课 程 设 计 报 告

设计题目：简单C语言集成开发环境的设计与实现

班　　级：计算机1308班

组长学号：20134017

组长姓名：刘皓冰

指导教师：张俐

设计时间：2015 年 12 月

设计分工

组长学号及姓名：20134017 刘皓冰

分工：文法设计；符号表数组表等结构设计；主体框架实现；算数表达式、赋值语句、变量定义语句语法语义分析；填表；语法分析的出错处理；划分基本块；对数组全面支持；可视化

组员 1 学号及姓名：20134008 安泊舟

分工：词法分析；常数处理；符号表数组表等结构修改完善；对注释支持

组员 2 学号及姓名：20133647 纪钰婷

分工：帮助设计文法；if 语句和 while 语句语法语义分析；生成可直接在汇编环境下运行的目标代码，并对目标代码进行了优化

组员 3 学号及姓名：20134028 袁堃皓

分工：中间代码分块 DAG 优化；符号表动态维护——删除无用中间变量，更新符号表 addr（即地址）内容

摘　要

编译原理是计算机专业的一门重要专业课，旨在介绍编译程序构造的一般原理和基本方法。内容包括语言和文法、词法分析、语法分析、语法制导翻译、中间代码生成、存储管理、代码优化和目标代码生成。 编译原理是计算机专业设置的一门重要的专业课程。虽然只有少数人从事编译方面的工作，但是这门课在理论、技术、方法上都对学生提供了系统而有效的训练，有利于提高软件人员的素质和能力。

本次课设我们分工明确，时时交流沟通，由小做大，设计实现了一个简单文法的编译器的前端加后端并最后做成了简单文法的集成开发环境。我们虽然分工不同，但我们每个人都从本次课设中了解到了编译程序工作的全过程，对编译程序工作过程的各阶段的认识有了质的飞跃。

关键词：编译程序，简单文法，前端，后端，集成开发环境

目　录

1 概述

编译程序的工作过程一般可以分为五个阶段：词法分析、语法分析、语义分析与中间代码产生、优化、目标代码生成。每一个阶段在功能上是相对独立的，它一方面从上一个阶段获取分析的结果来进行分析，另一方面由将结果传递给下一个阶段。由编译程序的五个阶段就对应了编译系统的结构。

其中词法分析器利用超前搜索、状态转换等方法，将源程序转化成为一个一个的单词符号二元式。一般程序语言的单词符号包括标识符、常量、关键字、运算符和界符。我们设计的词法分析器添加对注释的支持，会将源程序中的注释自动滤掉。能够拼出语言中的各个单词，会将拼出的标识符填入符号表项的 name 处，当然若是重复会只填一次。返回识别单词或符号的种族编码。

语法分析器将这些单词符号作为输入，对它进行语法分析。语法分析分为两种方法：自上而下分析法和自下而上分析法。针对不同程序语言的语法规则可以采取不同的分析方法，当然两种方法也可以同时使用。我们本次课设由于时间紧迫只采用了自上而下分析法之一的递归下降子程序法。语法分析完成后，可以确定源程序是否是由简单文法推出的正确程序。若源程序不是正确的程序，那么语法分析出错处理就会运转，给出错误所在，比如说“while 语句错误!”。然后调用 exit(1);程序退出不再继续运行。通过运行集成开发环境对源程序进行编辑、保存后或许错误便会解决再次编译不会报错。

语法分析器把语法单元作为输入供语义分析器使用。一般的语义分析

器主要采用的是语法制导方法，即在语法分析的同时进行语义分析，并产生一定的语义动作，来生成中间代码。

代码优化是将语义分析生成的中间代码进行优化，产生执行效率更高的代码。我们采用 DAG 优化，划分完基本块后优化器读入基本块入出口地址，根据块数初始化多个 DAG 树，分块进行优化（赋值语句、基本算数运算、数组下标赋值运算、if、while 等语句），每优化完成一个基本块，查找符号表，动态删除无用中间变量，更新符号表 addr 内容，优化完所有块之后，保存优化后的四元式序列用于下一步生成目标代码。

目标代码生成器将优化后的四元式转化为可直接在汇编环境下运行的目标代码。首先通过访问符号表，给变量分配空间，生成汇编的数据段代码。再调用一系列 OBJ 生成函数来生成汇编的代码段代码。在生成汇编代码的过程中，针对特殊情况，对汇编代码进行了优化，减少不必要、重复的语句，并检查优化结果是否正确。最后为验证其是否为正规的汇编语言，将生成的代码写入 asm 文件，并在汇编环境下直接运行它。

通过本次课设，我们一方面巩固验证了课堂上所学知识，把课堂上没能理解透的知识点比如说符号表不但弄懂而且实现了出来。很好地加深了而对所学编译知识的掌握和理解，也深刻的理解了编译器的工作过程和实现方法，明白了怎么从高级语言源程序到机器能够直接运行的目标代码。通过本次课设，我们一方面还明白了怎么更好地合作做一个较为大型的项目，接口问题和进度问题该如何妥善处理。我们另一方面还加强了运用 C++编程的能力和更好的调试能力。不过由于时间问题，要在规定的时间内做成一个我们较为满意的作品不太可能，所以本编译器不可避免可能存在一些问题，但作为一个具有基本功能的、可扩充的系统，完全达到了巩固编译原理的理论知识，并将其运用于实践的目的。

2 课程设计任务及要求

2.1 设计任务

1、定义一个简单程序设计语言文法（包括变量说明语句、算术运算表达式、赋值语句；扩展包括逻辑运算表达式、if 语句、while 语句等）；

2、扫描器设计实现；

3、语法分析器设计实现；

4、中间代码设计；

5、中间代码生成器设计实现；

7、中间代码优化；

8、生成目标代码。

2.2 设计要求

1、在深入理解编译原理基本原理的基础上，对于选定的题目，以小组为单位，先确定设计方案；

2、设计系统的数据结构和程序结构，设计每个模块的处理流程。要求设计合理；

3、编程序实现系统，要求实现可视化的运行界面，界面应清楚地反映出系统的运行结果；

4、确定测试方案，选择测试用例，对系统进行测试；

5、运行系统并要通过验收，讲解运行结果，说明系统的特色和创新之处，并回答指导教师的提问；

2.3 总体设计方案

在清楚课程设计任务要求后，我和纪钰婷制定出一套简单文法：

<A> -> void main ( ) { <B> }

<B> -> <definsen> <B1> | 空

<B1> -> <eqsen> <B1> | <ifsen> <B1> | <whilesen> <B1> | 空

<definsen> -> int 标识符 <definsen1> | int 标识符 [常数] <definsen1>

<definsen1> -> , 标识符 <definsen1> | , 标识符 [常数] <definsen1> | ;

<eqsen> -> 标识符 = <E> ; | 标识符 [常数] = <E> ;

<E> -> <T> {加减算符 <T>}//花括号代表 0 个或多个

<T> -> <F> {乘除算符 <F>}//花括号代表 0 个或多个

<F> -> 标识符 | 常数 | 标识符 [常数] | (<E>)

<ifsen> -> if (<E> 关系算符 <E>) { <B1> } [else { <B1> }]/*方括号表示可有可无*/

<whilesen> -> while (<E> 关系算符 <E>) { <B1> }

注：<A>代表简单 C 语言源程序；<B>代表程序体；<definsen>代表变量定义语句；<eqsen>代表赋值语句；<ifsen>代表 if-else 语句；<whilesen>代表 while 语句；<E>代表算算术表达式；<T>代表项；<F>代表因子。尖括号括起来的都是非终结符，没括的都是终结符。int 我们定的是两个字节，变量声明语句有的话在程序体开头占一句。变量避免以字母 T 开头因为中间变量从 T1 开始。

我和安泊舟制定了各种数据结构:

```
class TokenList
{
public:
    int TYPCOD;
    int pp; //关键字、界符为-1，非-1指向符号表或常量表
……
}TokenList[MAX]; //TOKEN 数组

int pToken = 0;//pToken++;相当于 NEXT(w);


vector<float> cons; //常数表，词法分析结束已经填好


typedef struct SMBLTABLitem {
    string name;
    int type; //int为-1，非-1指向整形一维数组表
    int addr;
    ……
}SMBLTABLi;//标识符表项或说符号表项
vector<SMBLTABLi> SMBLTABL; /*标识符表，词法分析结束已经填好 name 项。数组元素显示以及中间变量开辟直接用 SMBLTABL.push_back(SMBLTABLitem);堆到后面*/
```

```
struct AINFLitem {
    int low;
    int up;
    int ctp;
    int clen;
}AINFL[100];//数组表
int pAINFL = 0; //需要我一个个来填，由于没用 vector 动态数组，只能靠全局变量


struct Quat {
    string op;      //操作符
    string op1;     //操作数1
    string op2;     //操作数2
    string result;  //结果
}QT[50],NEWQT[30]; //四元式数组
int pQT = 0;        //QT个数
int NpQT = 0;       //NEWQT个数
```

3 算法及数据结构

3.1 算法的总体思想

下图给出了编译系统的结构框图：

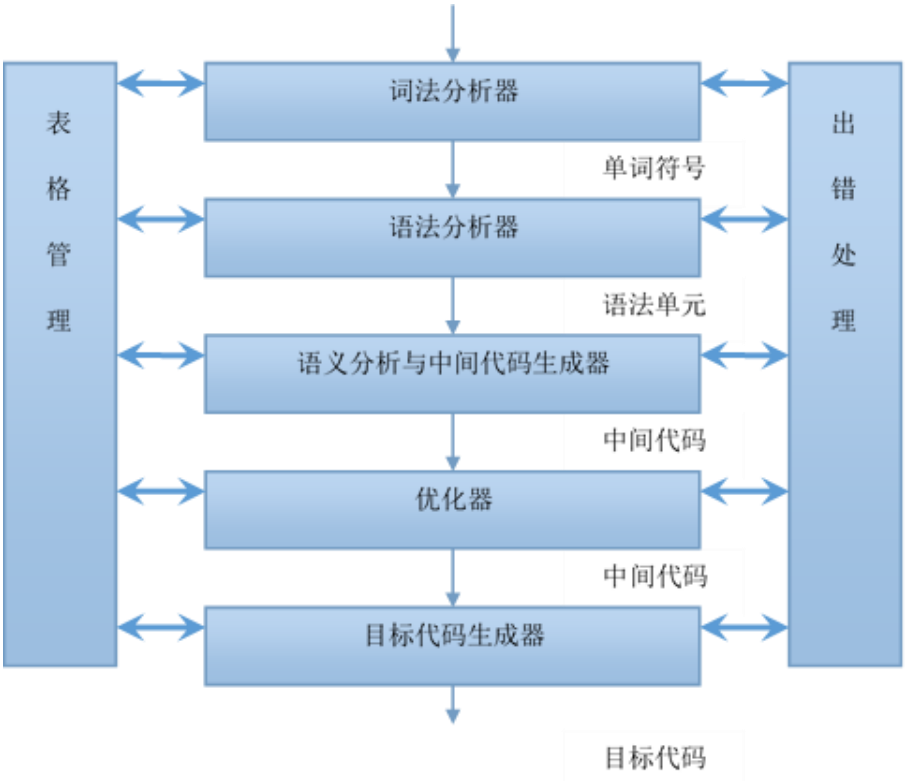


图 3.1-1 编译系统的结构框图

下面是我设计的简单 C 语言编译程序的总流程图或者说是 main()函数的结构图：

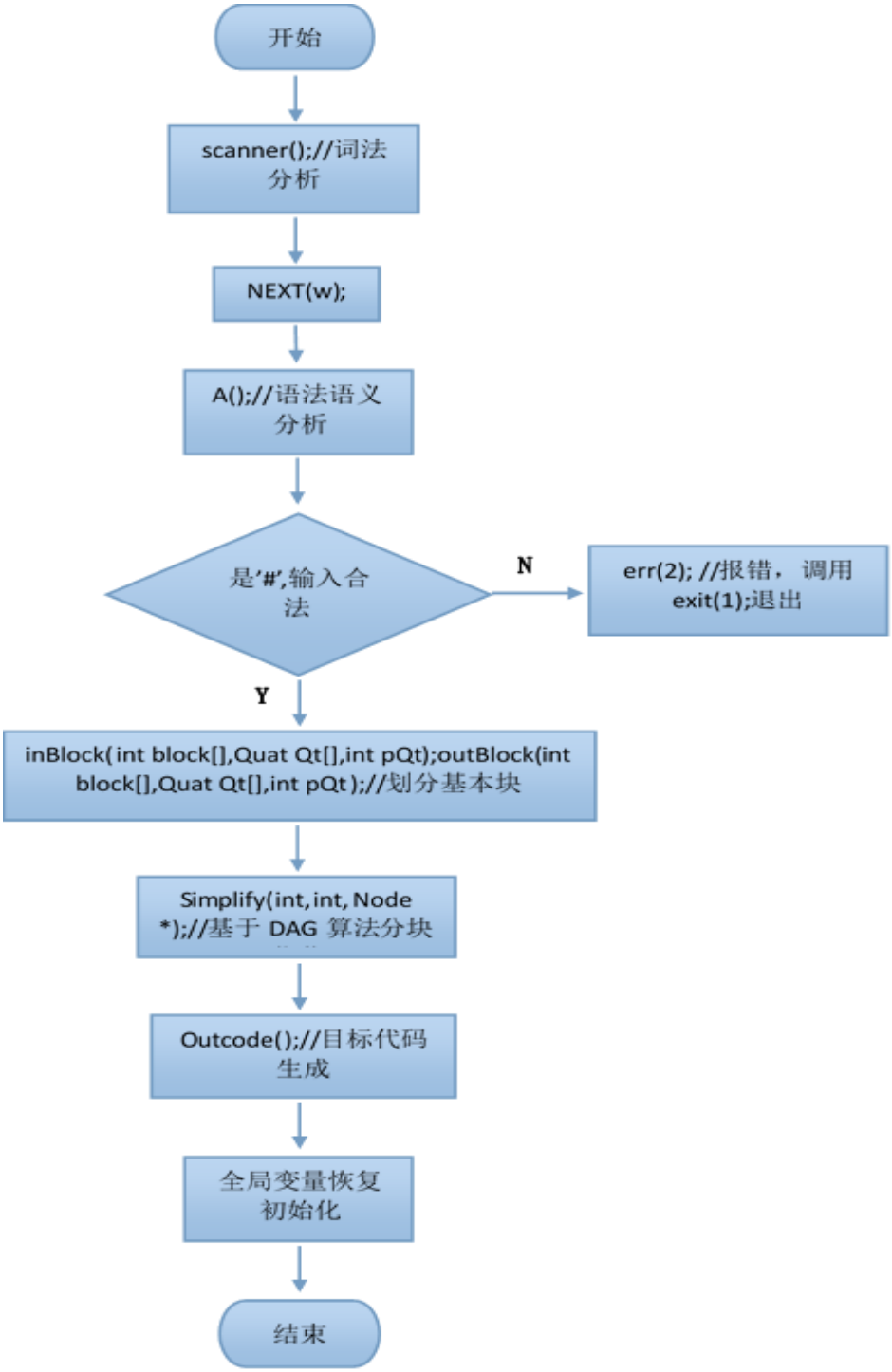


图 3.1-2 编译程序的总流程图

3.2 词法分析器模块

3.2.1 功能

利用超前搜索、自动机等方法，将源程序转化成为一个一个的单词符号二元式。单词符号的种类一般可分为下列 4 种。

1、关键字　是由程序语言定义的具有固定意义的标识符，又称保留字。

2、标识符　用来标示各种名字，如变量名，数组名，函数名等。

3、常量　程序中的数值或字符串。

4、界符　程序中用来分割的符号。

我们设计的词法分析器添加对注释的支持，会将源程序中的注释自动滤掉。能够拼出语言中的各个单词，会将拼出的标识符填入符号表项的 name 处，当然若是重复会只填一次。返回识别单词或符号的种族编码。

能够识别的关键字有：int, main, void, if, else, float, while, bool, char, for,

能够识别的界符有：">=", "<=", "==", "!=", "=", ">", "<", "+", "-","*", "/", "{", "}", ",", ";", "(", ")","[", "]", ".", "!", "&&", "||", "/*", "*/", "//"

1、标识符、常量、关键字表（部分）

标识符	0
常量	3
int	4
main	5
void	6
if	7

else	8
float	9
while	10
char	12
for	13

2、界符表

>=	51
<=	52
==	53
!=	54
=	55
>	56
<	57
+	58
-	59
*	60
/	61
{	62
}	63
,	64
;	65
(	66
)	67
[	68
]	69
.	70
#	71
!	72
&&	73
||	74

3.2.2 数据结构

```
string keywords[MAX] = {"int", "main", "void", "if", "else",
```

```
"float", "while", "bool", "char"};
int length_of_keywords = 9;
/*界符表。*/


string definitions[MAX] = {">=", "<=", "==", "!=", "=", ">",
"<", "+", "-","*", "/", "{", "}", ",", ";", "(", ")","[", "]",
".", "!", "&&", "||", "/*", "*/", "//" };


vector<float> cons;  //常数表
int length_of_cons = 0, length_of_signal = 0;


typedef struct SMBLTABLitem{
    string name;
    int type;
    int addr;
    enumcat cat;
    bool operator == ( string i );//重载==
}SMBLTABLi;//标识符表


class TokenType
```

```
{
public:
    int TYPCOD;
    int pp;

    int getpp() { return pp;}
     int getTYPCOD() { return TYPCOD;}

     void mkTYPCOD(int typecode) { TYPCOD = typecode;}

    void mkpp(string itoken)
    {
        unsigned int j;
        if (TYPCOD == 0)
        {
             for (j=0; j<SMBLTABL.size(); j++)
            {
                if (SMBLTABL[j].name == itoken)
                {
                    pp = j;
                    return;
                }
            }
            SMBLTABLitem.name = itoken;
```

```
    pp = SMBLTABL.size();
    SMBLTABL.push_back(SMBLTABLitem);
    return;
}
else if (TYPCOD == 3)
{
    int x;

    for (j=0; j< cons.size(); j++)
    {
        x = cons[j] - atof(itoken.c_str());
        if (x<=EPSINON && x>=EPSINON*(-1))
        {
            pp = j;
            return;
        }
    }
    pp = cons.size();
    cons.push_back(atof(itoken.c_str()));
    return;
}
else if (itoken == "#")
{
    pp = -2;
```

```
        }
        else
            pp = -1;
    }
}Token[MAX];


class TokenList
{
public:
    int TYPCOD;
    int pp;

     int getpp() { return pp;}
     int getTYPCOD() { return TYPCOD;}

     void mkTYPCOD(int code) { TYPCOD = code;}

     void mkpp(string itoken)
     {
        unsigned int j;
        float x;
        if (TYPCOD == 0)
        {
```

```
      for (j=0; j<SMBLTABL.size(); j++)
    {
        if (SMBLTABL[j].name == itoken)
        {
            pp = j;
            return;
        }
    }
    return;
}
else if (TYPCOD == 3)
{
    for (j=0; j< cons.size(); j++)
    {
        x = atof(itoken.c_str()) - cons[j];
        if (x<=EPSINON && x>=EPSINON*(-1))
        {
            pp = j;
            return;
        }
    }
}
else if (itoken == "#")
{
```

```
            pp = -2;
        }
        else
            pp = -1;
    }
}TokenList[MAX];
int pToken=0;
```

3.2.3 算法

词法分析用到的确定有限自动机如下图:

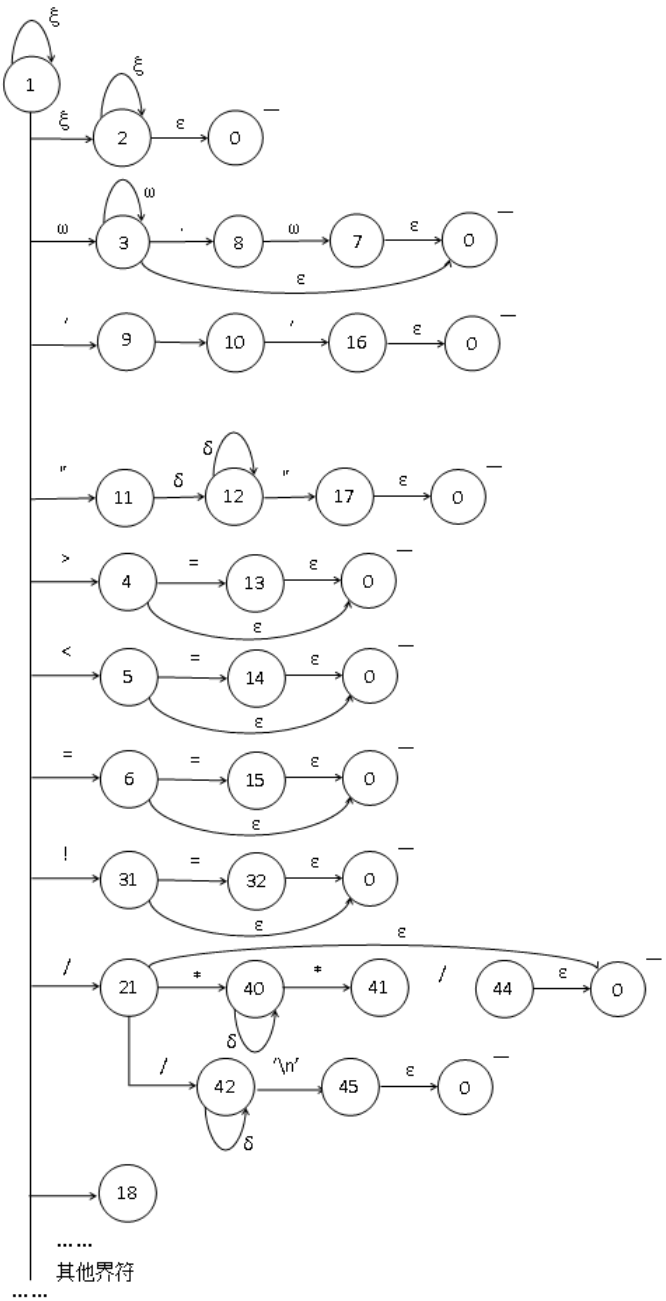


图 3.2-1 确定有限自动机图

3.3 语法语义分析与中间代码生成器模块

3.3.1 功能

语法分析是编译过程的核心部分。它的任务是在词法分析识别出单词符号串的基础上，分析并判定源程序的语法结构是否符合语法规则。源程序的语法结构由上下文无关文法描述。

目前，已存在许多语法分析的方法。但就产生语法树的方向而言，可大致把他们分为自上而下和自下而上两大类。我们本次课设由于时间紧迫只采用了自上而下分析法之一的递归下降子程序法，因其优点是简单、直观，易于构造分析程序。当然，我们事先已把所设计的文法做成了 LL(1)文法。

自上而下的分析方法就是从开始符号出发，采用推导运算，试图自顶向下构造语法树，通常采用“最左推导法”。语法分析的核心技术是“文法”的机内表示问题，递归下降子程序法直接把文法变成程序。其具体的设计原理为——对每一个非终结符，构造一个子程序，用以识别该非终结符所定义的符号串。每个子程序以产生式左部非终结符命名，以产生式右部构造子程序内容。

语义分析主要采用的是语法制导方法，即在语法分析的同时进行语义分析，并产生一定的语义动作，来生成中间代码。

中间代码是高级程序语言中，各种语法成分的语义结构表示，它介于源语言和目标语言之间。中间代码设置的目的是便于编译的后期处理（如与机器无关的优化和目标代码生成）。中间代码有多种形式，常见的有逆波兰记号、四元式和三元式。我们采用的是四元式形式。

3.3.2 数据结构

这部分数据结构除了用到上一部分的 TOKEN 数组、常数表、符号表外，还有：

```
struct AINFLitem {
    int low;
    int up;
    int ctp;
    int clen;
}AINFL[100];//数组表
int pAINFL = 0; //需要我一个个来填，由于没用 vector 动态数组，只能靠全局变量


stack<string>SEM;//语义栈


struct Quat {
    string op;      //操作符
    string op1;     //操作数1
    string op2;     //操作数2
    string result;  //结果
}QT[50]; //四元式数组
int pQT = 0;        //QT个数
```

3.3.3 算法

主控程序为 main()函数，详情见图 3.1-2。

A()子程序流程图：

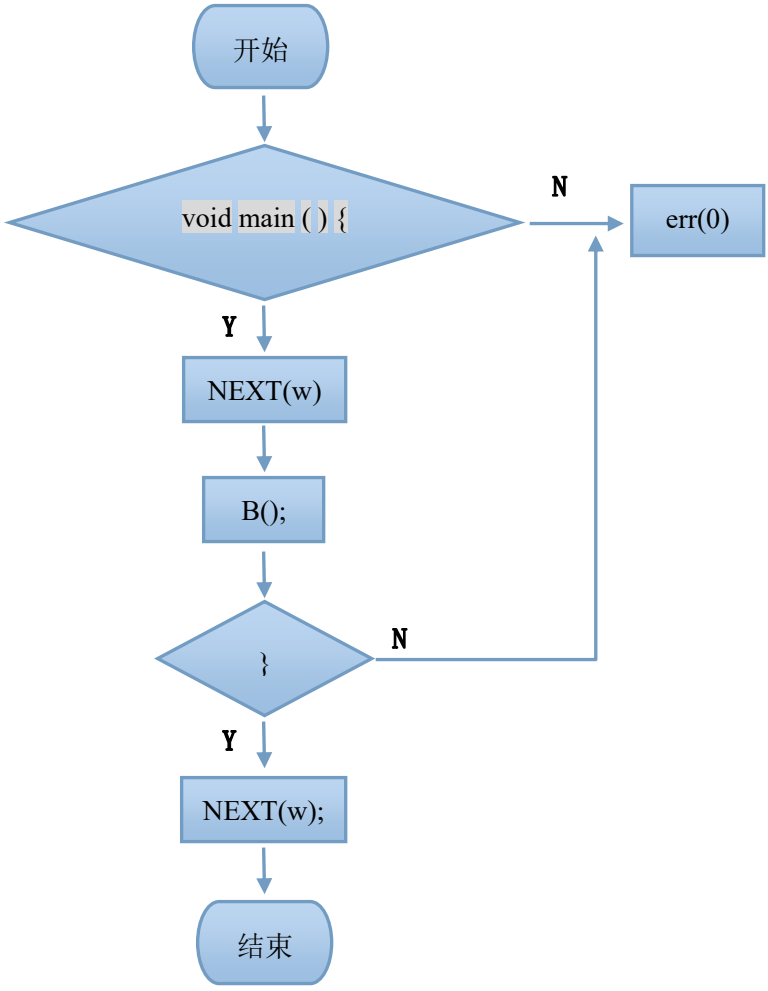


图 3.3-1 A()子程序流程图

B()子程序流程图：

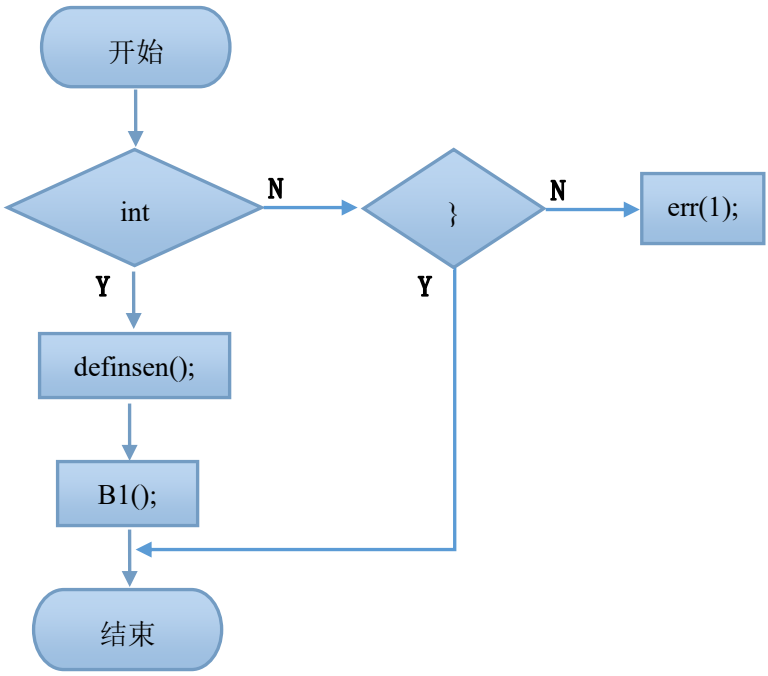


图 3.3-2 B()子程序流程图

B1()子程序流程图：

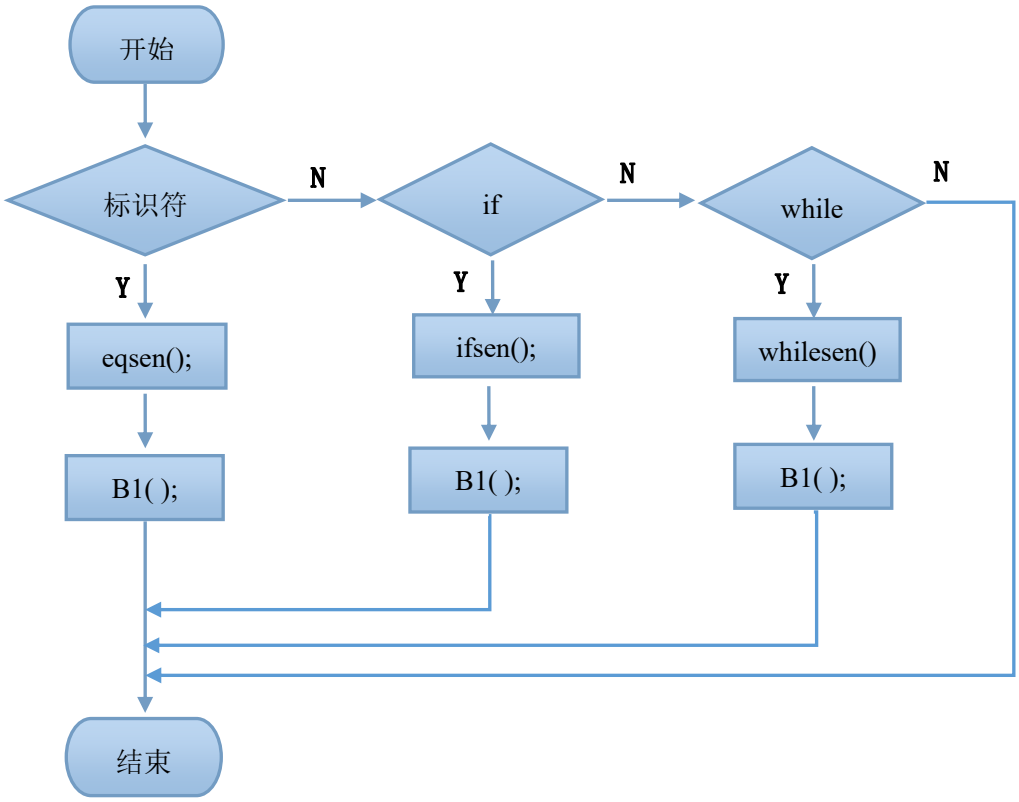


图 3.3-3 B1()子程序流程图

<definsen> -> int 标识符 语义动作:填符号表 <definsen1> | int 标识符 [常数] 语义动作:填符号表、填数组表 <definsen1>

definsen()子程序流程图：

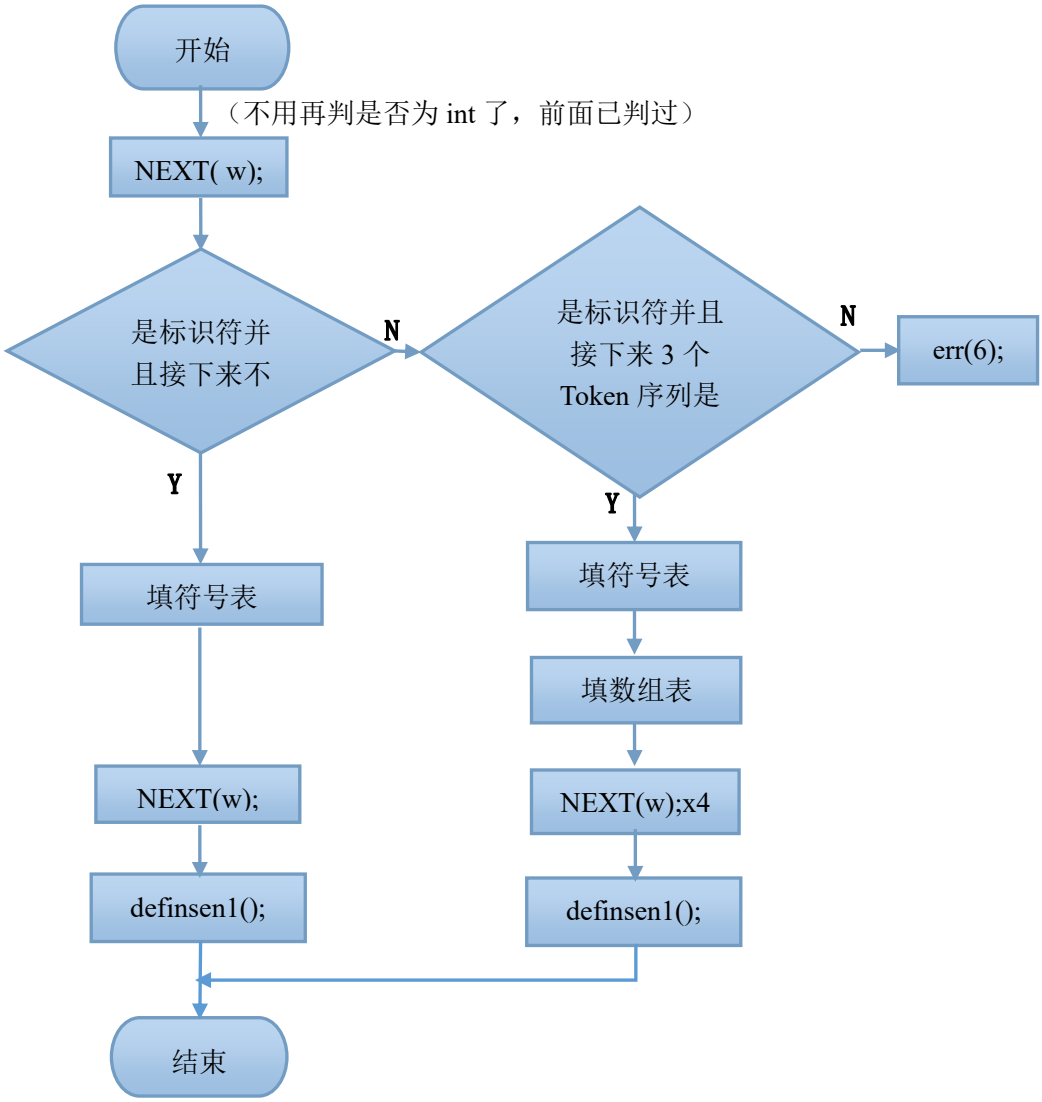


图 3.3-4 definsen()子程序流程图

<definsen1> -> ， 标识符 语义动作:填符号表 <definsen1> | ，标识符 [常数] 语义动作:填符号表、填数组表 <definsen1> | ；

definsen1()子程序流程图：

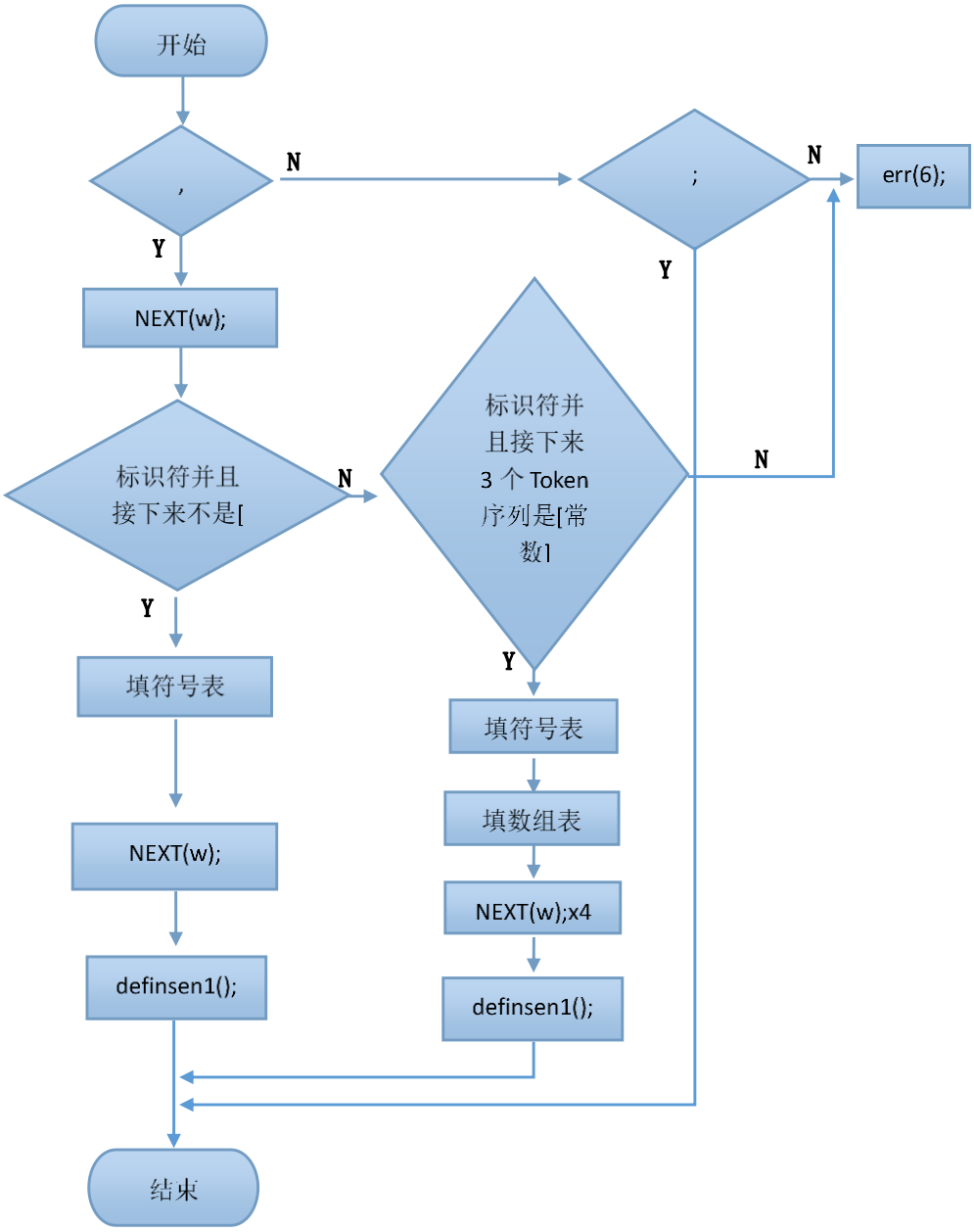


图 3.3-5 definsen1()子程序流程图

<eqsen> -> 标识符 语义动作:压进语义栈 = <E> 语义动作:生成赋

值四元式同时对语义栈操作 ； | 标识符 [常数] 语义动作:压进语义栈 = <E> 语义动作:生成赋值四元式同时对语义栈操作 ；

eqsen()子程序流程图：

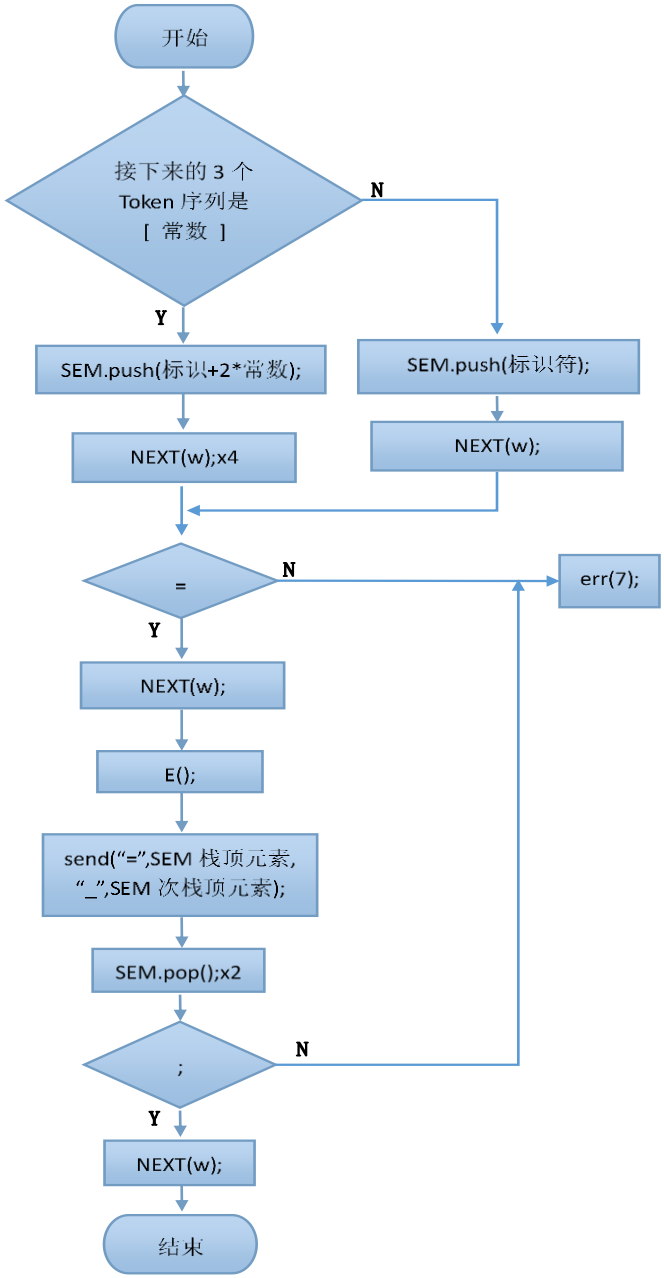


图 3.3-6 eqsen()子程序流程图

<E> -> <T> {加减算符 <T> 语义动作:生成加或减四元式同时对语义栈操作}//花括号代表 0 个或多个

E()子程序流程图：

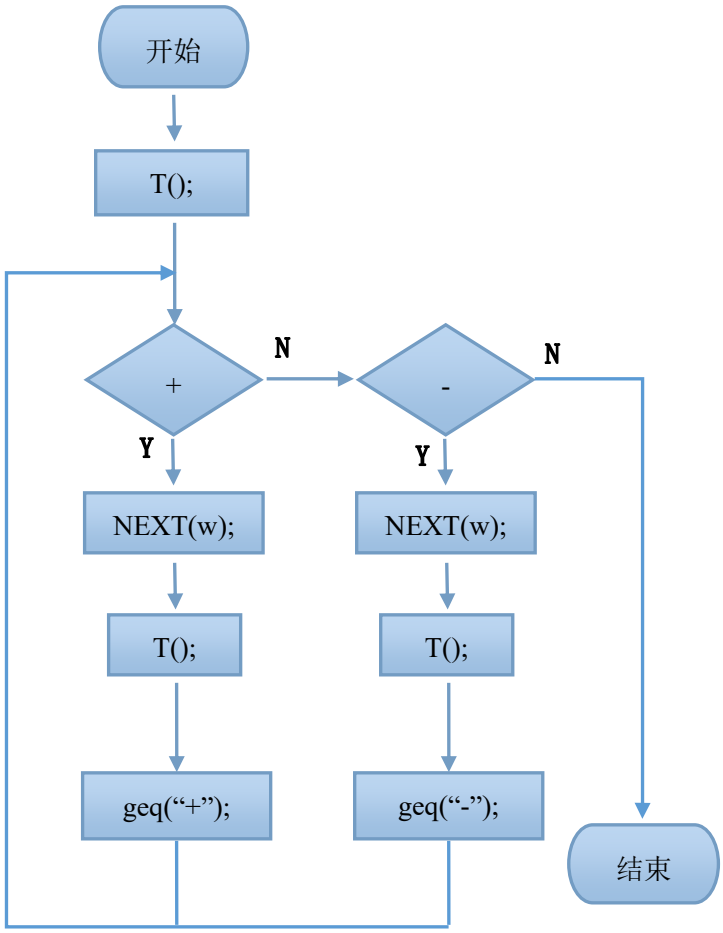


图 3.3-7 E()子程序流程图

<T> -> <F> {乘除算符 <F> 语义动作:生成乘或除四元式同时对

语义栈操作}//花括号代表 0 个或多个

T()子程序流程图：

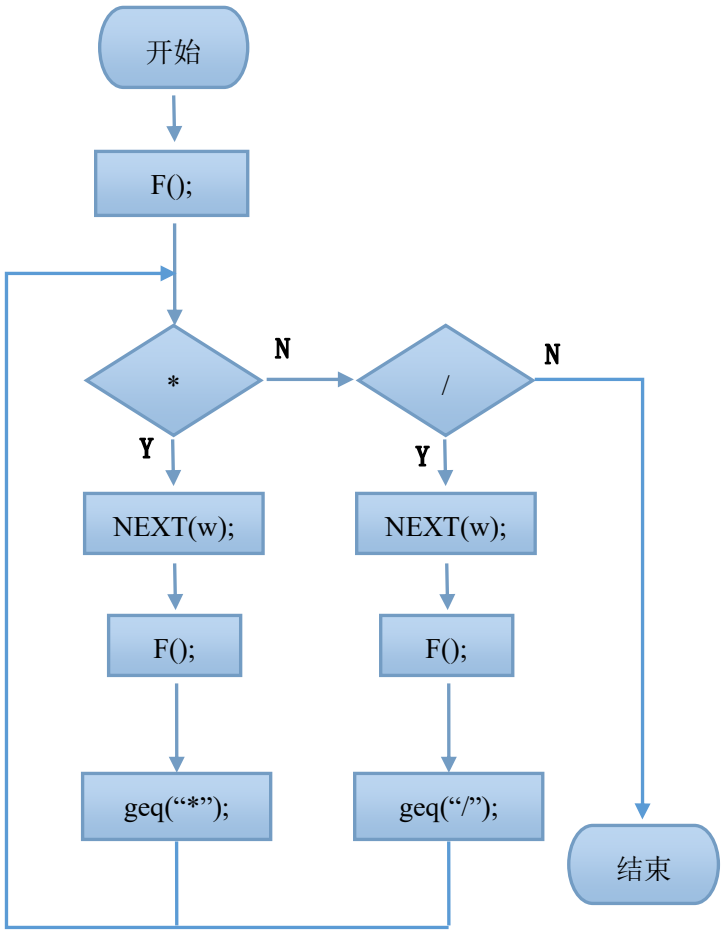


图 3.3-8 T()子程序流程图

<F> -> 标识符 语义动作:压进语义栈 | 常数 语义动作:压进语义栈 | 标识符 [常数] 语义动作:压进语义栈 | (<E>)

F()子程序流程图：

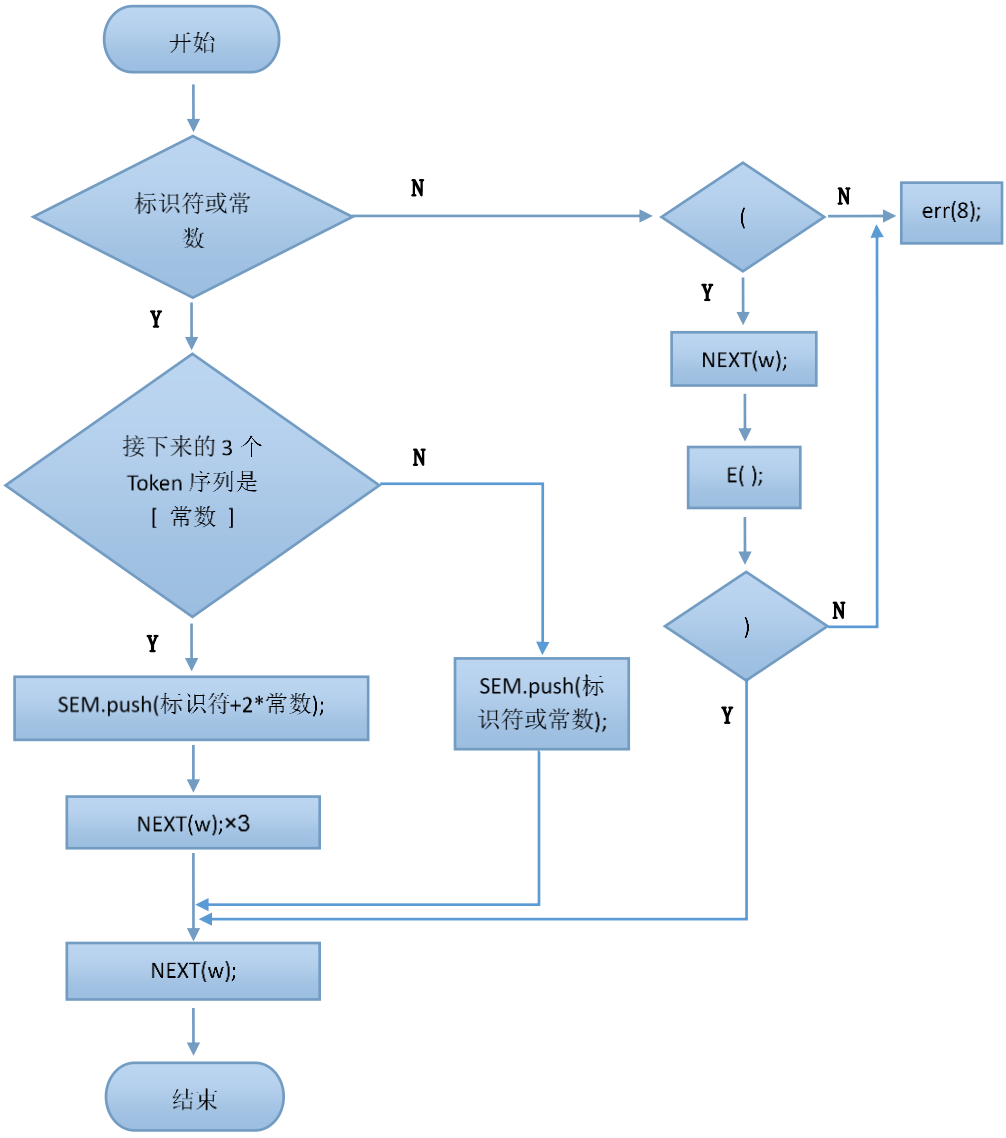


图 3.3-9 F()子程序流程图

条件语句算法

<ifsen> -> if （ <E> 关系算符 <E> ） { <B1> } [else { <B1> }]

1.语法分析算法

语法分析过程见流程图。

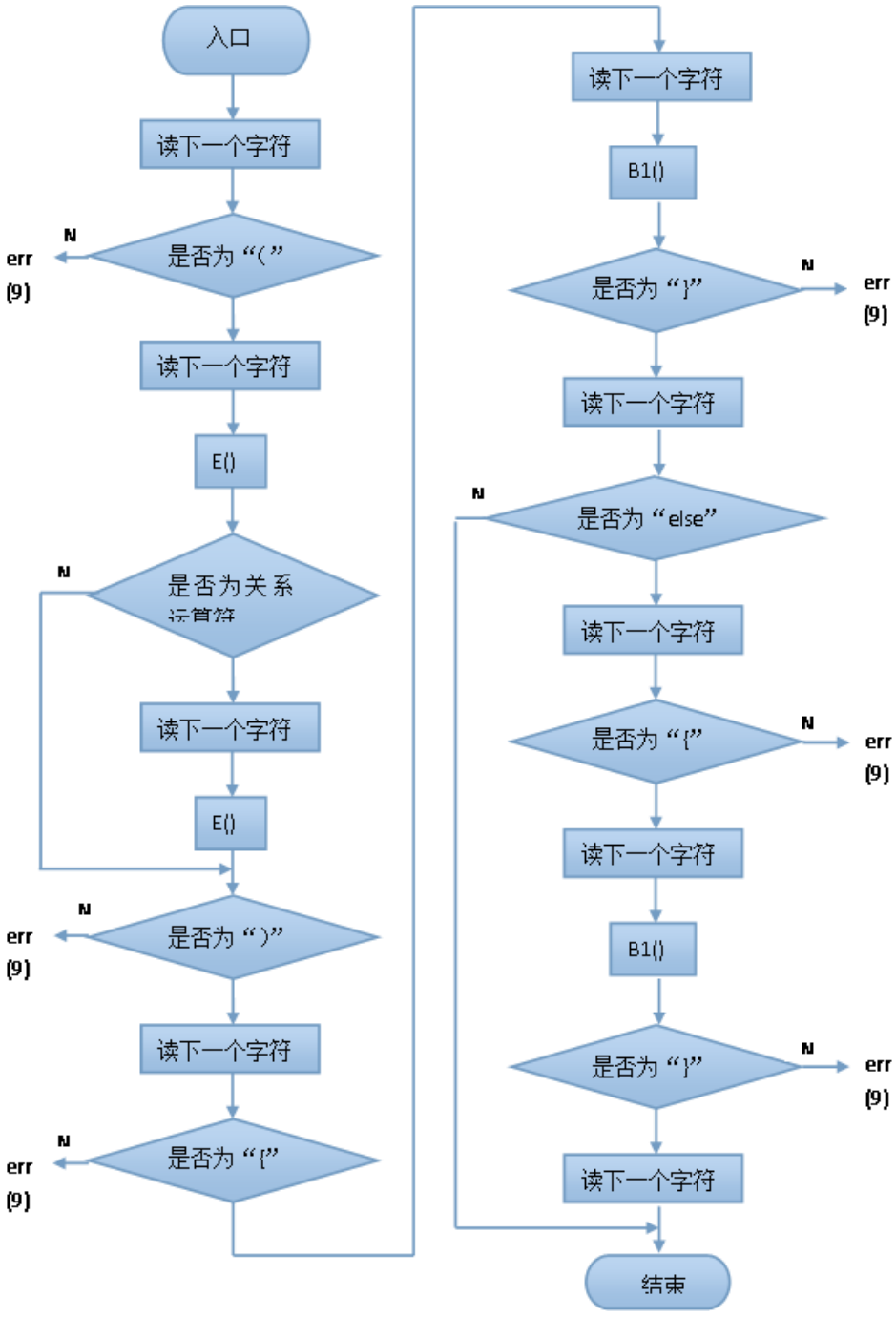


图 3.3-10 ifsen()语法分析子程序流程图

2.语义分析算法

在语法分析的基础上，插入相应的语义动作即可，如下所示。

if （<E> 关 系 算 符 {ACT0}<E>{ACT1}){ACT2}{<B1>}[else{ACT3}{ <B1> }{ACT4}]

其中，{ACTi}即为要插入的语义动作，具体如下表所示。

ACT0	将关系算符对应编码存入 code 中
ACT1	根据不同的 code 选择不同的关系符，再调用表达式的四元式生成函数 geq(“”)，生成相应关系表达式的四元式
ACT2	调用 send(“if”,SEM.top(),”_”,”_”)函数,生成一个标号 if 四元式送 QT[q]，再弹栈
ACT3	调用 send(“else”,”_”,”_”,”_”)函数,生成一个标号 else 四元式送 QT[q]
ACT4	调用 send(“ifend”,”_”,”_”,”_”)函数，生成一个标号 ifend 四元式送 QT[q]

while 语句算法

<whilesen> -> while （ <E> 关系算符 <E> ） { <B1> }

1.语法分析算法

见流程图

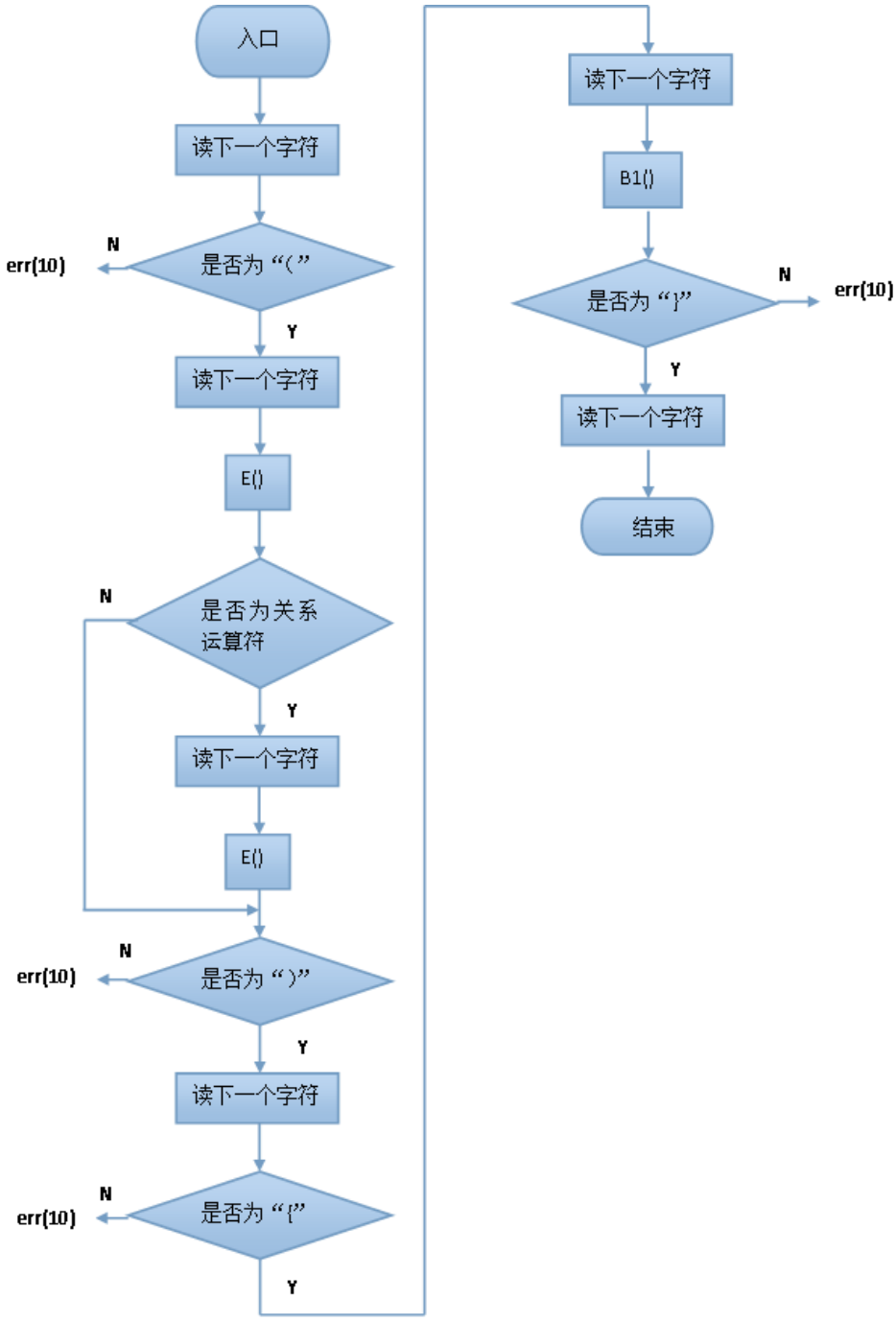


图 3.3-11 whilesen()语法分析子程序流程图

2.语义分析算法

在语法分析的基础上，插入相应的语义动作即可，如下所示。

<whilesen> -> while{ACT0} (<E>关系算符{ACT1}<E>{ACT2}){ACT3}{ <B1> }{ACT4}

其中，{ACTi}即为要插入的语义动作，具体如下表所示。

ACT0	调用 send(“while”,”_”,”_”,”_”)函数，生成一个标号 while 四元式送 QT[q]
ACT1	将关系算符对应编码存入 code 中
ACT2	根据不同的 code 选择不同的关系符，再调用表达式的四元式生成函数 geq(“”)，生成相应关系表达式的四元式
ACT3	调用 send(“do”,SEM.top(),”_”,”_”)函数，生成一个标号 do 四元式送 QT[q]，再弹栈
ACT4	调用 send(“whileend”,”_”,”_”,”_”)函数，生成一个标号 whileend 四元式送 QT[q]

3.4 划分基本块模块

3.4.1 功能

局部优化算法是以基本块为单位进行的。基本块其实也是目标代码生成的基本单位。

那何谓基本块——基本块是程序中一段顺序执行的语句序列，其中只有一个入口和一个出口。

3.4.2 数据结构

```
int block1[10];//block1数组用来存放入口四元式标号
int l1=0;/*block1数组存的入口四元式标号的个数，其值等于block2四元式存的出口四元式标号的个数，其值由inBlock();函数返回值赋值*/
int block2[10];//block2 数组用来存放出口四元式标号
```

3.4.3 算法

1、找基本块的入口语句，它们是：

①程序的第一个语句；

②转向语句（比如说 if 语句、else 语句、do 语句、whileend 语句、goto 语句等）转移到的语句（比如说 ifend 语句、while 语句、标号语句等）；

③紧跟在转向语句后面的语句。

2、对每一入口语句，构造其所属的基本块：

①从该入口语句到另一入口语句之间的语句序列；

②从该入口语句到另一转移语句(或停止语句)之间的语句序列。

3、具体设计如下：

```
int outBlock(int block[], Quat Qt[], int pQt) {
    int l = 0;
    for (int i = 0; i <= pQt - 1; i++) {
        if (Qt[i].op == "if" || Qt[i].op == "else" || Qt[i].op == "do" || Qt[i].op == "whileend") { block[l] = i; l++; }/*转向语句为出口*/
        else if (Qt[i].op == "ifend"|| Qt[i].op == "while") { block[l] = i - 1; l++; }//转向语句转移到的语句上一个为出口
```

```
    }
    if (block[1 - 1] != pQt - 1) { block[1] = pQt - 1; 1++; }/*
最后一个语句一定为出口*/
    return 1;
}

int inBlock(int block[], Quat Qt[], int pQt) {
    int 1 = 0;
    if (Qt[0].op == "ifend"|| Qt[0].op == "while") { block[0] =
0; 1++; }//转向语句转移到的语句恰为第一个语句
    else { block[0] = 0; 1++; }//第一个语句为入口
    for (int i = 1; i <= pQt - 1; i++) {
        if (Qt[i].op == "if" || Qt[i].op == "else" || Qt[i].op
== "do" || Qt[i].op == "whileend") { block[1] = i + 1;
1++; }//转向语句的下一个语句为入口
        else if (Qt[i].op == "ifend"|| Qt[i].op == "while")
{ block[1] = i; 1++; }//转向语句转移到的语句为入口
    }
    if (block[1 - 1] == pQt) { block[1 - 1] = '\0', 1--; }/*如
果入口地址超出了四元式地址，那么就不算了。此种情况do语句可能发
生*/
    return 1;
    }
```

3.5 优化器模块

3.5.1 功能

优化处理是指产生更高效的目标代码所做的工作。它可以分为在中间代码级上的优化和在目标代码上的优化。在本次课设中，我做的是在中间代码级上的优化。这类优化不依赖于具体的计算机。上一阶段划分完基本块后读入基本块入出口地址，根据块数初始化多个 DAG 树，分块进行优化（赋值语句、基本算数运算、数组下标赋值运算、if、while 等语句），每优化完成一个基本块，查找符号表，动态删除无用中间变量，更新符号表里中间变量地址，优化完所有块之后，输出优化完四元式组给 NEWQT 用于生成目标代码。由优化编译程序提供的对代码的各种变换必须遵循一定的原则:

1、等价原则。经过优化后不改变程序的运行结果。

2、有效原则。使优化后所产生的目标代码运行时间较短，占用的存储空间较小。

3、合算原则。应尽可能以较低的代价取得较好的优化效果。

说明：

1、优化中加入对 if、while、else 等语句的支持（虽然未对这类块入口和出口四元式进行处理，但是要进入 DAG 树，保持出口入口地址及最后输出顺序正确）。

2、优化中加入对数组及其他部分优化（包括常数运算、多次赋值、公共表达式、节点优先级交换等）。

3.5.2 数据结构

```
struct Node{
  int NodeNum;                    //该节点的编号
  bool Empty;                     //该节点是否为空
  int RChild,LChild;          //左右子节点的编号
  int TailNum;                //副标记个数
  int Num;          //DAG 节点个数
  string fun;
  string prim;
  string tail[5];
  string special1; //三个特殊标记 用于保存块出入口操作数
  string special2;
  string special3;
  Node(){  //初始化 DAG 节点 并把节点置空
      fun = "";
      prim = "";
      RChild = -1;
      LChild = -1;
      Empty = true;
      tail[0]="";
      tail[1]="";
      tail[2]="";
      tail[3]="";
```

```
        tail[4]="";
        TailNum = 0;
        special1="_";
        special1="_";
        special1="_";
    }
}DAG1[30],DAG2[30],DAG3[30],DAG4[30],DAG5[30],DAG6[30],DAG7[30
],DAG8[30],DAG9[30];


typedef struct SMBLTABLitem {
    string name;
    int type; //int为-1，非-1指向整形一维数组表
    int addr;
    bool operator == (string i);//重载==
}SMBLTABLi;//标识符表项或说符号表项
```

3.5.3 算法

本模块主流程图如下：

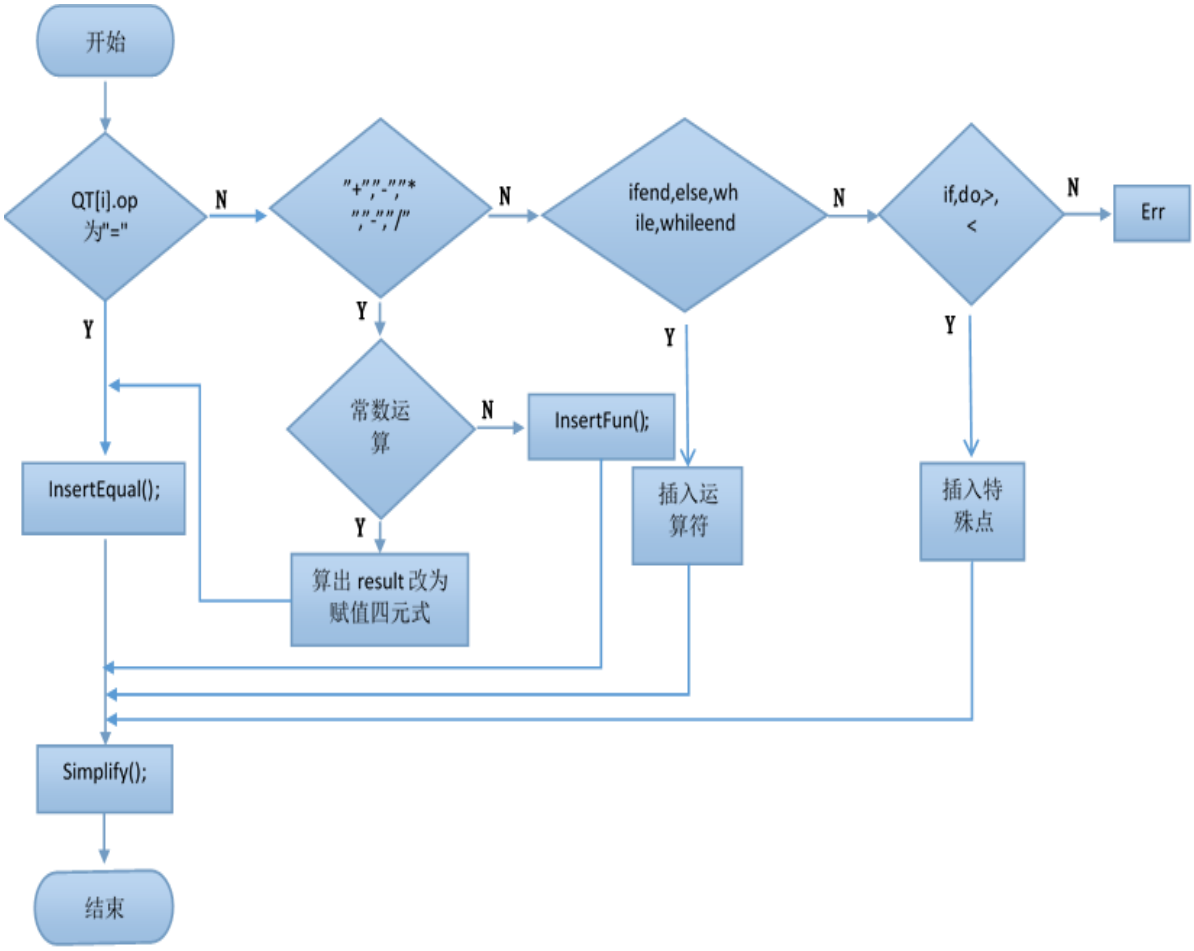


图 3.5-1 优化模块主流程图

DAG 节点插入流程：

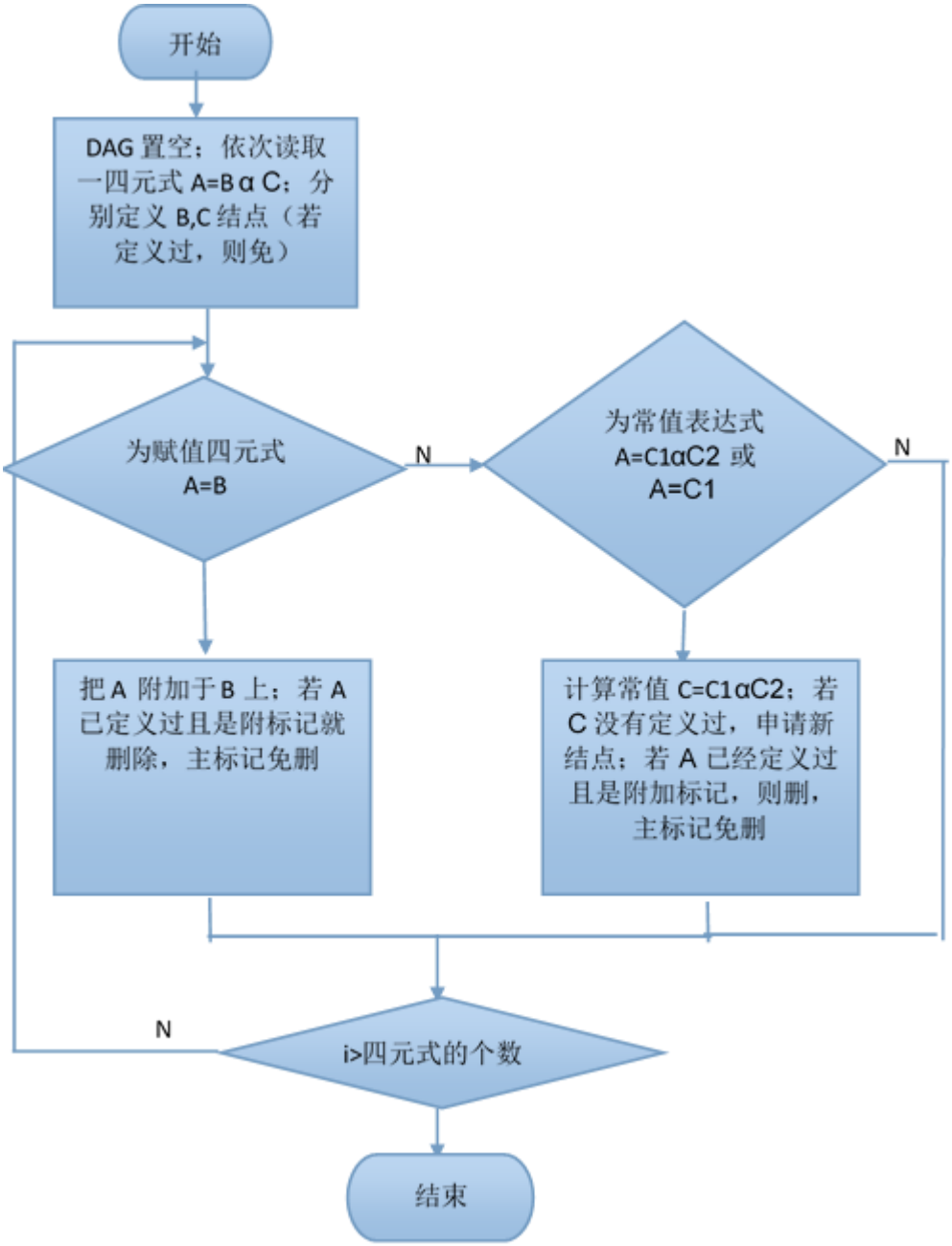


图 3.5-2 DAG 节点插入流程图

以上为优化器的第一个模块，构造基本块内优化的 DAG。

出口之后是另外一个模块。有两个假设：1. 临时变量的作用域是基本块内；2. 非临时变量的作用域也可以是基本块内。

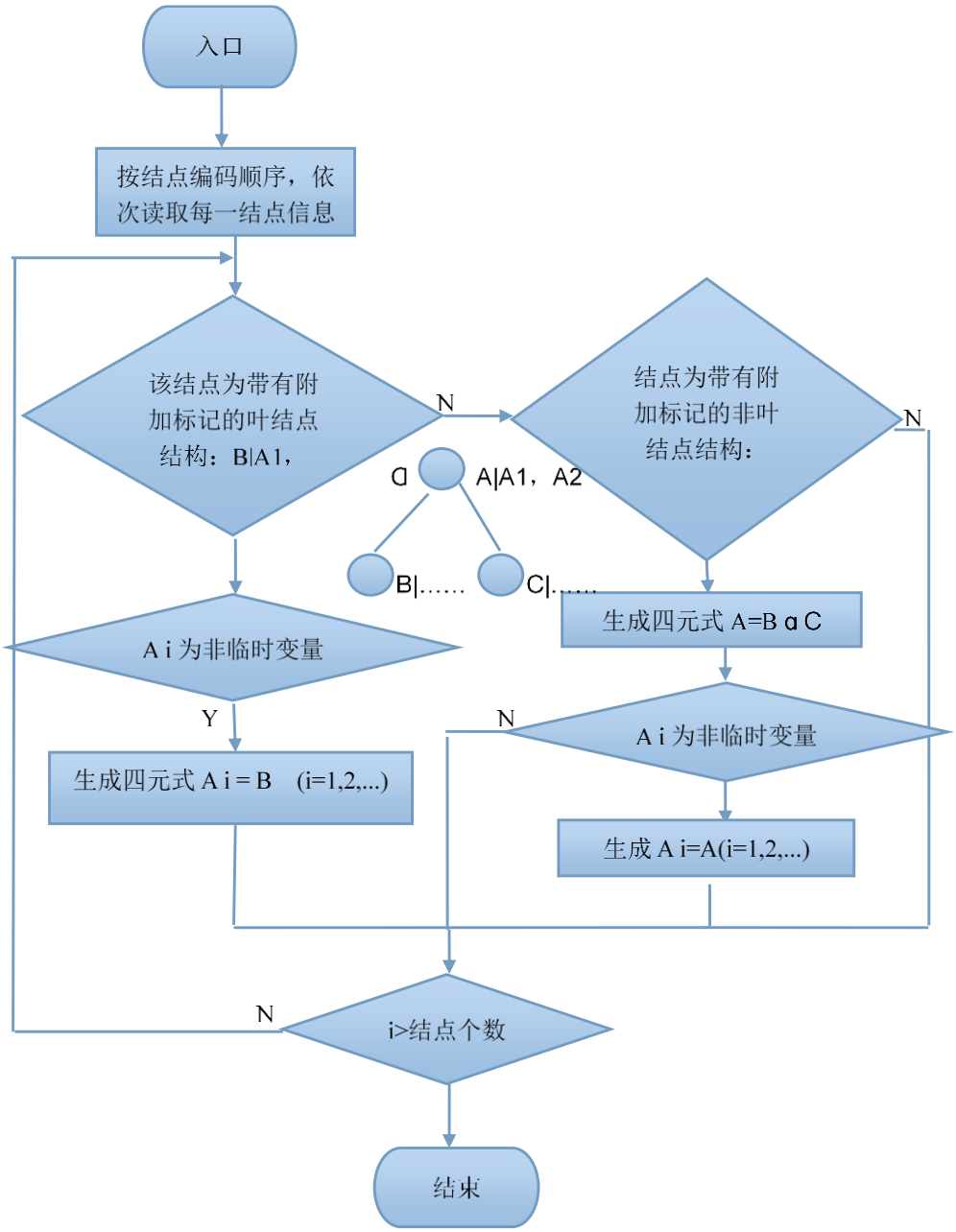


图 3.5-3 四元式生成流程图

3.6 目标代码生成器模块

3.6.1 功能

将优化后的四元式转化为可直接在汇编环境下运行的目标代码。通过访问符号表，给变量分配空间，生成汇编的数据段代码。调用一系列 OBJ 生成函数来生成汇编的代码段代码。在生成汇编过程中，针对特殊情况，对汇编代码进行了优化，减少不必要、重复的语句，并检查优化结果是否正确。最后为验证其是否为正规的汇编语言，将生成的代码写入 asm 文件，并在汇编环境下直接运行它。

3.6.2 数据结构

用二维结构体数组来存储与四元式相对应的汇编代码，之所以这么做是因为一个四元式可生成好几条汇编代码。obj 二维数组的行号就是相应四元式的序号。

```
struct OBJ//OBJ
{
    string obj;//操作符
    string obj1;//操作数 1
    string obj2;//操作数 2
    int j_num;//i 行的列数默认放在零号单元中
    string sign;//标号
    OBJ(){sign = " ";}//标号初始化
```

}obj[50][20];//用来存储汇编代码

stack<int> stack_P1;//程序中使用的栈

int flag[50]={0};//记录汇编代码是否进行了简化，如当上一句的结果是这一句的操作数，即 AX 不需要再置数，此时将对应的 flag 置数

string acc="0";//寄存器空闲与否的标志，用于优化代码

3.6.3 算法

使用的算法与老师所给的有所不同，但也是在此基础上进行的修改。舍弃了分块思想，为此也做出了补充的检查函数。

1.算术表达式 OBJ 生成算法

1）obj 二维数组的列号 j 初始化为 0，代表当前四元式生成的对应的汇编代码为 0 条。

2)查表生成目标代码(w—操作符,RE—四元式用于存储结果的变量,Ti—其他值)

表一（表中对 flag 数组的操作是为最后的 check()函数服务，记录下进行化简的语句序号：若化简，则置数）

w	O1	O2	acc	OBJ

+ - * /	X1	X2	0/Ti	MOV AX,X1
				对 w 查表二(X2)
				MOV RE,AX

+ - * /	X1	X2	X1	对 w 查表二(X2) MOV RE,AX flag[i]置 1

+ *	X1	X2	X2	对 w 查表二(X1) MOV RE,AX flag[i]置 2

- /	X1	X2	X2	MOV AX,X1
				对 w 查表二(X2)
				MOV RE,AX

表二（X 的值根据传过来的操作数来确定）

w	OBJ
+	ADD AX,X
-	SUB AX,X
*	MOV BX,X MUL BX
/	MOV BX,X XOR DX,DX DIV BX

3）计算所生成汇编代码的数目，存入 obj[i][0].j_num

4）将 RE 结果存入 acc

2.赋值语句 OBJ 生成算法

1）obj 二维数组的列号 j 初始化为 0，代表当前四元式生成的对应的汇编代码为 0 条。

2）查遍生成目标代码（O1--X1，O2--X2，RE—四元式用于存储结果的变

量，Ti—其他值）

表三（flag 数组记录下进行化简的语句序号）

acc	OBJ
0/Ti	MOV AX,X1 MOV RE,AX
X1	MOV RE,AX flag[i]置 1

3）计算所生成汇编代码的数目，存入 obj[i][0].j_num

4）将 RE 结果存入 acc

3.布尔表达式 OBJ 生成算法

1）obj 二维数组的列号 j 初始化为 0，代表当前四元式生成的对应的汇编代码为 0 条。

2）查表生成目标代码(w—操作符，W—w 对应的汇编指令，RE—四元式用于存储结果的变量，Ti—其他值)

表四（flag 数组记录下进行化简的语句序号）

acc	OBJ
0/Ti/X2	MOV AX,X1 CMP AX,X2
X1	CMP AX,X2 flag[i]置 1

3）计算所生成汇编代码的数目，存入 obj[i][0].j_num

4）将 RE 结果存入 acc

4.条件语句 OBJ 生成算法

针对汇编代码的特点对原算法进行了修改。具体算法如下所示：

1）obj 二维数组的列号 j 赋为 0，因为以 if,else,ifend 开始的四元式所生成的汇编代码最多只有一句。

2）查表进行相应的操作

表五（以下都是对第 i 个四元式的处理）

四元式标号	操作	备注
if	a.通过访问上一个四元式的操作符如“>”，即调用函数 relation_obj(i-1)，查表六，生成跳转指令，如“JBE” b.把当前四元式序号 i 压栈	//访问布尔表达式四元式生成条件跳转指令，跳转地址待回填

else	a.给 i+1 个四元式生成的汇编代码标号栏标上标号 00
	b.将栈顶元素赋给 k
	c.将标号 00 填入第 k 个四元式生成的汇编代码的操作数 1 上
	d.弹栈
	e.将第 i 个四元式生成的汇编代码的操作符赋为“JMP”
	f.把当前四元式序号 i 压栈
		//标号 00,01,02 均从事先定义好的 sign_table 数组中取出
		//地址回填
		//else 的无条件跳转指令，跳转地址待回填

ifend	a.若下一个四元式时(while, , ,),	//ifend,while 为开头的

	则给第 i+2 个四元式所生成的汇编代码标号栏标上标号 01 否则，给第 i+1 个四元式所生成的汇编代码标号栏标上标号 01 b.将栈顶元素赋给 k c.将标号 01 填入第 k 个四元式生成的第一条汇编代码的操作数 1 上 d.弹栈	四元式不生成汇编代码，需要跳过 //地址回填

表六（在不相符的情况下跳转，跳转地址等待回填）

操作符	OBJ
>=	JB
>	JBE
<=	JA
<	JAE
==	JNE
!=	JE

注意：生成的汇编代码是赋给行号为 i+1 的 obj 数组。（传送过来的序号为 i）

5.while 语句 OBJ 生成算法

针对汇编代码的特点对原算法进行了修改。具体算法如下所示：

1）obj 二维数组的列号 j 赋为 0，因为以 while,do,whileend 开始的四元式所生成的汇编代码最多只有一句。

2）查表进行相应的操作

表七（以下都是对第 i 个四元式的处理）

四元式标号	操作	备注
while	将下一个四元式的序号 i+1 压栈	
do	a. 通过访问上一个四元式的操作符如“>”,即调用函数 relation_obj(i-1)，查表六，生成跳转指令，如“JBE” b. 把当前四元式序号 i 压栈	//访问布尔表达式四元式生成条件跳转指令，跳转地址待回填

whileend	a. 将栈顶元素赋给 k
	b. 给第 i+1 个四元式所生成的汇编代码标号栏标上标号 02
	c. 将栈顶元素赋给 k
	d. 将标号 02 填入第 k 个四元式生成的第一句汇编代码的操作数 1 上
	e. 弹栈
	f. 将第 i 个四元式生成的第一句汇编代码的操作符赋为“JMP”
	g. 将栈顶元素加一后赋给 k
	h. 若第 k 个四元式生成的汇编代码标号栏为空，则给第 k 个四元式所生成的汇编代码标号栏标上标号 03，并将标号 03 填入第 i 个四元式生成的第一句汇编代码的操作数 1 上
	否则，将第 k 个四元式生成的汇编代码标号填入第 i 个四元式生成的汇编
		//标号 00,01,02 均从事先定义好的 sign_table 数组中取出
		//地址回填
		//while 语句的无条件跳转指令，跳转地址待回填
		//加一因为以 ifend 和 while 为标号的四元式不生成汇编代码，需要跳过 k，对 k+1 进行处理
		//地址回填
		//标号已定义，则不再重新定义

	代码的操作数 1 上 i.弹栈	//地址回填

6.check()函数的算法

在老师所给的原算法中，优化部分利用分块思想，并且仅能对算术表达式类型的四元式进行优化，在生成目标代码时也是利用分块思想生成的。

对于目标代码生成而言，我一开始认为没有必要使用分块技术，对普通四元式只要一句句生成即可，对条件四元式，while 四元式，需要地址回填，但由于我将汇编代码都存入二维数组里，所以地址回填只需要修改数组里的内容即可。故按照自己的算法写了下去，在完成时发现了许多不严谨的地方，于是补充了 check()函数，用来检查由化简引起的相关错误。若出错，则取消化简，否则，不进行操作。

具体算法如下所示：

1）对所有生成的汇编代码的标号段进行扫描，当发现不为空时（即有标号）判断其对应的 flag[p]的值是否 1 或 2。flag 数组的编号与每一段汇编代码相对应，当发现 flag[p]的值为 1 或 2 时，说明该段代码进行了化简。

2）若 flag 为 1，则把这一段汇编代码向后调整一列，空出第 0 列并填入“MOV AX,X1”

若 flag 为 2，则把这一段汇编代码向后调整一列，空出第 0 列并填入“MOV AX,X2”

即将化简的效果取消。

3）调整所生成的这段汇编代码的长度值 obj[p][0].j_num。

以上算法是为了防止遇到跳转指令，跳到这条汇编代码时，因为被化简，导致所执行的代码缺失。因为汇编代码是按照所给的四元式顺序所生成的，并且同时进行了化简，所以遇到不能顺序执行的情况时，优化后就会出现问题。

7.Outcode()函数的算法

这是该模块的主函数，负责目标代码生成和输出。

1）首先通过访问符号表，给变量分配空间，并生成汇编数据段代码。

2）再通过调用优化后的四元式，和以上说明的相关函数生成汇编代码段的代码。

3）调用 check()函数，检查可能由优化引起的错误。

4）最后依次输出所有汇编代码并写入 output.asm 文件，以便之后对所生成的汇编代码进行运行测试。

注意：对于最终版本的代码，写入 asm 文件是在可视化中完成，此处写入文件是不考虑可视化的情况。

3.7 可视化模块

3.7.1 功能

我把我们做的编译程序可视化了，做成了简单 C 语言小型集成开发环境。用户能够通过我们所做的平台选择 C 源程序文件，进行编辑、保存、编译。编译时会给出编译每个阶段的结果，比如词法分析完毕会给出 Token 序列，词法分析完毕后会给出源程序通没通过。最终会在 C 源程序所在源目录生成一个与 C 源程序同名但后缀改为 asm 的汇编源程序，即

目标代码。

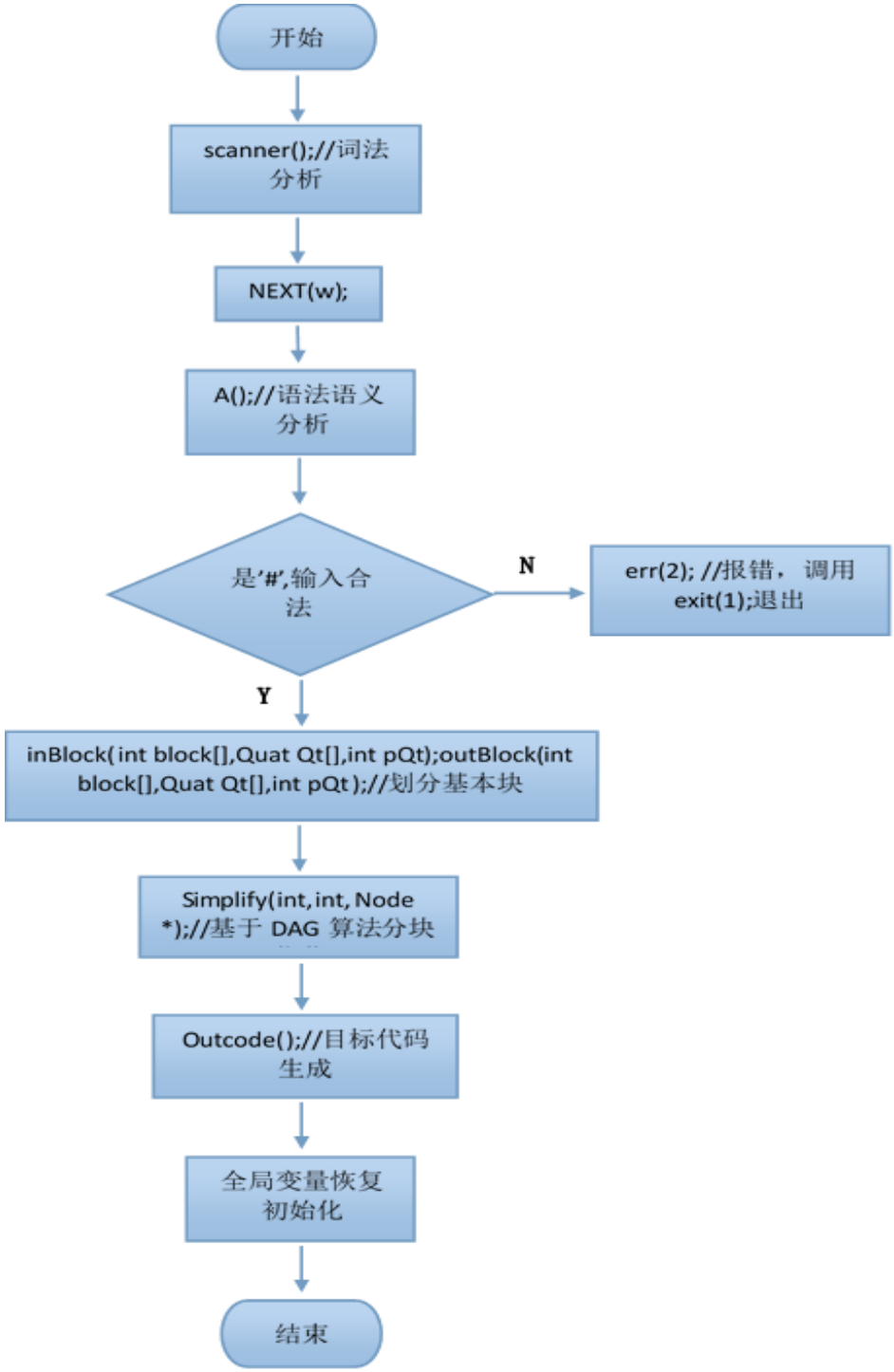
3.7.2 实现

我们编译程序是用 C++实现的，使用 CodeBlocks 13.12 集成开发环境进行开发。最后我是使用 Visual Studio 2015 实现的可视化。我先用 VS2015 建 MFC 基于对话框的工程，画出主界面的样子，添加控件。再把我们的工程项目做成一个 cpp 文件添加到 MFC 工程中。

4 程序设计与实现

4.1 程序流程图

下图是我设计的简单 C 语言编译程序的总流程图或者说是 main()函数的结构图，在 3.1 算法的总体思想 处已给出一次：


开始
scanner();//词法分析
NEXT(w);
A();//语法语义分析
是'#',输入合法
N
err(2); //报错，调用exit(1);退出
Y
inBlock(int block[],Quat Qt[],int pQt);outBlock(int block[],Quat Qt[],int pQt);//划分基本块
Simplify(int,int,Node *);//基于 DAG 算法分块
Outcode();//目标代码生成
全局变量恢复初始化
结束

编译程序的总流程图

各个模块具体流程图在　3 算法与数据结构　中都已给出，在这不再一一列出。

4.2 程序说明

int main();//调用各模块

4.2.1 词法分析：

scanner(): 词法分析{可以生成 Token 序列及静态符号表并输出}

IsLetter():判断字符是否为字母

IsDigit():判断字符是否为数字

state_to_code(): 根据 Token 输出对应的编码。

ch_to_num(): 把字符变为对应的编码，便于处理。

state_check(): 执行自动机。

searchcons(): 查找常数表中是否有对应的常数。

searchadmdefine(): 在符号表中查找符号，返回 1 表示为标识符，返回 2 表示是中间变量。返回 0 表示不存在。

4.2.2 语法语义分析与中间代码生成模块：

```
void A();/*A();子程序，下面一直到whilesen();都是递归下降子程序法的子程序，当然都能产生相应的语义动作产生四元式*/
void B();
void B1();
```

```
void definsen();
void definsen1();
void eqsen();
void E();
void T();
void F();
void ifsen();
void whilesen();
void err(int k);//语法分析出错处理函数
void geq(string op);//表达式四元式生成函数
void send(string op, string op1, string op2, string result);/*
生成一个四元式送到四元式数组QT中去*/
void ENT(char type, int addr);//填写符号表函数
void ENT1(int low, int up, int ctp, int clen);//填写数组表函数
```

4.2.3 划分基本块模块：

```
int inBlock(int block[], Quat Qt[], int pQt);/*不但可以对未优
化前的四元式数组QT找出基本块入口，还可以对优化后的四元式数组
NEWQT找出基本块入口，返回值为基本块入口个数*/
int outBlock(int block[], Quat Qt[], int pQt);/*对四元式数组找
出基本块出口，返回值为基本块出口个数，其值等于基本块入口个数*/
```

4.2.4 优化模块：

程序段特色函数（部分）

1、动态维护符号表，并重新分配中间变量地址

实现函数1：结构体中重载==运算以查找是否是需要删除的中间变量

```
typedef struct SMBLTABLitem {
    string name;
    int type; //int为-1，非-1指向整形一维数组表
    int addr;
    bool operator == (string i);//重载==
}SMBLTABLi;//标识符表项或说符号表项
bool SMBLTABLi::operator == (string i)
{
    return this->name == i;
}
```

实现函数 2：维护符号表，实现无用中间变量删除

```
void Maintain(struct Node *p);
```

2、优化中加入对 if、while、else 等语句的支持（虽然未对这类块入口和出口四元式进行处理，但是要进入 DAG 树，保持出口入口地址及最后输出顺序正确）

DAG 数据结构：

```
struct Node{
  int NodeNum;                    //该节点的编号
  bool Empty;                     //该节点是否为空
  int RChild,LChild;          //左右子节点的编号
  int TailNum;                //副标记个数
```

```
  int Num;          //DAG 节点个数
  string fun;
  string prim;
  string tail[5];
  string special1; //三个特殊标记 用于保存块出入口操作数
  string special2;
  string special3;
  Node(){  //初始化 DAG 节点 并把节点置空
      fun = "";
      prim = "";
      RChild = -1;
      LChild = -1;
      Empty = true;
      tail[0]="";
      tail[1]="";
      tail[2]="";
      tail[3]="";
      tail[4]="";
      TailNum = 0;
      special1="_";
      special1="_";
      special1="_";
  }
}DAG1[30],DAG2[30],DAG3[30],DAG4[30],DAG5[30],DAG6[30],DAG7[30
```

```
],DAG8[30],DAG9[30];

实现函数 1：插入 DAG 节点时判断是赋值还是算术运算还
是块出入口四元式，并按不同方式进行插入
int InsertNode(int i,struct Node *DAG);
```

3、数组及其他部分优化（包括常数运算、多次赋值、公共表达式、节点优先级交换等）

```
实现函数 1：运算符为“=”或者能通过常数运算转换成“=”的情况插
值
int InsertEqual(int i , struct Node *DAG);

实现函数 2：运算符为算术运算符的情况插值
int InsertFun(int i,struct Node *DAG);

实现函数 3：按节点优先级交换主副标记
void ShiftNode(int n,struct Node *DAG);

//寻找相同部分
int Mutpart(string str,struct Node *DAG);

//检查是否重定义过 返回 redifeined1 表示定义过且是副节点，
redifined2 表示定义过且是主节点
string VerRedefine(string str , struct Node *DAG);
```

```
//找到所在的节点并返回 NodeNum
int FindNum(string str , struct Node *DAG);
```

4.2.5 目标代码生成模块：

用二维结构体数组来存储与四元式相对应的汇编代码，因为一个四元式可生成好几条汇编代码。obj 二维数组的行号就是相应四元式的序号。通过访问符号表，给变量分配空间，生成汇编数据段代码。通过调用以下生成 OBJ 函数来生成汇编代码段代码。各 OBJ 生成函数里，针对特殊情况，对汇编代码进行了优化。对普通四元式一句一句地生成，对条件四元式，while 四元式，需要地址回填，因为我将汇编代码都存入二维数组里，所以地址回填只需要修改数组里的内容即可。之后调用 check()函数检查可能由优化引起的错误。最后按顺序输出全部的汇编代码。将输出的汇编代码存入 asm 文件，并在汇编环境下直接执行文件。

全部函数声明如下所示：（具体实现见算法说明）

```
int asmd_obj(int i,int j,string ops);//"+-*/"类四元式-->OBJ
void expression_obj(int i);//算术表达式 OBJ
void assignment_obj(int i);//赋值语句 OBJ
void relation_obj(int i);//关系运算符汇编跳转指令的生成
void boolexp_obj(int i);//布尔表达式 OBJ
void ifsen_obj(int i);//IF 语句 OBJ
void whilesen_obj(int i);//while 语句 OBJ
void check();//检查可能由优化引起的错误
void err();//错误提醒
```

```
int Outcode();//对符号表进行访问，生成并输出目标代码
```

4.3 实验结果

采用如下一段 C 语言程序进行验证。这是一段正确的程序，符合我们所定义的文法。

```
    //main.c
void main(){
    int a,b,c,d,e,p,q,t,r,arr[10],bb[20];//definsen
    a=9;arr[0]=39;/*eqsen*/
    bb[0]=0;bb[1]=bb[0]+arr[0];
    if(a<3){
    b=a+2;
    c=5+3;
    d=b*c;
    e=b*c+3;
    p=t-r;
    q=(t-r)*9;
    }
    else {b=a+2;}
    while(b>6){
    c=b+a;
    b=b-1;}
}
```

4.3.1 词法分析模块输出结果如下：

Token 序列如下（部分）：

```
6       -1          0       9       arr
5       -1          68      -1
66      -1          3       3       0
67      -1          69      -1
62      -1          55      -1
4       -1          3       4       39
0       0       a   65      -1
64      -1          0       10      bb
0       1       b   68      -1
64      -1          3       3       0
0       2       c   69      -1
64      -1          55      -1
0       3       d   3       3       0
64      -1          65      -1
0       4       e   0       10      bb
64      -1          68      -1
0       5       p   3       5       1
64      -1          69      -1
0       6       q   55      -1
64      -1          0       10      bb
0       7       t   68      -1
64      -1          3       3       0
0       8       r   69      -1
64      -1          58      -1
0       9       arr 0       9       arr
68      -1          68      -1
3       0       10  3       3       0
69      -1          69      -1
64      -1          65      -1
0       10      bb  7       -1
68      -1          66      -1
3       1       20  0       0       a
69      -1          57      -1
65      -1          3       6       3
0       0       a   67      -1
55      -1          62      -1
3       2       9   0       1       b
65      -1          55      -1
```

```
65      -1
63      -1
63      -1
endscanner.
```

4.3.2 语法语义分析与中间代码生成模块加基本块划分模块输出结果如下：

```
请按任意键继续. . .
语法分析通过.
请按任意键继续. . .
四元式未优化前总计33个.分别为:
0:      =       9       _       a
1:      =       39      _       arr+0
2:      =       0       _       bb+0
3:      +       bb+0    arr+0   T1
4:      =       T1      _       bb+2
5:      <       a       3       T2
6:      if      T2      _       _
7:      +       a       2       T3
8:      =       T3      _       b
9:      +       5       3       T4
10:     =       T4      _       c
11:     *       b       c       T5
12:     =       T5      _       d
13:     *       b       c       T6
14:     +       T6      3       T7
15:     =       T7      _       e
16:     -       t       r       T8
17:     =       T8      _       p
18:     -       t       r       T9
19:     *       T9      9       T10
20:     =       T10     _       q
21:     else    _       _       _
22:     +       a       2       T11
23:     =       T11     _       b
24:     ifend   _       _       _
25:     while   _       _       _
26:     >       b       6       T12
27:     do      T12     _       _
28:     +       b       a       T13
29:     =       T13     _       c
30:     -       b       1       T14
31:     =       T14     _       b
32:     whileend        _       _       _
```

图 4.3.2-1 四元式序列

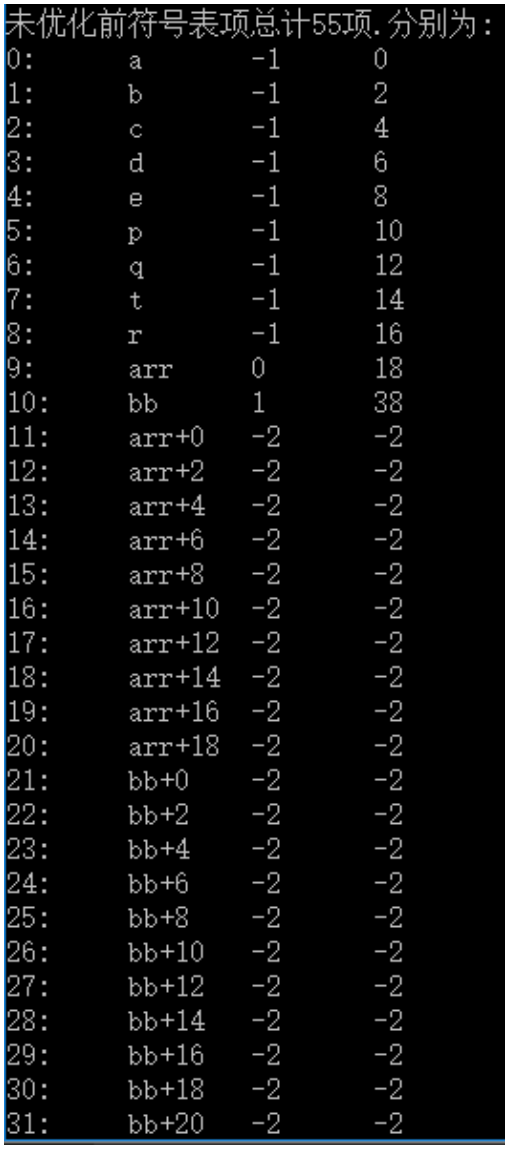

未优化前符号表项总计55项.分别为:
0: a -1 0
1: b -1 2
2: c -1 4
3: d -1 6
4: e -1 8
5: p -1 10
6: q -1 12
7: t -1 14
8: r -1 16
9: arr 0 18
10: bb 1 38
11: arr+0 -2 -2
12: arr+2 -2 -2
13: arr+4 -2 -2
14: arr+6 -2 -2
15: arr+8 -2 -2
16: arr+10 -2 -2
17: arr+12 -2 -2
18: arr+14 -2 -2
19: arr+16 -2 -2
20: arr+18 -2 -2
21: bb+0 -2 -2
22: bb+2 -2 -2
23: bb+4 -2 -2
24: bb+6 -2 -2
25: bb+8 -2 -2
26: bb+10 -2 -2
27: bb+12 -2 -2
28: bb+14 -2 -2
29: bb+16 -2 -2
30: bb+18 -2 -2
31: bb+20 -2 -2

图 4.3.2-2 符号表 1

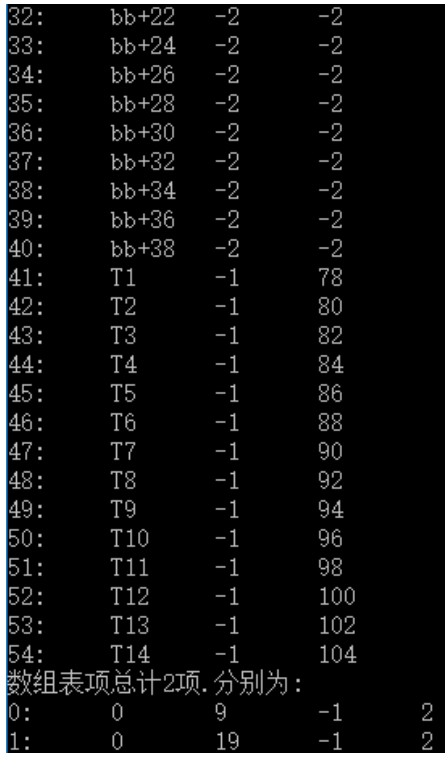


图 4.3.2-2 符号表 2 加数组表

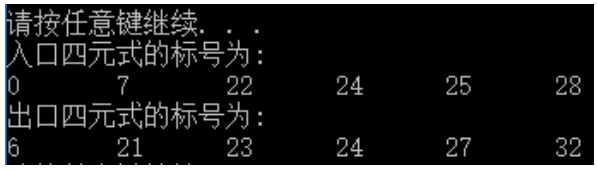


图 4.3.2-3 划分基本快，步入编译后端

可以看出，语法分析通过、四元式正确生成、符号表和数组表填的也是正确的、基本块划分的入口、出口四元式标号也对（可以把 图 4.3.2-1 和 图 4.3.2-3 放一起看）。注意，每个数组元素都进了符号表便于优化工作，类型为-2 表示无意义。地址为-2 也表示无意义。

因为用来验证的源程序没有错误，所以需要人为的添加错误。具体情

况请看 4.3.5。

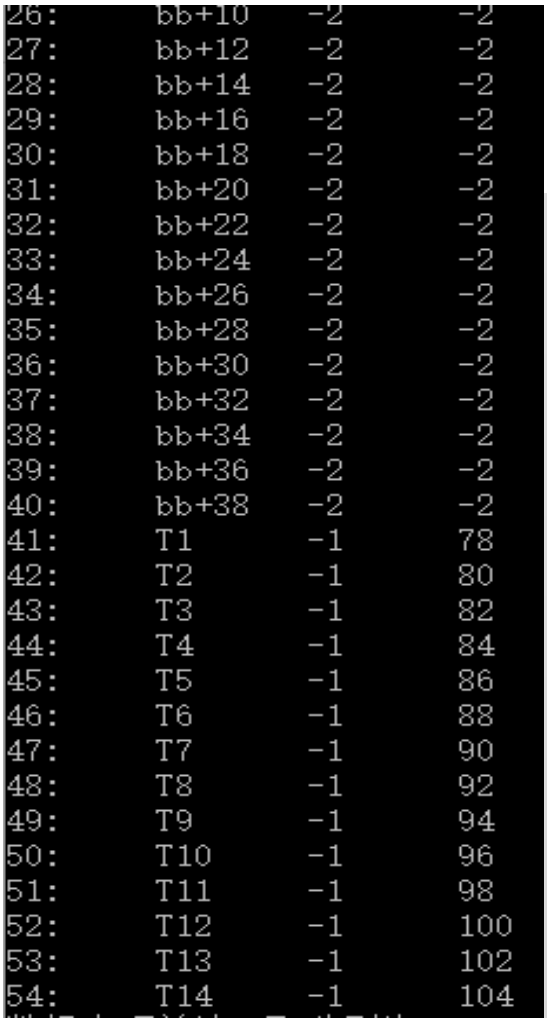
4.3.3 优化器模块输出结果如下：

分块的 DAG 树如下（部分）：

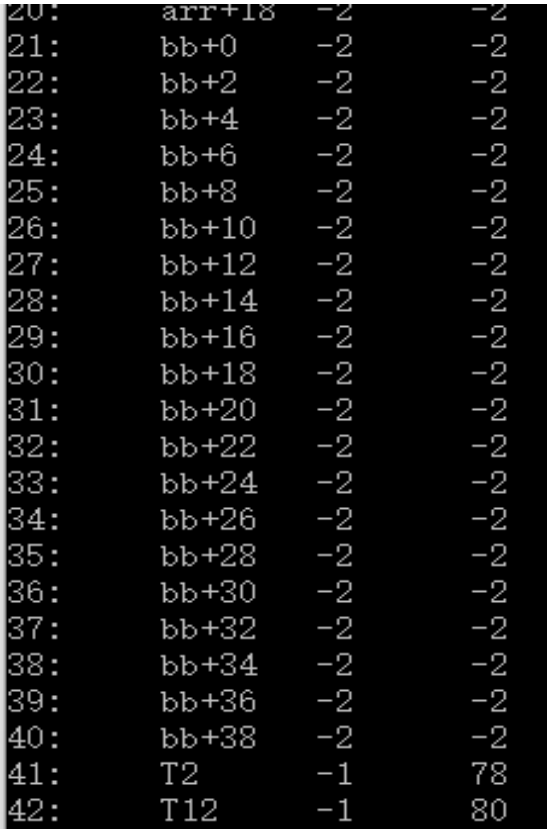
```
块0信息如下:
DAG节点如下:
该节点编号是:1操作符是:=主标记是:9副标记是: a,
该节点编号是:2操作符是:=主标记是:39副标记是: arr+0,
该节点编号是:3操作符是:=主标记是:0副标记是: bb+0,
该节点编号是:4操作符是:+主标记是:bb+2副标记是: T1, 左孩子是:0右孩子是:39
该节点编号是:5操作符是:<主标记是:副标记是:
该节点编号是:6操作符是:if主标记是:副标记是:
该块生成的四元式如下:
(= 9 _ a)
(= 39 _ arr+0)
(= 0 _ bb+0)
(+ 0 39 bb+2)
(< a 3 T2)
(if T2 _ _)


块1信息如下:
DAG节点如下:
该节点编号是:1操作符是:+主标记是:b副标记是: T3, 左孩子是:a右孩子是:2
该节点编号是:2操作符是:主标记是:a副标记是:
该节点编号是:3操作符是:主标记是:2副标记是:
该节点编号是:4操作符是:=主标记是:8副标记是: T4, c,
该节点编号是:5操作符是:*主标记是:d副标记是: T5, T6, 左孩子是:b右孩子是:8
该节点编号是:6操作符是:+主标记是:e副标记是: T7, 左孩子是:d右孩子是:3
该节点编号是:7操作符是:主标记是:3副标记是:
该节点编号是:8操作符是:-主标记是:p副标记是: T8, T9, 左孩子是:t右孩子是:r
该节点编号是:9操作符是:主标记是:t副标记是:
该节点编号是:10操作符是:主标记是:r副标记是:
该节点编号是:11操作符是:*主标记是:q副标记是: T10, 左孩子是:p右孩子是:9
该节点编号是:12操作符是:主标记是:9副标记是:
该节点编号是:13操作符是:else主标记是:副标记是:
该块生成的四元式如下:
(+ a 2 b)
(= 8 _ c)
(* b 8 d)
(+ d 3 e)
(- t r p)
(* p 9 q)
(else _ _ _)
```

优化前后符号表和优化前后四元式组截图：

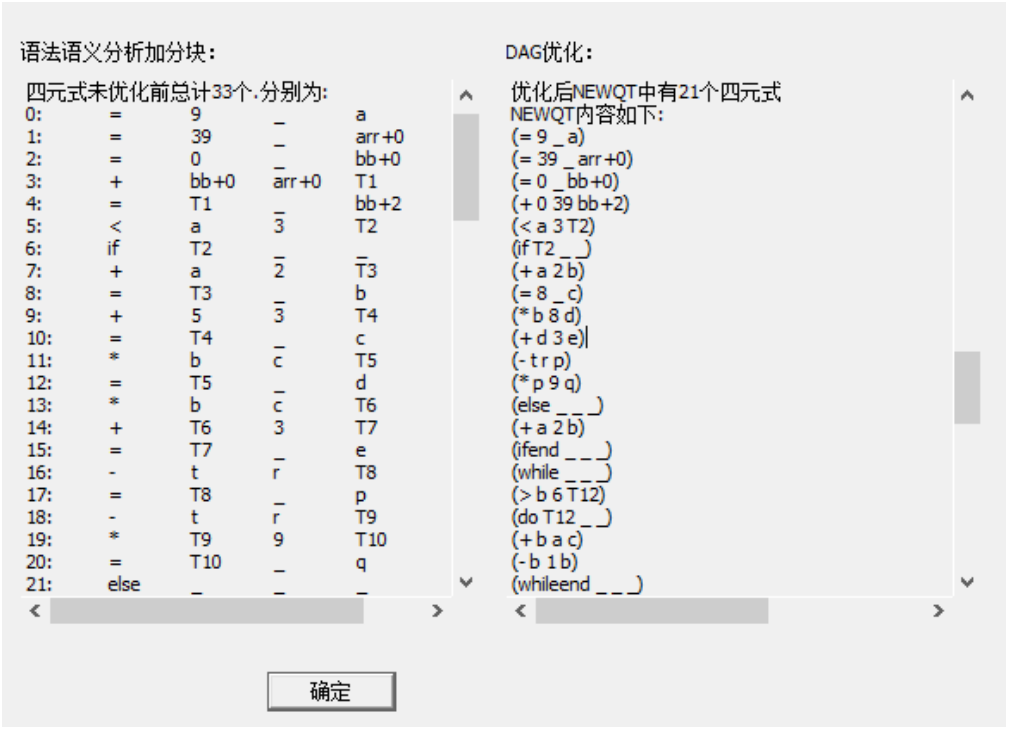


26:	bb+10	-2	-2
27:	bb+12	-2	-2
28:	bb+14	-2	-2
29:	bb+16	-2	-2
30:	bb+18	-2	-2
31:	bb+20	-2	-2
32:	bb+22	-2	-2
33:	bb+24	-2	-2
34:	bb+26	-2	-2
35:	bb+28	-2	-2
36:	bb+30	-2	-2
37:	bb+32	-2	-2
38:	bb+34	-2	-2
39:	bb+36	-2	-2
40:	bb+38	-2	-2
41:	T1	-1	78
42:	T2	-1	80
43:	T3	-1	82
44:	T4	-1	84
45:	T5	-1	86
46:	T6	-1	88
47:	T7	-1	90
48:	T8	-1	92
49:	T9	-1	94
50:	T10	-1	96
51:	T11	-1	98
52:	T12	-1	100
53:	T13	-1	102
54:	T14	-1	104



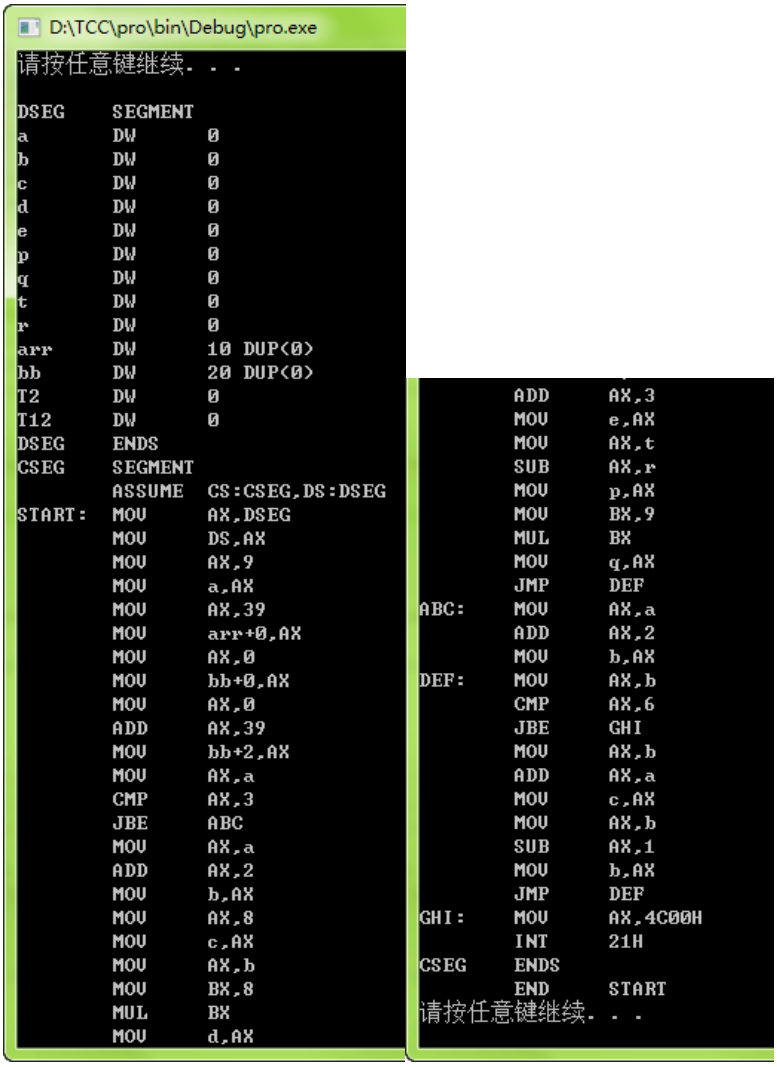
20:	arr+18	-2	-2
21:	bb+0	-2	-2
22:	bb+2	-2	-2
23:	bb+4	-2	-2
24:	bb+6	-2	-2
25:	bb+8	-2	-2
26:	bb+10	-2	-2
27:	bb+12	-2	-2
28:	bb+14	-2	-2
29:	bb+16	-2	-2
30:	bb+18	-2	-2
31:	bb+20	-2	-2
32:	bb+22	-2	-2
33:	bb+24	-2	-2
34:	bb+26	-2	-2
35:	bb+28	-2	-2
36:	bb+30	-2	-2
37:	bb+32	-2	-2
38:	bb+34	-2	-2
39:	bb+36	-2	-2
40:	bb+38	-2	-2
41:	T2	-1	78
42:	T12	-1	80

优化前后四元式组截图：


语法语义分析加分块：
四元式未优化前总计33个.分别为:
0: = 9 _ a
1: = 39 _ arr+0
2: = 0 _ bb+0
3: + bb+0 arr+0 T1
4: = T1 _ bb+2
5: < a 3 T2
6: if T2 _ _
7: + a 2 T3
8: = T3 _ b
9: + 5 3 T4
10: = T4 _ c
11: * b c T5
12: = T5 _ d
13: * b c T6
14: + T6 3 T7
15: = T7 _ e
16: - t r T8
17: = T8 _ p
18: - t r T9
19: * T9 9 T10
20: = T10 _ q
21: else _ _ _
DAG优化：
优化后NEWQT中有21个四元式
NEWQT内容如下:
(= 9 _ a)
(= 39 _ arr+0)
(= 0 _ bb+0)
(+ 0 39 bb+2)
(< a 3 T2)
(if T2 _ _)
(+ a 2 b)
(= 8 _ c)
(* b 8 d)
(+ d 3 e)
(- t r p)
(* p 9 q)
(else _ _ _)
(+ a 2 b)
(ifend _ _ _)
(while _ _ _)
(> b 6 T12)
(do T12 _ _)
(+ b a c)
(- b 1 b)
(whileend _ _ _)
确定

4.3.4 目标代码生成模块输出结果如下：

所生成的汇编代码，如下图所示：



按照 C 源程序，程序运行后，各变量的结果应该为：

```
a=9
b=6
c=16(10H)
d=88(58H)
e=91(5BH)
arr[0]=39(27H)
bb[0]=0
bb[1]=39(27H)
```

我在 win7 系统上使用 DOSBOX 来运行汇编程序，运行过程如下所示：

```
Z:\>mount c d:\TCC\pro
Drive C is mounted as local directory d:\TCC\pro\

Z:\>c:

C:\>masm output.asm
Microsoft (R) Macro Assembler Version 5.00
Copyright (C) Microsoft Corp 1981-1985, 1987.  All rights reserved.

Object filename [output.OBJ]:
Source listing  [NUL.LST]:
Cross-reference [NUL.CRF]:

  51706 + 464838 Bytes symbol space free

      0 Warning Errors
      0 Severe  Errors
```

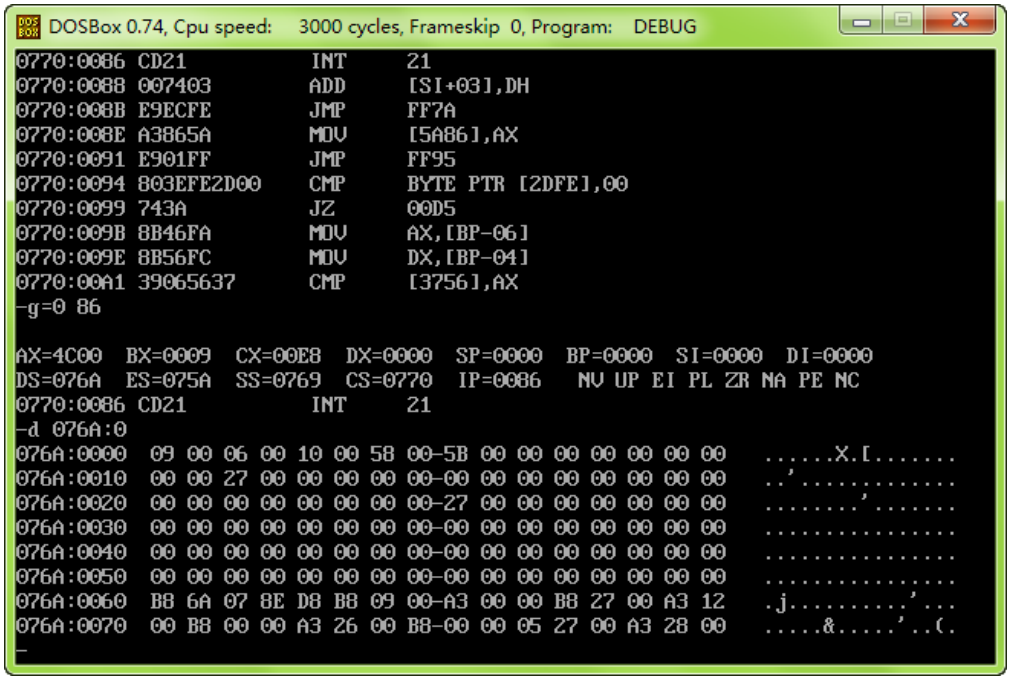
```
C:\>link output

Microsoft (R) Segmented-Executable Linker  Version 5.13
Copyright (C) Microsoft Corp 1984-1991.  All rights reserved.

Run File [output.exe]:
List File [NUL.MAP]:
Libraries [.LIB]:
Definitions File [NUL.DEF]:
LINK : warning L4021: no stack segment
```

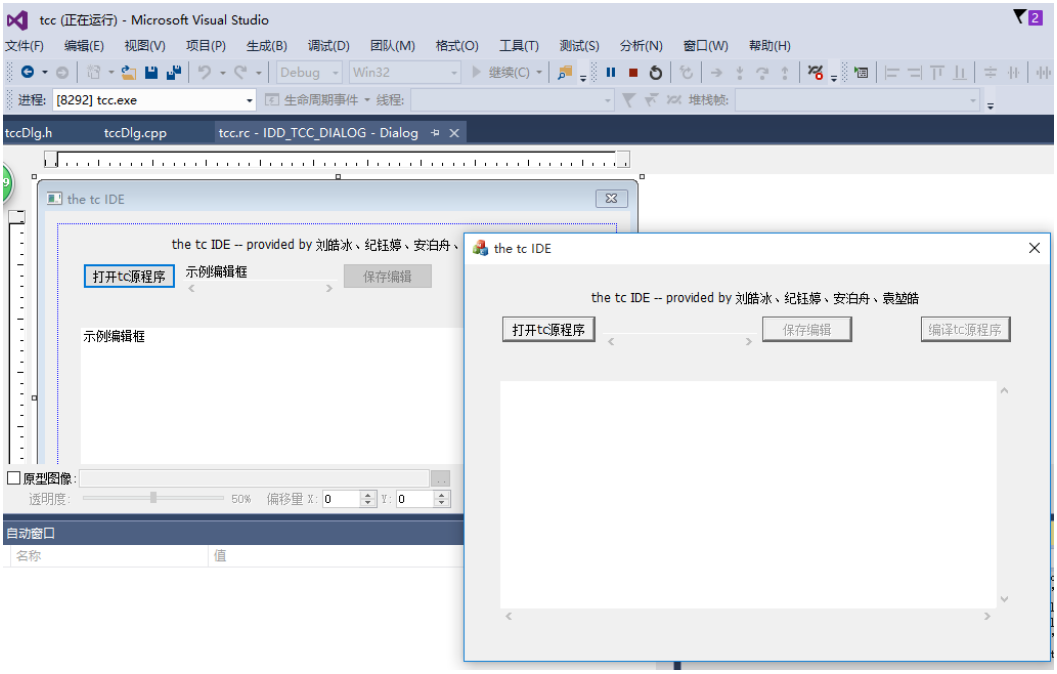
在 debug 下用 G 命令执行汇编程序后，并用 D 命令查看内存，与预期

结果进行比较，变量的值相同，验证成功。如下图所示：



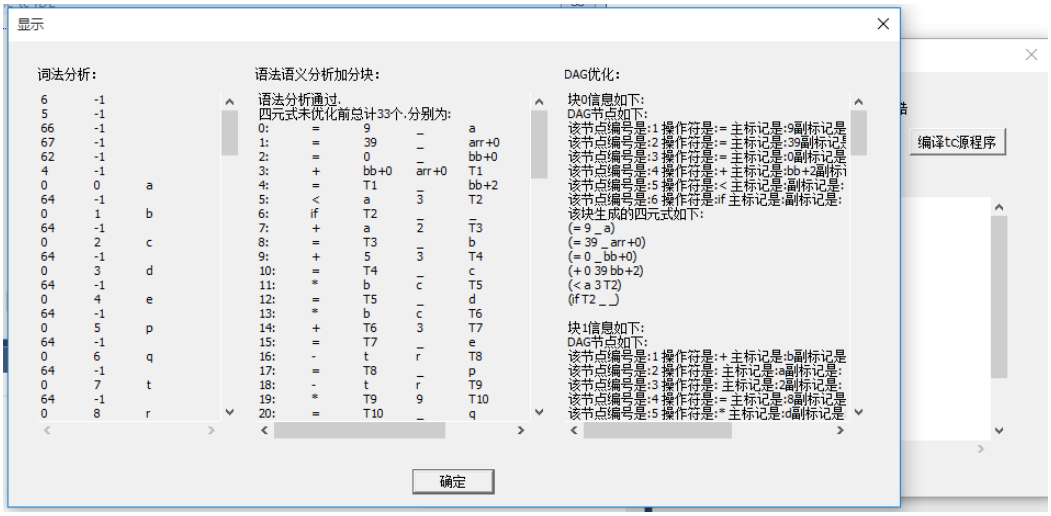
4.3.5 可视化界面如下图所示：

初始运行样子如下所示，此时还没选中 C 源程序文件，后两个 Button 是不可用的：

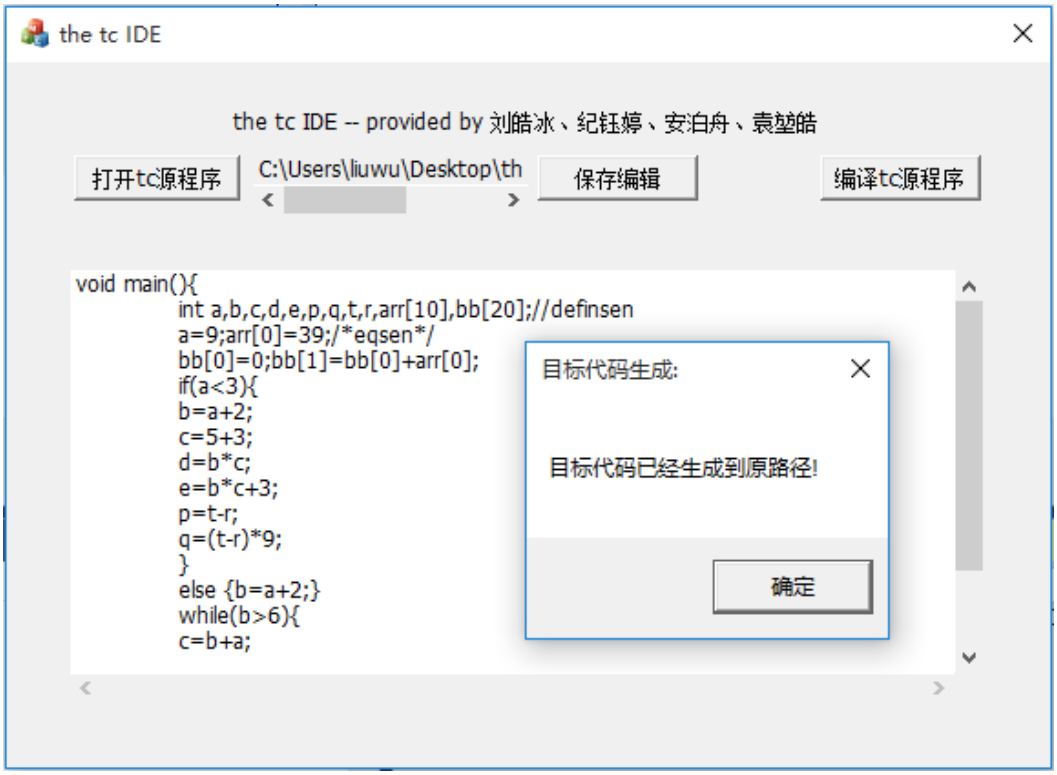


在点击 打开 tc 源程序 Button 选中源程序之后，源程序会显示到 Edit Control 里，此时后两个 Button 可用:

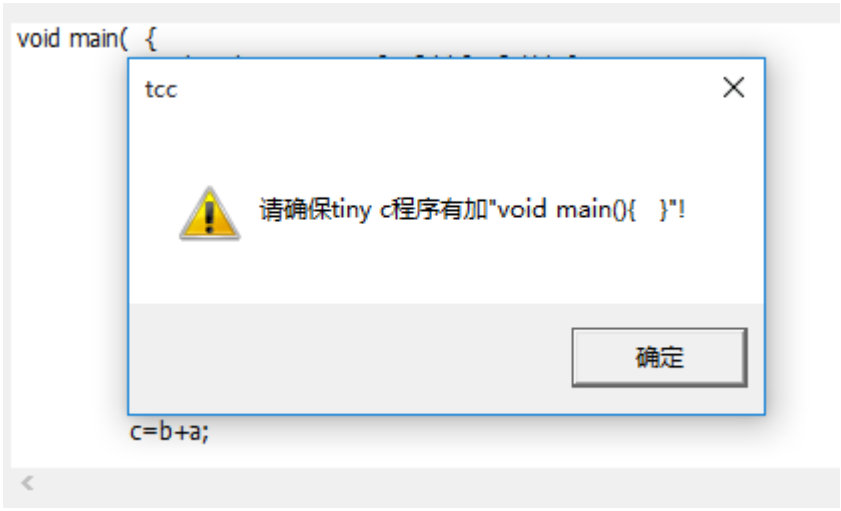
点击 编译 tc 源程序 Button，会弹出另一个对话框，若语法分析通过的话，编译每个阶段的结果都会显示出来：



点击 确定 按钮，会弹出一个 MessageBox，告诉用户目标代码已生成到原路径：

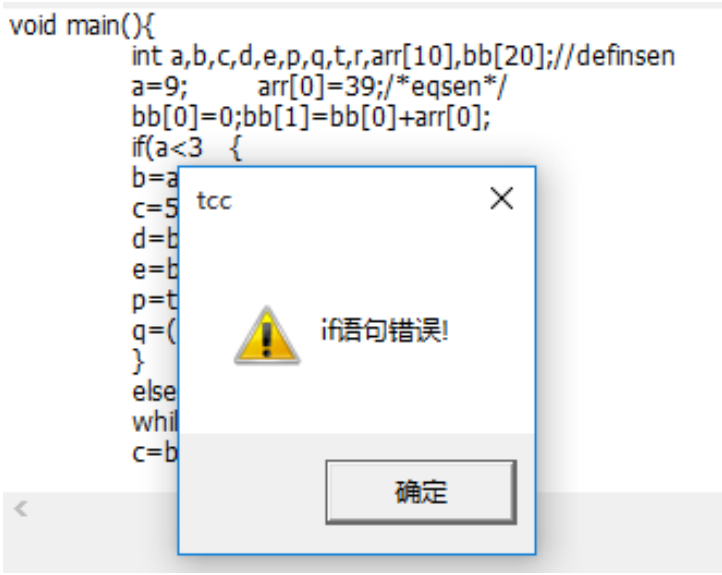


通过编辑，把 main()的右括号去掉，点击 保存编辑 Button，再次编译，会报错：



加上右括号，去掉赋值语句的分号：

同样，if 语句错了也会报错，不再一一列举：


void main(){
int a,b,c,d,e,p,q,t,r,arr[10],bb[20];//definsen
a=9;　　arr[0]=39;/*eqsen*/
bb[0]=0;bb[1]=bb[0]+arr[0];
if(a<3　{
b=a
c=5
d=b
e=b
p=t
q=(
}
else
whil
c=b
tcc
if语句错误!
确定

5 结论

我们所设计的 C 语言编译程序可以根据我们所定义的文法成功地进行词法分析，生成相应的 Token 序列；成功地进行语法语义分析，生成中间代码——四元式序列，若语法分析有错会给出错误所在；成功地用 DAG 进行优化;成功地生成目标代码——汇编代码;成功地进行了可视化，做成了集成开发环境。

因此，整体来说，我们本次课设是成功的。但我们也有遗憾，我们本来要把函数加进来。我们已经实现了函数声明的语法语义分析，但是由于时间原因我们没法做到调用函数，没法实现调用函数只有函数声明是没有意义的，最后我们放弃了把函数加进来的想法。我们还准备加入“++i”和“--i”，其实只要是扫描器能够识别“++”和“--”，语法语义分析看到“++”或“--”的种族编码就可以自动生成(+ i 1 i)或(- i 1 i)的四元式，由于时间关系我们没能赶在最后加上。

6 收获、体会和建议

参考文献

1、陈火旺. *《程序设计语言编译原理》（第 3 版）*. 北京：国防工业出版社. 2000.

2、百度百科.

3、百度学术.

4、东北大学编译原理课程课件讲义.