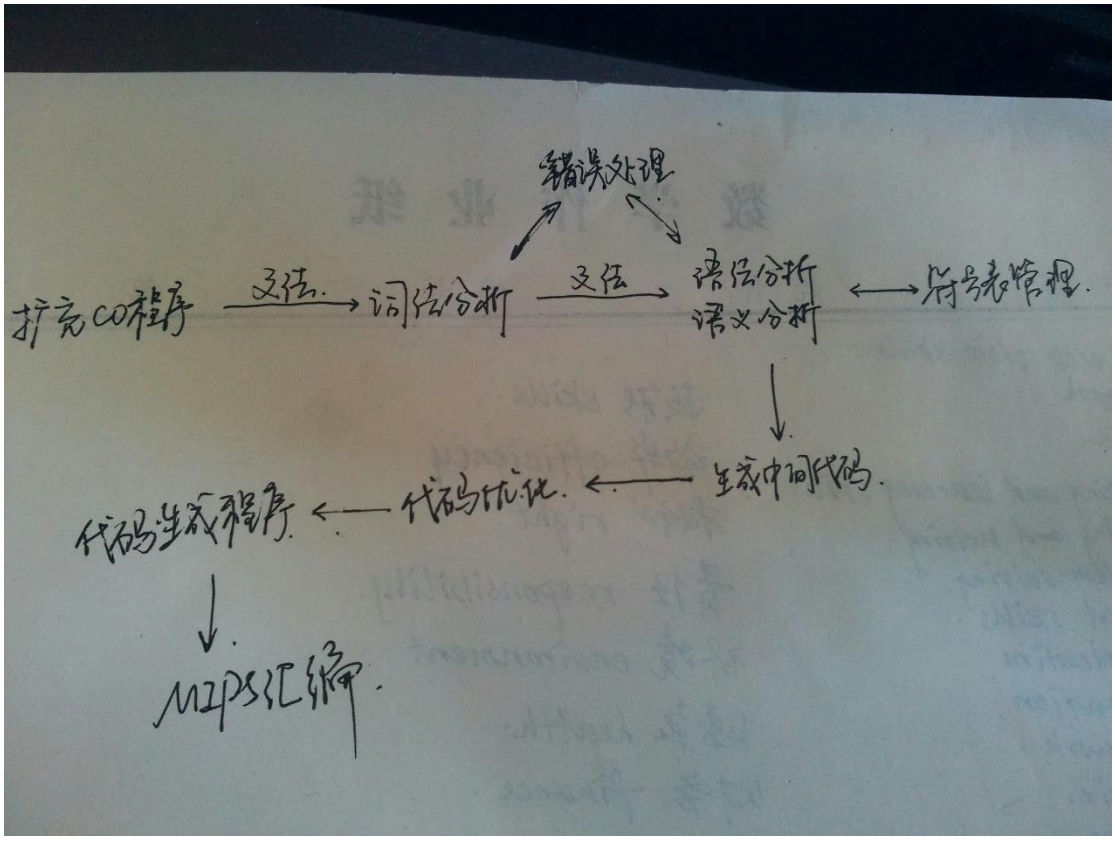
基本块内部的公共子表达式删除（DAG 图）；

全局寄存器分配（引用计数或着色算法）；

代码生成时合理利用临时寄存器，并能生成较高质量的目标代码；

二．详细设计

1．程序结构


错误处理
扩充C0程序
文法
词法分析
文法
语法分析
语义分析
符号表管理
生成中间代码
代码优化
代码生成程序
MIPS汇编

2．类/方法/函数功能

各个函数的功能见注释。

2.1 词法分析

```
//token的类型
enum TOKEN_TYPE
{
    IDEN, INTCON, CHARCON, STRCON, CONSTTK, INTTK, CHARTK, VOIDTK, MAINTK,
    IFTK, THENTK, ELSETK,
    WHILETK, SWITCHTK, CASETK, DEFAULTTK, FORTK, SCANFTK, PRINTFTK, RETURNTK,
    PLUS, MINU, MULT, DIV, LSS, LEQ, GRE, GEQ, EQL, NEQ, ASSIGN,
    SEMICN, COMMA, COLON, LPARENT, RPARENT, LBRACK, RBRACK, LBRACE, RBRACE,
    ERRORTK, NONE
};

//token结构体
typedef struct TOKEN
{
    TOKEN_TYPE type;//token类型
    union Value     //token的值
    {
        int i;
        float f;
        char c;
        char str[MAX_STR_LEN];
        int line;   //error line
    }value;
}TOKEN;
```

```
//词法分析类
class Lexer
{
private:
    int curChar ;    //the place of current char
    char program[MAX_PRO_LEN];  //读入程序

    char NextChar();         //读入下一个字符
    void BackChar(int mark);//curChar回到mark
    TOKEN SingleSeparator(char c);  //返回一个SingleSeparator
    TOKEN DoubleSeparator(char c, char d);  //返回一个DoubleSeparator
    TOKEN Reserver(char str[]); //Identifier or Reserver
public:
    Lexer();    //构造函数，初始化curChar=0
    void getProgram(char filename[]);    //getProgram程序输入到字符串program中
    TOKEN getToken();                    //获取当前token
    TOKEN LookAhead(int k);              //获取下k个token
};
```

和其他调用的函数：

```
bool isSpace(char c);    // 判断字符c是否为' '
bool isNewline(char c); // 判断字符c是否为'\n'
bool isTab(char c);      // 判断字符c是否为'\t'

bool isaddop(char c);    // 判断字符c是否为+ -
bool ismultop(char c);   // 判断字符c是否为* /

bool isDigit(char c);    // 判断字符c是否为数字
bool isNonZeroDigit(char c);//判断字符c是否为非零数字
```

```
bool isLetter(char c);  //判断字符c是否为字母
bool isChar(char c);    //判断字符c是否为字符
bool isString(char *str);   //判断字符串str是否为字符串
bool isSeparator(char c);   //判断字符c是否为分割符
bool isSingleSeparator(char c); //判断字符c是否为单分割符
bool isDoubleSeparator(char c); //判断字符c是否为双分割符

int Ctod(char c);   //%c --> %d 将字符 c 转换为对应的 ASCII 码
```

关键算法：getToken()是循环读入字符直到分隔符出现，判断读入的什么类型的 token，并返回。LookAhead(int k)是表示获取第 k 个 token，循环调用 getToken()函数，然后将 curChar 的位置重置回调用前的状态。

2.2 语法、语义分析、生成中间代码及错误处理

```
//语法分析，包含语义分析、生成中间代码及错误处理
class Parser
{
public:
    TOKEN token;//当前token
    Lexer lexer;//调用词法分析
//错误处理
    void error(int errid);              //错误处理
    void error(int errid, char str[]);  //错误处理
//语法分析的函数
    bool Program();             //程序
    bool ConstDeclaration();    //常量声明
    bool ConstDefinition();     //常量定义
    bool VarDeclaration(int *num);      //变量声明
    bool VarDefinition(int *num, int *varnum);      //变量定义
```

```
    bool RtnFuncDefinition();        //有返回值函数定义
    bool NonRtnFuncDefinition();//无返回值函数定义
    bool MainProgram();              //主函数
    bool Args(int *size);                //参数
    bool CompoundState(int *num);        //复合语句
    bool StateList();            //语句列
    bool Statement();            //语句
    bool IfStatement();          //条件语句
    bool Condition(char *r, int iflabel);    //条件
    bool WhileStatement();               //循环语句
    bool CallRtnFunc(char *r);           //有返回值函数调用语句
    bool CallNonRtnFunc(char *r);        //无返回值函数调用语句
    bool ParameterList(int *size);       //值参数列
    bool AssignStatement();      //赋值语句
    bool ReadStatement();        //读语句
    bool WriteStatement();       //写语句
    bool SwitchStatement();      //情况语句
    bool ReturnStatement();      //返回语句

    bool Constant(char *con);    //常量
    bool Integer(int *num);      //整数
    bool Expression(char *r, int *arr); //表达式
    bool Term(char *r, int *arr);        //项
    bool Factor(char *r, int *arr);      //因子
    bool CaseList(char *p, int *switchlabel);    //情况列

//语义分析及生成中间代码
    void genVardef(TabEntry entry);      //生成变量定义
    void genVar(char *r, int i, bool flag);      // r = iflag为真，则i是整数，否则
是字符的ASCII码
```

```
    void genVar(char *r, char c);                // r = c
    void genALU(char *r, char op, char *p);      // r = op p
    void genALU(char *r, char *p, bool arr);            // r = p
    void genALU(char *r, char *p, char op, char *q, TEMP_TYPE ttype);    // r =
p op q  ttype表示是否是字符串运算
    void genLabel(char str[]);          //生成函数标签    _func:
    void genLabel(int i);               //生成标签        _Lx:
    void genGOTOLabel(int i);           //无条件跳转      GOTO _Lx
    void genGOTOLabel(char str[]);      //无条件跳转      GOTO str
    void genCall(char str[]);           //调用无返回值的函数      LCALL _func
    void genCall(char *r, char str[], ENTRY_TYPE etype);         //调用有返回值的
函数     _tx = LCALL _func
    void genAssign(char str1[], char str2[], bool arr); //（最后的）赋值语句
    str1 = str2  or *(str1) = str2
    void genRtn(char *p);               //返回语句    Return _tx
    void genCondition(char *r, char *p, int iflabel);    //  <条件> = <表达式>
    IFZ %s GOTO _L%d
    void genCondition(char *r, char *p, char *op, char *q, int iflabel);//<条
件> = <表达式> <关系运算符> <表达式>
    void genCondition(char *r, char *p, char *op, char *q, char str[]);
    void genPush(std::string p);//入栈
    void genPop(int size);  //出栈
    void genPrintf(char *p);//输出
    void genScanf(char *p); //输入
    void genBeginFunc();    //开始函数
    void genEndFunc();      //函数结束
    void genExit();         //退出程序

    int getnum(char str[]);     //对形如$tx的临时寄存器取得数字x
    int getarrnum(char str[]);  //对形如*($tx)的临时寄存器取得数字x
```

```
};
```

关键算法：

语法分析就是对着语句的文法进行匹配，几乎每个非终结符都有一个函数，读 token 判断是否符合文法，满足语句条件就返回 true 否则会报错返回 false。

语义分析的代码加在语法分析中，然后生成相应的四元式。比如 a = b + c,就会生成四元式 a b + c 。

错误处理也加在语法分析中，读到的 token 若不满足语句的文法则会返回输出相应的错误，具体错误类型见下文“出错处理”。

2.3 中间代码的形式

```
//中间代码，结构如 result arg1 op arg2
typedef struct MidCode
{
    char result[MAX_STR_LEN];    // result
    char arg1[MAX_STR_LEN];      // arg1
    char op[MAX_STR_LEN];        // op
    char arg2[MAX_STR_LEN];      // arg2
    int block;   //划分的基本块的编号
    bool flag;   //是否可优化,$tx中x是否会修改
}MidCode;
```

2.4 符号表的形式

```
//符号表项的类型
enum ENTRY_TYPE
{
```

```
    INT, CHAR, INTARRAY, CHARARRAY, VOID, NOTYPE
};

//符号表项的种类
enum ENTRY_KIND
{
    CONST, VAR, ARGS, RTNFUNCTION, NONRTNFUNCTION, NOKIND
};

//符号表项
typedef struct TabEntry
{
    char name[MAX_STR_LEN];
    ENTRY_KIND ekind;    // 种类const var rtnfunction nonrtnfunction
    ENTRY_TYPE etype;    // 类型int char intarray chararray void
    int declarLine;      // 声明时的行数
    int size;            // 数组的大小，函数形参的个数
    int argsnum;// 函数变量的形参个数
    int varnum;      // 函数变量的个数
    union Value      // 符号表项的值
    {
        int i;
        char c;
        int intarr[MAX_STR_LEN];
        char chararr[MAX_STR_LEN];
    }value;
}TabEntry;

typedef std::vector<TabEntry> TABLE;//定义符号表的结构
```

```
//符号表
class Table{
    TABLE table;//符号表
public:
    bool insertTab(TabEntry entry); // 将entry插入符号表
    bool beDef(char str[]);          //查找符号表看是否已经定义过str    防止重定义
    TabEntry canBeUse(char str[]);  //查找符号表看是否已经定义过str    防止未声明
    TabEntry search(char str[]);//查表找调用的标识符和函数
    void Modify(int size);          //修改函数形参个数
    int getFuncLocation(char str[]);//返回str所在的位置
    TabEntry getLocalVarLocation(int k, char str[]);//返回局部变量str所在的位置
    TabEntry getGlobalVarLocation(char str[]);  //返回全局变量str的entry
    void setGlobalVar();//生成汇编时调用，得到所有全局变量
};
```

2.5 优化结构

```
//节点表中的节点项
typedef struct NodeEntry
{
    char name[MAX_STR_LEN]; //节点名
    int node;       //所在节点
}NodeEntry;

//节点表
typedef struct NodeTable
{
    NodeEntry nentry[MAX_STR_LEN];  //节点项数组
    int size;       //节点表中几个节点项
```

```
//  int nodenum;//DAG图中有几个节点
}NodeTable;

//DAG图中节点结构体
typedef struct Node
{
    int node;   //第几个节点
    char op[MAX_STR_LEN];   //符号 or name
    char repr[MAX_STR_LEN]; //代表该节点的名字
    int reprnum;//标识符数量
    int left;       //左节点
    int right;      //右节点
    bool ready;     //是否已入栈，false为未入栈
}Node;

//DAG图结构
typedef struct DAGTree
{
    Node nodes[MAX_STR_LEN];//节点数组
    NodeTable nodetable;    //节点表
    std::stack<int> nodestack;  //中间节点序列
    int nodenum;            //节点数量
}DAGTree;

//优化类
class Optimizing
{
    DAGTree dagtree;
public:
    Optimizing();   //初始化dagtree.nodenum=0,dagtree.nodetable.size=0
```

```
    void DAGUpdate();                       //DAG更新（初始化）
    int genDAG(int start, int block);       //生成DAG
    void genOpCode();                       //生成优化后代码

    void LocalOptimize();                   //局部优化
    void GlobalOptimize();                  //全局优化

    int FindOrBuild(char str[]);//在nodetable中找str，找到了返回节点i；未找到则在DAG图中新建节点i，节点表中添加(str,i)
    int FindOP(char op[], int i, int j);//在DAG中寻找中间节点op
    void FindResult(char str[], int k); //在节点表中找str，找到了将其节点号更改为k;未找到则在节点表中新建(str,k)

    bool CheckOP();             //检查是否有中间节点未进入队列
    int FindNoParentNode();     //找无父节点的节点
    bool isSon(int i);          //检查节点i是否是其他节点的子节点
};
```

2.6 生成目标代码

```
//生成mips汇编
class MIPS
{
public:
    Parser parser;  //调用语法分析
    MidCode m;  //中间代码m
    int t[10];  //register t0 - t9   1 : invalid  0 : valid
    int i;      //当前读到四元式的标号
```

```
int regnum; //临时寄存器t，初始化为-1
int sregnum;//临时寄存器s，初始化为-1
int nowfuncflag;//当前所在函数在符号表中的位置
int ValidRegister();//取可用t寄存器的标号
int ValidSRegister();    //取可用s寄存器的标号

MIPS(); //构造函数

void EmitTotal(int size);    //遍历四元式，生成mips汇编

void EmitLoadConstant();//常量赋值汇编生成
void EmitLoadString();       //字符串赋值汇编生成
void EmitBeginFunc();        //函数开始标志汇编生成
void EmitPushStack();        //入栈汇编生成
void EmitCallNonRtnFunc();   //调用无返回值函数汇编生成
void EmitCallRtnFunc();      //调用有返回值函数汇编生成
void EmitPopStack();     //出栈汇编生成
void EmitIFZ();              //if判断语句汇编生成
void EmitGoto();         //无条件跳转汇编生成
void EmitReturn();           //返回语句汇编生成
void EmitEndFunc();          //函数结束标志汇编生成
void EmitEndProgram();       //程序结束标志汇编生成
void EmitLabel();            //函数标签、跳转标签汇编生成
void EmitPrintf();           //输出函数汇编生成
void EmitScanf();            //输入函数汇编生成
void EmitIDENAssign();       //标识符赋值汇编生成
void EmitArrayOp();          //数组操作汇编生成

void EmitAdd();      //加法汇编生成
void EmitSub();      //减法汇编生成
```

```
    void EmitMultDiv();      //乘除法汇编生成
    void EmitComparison();   //比较运算汇编生成

    void storereg(int t);    //存寄存器t被使用
    int getnum(char p[]);    //取得形如$tx的临时寄存器的值x
    int getarrnum(char p[]);//取得形如*($tx)的临时寄存器的值x
    void MarkValidReg(int reg);//释放寄存器t(reg)使之可以被再次使用
    void MarkInvalidReg(int reg);//使用寄存器t(reg)，标记其为不可使用
};
```

关键算法：

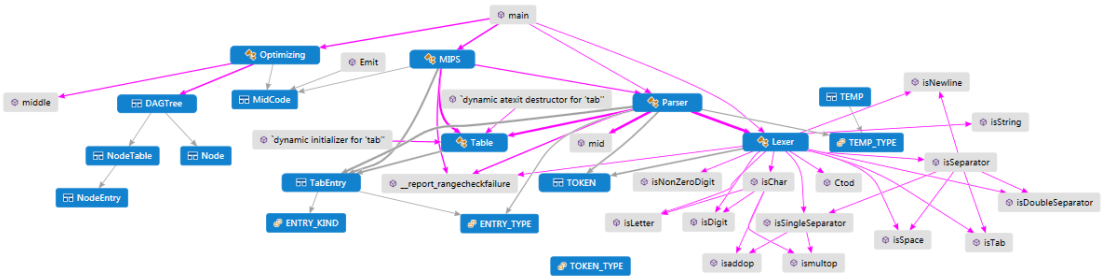
每种类型的四元式都对应了相应的生成汇编的函数，比如 `Printf  (iden | $tx)`就对应了函数 `EmitPrintf()`，`Scanf iden` 对应了函数 `EmitScanf()`。

3. 调用依赖关系

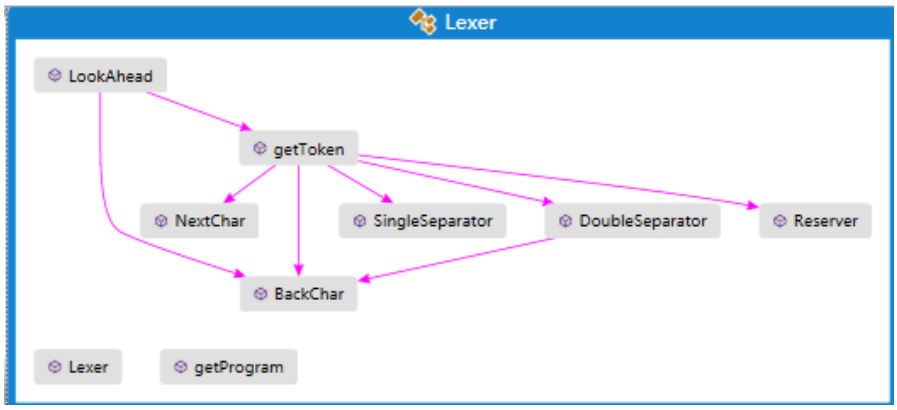
Parser 类调用 Lexer 类，对于词法分析的函数 getToken()每次返回的 token 进行解读和判断，对应各个部分的文法判断所读取的内容是哪个部分，进行语法分析生成四元式，出错部分由错误处理部分进行处理。Optimizing 类对四元式遍历进行优化。MIPS 对四元式进行解读生成 MIPS 汇编。

以下是 VS 自动生成的函数调用关系。

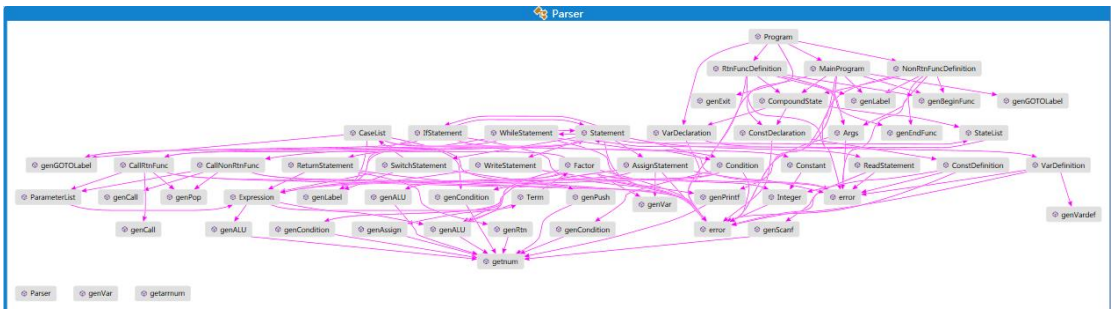
整体调用关系：



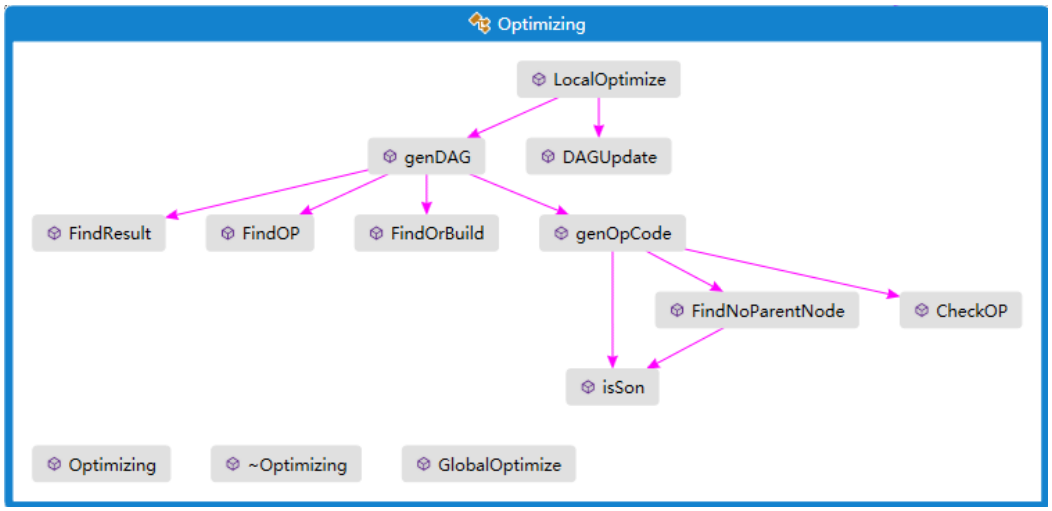
Lexer 类中调用关系：



Parser 类中调用关系：



Optimizing 类中调用关系：



MIPS 类中调用关系：

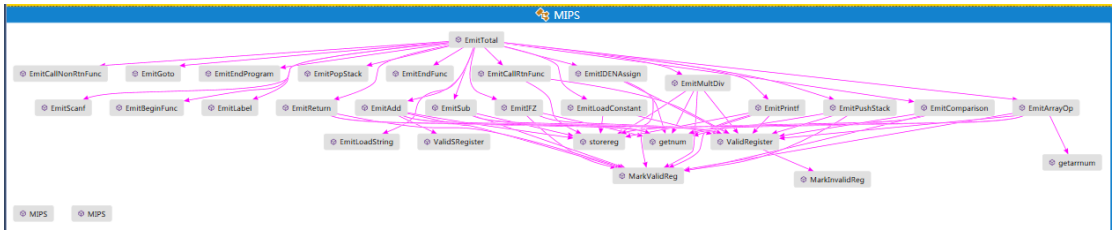
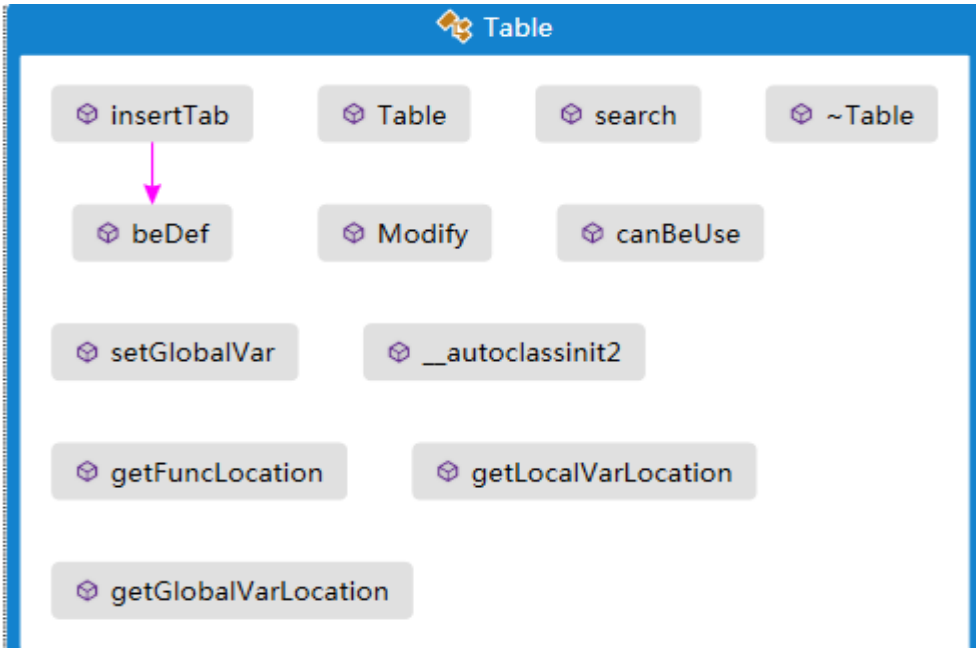


Table 类中的调用关系：



4. 符号表管理方案

```
//符号表项的类型
enum ENTRY_TYPE
{
    INT, CHAR, INTARRAY, CHARARRAY, VOID, NOTYPE
};

//符号表项的种类
enum ENTRY_KIND
{
    CONST, VAR, ARGS, RTNFUNCTION, NONRTNFUNCTION, NOKIND
};

//符号表项
typedef struct TabEntry
{
    char name[MAX_STR_LEN];
    ENTRY_KIND ekind;   // 种类const var rtnfunction nonrtnfunction
    ENTRY_TYPE etype;   // 类型int char intarray chararray void
    int declarLine;     // 声明时的行数
    int size;           // 数组的大小，函数形参的个数
    int argsnum;// 函数变量的形参个数
    int varnum;     // 函数变量的个数
    union Value     // 符号表项的值
    {
        int i;
        char c;
        int intarr[MAX_STR_LEN];
        char chararr[MAX_STR_LEN];
```

```
    }value;
}TabEntry;

typedef std::vector<TabEntry> TABLE;//定义符号表的结构

//符号表
class Table{
    TABLE table;//符号表
public:
    bool insertTab(TabEntry entry); // 将entry插入符号表
    bool beDef(char str[]);          //查找符号表看是否已经定义过str    防止重定义
    TabEntry canBeUse(char str[]);  //查找符号表看是否已经定义过str    防止未声明
    TabEntry search(char str[]);    //查表找调用的标识符和函数
    void Modify(int size);          //修改函数形参个数
    int getFuncLocation(char str[]);//返回str所在的位置
    TabEntry getLocalVarLocation(int k, char str[]);//返回局部变量str所在的表项
    TabEntry getGlobalVarLocation(char str[]);  //返回全局变量str的entry
    void setGlobalVar();    //生成汇编时调用，得到所有全局变量
};
```

管理算法：

符号表在语法分析时开始使用，符号表项包括名字、类型、种类、声明时的行数、数组大小或函数形参个数、函数形参排序和函数变量排序(只针对函数相关变量)、值等，读到声明新标识符则存入符号表，存入时检查符号表是否已经存在该标识符(全局常量、变量和函数检测全局，函数内声明检测函数内部分)，若存在则报错，若不存在则加入符号表，使用到的函数是 `insertTab(TabEntry entry)`和 `beDef(char str[])`。赋值语句等使用标识符时要检测符号表中是否已经存在该标识符，存在才可使用，否则报错。使用到的函数是 `canBeUse(char str[])`和 `search(char str[])`。(具体函数作用请参照注释)

`Modify(int size)`是修改函数名对应符号表项的 `size` 的修改以及函数形参 `argsnum` 的修改，因为函数名刚存入符号表时还不知道形参的个数。

`getFuncLocation(char str[])`是返回符号表中函数 `str` 所在的位置。

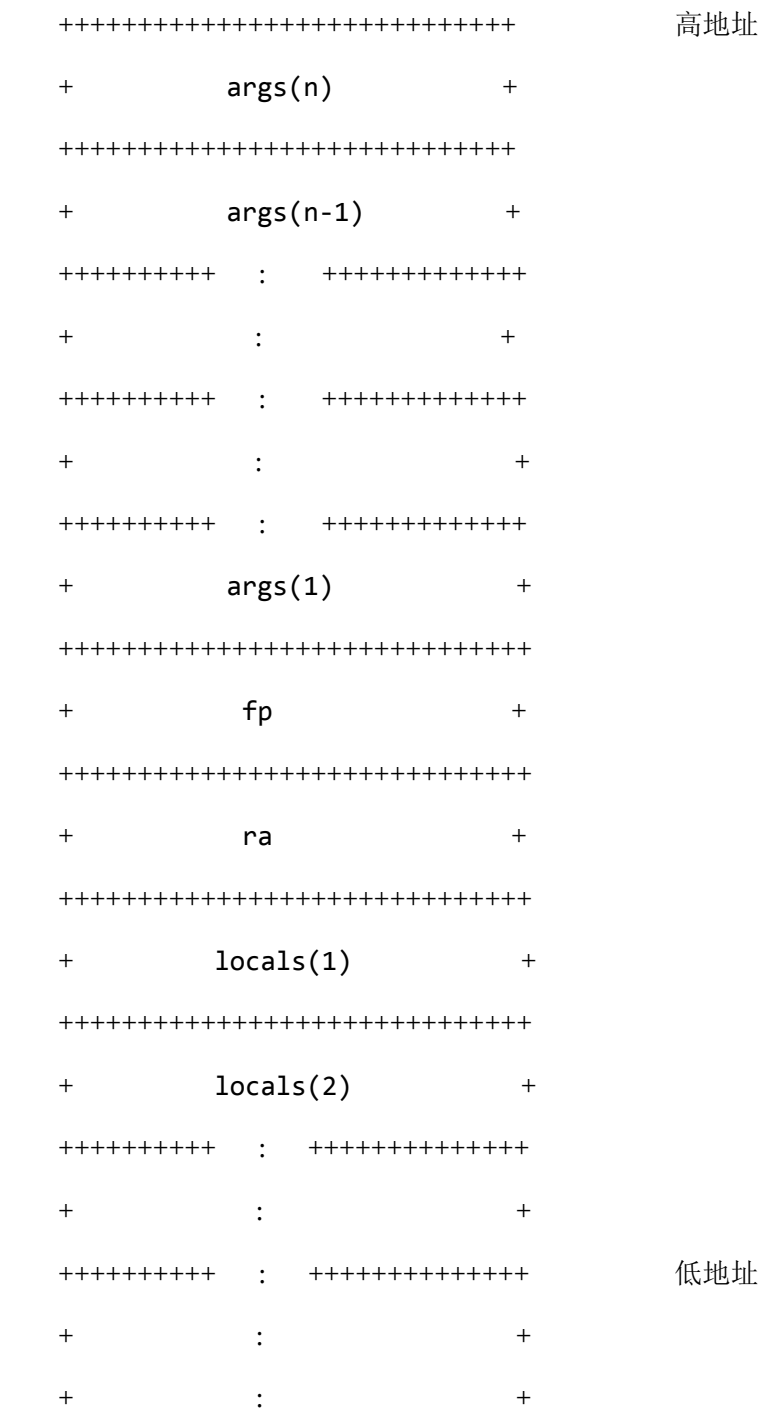
`getLocalVarLocation(int k, char str[])`返回局部变量 `str` 的符号表项。

`getGlobalVarLocation(char str[])`返回全局变量 `str` 的符号表项。

`setGlobalVar()`生成 `mips` 汇编时生成所有全局变量。

5．存储分配方案

运行栈结构：



如运行栈所示，函数的形参存入栈指针$fp 上方(通过符号表项中 argsnum 寻找)，函数局部变量存入栈指针$fp 下方(通过符号表项中 varnum 寻找)，寻址时若是形参则$fp+argsnum*4；若是局部变量则$fp-(varnum+1)*4。通过 lw 和 sw 可存取运行栈中的值。

6. 四元式设计*

采用的四元式结构形如 result arg1 op arg2

中间代码四元式会出现的形式：

```
_funcname:                  //函数名标签
$Lx:                        //跳转标签
EndFunc                     //函数结束
EndProgram                  //程序结束
BeginFunc   num             //函数开始,num 表示函数局部变量所占空间大小
PushStack  iden($tx)        //iden 或$tx 入栈
LCALL _func                 //调用无返回值函数
PopStack num                //空间大小 num 出栈
GOTO  $Lx(str)              //无条件跳转到$Lx 或 str
Return  $tx(iden)           //返回$tx 或 iden
VAR  iden                   //声明变量 iden
Printf   iden($str  $tx)    //输出 iden  |  $str  |  $tx
Scanf   iden                //输入 iden
IDEN $t(IDEN num)           //给标识符 iden 赋值
$t *($t)                    //将字符串的某个值赋给$t
*($t) $t(num)               //将某个值存入字符串
$tx LCALL _func             //调用有返回值函数
Result arg1 op arg2         //加减乘除运算、关系运算符运算
IFZ ($tx | iden)  GOTO  $Lx   //判断，如果($tx | iden)不成立则跳到$Lx
```

7. 优化方案*

```
//节点表中的节点项
typedef struct NodeEntry
{
    char name[MAX_STR_LEN]; //节点名
    int node;               //所在节点
}NodeEntry;

//节点表
typedef struct NodeTable
{
    NodeEntry nentry[MAX_STR_LEN];  //节点项数组
    int size;                       //节点表中几个节点项
}NodeTable;

//DAG图中节点结构体
typedef struct Node
{
    int node;               //第几个节点
    char op[MAX_STR_LEN];   //符号 or name
    char repr[MAX_STR_LEN]; //代表该节点的名字
    int reprnum;            //标识符数量
    int left;               //左节点
    int right;              //右节点
    bool ready;             //是否已入栈，false为未入栈
}Node;

//DAG图结构
typedef struct DAGTree
```

```
{
    Node nodes[MAX_STR_LEN];     //节点数组
    NodeTable nodetable;         //节点表
    std::stack<int> nodestack;   //中间节点序列
    int nodenum;                 //节点数量
}DAGTree;

class Optimizing
{
    DAGTree dagtree;
public:
    Optimizing();   //初始化dagtree.nodenum=0,dagtree.nodetable.size=0

    void DAGUpdate();                       //DAG更新（初始化）
    int genDAG(int start, int block);       //生成DAG
    void genOpCode();                       //生成优化后代码

    void kuikong();                         //窥孔优化
    void LocalOptimize();                   //局部优化
    void GlobalOptimize();                  //全局优化

    int FindOrBuild(char str[]);//在nodetable中找str，找到了返回节点i；未找到则在DAG图中新建节点i，节点表中添加(str,i)
    int FindOP(char op[], int i, int j);//在DAG中寻找中间节点op
    void FindResult(char str[], int k); //在节点表中找str，找到了将其节点号更改为k;未找到则在节点表中新建(str,k)

    bool CheckOP();             //检查是否有中间节点未进入队列
    int FindNoParentNode();     //找无父节点的节点
    bool isSon(int i);          //检查节点i是否是其他节点的子节点
```

};

LocalOptimize()做代码的局部优化，主要是进行窥孔优化和构建DAG图删除局部公共子表达式，生成DAG图前前对DAG图进行初始化DAGUpdate()，然后调用genDAG(int start, int block)生成DAG图，函数中调用的函数有FindOrBuild(char str[])在nodetable中找str，找到了返回节点i；未找到则在DAG图中新建节点i，节点表中添加(str,i)、FindOP(char op[], int i, int j)在DAG中寻找中间节点op、FindResult(char str[], int k)在节点表中找str，找到了将其节点号更改为k;未找到则在节点表中新建(str,k)。构造DAG图以后生成优化后的代码，使用从DAG图导出中间代码的启发式算法，用到的函数有CheckOP()检查是否有中间节点未进入队列、FindNoParentNode()找无父节点的节点、isSon(int i)检查节点i是否是其他节点的子节点。删除局部公共子表达式以后进行窥孔优化kuikong()。

代码生成时合理利用临时寄存器，临时寄存器采用用完即释放的原则，使得寄存器高效利用。

8. 出错处理

错误信息处理表:

编号	具体信息	处理	
101	常变量定义缺少;	找下一个; flag = false	
102	非常变量定义缺少;	flag=false	
201	常量变量定义缺少 int char	找下一个; return false	类型标识缺失
202	函数参数声明缺少 int char	找下一个，或) flag = false	
301	IDEN 缺失 常变量定义 读语句	找下一个; return false	IDEN 缺失
302	IDEN 缺失 参数	找下一个，) flag = false	
401	Redefine 重定义	报错	重定义和未定义
402	调用函数未定义	报错	
403	调用标识符未定义	报错	
501	Assign = 缺失	找下一个; return false	关系运算符缺失
502	条件语句 关系运算符 缺失	找下一个) return false	
503	Colon : 缺失	报错	
601	非整数	找下一个; return false	数字字符不符合
602	非字符	找下一个; return false	
603	非无符号整数	找下一个，或; flag = false	

70201	非] 数组定义错误	找下一个，或; flag = false	括号缺失
70202	数组赋值错误	找下一个; return false	
70203	因子：标识符[表达式]	直接读 return false	
70301	(缺失	函数定义、调用(不可能)	703： (缺失
70302	条件语句 循环语句 情况语句	直接读 flag = false	
70303	读写语句	找下一个; return false	
70401	) 缺失 函数定义 情况语句 Main 函数	找下一个) 或 { flag = false	704：) 缺失
70402	条件语句 循环语句 因子	直接读 flag = false 直接读 return false	
70403	函数调用 读写语句 返回语句	找; flag= false 找; return false	
705	{ 缺失 函数定义 main 情况语句	直接读 flag = false	{ 缺失
706	} 缺失	报错	}缺失
801	Case 保留字缺失	找 case | }	
901	函数参数量不对	flag = false	
902	写语句无参量	flag = false	
903	数组越界	flag = false	
904	给 const 常量赋值	flag = false	
1000	Const 应该是 var 之前	报错	
1001	无主函数	报错	
1002	多个主函数	报错	

1003	主函数之后没有语句	报错	

三．操作说明

1．运行环境

使用 Visual Studio 2013 编写程序，最后在 Visual Studio2010 上也可以跑通

2．操作步骤

运行程序，输入程序的文件名，会告诉你文件是否成功打开，若未成功则文件不存在；若成功打开会提示是否有错误，若有错误则不会生成四元式和汇编代码，错误信息在 errfile.txt 中；若没有错误则随意输入一个符号然后回车退出运行，四元式在 midfile.txt 中，汇编代码在 finalfile.txt 中。

四．测试报告

1．测试程序及测试结果

【给出提供的测试程序以及每个程序的测试结果，无需截屏】

1.1 测试 1：

```
int str[10];

int get(int a,int b){
	int temp;
	printf(a);
	printf(b);
	temp =a;
	a = b;
	b = temp;
	printf(a);
	printf(b);
	return (1);
}

void main()
{
	int a,b,c,d;
	scanf(a);
	printf(a);
	b=2;
	c=4;
	d = get(b,c) + b *(-c);
	a = c + b*(-c);
	printf(d);
	printf(a);
	str[0] = 2;
	str[1] = 3;
	str[2] = str[1] + str[0];
	printf("str[2] is ",str[2]);
}
```

输入：1

输出结果：1 2 4 4 2 -7 -4 str[2] is 5

1.2 测试 2

```
void main(){
    int c;
    c = a+b;
    if (0){

        c = a +a+a;
        b = a +97;
    }
    else{
        b = 'c';
    }

    if (1<2){

        c = a +a+a;
        b = a +97;
    }
    else{
        b = 'c';
    }
    printf(b);
    printf(c);
}
```

输出结果：99 100 98 3

1.3 测试 3：

```
const int a=1;
int b;
void main(){
    int c;
    b = 'c';

    switch (a){
        case 1 : c = 2;
        case 2 : c=3 ;
    }

    printf(b);
```

```
    printf(c);

}
```
输出结果：99 2

1.4 测试 4：

```
int str[10];

int get(int a,int b){

    return (a+b);
}

void foo(){
    int c ;
    c = 1+2;
    printf(c);
}

void main()
{
int a,b,c,d;


    b=2;
    c=4;
    printf(get(b,c));
    foo();
}
```

输出结果：6 3

1.5 测试 5

```
int str[4];

int get(int a,int b){

    return (a+b);
}
```

```
void main()
{
	int a;
	a =1;
	str[0]=1;
	str[1]=2;
	str[2]=str[0]+str[1];
	str[3]=a;
	printf(str[3]);

}
```

输出 ： 1

1.6 测试 6

```
int str[4];

int get(int a,int b){

	return (a+b);
}


void main()
{
	int a;
	a =1;
	str[0]=1;
	str[1]=2;
	str[2]=str[0]+str[1];
	str[5]=a;
	printf(str[3]);

}
```

输出： array overflow

1.7 测试 7

```
const float const1 = 0.1, const2 = -100;
const char const3 = '_';
int change1;
float change2;
char change3;
int gets1(int var1,int var2){
	change1 = var1 + var2;
	return (change1);
}
```

文法中没有 float，故 float 被识别为标识符，0.1 被分割成 0 和 1，‘.’为不能识别的字符，报错。没有 main 函数。

1.8 测试 8

```
const int const1 = 0, const2 = -100;
const char const3 = '_a';
int change1;
float change2;
char change3;
int gets1(int var1,int var2){
	change1 = var1 + var2;
	return (change1);
}
```

字符错误，没有 main 函数

1.9 测试 9

```
const int const1 = 0, const2 = -100;
const char const3 = '&';
int change1;
float change2;
char change3;
int gets1(int var1,int var2){
	change1 = var1 + var2;
	return (change1);
}
```

缺少一个字符，无 main 函数

1.10 测试 10

```
void main(){
    const int const1 = 0, const2 = -100;
    int change1
}
```

少了一个分号

1.11 测试 11

```
int foo(){
    int x = 0;
    x = x + 1;
```

少一个;、少了一个}，没有 main 函数

2．测试结果分析

【说明测试程序对语法成分的覆盖情况】

测试 1 覆盖了赋值语句、输入输出、函数调用

测试 2 覆盖了赋值语句、条件语句

测试 3 覆盖了赋值语句、switch 语句

测试 4 覆盖了赋值语句函数调用语句，返回语句

测试 5 覆盖了数组的运算

测试 6 覆盖了数组越界的判断

五．总结感想

在这学期开始之前就对编译实验的难度略有耳闻，所有学长学姐都认为这是最难的一门课，让我在课设开始之前惶恐不已，听闻要经常熬夜我的内心几乎是崩溃的。但是课设开始选题的时候我又犹豫了，是选中级还是高级，因为听说很难怕自己做不出优化，犹豫再三还是选择了高级，挑战一下自己咯。于是现在我拖着疲惫的身躯在写文档，怎么说呢，也还是有很大收获的吧。毕竟学计算机从来没有写过这么长的代码，编写“较大”工程时对于各个模块之间的结构要设计好，知道他们之间的联系，在这方面还是学到了不少。先思考设计然后再写，不然写出来的东西又乱也不好调试，白白浪费时间。

这学期还是事情太多了，编译作业的战线拉得太长，中间因为别的事情停了好一段时间，然后重新开始写的时候发现已经忘了前面写的东西了，所以说写代码还是一次性写完比较好，这样对整体的把握都有了，像这次我前面写符号表用向量结构，后来写节点表时忘记了向量只好用数组，不是说数组不好，只是明明同样的结构用两种表示方法让强迫症有点不好受，并且向量有些函数而数组没有，写到后面简直要崩溃了但是还好我坚持写下来了。

其实编译课设并没有那么难，分成模块写其实都很好懂，但是主要就是麻烦，要思考很多，一些结构要考虑好，一块块的写其实都能写出来的。最怕的是未战先退缩，恩，还好我没有退缩。接下来也要继续努力了！