**课 程 设 计 报 告**

设计题目：编译原理课程设计

班 级：计算机科学与技术1605班

组长学号：20164436

组长姓名：陈弘超

指导教师：肖桐

设计时间：2018年12月

设计分工

组长学号及姓名：20164436 陈弘超

分工：总体负责算法的设计，总体布局接口的定义及设计，文法的设计及扩充，自动生成分析表，语义分析的设计及实现。

组员1学号及姓名：20164417 曹越

分工：辅助算法的设计，四元式的优化（dag），基本块的划分，目标代码的生成（PPT），目标代码的生成（8086）。

组员2学号及姓名：20164382 张亚强

分工：辅助算法的设计，自动机实现的词法分析，活跃信息的填写，常数处理机的设计与实现，符号表的设计与实现。

组员3学号及姓名：20164515 刘键

分工：辅助算法的设计，语义动作的设计及实现，四元式生成代码实现，测试样例的设计及调试

# 

# 摘 要

本次课程设计我们用C++，运用面向对象的编程思想，完成了一个编译器的设计，编译器的设计主要可以分为前端和后端两个部分。  
 编译器前端可大致分为三个部分，词法分析，语法分析和语义分析。我们语法分析在实现上使用了自动生成LL1分析表的表驱动的LL1分析法。在LL1分析法中，采用在产生式中插入语义动作的方法进行语法制导翻译产生四元式。在前端分析完成后，所有信息存储在一个较为综合的符号表中，包含了后端需要的所有信息，并传递给后端。同时，为后端提供了一些读取，查找符号表的接口函数，便于后端使用。编译器后端可分为两部分，四元式的优化及目标代码生成。其中优化算法采用了DAG优化算法，划分基本快后，以基本块为单位对四元式进行优化。产生优化后的四元式后，对四元式进行活跃信息的填写，再对目标代码生成采用单寄存器分配方法，首先分配寄存器，处理操作数寻址方式，并生成对应的目标代码，然后生成运算部分的目标代码。生成的汇编代码基于IntelX86。前端后端中间结合使用了接口函数来适应二者数据结构的细微区别。

**关键词**：编译原理，编译器，前端，后端

**目录**

**摘要** 1

**1 概述** 4

**2 课程设计任务及要求** 5

**3 算法与数据结构** 5

3.1 算法的总体思想 5

3.2 词法分析器 6

3.2.1 功能 6

3.2.2 数据结构 6

3.2.3 算法 7

3.3 语法分析 10

3.3.1 功能 10

3.3.2 数据结构 10

3.3.3 算法 11

求推出空的非终结符集合Vnull 11

求First集合 13

求Follow集合 15

求Select集合 17

构造分析表 19

LL 1 分析过程 20

3.4 语义分析（产生四元式） 22

3.4.1 功能 22

3.4.2 数据结构 22

3.4.3 算法 23

3.5 文法设计 26

3.5.1 功能 26

3.5.2 数据结构 26

3.5.3 算法 27

动作函数 27

文法输入 30

3.6 符号表设计 36

3.6.1 功能 36

3.6.2 数据结构 36

3.6.3 算法 36

3.7 四元式优化 38

3.7.1 功能 38

3.7.2 数据结构 38

3.7.3 算法 39

基本块的划分 39

基于DAG的四元式优化 39

3.8 活跃信息生成 43

3.8.1 功能 43

3.8.2 数据结构 43

3.8.3 算法 45

3.9 目标代码生成 47

3.9.1 功能 47

3.9.2 数据结构 47

3.9.3 算法 48

**4 程序设计与实现** 53

4.1 程序流程图 53

4.2 程序说明 54

4.3 实验结果 55

**5 结论** 58

**6 参考文献** 58

**7 收获、体会和建议** 59

# 1.概述

编译器前端使用了以语法分析为中心的方法。

词法分析器作为一个函数提供给语法分析器，函数在每次调用时返回一个token。在遇到文件结尾时，前端会返回一个标识符表示文件结尾，语法分析到此处停止。

在语法分析的实现上，使用递归下降分析法识别结构较为复杂的语法结构，如，函数，if-else, while 等语法成分。由于这些部分较为稳定，即在设计完成后，不会进行较大的修改。在递归下降的语义分析部分，将语法动作插入到产生式中。并在分析过程中填写符号表，为了简化数据结构，在符号表中添加了四元式区。这样，只需将符号表传递给后端，降低了编码复杂性。

LL1分析法负责识别结构较为简单，但是需要处理较多细节的部分，如算数表达式，逻辑表达式。由于此处在编码时可能涉及到较多的修改，为了降低修改语法所带来的大量代码修改。在此处实现了LL1分析表的自动生成，修改语法仅需要修改储存文法的文件即可，不需要对分析器本身进行修改。大量减少了修改，新建文法所带来的编码工作，且易于调试。而且，分析表的自动生成可以实现复杂的算数逻辑表达式，能够使语言具有更强的可用性。在语义分析上采用了属性文法的方法，在归约时执行语义动作。

编译器后端包括四元式优化和目标代码生成两个部分。

后端大致思路为，将前端四元式用接口函数转换为后端待处理四元式的数据结构（大体相同，接口函数只处理细节），经过以基本块为单位的优化算法实现四元式的优化，并且将标志活跃信息的算法动作插入划分基本块算法后。以得到含有活跃信息的优化后的四元式序列，接下来目标代码生成算法处理此序列得到目标汇编指令。其中，优化算法使用DAG算法，目标代码生成采用多寄存器分配方法，首先分配寄存器，处理操作数寻址方式，并生成对应的目标代码，然后生成运算部分的目标代码。处理了很多算法细节使算法更加健壮完备，生成的代码是基于IntelX86的汇编指令。

# 2.课程设计任务及要求

设计任务：一个简单文法的编译器的设计与实现。

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；

6、提交课程设计报告。

# 3.算法与数据结构

## 3.1算法的总体思想

编译器前端大致可分为三个部分：词法分析，语法分析以及语义分析。

编译器前端使用了以语法分析为中心的方法。词法分析器作为一个封装类提供给语法分析器，词法分析器的输入为string流，输出为token串流。LL1分析法中，采用了自动生成LL1分析表的算法，将语法动作插入到产生式中。并在分析过程中填写符号表，同时进行了类型检查的判断。这样，只需将符号表传递给后端，降低了编码复杂性。

使用自动生成LL1分析表驱动的语法分析算法，使前端具有较强的灵活性。 如goto，if-else, while 等语法成分。由于这些部分较为稳定，即在设计完成后，不会进行较大的修改。LL1分析法负责细节部分，如算数表达式，逻辑表达式。在LL1 部分，修改与新建文法只需要修改储存文法的文件即可，不需要对分析器本身进行修改。可以使较为复杂的文法并大量减少调试工作。

后端算法目标在于，优化前端输出的四元式序列，在此基础上生成目标代码（Intel x86汇编指令）。

具体思路大致可以描述为：将前端四元式用接口函数转换为后端待处理四元式的数据结构（大体相同，接口函数只处理细节），经过以基本块为单位的优化算法实现四元式的优化，并且将标志活跃信息的算法动作插入划分基本块算法后，以得到含有活跃信息的优化后的四元式序列，接下来目标代码生成算法处理此序列得到目标汇编指令。目标代码生成采用多寄存器分配方法，首先分配寄存器，处理操作数寻址方式，并生成对应的目标代码，然后生成运算部分的目标代码。

## 3.2 词法分析器

### **3.2.1 功能**

词法分析器主要功能是为语法分析器提供TOKEN串。该部分包含一个头文件：lexical.h 。词法分析器将准备编译的源代码作为输入，利用设计好的确定的有限自动机的状态转移函数，实现状态的转移和对语句的分词功能。根据终结状态将分词划分为六类：关键字、标识符、界符、常数常量、字符常量和字符串常量。最后再生成对应的TOKEN串。

### **3.2.2 数据结构**

TOKEN的结构定义如下：

struct TOKEN

**{**

int i**;** //对应分词的种类

int j**;** //对应分词在相应表中的位置

int Vt\_id**;** //记录分词在文法中的终结符信息

TOKEN**(**int a **=** 0**,** int b **=** 0**,** int c **=** 0**)**

**{**

i **=** a**,** j **=** b**,** Vt\_id **=** c**;**

**}**

**};**

vector**<**string**>**TOKEN\_k**;** //关键字表

vector**<**string**>**TOKEN\_p**;** //界符表

vector**<**string**>**TOKEN\_i**;** //标识符表

vector**<**string**>**TOKEN\_c**;** //常数常量表

vector**<**string**>**TOKEN\_strc**;** //字符串常量表

vector**<**string**>**TOKEN\_charc**;** //字符常量表

map**<**string**,** int**>**Vt**;** //文法中终结符类别到整数的map映射

vector**<**TOKEN**>**Token**;** //最后生成的Tken串

### 3.2.3 算法

lexical\_analysis()主函数，完成分词并生成Token串：

初始化变量；

While 从文件中按行读入数据：

If 读入字符串 != “#”：

i=0：

While i< input\_str.size()：

next\_state(input\_str[i])；

If is\_KI()：

查询TOKEN\_k表；

If 在TOKEN\_k表中：生成关键字Token字

Else：

查询TOKEN\_i表；

If 在TOKEN\_i表中：生成标识符Token字

Else：

将当前分词加入TOKEN\_i表中，生成Token字

If is\_P()：

查询TOKEN\_p表；

If 在TOKEN\_p表中：生成界符Token字

Else：编译程序出错，break

If is\_C()：

查询TOKEN\_c表；

If 在TOKEN\_c表中：生成常数常量Token字

Else：

将当前分词加入TOKEN\_c表中，生成Token字

If is\_STRC()：

查询TOKEN\_strc表；

If 在TOKEN\_strc表中：生成字符串常量Token字

Else：

将当前分词加入TOKEN\_strc表中，生成Token字

If is\_CHARC()：

查询TOKEN\_charc表；

If 在TOKEN\_charc表中：生成字符常量Token字

Else：

将当前分词加入TOKEN\_charc表中，生成Token字

i++；

If 编译出错：报错，break

While end；

While end；

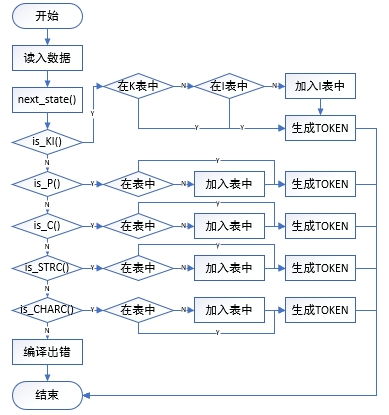
 该算法流程图如图3.2.3.1所示：

图3.2.3.1 lexical\_analysis()函数流程图

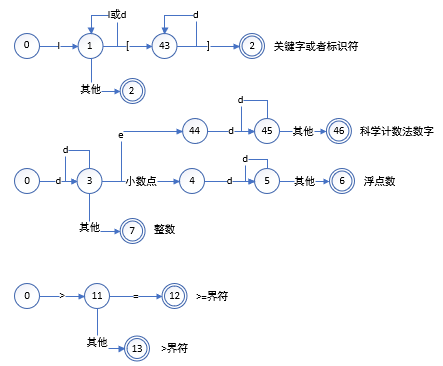
 所设计的确定有限自动机的部分状态转移函数如图3.2.3.2所示：

图3.2.3.2 有限自动机的部分状态转移函数

is\_KI()函数，判断分词是否属于关键字或标识符：

If state ==2：return true；

Else： return false；

is\_C()函数，判断分词是否属于常数常量：:

If state == 6或7: return true；

Else: return false；

is\_P()函数，判断分词是否属于界符：

If state == 9、10、12、13、15...38：return true；

Else: return false；

is\_STRC()函数，判断分词是否属于字符串常量：

If state == 42：return true；

Else：return false；

is\_CHARC()函数，判断分词是否属于字符常量：

If state == 40：return true；

Else：return false；

## 3.3语法分析

### 3.3.1功能

输入为词法分析器产生的token串流，include”grammar.h”的头文件以及”lexer.h”词法分析器的头文件，通过对文法的解析，自动生成First，Follow，Select等集合，进而有Select集合自动构建LL1分析表，根据自动生成的LL1分析表，进行语法分析，以及为后续的四元式产生及语义分析做前置准备工作。语法分析的同时可以利用符号表中的信息进行类型检查。

### 3.3.2 数据结构

struct Grammar {

vector<string> names;

set<int> Vt, Vn;

int Start = -1;

vector<pair<int, vector<int>>> Gram;

set<int> Vnull;

map<int, set<int>> First;

map<int, set<int>> Follow;

map<int, set<int>> Select;

int Id(string str);

void Get\_Set(set<int> &);

void Get\_Vn();

void Get\_Vt();

void Get\_Gram();

void Get\_Vnull();

bool Check\_Vnull(vector<int> &);

void Get\_First();

void Get\_Follow();

void Calc\_Select(int);

void Get\_Select();

void Print\_Follow();

void Print\_First();

void Print\_Select();

void Print\_Vnull();

};

### 3.3.3 算法

#### 求推出空的非终结符集合Vnull

**算法思想：**

非终结符X属于集合Vnull，当且仅当：

 基本情况：

 X -> 空

 归纳情况:

 X -> Y1 … Yn

 Y1, …, Yn 是n个非终结符，且都属于Vnull集

**伪代码：**

NULLABLE = {};

while (NULLABLE is still changing)

foreach (production p: X->β)

if (β == 空)

NULLABLE ∪= {X}

if (β == Y1 … Yn)

if (Y1属于NULLABLE && … && Yn属于NULLABLE)

NULLABLE ∪= {X}

**代码实现：**

void Grammar::Get\_Vnull() {

bool changing = true;

while(changing) {

changing = false;

for(auto p : Gram) {

vector<int> R = p.se;

if(names[R[0]] == "~" && !Vnull.count(p.fi)) {

Vnull.insert(p.fi);

changing = true;

}

else if(Check\_Vnull(R) && !Vnull.count(p.fi)) {

Vnull.insert(p.fi);

changing = true;

}

}

}

}

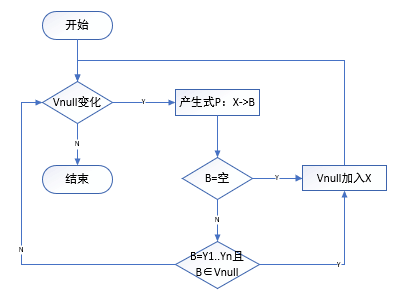


图3.3.3.1 推出空的非终结符集合Vnull算法流程图

#### 求First集合

**算法思想：**

 X -> a

 FIRST (X) ∪= {a}

 归纳情况：

 X -> Y1 Y2 … Yn

 FIRST (X) ∪= FIRST(Y1)

 if Y1NULLABLE, FIRST (X) ∪= FIRST(Y2)

 if Y1,Y2 NULLABLE, FIRST(X) ∪= FIRST(Y3)

**伪代码：**

foreach (nonterminal N)

FIRST(N) = {}

while(some set is changing)

foreach (production p: N->β1 … βn)

foreach (βi from β1 upto βn)

if (βi== a …)

FIRST(N) ∪= {a}

break

if (βi== M …)

FIRST(N) ∪= FIRST(M)

if (M is not in NULLABLE)

break;

**代码实现：**

void Grammar::Get\_First() {

bool changing = true;

while(changing) {

// Print\_First();

changing = false;

for(auto p : Gram) {

vector<int> R = p.se;

for(auto i : R) {

if(Vt.count(i)) {

if(!First[p.fi].count(i)) {

First[p.fi].insert(i);

changing = true;

}

break;

}

else if(Vn.count(i)) {

for(auto vt : First[i]) {

if(!First[p.fi].count(vt)) {

changing = true;

First[p.fi].insert(vt);

}

}

if(!Vnull.count(i)) break;

}

}

}

}

}

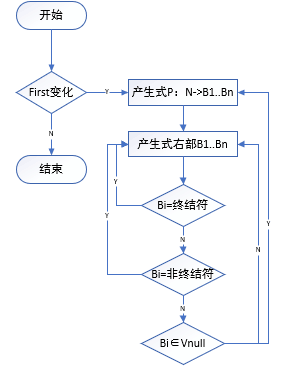


图3.3.3.2 求First集合算法流程图

#### 求Follow集合

**伪代码：**

foreach (nonterminal N)

FOLLOW(N) = {}

while(some set is changing)

foreach (production p: N->β1 … βn)

temp = FOLLOW(N)

foreach (βi from βn downto β1) // 逆序！

if (βi== a …)

temp = {a}

if (βi== M …)

FOLLOW(M) ∪= temp

if (M is not NULLABLE)

temp = FIRST(M)

else temp ∪= FIRST(M)

**代码实现：**

void Grammar::Get\_Follow() {

bool changing = true;

while(changing) {

changing = false;

for(auto p : Gram) {

set<int> temp\_f = Follow[p.fi];

vector<int> R = p.se;

reverse(R.begin(), R.end());

for(auto i : R) {

if(Vt.count(i)) {

temp\_f.clear();

temp\_f.insert(i);

}

else if(Vn.count(i)) {

for(auto vt : temp\_f) {

if(!Follow[i].count(vt)) {

changing = true;

Follow[i].insert(vt);

}

}

if(!Vnull.count(i)) temp\_f = First[i];

else {

for(auto vt : First[i]) {

temp\_f.insert(vt);

}

}

}

}

}

}

// Print\_Follow();

}

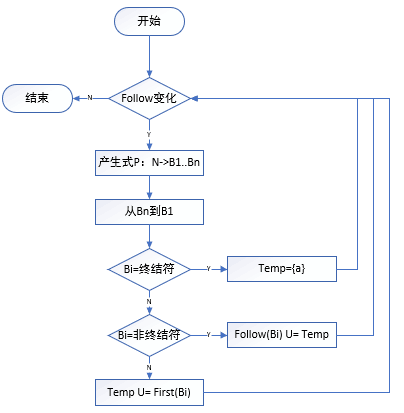


图3.3.3.3 求Follow集合算法流程图

#### 求Select集合

**伪代码：**

foreach (production p)

FIRST\_S(p) = {}

calculte\_Select(production p: N->β1 … βn)

foreach (βi from β1 to βn)

if (βi== a …)

Select(p) ∪= {a}

return;

if (βi== M …)

Select(p) ∪= FIRST(M)

if (M is not NULLABLE)

return;

Select(p) ∪= FOLLOW(N)

**代码实现：**

void Grammar::Calc\_Select(int i) {

pair<int, vector<int>> p = Gram[i];

vector<int> R = p.se;

for(auto j : R) {

if(Vt.count(j)) {

Select[i].insert(j);

return;

}

else if(Vn.count(j)) {

for(auto f : First[j]) {

Select[i].insert(f);

}

if(!Vnull.count(j)) return;

}

}

for(auto f : Follow[p.fi]) {

Select[i].insert(f);

}

}

void Grammar::Get\_Select() {

for(int i = 0; i < Gram.size(); ++i) {

Calc\_Select(i);

}

}

#### 构建分析表

**代码实现：**

void Parser::Get\_LL1Table() {

for(auto i : gram.Select) {

pair<int, vector<int>> p = gram.Gram[i.fi];

//p : product A -> ab

for(auto j : i.se) {

LL1Table[{p.fi, j}] = i.fi;

// printf("(%s, %s) = %d\n", gram.names[p.fi].c\_str(),

// gram.names[j].c\_str(), i.fi);

}

}

}

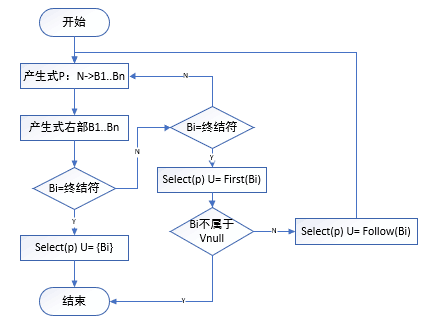


图3.3.3.4 求Select集合算法流程图

#### LL1分析过程

**伪代码：**

tokens[]; // all tokens

i=0;

stack = [S] // S是开始符号

while (stack != [])

if (stack[top] is a terminal t)

if (t==tokens[i++])

pop();

else error(…);

else if (stack[top] is a nonterminal T)

pop()

push(table[T, tokens[i]])

**代码实现：**

void Parser::LL1() {

Get\_LL1Table();

stack<int> S;

S.push(gram.Start);

int d = 0;

// printf("Start: %s\n", gram.names[gram.Start].c\_str());

while(!S.empty()) {

// printf("sz = %d\n", S.size());

int i = S.top(), j = tokens[d].Vt\_id;

// Debug\_LL1(S, j, d);

if(gram.Vt.count(i)) {

if(S.top() == j) {

d++;

S.pop();

}

else {

puts("Not Match!");

exit(0);

}

}

else if(gram.Vn.count(i)) {

S.pop();

// puts("yes");

if(LL1Table.count({i, j})) {

//puts("yes");

vector<int> rev;

for(auto x : gram.Gram[LL1Table[{i, j}]].se) {

// printf("%s", gram.names[x].c\_str());

if(gram.names[x] == "~") continue;

rev.pb(x);

}

// cout << endl;

reverse(rev.begin(), rev.end());

for(auto x : rev) S.push(x);

}

else {

puts("Don't have LL1Table !");

exit(0);

}

}

}

puts("exp is right!");

}

## 3.4语义分析（产生四元式）

### 3.4.1功能

包含了文法的头文件以及符号表的头文件，根据输入的语义动作进行语法制导翻译，通过语法分析中自动生成的LL1分析表，将动作当作文法右部变元同等处理，逆序压栈，当遇到语义动作的非终结符时，执行相应的语义动作，同时将产生的四元式输出并存储到vector<quat>的结构中，作为下一阶段四元式优化的输入部分。

### 3.4.2 数据结构

struct Parser {

map<pair<int, int>, int> LL1Table;

Grammar gram;

Lexer lex;

vector<Token> tokens;

stack<int> SEM;

stack<int> SToken;

int cnt\_t = 0; //for encode t1..t2

vector<Quat> quats;

Action acts[maxn];

vector<string> acttype{"GEQ", "PUSH", "ASSI", "IF", "ELSE", "ENDIF", "LAB", "GOTO"

"WE", "WH", "DO"};

SymTable symtbl;

void Get\_Action();

void Debug\_LL1(stack<int>, int, int);

void Get\_LL1Table();

void LL1();

void GEQ(int, int);

void PUSH(int, int);

void ASSI();

void IF();

void ELSE();

void ENDIF();

void LAB();

void GOTO();

void WE();

void WH();

void DO();

void Get\_Quats();

void Print\_Quat(Quat);

void Print\_Act(Action);

void Debug\_Quats(stack<int>, stack<Action>, int, int);

int Get\_Type(int, int);

};

### 3.4.3 算法

**产生四元式**

**代码实现**

void Parser::Get\_Quats() {

stack<int> S;

stack<Action> SACT;

S.push(gram.Start);

int d = 0;

while(!S.empty()) {

int i = S.top(), j = tokens[d].Vt\_id;

// printf("level = %d\n", tokens[d].level);

// Debug\_Quats(S, SACT, j, d);

if(i == -1) { //action is the top

S.pop();

if(SACT.empty()) { puts("SACT is null!"); exit(0); }

Action act = SACT.top(); SACT.pop();

// printf("act = %s\n", acttype[act.type-1].c\_str());

// printf("act = %d\n", act.type);

if(act.type == 1) { //GEQ

GEQ(act.object, act.tokenid);

}

else if(act.type == 2) {

PUSH(act.object, act.tokenid);

}

else if(act.type == 3) {

ASSI();

}

else if(act.type == 4) {

IF();

}

else if(act.type == 5) {

ELSE();

}

else if(act.type == 6) {

ENDIF();

}

else if(act.type == 7) {

LAB();

}

else if(act.type == 8) {

GOTO();

}

else if(act.type == 9) {

WE();

}

else if(act.type == 10) {

WH();

}

else if(act.type == 11) {

DO();

}

}

else if(gram.Vt.count(i)) {

if(S.top() == j) {

d++;

S.pop();

}

}

else if(gram.Vn.count(i)) {

S.pop();

if(LL1Table.count({i, j})) {

vector<int> rev;

int k = 0; //count the position

int t = LL1Table[{i, j}]; //the index of product

for(auto x : gram.Gram[t].se) {

if(gram.names[x] == "~") continue;

rev.pb(x);

if(acts[t].type == 0) continue;

if(k == acts[t].pos) {

rev.pb(-1);

int r = tokens[d].i, c = tokens[d].j;

acts[t].object = lex.Table[r][c];

acts[t].tokenid = d;

// Print\_Act(acts[t]);

// printf("t = %d\n", t);

SACT.push(acts[t]);

}

++k;

}

reverse(rev.begin(), rev.end());

for(auto x : rev) S.push(x);

}

}

}

}

## 3.5 文法设计

### 3.5.1功能

LL1 分析使用的文法，识别表达式，语法动作的定义及声明，输入为词法分析器产生的token串流，include”grammar.h”的头文件以及”lexer.h”词法分析器的头文件，通过对文法的解析，自动生成First，Follow，Select等集合。根据输入的动作序列以及相应的动作函数，进行四元式生成。

### 3.5.2 数据结构

struct Action {

int pos;

int type = 0;

int object;

int tokenid;

};

struct Quat {

int op, a, b, res;

Quat(int op = 0, int a = 0, int b = 0, int res = 0) :

op(op), a(a), b(b), res(res) { }

};

enum { GEQ = 1, PUSH, ASSI, IF, ELSE,

ENDIF, LAB, GOTO, WE, WH, DO };

### 3.5.3 算法

#### 动作函数

**代码实现**

void Parser::ASSI() {

int lop = SEM.top(); SEM.pop();

int res = SEM.top(); SEM.pop();

quats.push\_back(Quat(lex.Get\_Id("="), lop, -1, res));

Print\_Quat(quats.back());

}

void Parser::IF() {

int lop = SEM.top(); SEM.pop();

quats.push\_back(Quat(lex.Get\_Id("if"), lop, -1, -1));

Print\_Quat(quats.back());

}

void Parser::ELSE() {

quats.push\_back(Quat(lex.Get\_Id("el"), -1, -1, -1));

Print\_Quat(quats.back());

}

void Parser::ENDIF() {

quats.push\_back(Quat(lex.Get\_Id("ie"), -1, -1, -1));

Print\_Quat(quats.back());

}

void Parser::LAB() {

int res = SEM.top(); SEM.pop();

quats.push\_back(Quat(lex.Get\_Id("lb"), -1, -1, res));

SEM.push(res);

Print\_Quat(quats.back());

}

void Parser::GOTO() {

int res = SEM.top(); SEM.pop();

quats.push\_back(Quat(lex.Get\_Id("gt"), -1, -1, res));

Print\_Quat(quats.back());

}

void Parser::WE() {

quats.push\_back(Quat(lex.Get\_Id("we"), -1, -1, -1));

Print\_Quat(quats.back());

}

void Parser::WH() {

quats.push\_back(Quat(lex.Get\_Id("wh"), -1, -1, -1));

Print\_Quat(quats.back());

}

void Parser::DO() {

int lop = SEM.top(); SEM.pop();

quats.push\_back(Quat(lex.Get\_Id("do"), lop, -1, -1));

Print\_Quat(quats.back());

}

void Parser::GEQ(int op, int tokenid) {

int tr = SToken.top(); SToken.pop();

int tl = SToken.top(); SToken.pop();

int rop = SEM.top(); SEM.pop();

int lop = SEM.top(); SEM.pop();

// printf("level = %d\n", tokens[tl].level);

int typer = Get\_Type(lop, tl), typel = Get\_Type(rop, tr);

if(typer != typel) {

printf("Error(%d): Type not match : (%s, %s)!\n",

tokens[tl].level+1, lex.names[lop].c\_str(), lex.names[rop].c\_str());

exit(0);

}

string t = "t";

char num[10];

sprintf(num, "%d", cnt\_t);

t += num;

++cnt\_t;

quats.push\_back(Quat(op, lop, rop, lex.Get\_Id(t)));

SEM.push(lex.Get\_Id(t));

Token tt;

tt.level = tokens[tl].level;

tt.Vt\_id = tokens[tl].Vt\_id;

tt.type = tokens[tl].type;

tt.i = -1;

// tt.type = typel;

tokens.push\_back(tt);

SToken.push(tokens.size()-1);

Print\_Quat(quats.back());

}

void Parser::PUSH(int x, int tokenid) {

SEM.push(x);

SToken.push(tokenid);

}

#### 文法输入

**Vn** 非终结符

28

Prog\_

Prog

DivProg

VarDecl

IdenTab

ComSta

StaTab

AssiSta

AssiSta\_

IdenTab\_

StaTab\_

Exp

Exp\_

Item

Item\_

Fac

Type

IfSta

ElseSta

IfSta\_

LabSta

LabSta\_

LabSta'

WSta

Swh

Sdo

L

L\_

**Vt** 终结符

29

Iden

{

}

,

:

;

=

+

-

\*

/

Num

(

)

int

double

char

bool

if

else

goto

while

>

<

>=

<=

==

!=

#

**Productor** 产生式

51//产生式条数

左部 右部变元数目 右部Prog\_ 2 Prog #

Prog 5 Type Iden ( ) ComSta

ComSta 5 { VarDecl StaTab LabSta }

VarDecl 4 Type IdenTab ; VarDecl

VarDecl 1 ~

IdenTab 2 Iden IdenTab\_

IdenTab\_ 2 , IdenTab

IdenTab\_ 1 ~

StaTab 2 AssiSta StaTab\_

StaTab 2 ComSta StaTab\_

StaTab 2 IfSta StaTab\_

StaTab\_ 2 ; StaTab

StaTab\_ 1 ~

IfSta 5 if ( Exp ) IfSta\_

IfSta\_ 2 ComSta ElseSta

ElseSta 1 ~

ElseSta 2 else ComSta

AssiSta 3 AssiSta\_ = Exp

AssiSta\_ 1 Iden

L 2 Item L\_

L\_ 3 + Item L\_

L\_ 3 - Item L\_

L\_ 1 ~

Item 2 Fac Item\_

Item\_ 3 \* Fac Item\_

Item\_ 3 / Fac Item\_

Item\_ 1 ~

Fac 1 Num

Fac 1 Iden

Fac 3 ( Exp )

Type 1 int

Type 1 double

Type 1 char

LabSta 3 LabSta\_ StaTab LabSta

LabSta\_ 2 LabSta' :

LabSta' 1 Iden

LabSta 3 goto Iden ;

LabSta 1 ~

StaTab 2 WSta StaTab\_

WSta 3 Swh Sdo ComSta

Swh 1 while

Sdo 3 ( Exp )

Exp 2 L Exp\_

Exp\_ 3 > L Exp\_

Exp\_ 3 < L Exp\_

Exp\_ 3 >= L Exp\_

Exp\_ 3 <= L Exp\_

Exp\_ 3 == L Exp\_

Exp\_ 3 != L Exp\_

Exp\_ 1 ~

Type 1 bool

Action语义动作

23 //语义动作条数

产生式编号 动作种类 动作位于右部的位置

14 4 3

15 6 1

17 5 0

18 3 2

19 2 0

21 1 1

22 1 1

25 1 1

26 1 1

28 2 0

29 2 0

35 7 1

36 2 0

37 8 1

40 9 2

41 10 0

42 11 2

44 1 1

45 1 1

46 1 1

47 1 1

48 1 1

49 1 1

## 3.6符号表设计

### 3.6.1功能

存储用户定义变量，数据类型，函数，并基于作用域的语义检查功能。

收集符号属性，编译程序扫描说明部分收集有关标识符的属性，并在符号表中建立符号的相应属性信息。上下文语义的合法性检查的依据 同一个标识符可能在程序的不同地方出现，而有关该符号的属性是在这些不同情况下收集的。通过符号表中属性记录可进行相应上下文的语义检查。

### 3.6.2数据结构

enum {INT, DOUBLE, CHAR, BOOL, ARRAY, FUNCTION, STRUCT};

struct Record {

int name;

int type;

int range;

int line = 0;

int ival;

double dval;

char cval;

bool bval;

int low, up, ctp, clen;

int level = 0, foff, fn = 0, param, entry;

void Init\_Var(double v) {

switch(type) {

case INT : ival = v; break;

case DOUBLE : dval = v; break;

case CHAR : cval = v; break;

case BOOL : bval = v; break;

}

}

void Init\_Arr(int a, int b, int c, int d) {

type = ARRAY;

low = a, up = b, ctp = c, clen = d;

}

void Init\_Fun(int a, int b, int c, int d) {

type = FUNCTION;

foff = a;

fn = b;

param = c;

entry = d;

}

};

struct SymTable {

Lexer lex;

vector<map<int, vector<Record>>> Infos;

vector<string> lines;

set<string> Def{"int", "double", "char", "bool"};

vector<Record> synbl;

vector<Record> arr;

map<string, string> mT;

map<string, int> mR;

vector<pair<pair<string, string>, int>> vp;

map<string, int> moffset;

vector<pair<string, int>> data;

void Get\_SymTable();

void Get\_NewRecord(string, int, vector<Record> &, int, int);

void Pop\_Record(int, map<int, vector<Record>> &);

void Print\_Table(map<int, vector<Record>>);

void Print\_RegularTable(int &);

};

## 3.7四元式优化

### 3.7.1 功能

将语法制导翻译生成的四元式按照一定的规律划分成不同的基本块，然后再依次分别对基本块内的四元式通过把其转化为dag结构进行优化，能使常值表达式节省、公共表达式节省、删除无用赋值，该部分包含一个optimizequa.h头文件和一个optimizequa。cpp文件。

### 3.7.2 数据结构

struct token {

string name;

string type; //2为临时变量，3为数字，1为非临时变量

};

struct Quaternion { //四元式结构体

int block; //基本块标识

int type; //类型-0 1 2 3

string op; //操作

token num1; //操作数1

token num2; //操作数2

token ans; //结果

};

struct Quaternion qua[20]; //四元式数组

struct Node { //DAG节点

int left; //左子节点标识

int right; //右子节点标识

int id; //节点标识

string op; //操作

token m\_sign; //主标记

token sign[10]; //附加标记

};

struct DAG{ //DAG

int num=0;

Node node[20];

}dag;

### 3.7.3 算法

#### 基本块的划分

Divide() //划分基本块

{

num=1;

While(四元式操作符不为空)

If(不是特殊四元式或op为lb wh )

四元式的所在块=num

If(op为if el ie do we gt)

四元式的所在块=num num++

If(op为if el ie do we gt) num--

return num

}

#### 基于DAG的四元式优化

optimizequa(int block\_num) //优化四元式函数，构建dag

{

for(基本块的块数) //对每个块进行优化

while(四元式的操作符号不为空且四元式在当前基本块内)

switch(四元式的类型)

case 0: //（= ,B, ,A）

if(节点B不在)

建立B叶子节点

A附加到B上

case 2: //双目操作（op, B ,C ,A）

if(B和C的节点不存在)

建立B C叶子节点

if(四元式的运算符是否为\*/+ -且两个操作数为数字)

p=B op C

if(B C是此次新建立的节点)

删除节点

If(p的节点不存在)

建立p节点

Else

if (在DAG中查找,没有符合A=B op C的节点n)

建立符合A=B op C的n节点

A附加到n节点上

case 3: //特殊四元式（if else while goto等）

跳过

makequa(基本块); //遍历dag生成优化后的四元式

对dag清零

}

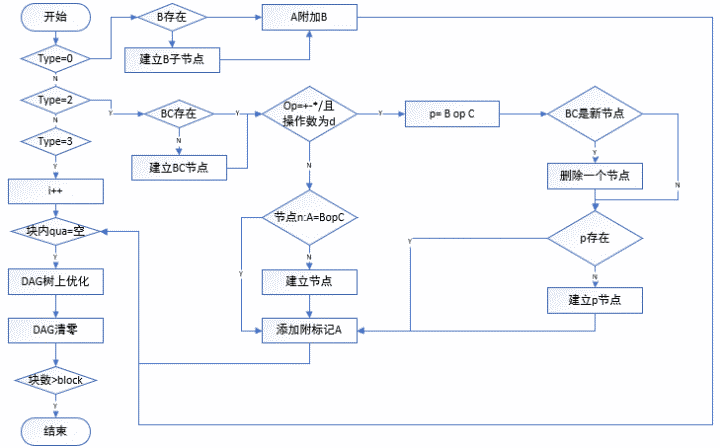


图3.7.3.1 基于DAG的四元式优化算法流程图

makequa(基本块) //遍历dag生成优化后的四元式

{

for(dag节点的个数)

If(是叶子节点)

If(主标记为数字或非临时变量，附标记有非临时变量)

生成四元式 (=, 主标记, ,附标记的非临时)

else

生成四元式 (op, 左子节点的主标记, 右子节点的主标记, 主标记)

}

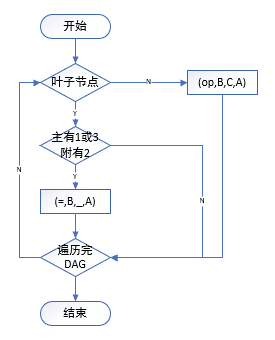


图3.7.3.2 遍历DAG生成优化后的四元式算法流程图

addition( n, A) //把标记A添加到节点n上，删除无用标记，调整顺序

{

If(有标记为A的节点)

在节点上删除附加标记A

If(节点n的主标记不为空)

在节点n上添加附标记A

else 在n上添加主标记A

If(主标记为临时变量，附标记为非临)

主标记与附标记互换位置

}

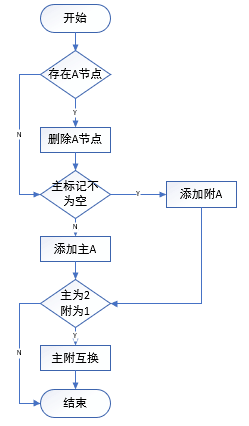


图3.7.3.3 addition()算法流程图

## 3.8 活跃信息生成

### 3.8.1 功能简介

活跃信息生成模块将划分完基本块的四元式作为输入，生成对应每个基本块的变量活跃信息，以便实现目标代码的生成。该部分包含一个头文件：optimizequa.h 。以基本块为单位，先对四元式中的变量初始化，临时变量为N，非临时变量为Y，再逆序填写四元式的各个变量的活跃信息。

### 3.8.2 数据结构

Token结构体的定义：

struct token

{

string name;

string type; //1-非临时变量，2-临时变量，3-数字

token(string a="",string b="")

{

name = a; type = b;

}

};

struct ACT\_INF //四元式的活跃信息

{

string num1;

string num2;

string ans;

ACT\_INF(string a="", string b="", string c="")

{

num1 = a; num2 = b; ans = c;

}

};

struct ACT\_SYNBL //活跃信息表的元素

{

string name;

string act\_inf;

ACT\_SYNBL(string a="", string b="")

{

name = a; act\_inf = b;

}

};

struct Quaternion //四元式

{

int block; //所属快的标识

int type; //类型，0-赋值语句 2-双操作数 3-if,else,wh...

string op; //操作

token num1; //操作数1

token num2; //操作数2

token ans; //结果

Quaternion(int a,int b, string c, token d, token e, token f)

{

block = a; type = b; op = c; num1 = d; num2 = e; ans = f;

}

};

vector< vector<ACT\_INF> >Act\_inf; //四元式的变量活跃信息

vector< vector<Quaternion> >qua; //四元式

vector< vector<ACT\_SYNBL> >Act\_synbl; //活跃信息表

### 3.8.3 算法

get\_act\_inf(vector< vector<Quaternion> >qua)函数：

//先初始化活跃信息表

For i=0;i<qua.size();i++：

初始化活跃信息表；

If 四元式num1不为空且不是数字时：

If num1 不在活跃信息表中：

If num1是临时变量：表中填入(num1.N)；

If num1是非临时变量：表中填入(num1,Y)；

If num2 不在活跃信息表中：

If num2是临时变量：表中填入(num2.N)；

If num2是非临时变量：表中填入(num2,Y)；

If ans 不在活跃信息表中：

If ans是临时变量：表中填入(ans.N)；

If ans是非临时变量：表中填入(ans,Y)；

For end；

//填四元式变量的活跃信息

For i=0;i<qua.size();i++：

For j=qua[i].size()-1;j>=0;j--：

For k=0;k<Act\_synbl[i].size();k++：

If num1== Act\_synbl[i][k]：

填num1活跃信息，更新活跃信息表；

If num2== Act\_synbl[i][k]：

填num2活跃信息，更新活跃信息表；

If ans== Act\_synbl[i][k]：

填ans活跃信息，更新活跃信息表；

For end；

For end；

将所得的活跃信息反转；

For end；

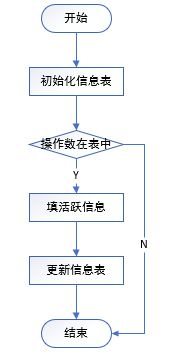
 该函数的流程图如图3.8.3.1所示：

图3.8.3.1 get\_act\_inf()函数的流程图

## 3.9 目标代码生成

### 3.9.1 功能

目标代码生成模块将经过DAG优化、划分完基本块的四元式和对应的变量活跃信息作为输入，可以生成表达式语句、条件语句if-else、循环语句while和跳转语句goto的目标代码（汇编语言），生成的目标代码有两种形式：（1）编译原理课上讲的基于单寄存器的类汇编语言代码（2）包含变量的内存分配信息的基于单寄存器的汇编语言代码，该部分包含一个头文件：optimizequa.h 。

### 3.9.2 数据结构

四元式变量活跃信息结构体定义如下：

struct ACT\_INF //四元式的活跃信息

{

string num1;

string num2;

string ans;

ACT\_INF(string a="", string b="", string c="")

{

num1 = a; num2 = b; ans = c;

}

};

四元式的结构体定义如下：

struct Quaternion

{

int block; //所属快的标识

int type; //类型，0-赋值语句 2-双操作数 3-if,else,wh...

string op; //操作

token num1; //操作数1

token num2; //操作数2

token ans; //结果

Quaternion(int a,int b, string c, token d, token e, token f)

{

block = a; type = b; op = c; num1 = d; num2 = e; ans = f;

}

};

vector< vector<ACT\_INF> >Act\_inf; //四元式的变量活跃信息

vector< vector<Quaternion> >qua; //四元式

vector< vector<ACT\_SYNBL> >Act\_synbl; //活跃信息表

vector<string>objcode; //对应PPT上的目标代码

vector<string>Objcode; //包含变量内存分配情况的目标代码

stack<string>SEM; //保存待返填地址

map<string, int>mp;

vector< pair<string, int> >Data;

### 3.9.3 算法

生成目标代码to\_objcode函数：

初始化目标代码基本信息；

For i=0;i<qua.size();i++：

For j=0;j<qua[i].size();j++：

//两个操作数不可互换位置

If op== - / != >= <= > <： //(op,A,B,C)

If RDV==0：code(LD AX,A)，CODE(LD AX,A的符号表偏移量)；

Else：

If RDV==D：

If D(Y)：

code(ST AX,D；LD AX,A)；

CODE(ST AX,D的符号表偏移量)；

CODE(LD AX,A的符号表偏移量)；

Else：code(LD AX,A)，CODE(LD AX,A的符号表偏移量)；

CODE(op,AX,B)；

//两个操作数可以互换位置

Else if op== + \*： //(op,A,B,C)

If RDV==D：

If D(Y)：

code(ST AX,D)；CODE(ST AX,D的符号表偏移量)；

Else：

If RDV==A：

更新寄存器变量的活跃信息；

code(op,A,B);

CODE(op,A的符号表偏移量,B的符号表偏移量)；

Else if RDV==B:

更新寄存器变量的活跃信息；

code(op,B,A)；

CODE(op,B的符号表偏移量,A的符号表偏移量)；

Else:

code(LD AX,A)；

code(op,A,B)；

CODE(LD AX,A的符号表偏移量)；

CODE(op,A的符号表偏移量，B的符号表偏移量)；

Else if op==if： //(if,A,\_,\_)

If RDV==0：

code(LD AX,A；FJ AX,?)；

CODE(LD AX,A的符号表偏移量；FJ AX,?)；

PUSH(P)；

Else if RDV==A：

If A(Y)：

code(ST AX,A；FJ AX,?)；

CODE(ST AX,A的符号表偏移量；FJ AX,?)；

Else：

code(FJ AX,?)；CODE(FJ AX,?)；

PUSH(p)；RDV=0；

Else if RDV==B： //(B!=A)

If B(Y)：

code(ST AX,B；LD AX,A；FJ AX,?)；

CODE(ST AX,B的符号表偏移量)；

CODE(LD AX,A的符号表偏移量)；

CODE(FJ AX,?)；

Else:

code(LD AX,A；FJ AX,?)；

CODE(LD AX,A的符号表偏移量)；

CODE(FJ AX,?)；

PUSH(p)；RDV=0；

Else if op==el： //(el,\_,\_,\_)

If RDV==X && X(Y)：

code(ST AX,X)；

CODE(ST AX,X的符号表偏移量)；

POP(p’)；BACK(p’,p+2)；

code(JMP ?)；

CODE(JMP ?)；

Else if op==ie： //(ie,\_,\_,\_)

If RDV==X && X(Y)：

code(ST AX,X)；

CODE(ST AX,X的符号表偏移量)；

POP(p’)；BACK(p’,p+1)；

code(JMP ?)；

CODE(JMP ?)；

Else if op==wh： //(wh,\_,\_,\_)

PUSH(p)；

Else if op==do： //(do,A,\_,\_)

If RDV==0：

code(LD AX,A；FJ AX,?)；

CODE(LD AX,A的符号表偏移量；FJ AX,?)；

PUSH(p)；

Else if RDV==A：

If A(Y)：

code(ST AX,A；FJ AX,?)；

CODE(ST AX,A的符号表偏移量；FJ AX,?)；

Else：code(FJ AX,?)；CODE(FJ AX,?)；

PUSH(p)；RDV=0；

Else if RDV==D：

If D(Y)：

code(ST AX,D；LD AX,A；FJ AX,?)；

CODE(ST AX,D的符号表偏移量)；

CODE(LD AX,A的符号表偏移量；FJ AX,?)；

Else：

code(LD AX,A；FJ AX,?)；

CODE(LD AX,A的符号表偏移量；FJ AX,?)；

PUSH(p)；RDV=0；

Else if op==we： //(we,\_,\_,\_)

If RDV==X && X(Y)：

code(ST AX,X)；

CODE(ST AX,X)；

RDV=0；

POP(p’)；BACK(p’,p+2)；

POP(p’)；

code(JMP ?)；

CODE(JMP ?)；

Else if op==lb： //(lb,A,\_,\_)

code(A：)；

CODE(A：)；

Else if op==gt： //(gt,A,\_,\_)

code(GOTO A)；

CODE(FOTO A)；

Else if op== =： //(=,A,\_,B)

If RDV==C：

If C(Y)：

code(ST AX,C)；

CODE(ST AX,C的符号表偏移量)；

code(LD AX,C)；

CODE(LD AX,C的符号表偏移量)；

For end；

If 寄存器中变量活跃：code(ST AX,R)；CODE(ST AX,R)；

For end；

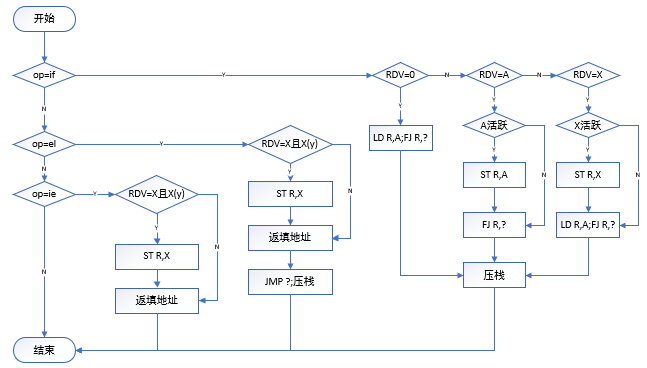
 该模块的流程图如下图所示：

图3.9.3.1 目标代码生成if-else语句流程图

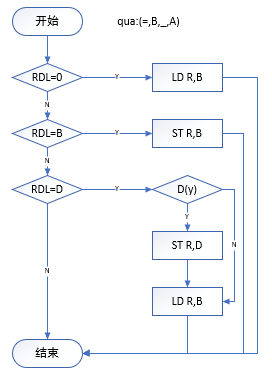


图3.9.3.2 目标代码生成赋值语句流程图

# 4.程序设计与实现

## 4.1程序流程图

总体程序流程图大致如下：

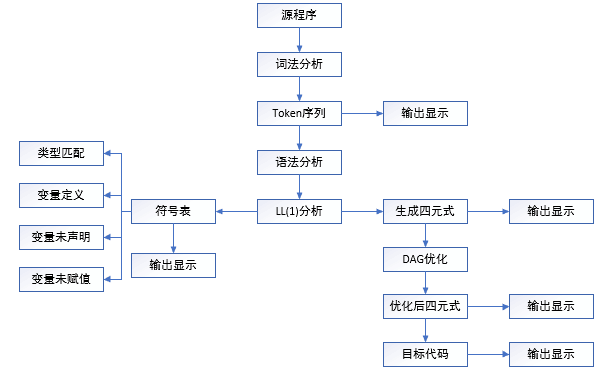


图4.1.1 总体程序流程图

具体细节请参照前面算法说明

## 4.2程序说明

编译器前端使用了以语法分析为中心的方法。

词法分析器作为一个函数提供给语法分析器，函数在每次调用时返回一个token。在遇到文件结尾时，前端会返回一个标识符表示文件结尾，语法分析到此处停止。

在语法分析的实现上，使用递归下降分析法识别结构较为复杂的语法结构，如，函数，if-else, while 等语法成分。由于这些部分较为稳定，即在设计完成后，不会进行较大的修改。在递归下降的语义分析部分，将语法动作插入到产生式中。并在分析过程中填写符号表，为了简化数据结构，在符号表中添加了四元式区。这样，只需将符号表传递给后端，降低了编码复杂性。

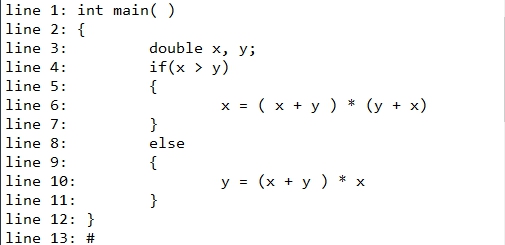
LL1分析法负责识别结构较为简单，但是需要处理较多细节的部分，如算数表达式，逻辑表达式。由于此处在编码时可能涉及到较多的修改，为了降低修改语法所带来的大量代码修改。在此处实现了LL1分析表的自动生成，修改语法仅需要修改储存文法的文件即可，不需要对分析器本身进行修改。大量减少了修改，新建文法所带来的编码工作，且易于调试。而且，分析表的自动生成可以实现复杂的算数逻辑表达式，能够使语言具有更强的可用性。在语义分析上采用了属性文法的方法，在归约时执行语义动作。

编译器后端包括四元式优化和目标代码生成两个部分。

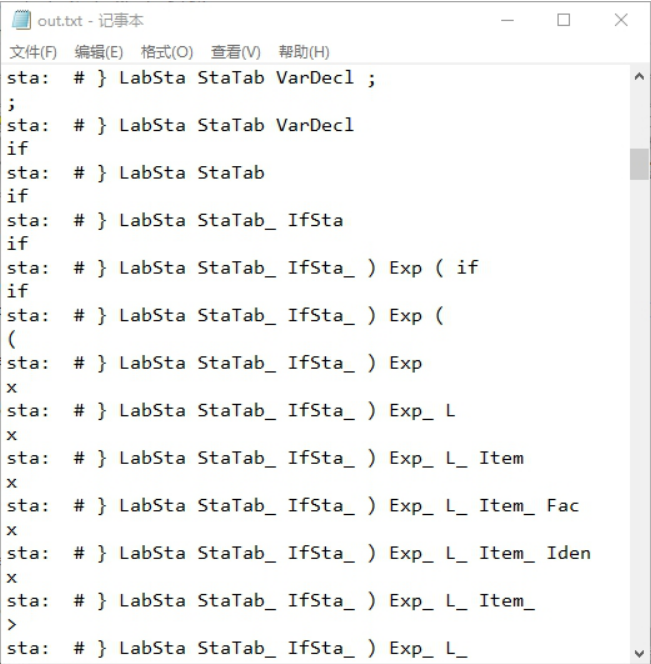
后端大致思路为，将前端四元式用接口函数转换为后端待处理四元式的数据结构（大体相同，接口函数只处理细节），经过以基本块为单位的优化算法实现四元式的优化，并且将标志活跃信息的算法动作插入划分基本块算法后。以得到含有活跃信息的优化后的四元式序列，接下来目标代码生成算法处理此序列得到目标汇编指令。其中，优化算法使用DAG算法，目标代码生成使用多寄存器分配方法并且处理了很多算法细节使算法更加健壮完备，生成的代码是基于IntelX86的汇编指令。

## 4.3实验结果

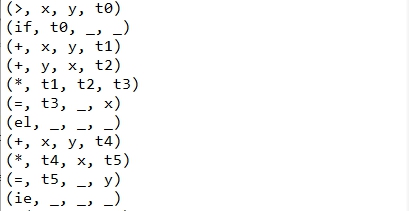
源程序：



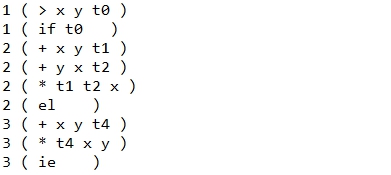
语法分析过程：



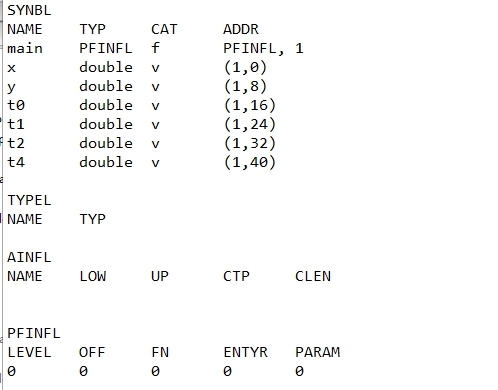
产生四元式序列：



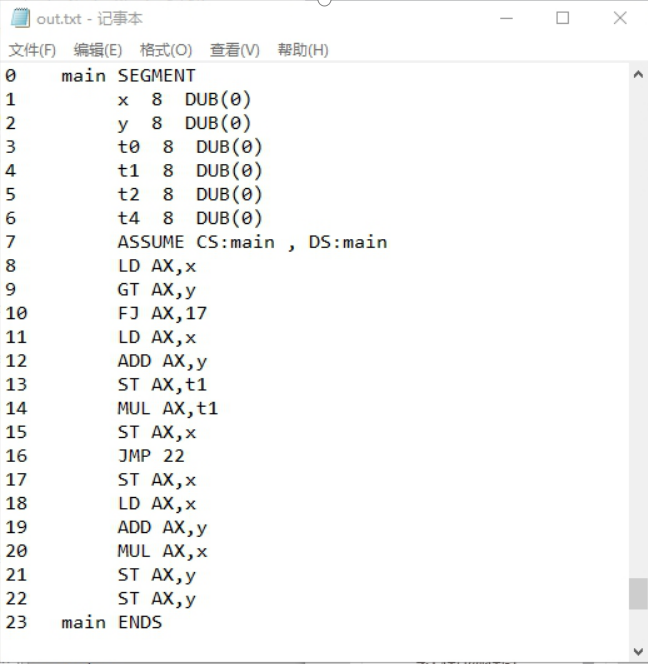
优化后四元式序列：



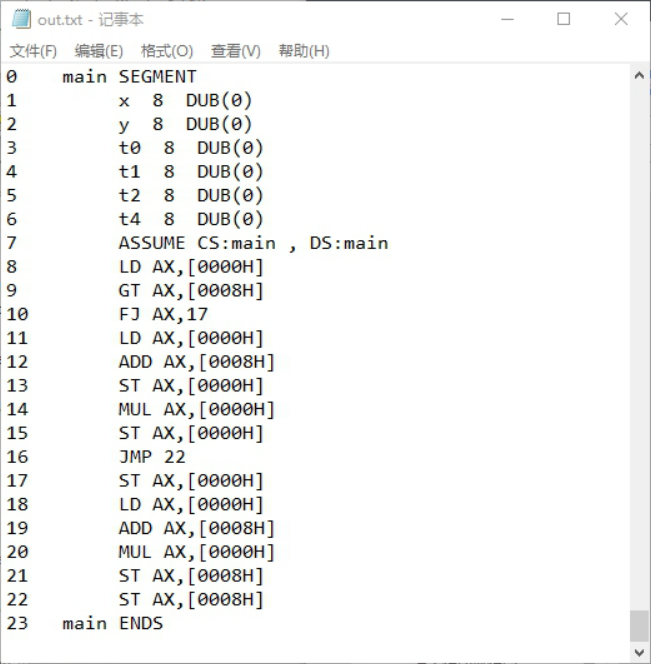
符号表：



ppt目标代码：



8086目标代码：



# 5.结论

编译原理课程设计，实现一个编译器（前端及后端）完成。

# 6.参考文献

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、美 Alfred V.Aho Ravi Sethi Jeffrey D. Ullman著.李建中，姜守旭译.《编译原理》.北京：机械工业出版社.2003.

3、美 Kenneth C.Louden著.冯博琴等译.《编译原理及实践》.北京：机械工业出版社.2002.

# 7.收获、体会和建议

陈弘超：

这一次作为队长，让我有了很多的收获。首先是自己实践了对于编译课程的理解，从文法开始入手，一点一点地探索编译器的整体。我自己在老师的基础上重新写了文法，对文法进行了扩展，在这个过程中我做了很多的尝试，也发现了诸多的问题，为此我查阅了很多的资料，也扩充了自己的知识储备，让自己能更好的理解一个编译器的工作流程和细节。在做实验时，我首先有了大致的思路，但是在实际上输入的时候，由于思路在执行上出现了问题，循环的嵌套，条件的判断出现了一些问题，导致我的思路经常被阻断，在执行的时候总是卡壳。而我又想一气呵成的把代码打完，所以我并没有怎么加注释。在这种情况下我就经常被自己之前的代码干扰，不确定自己的进度，要经常翻看前面的代码，并且在检查bug的时候会选择一部分一部分执行函数，但是因为没有注释，所以在执行的时候，要完整的看完整个函数，有的时候还会忘了自己之前的思路，导致自己要花很多时间去理解之前自己写下的东西，无疑是自己给自己添麻烦，这也导致了顺下来需要很多的时间，让我的工作效率十分的低，还有些拖进度。好在我后来及时为代码添加了注释，解决了这个在之前出现的问题，也让我意识到及时添加注释的重要性和必要性。

最重要的是，我对面向对象的编程思想有了更深的体会，明白了private与public的使用上的作用，也感受到了定义接口的重要意义，值得反思的是，我应该在写程序前，先做好完整的规划，这样就不会有那么多的修改了，毕竟很多时候，没有机会再返回修改前面步骤的结果。

此外，在当了队长之后，我学会先给整个课设一个总体的规划，给每个人适当的分工，例如给曹越同学分配优化，给张亚强同学分配图，给刘键同学分配四元式等。大家有差不多的工作量，我们几乎是同时开始各自的工作，在一个人有困难时我们也会一起解决困难。最后集体合作，把大家的代码捏到一起，再经过调整之后，成功编译的时候，我十分的喜悦。作为队长让我学会了从总体上看一件事情，学会了和大家分工合作，也增进了我们彼此之间的感情，对我来说受益匪浅。

曹越：

在这次编译原理课程设计中，我主要负责后端的大部分内容，把在课堂上学到的东西运用到了实践中，化抽象为具体，让我对编译器的整体运行有了更加深刻的理解。在实验过程中我参与了小组文法与算法的讨论并通过代码实现了基于dag的四元式的优化、基本块的划分、目标代码的生成（PPT）、目标代码的生成（8086）。  
 在编写后端程序的过程了，因为接触较少且算法需要自行设计，所以许多看起来较为简单的问题，因为思维不够缜密，在执行的时候也出现了很多的困难。例如结构体定义不全，临时增加许多变量，在定义的时候出现了不知道如何处置数据类型，结构的处理不够系统等问题，为此我通过查阅了资料与同学交流，在接下来解决了这个问题。在进行目标代码生成这个模块的设计时，一开始我并没有了解它真正额意义，于是设计的效果很不好，后期通过与老师的沟通，我使用了符号表，生成了8086形式的目标代码。在后期对特殊样例进行测试时，因为我自身考虑问题不够周到，设计的程序对一些特殊情况的处理会出现错误，我也逐一进行分析与修改，使我积累了很多的经验。  
 这次课程设计也使我的编码能力有了很大提高，而且是我更加深刻的理解了团队合作以及软件总体架构的重要性。在较短的时间内想要完成较多的编码任务，只有通过团队合作的方式。而要想使团队合作能够顺利进行，就需要使用模块化的的设计方法，合理安排软件的结构。总体上说，这次课程设计加深了我对理论的理解，增强了我的编码能力，让我积累了更多的有关于团队合作的经验，使我受益匪浅。

张亚强：

通过本次课设，我全面系统的了解了编译原理程序构造的原理和基本实现方法，并从词法分析器入手，为了便于词法分析器的修改和扩充，选择使用自动机的方法，从设计自动机到实现自动机状态转移函数过程中遇到了很多问题，对问题逐个击破，再到后面的符号表的生成和四元式活跃信息的生成，成功地作为队伍中的一员，实现了编译器前端和后端的生成。这让我在课堂上学习到的知识得到了充分的运用，也能找到在课堂上发现不了的错误，同时也能锻炼自己的编程能力。

与此同时我还锻炼了自己的思考问题的能力，在生成符号表之前，我自己进行了一定的知识学习和补充，观看了一些教学和视频，对于符号表的构建和使用还是不能有很好的掌握。然后我自己进行了深入的思考，符号表在编译器的运行中主要参与什么环节，在这之中起到的主要功能是什么，需要的输入是什么，输出的结果是怎么样子的，在和同组搭档和别的组的同学交流之后，我还是没能得到很好的答案。然后我在课上询问了肖桐老师，老师和我说“符号表的功能主要在后端部分，在目标代码生成阶段需要通过符号表来得到标识符的偏移量等信息”。在这之后我重新好好思考了一下，得到了自己的感悟。最终将其付诸实现，虽然最后实现的结果也不是太全面，没能真正实现多函数嵌套的活动记录，但是还是学到了很多的东西，也锻炼了自己的能力，懂得了及时和同学老师交流的重要性。

刘键：

这两周的编译原理课程设计让我明白了计算机中编译程序的实现过程，通过词法语法语义优化生成目标代码等一系列过程。并且学到了团队合作的重要性们对于这种大型的工程，需要几个人分工完成，一个人的精力是有限的，在这种分工合作的模式下，我们每个人都能更好的接受自己的任务并将其落实，最后做到捏合如一，拥有满满的自豪感。本次实验我负责四元式的生成和准备测试样例。 在课程设计开始时，我对课程设计的内容感到无从下手，因为基础知识的不扎实，我在编程的过程中经常需要翻看书籍和老师给的ppt，发现自己的知识没有很好的掌握，没有成为一个体系，在编程时总会出现这样那样的问题，在输出时出现格式上的问题，不过在我的努力调整和队友的帮助下，我克服了难关，成功完成了自己的任务。我们比较顺利地完成了任务，这超乎了我的想象，在这个过程中我们都曾饱受各种bug的折磨。

这次编译原理课程设计让我受益匪浅，不仅巩固了我平时在课堂上学到的知识，还为接下来的考试打下了坚实的基础。最后衷心的感谢老师们在实验过程中对我们的悉心指导，没有老师的耐心答疑我们很难这么顺利地完成课程设计。