

《编译技术》课程设计 文 档

学号： _____39061416_____

姓名： _____黄建宇_____

2012 年 1 月 1 日

目 录

目 录.....	2
一. 需求说明.....	3
1.实验要求	3
2.实现方案	3
3.文法说明	3
4.文法变更	10
5.语法图	13
二. 详细设计.....	17
1. 程序结构	17
2. 类/方法/函数功能	19
3. 调用依赖关系	23
4. 符号表结构	24
5. 运行栈结构	26
6. 出错信息编号及描述	27
7. 保留字、标识符定义	30
8. 原始四元式	30
9. 优化后四元式	31
10. 优化种类及优化模式	32
11. 中间结果显示	32
三. 操作说明.....	32
1. 运行环境	32
2. 操作步骤	33
四. 测试报告.....	37
4.1.正确程序演示	37
4.2.错误程序演示	46
4.3.优化程序演示	53
五. 总结感想.....	67

一. 需求说明

【说明实验的要求及实施方案等】

1.实验要求

目标：实现编译器，生成 X86 汇编或 MIPS 汇编

文法： 扩充 PL/0 文法

优化：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）；

数据流分析（通过活跃变量分析，或利用定义-使用链建网等方法建立冲突图）；

其它优化自选；

代码生成时合理利用临时寄存器（临时寄存器池），并能生成较高质量的目标代码；

中间代码：四元式

目标码：32 位 X86 汇编或 MIPS 汇编（任选一）

2.实现方案

使用 C++ 语言编写，生成 x86 汇编。

3.文法说明

<程序>::= <分程序>.

作用：说明<程序>的构成。特别注意：主程序是由<分程序>在结尾加一个'.'构成。

<分程序>::= [<常量说明部分>**][**<变量说明部分>**][**<过程说明部分>**][**<函数说明部分>**]**<复合语句>****

作用：说明<分程序>的构成。

限定条件：

<常量说明部分><变量说明部分>可以出现 1 次，也可以不出现，<过程说明部分>，<函数说明部分>还可以出现 0 次或者多次，且<过程说明部分>和<函数说明部分>出现顺序可以任意组合，<过程说明部分>或者 <函数说明部分>均可以先出现。

<常量说明部分>::= **const<常量定义>{<常量定义>;}**

作用：说明<常量说明部分>的构成。

限定条件：

<常量说明部分>若出现，只可以出现在<分程序>最先开始的部分，且以“const”作为规则开始符，以‘;’做文法的结束符。

句子示例:

const a=1,b='b',c=65535;

<常量定义>::= <标识符>= <常量>

作用：说明<常量定义>的构成。

限定条件：<常量定义>只可以出现在<常量说明部分>。

句子示例:

x='a'

<常量>::= [+|-]<无符号整数>|<字符>

作用：说明<常量>的构成。<常量>由‘+’/‘-’<无符号整数>或<字符组成>; ‘+’, ‘-’有无均可。

限定条件：<常量>只可以出现在<常量定义>部分和<情况常量表>部分。

句子示例:

-65535

<字符>::= '(<字母> | <数字>)'

作用：说明<字符>的构成。<字符>是由单引号中间加入一个<字母>或<数字>构成的。

句子示例:

'e'

<字符串>::= "{(<字母> | <数字>| ')}"

作用：说明<字符串>的构成。<字符串>是由双引号中间加入一个或多个<字母>或<数字>或空格构成的。

句子示例:

"buaa_scse"

<无符号整数>::= <数字>{<数字>}

作用：说明<无符号整数>的构成。<无符号整数>由一个或多个<数字>构成。

句子示例:

65535

<标识符>::= <字母>{<字母>|<数字>}

作用：说明<标识符>的构成。<标识符>由<字母>开始，后面跟零个或多个<字母>或<数字>

限定条件:标识符必须以<字符>开始。

示例:

buaav5 正确

<变量说明部分>::= var <变量说明>;{<变量说明>;}

作用：说明<变量说明部分>的构成。

限定条件：<变量说明部分>如果出现，只可以出现在<分程序>开始的部分或紧跟在<常量说明部分>之后，且以“var”作为规则开始符，以‘;’做文法的结束符。

<变量说明部分>示例：

```
var    x:integer;
        w,t:char;
        a,b: array[100] of integer;
```

<变量说明>::= <标识符>{,<标识符>}:<类型>

作用：说明<变量说明>的构成。<变量说明>由 1 个或多个<标识符>开始，中间以‘,’间隔，后面紧跟‘:’和<类型>

示例：

```
x:integer;
w,t:char;
a,b: array[100] of integer;
```

<类型>::= <基本类型> | array '<无符号整数>[' of <基本类型>

作用：说明<类型>的构成。<类型>分为<基本类型>和数组类型

<基本类型>::= integer | char

作用：说明<基本类型>的构成。

<过程说明部分>::= <过程首部><分程序>{;<过程说明部分>;}

作用：说明<过程说明部分>的构成。

句子示例：

```
procedure swap();
var temp:integer;
begin
    temp:=x;
    x:=y;
    y:=temp
end;
```

<函数说明部分>::= <函数首部><分程序>{;<函数说明部分>;}

作用：说明<函数说明部分>的构成。

句子示例：

```
function mod(var fArg1,fArg2:integer):integer;
```

```
begin
    fArg1:=fArg1-fArg1/fArg2*fArg2;
    mod:=fArg1
end;
```

<过程首部>::= procedure<标识符>('['<形式参数表>']');

作用：阐明<过程首部>的构成。

限定条件：

<标识符>作为过程名。

<过程首部>以 **procedure** 开始，以 ';' 结束。

<形式参数表>可有可无。

示例：

```
procedure swap();
```

<函数首部>::= function <标识符>('['<形式参数表>']'): <基本类型>;

作用：说明<函数首部>的构成。

限定条件：

<标识符>作为函数名。

<函数首部>以 **function** 开始，以 ';' 结束。

<形式参数表>可有可无。

<基本类型>说明函数返回值的类型且限定为基本类型。

示例：

```
function mod(var fArg1,fArg2:integer):integer;
```

<形式参数表>::= <形式参数段>{<形式参数段>}

<形式参数段>::= [var]<标识符>{<标识符>}: <基本类型>

作用:定义参数的格式。

限定条件:参数的类型仅可以是基本类型。<形式参数段>以“**var**”开始说明参数是变量形参，需传入变量地址，以<标识符>开始说明是传值形参，需传入变量的值。

特别注意：在汇编代码中务必注意传值与传地址的区别！

<语句>::= <赋值语句>|<条件语句>|<情况语句>|<过程调用语句>|<复合语句>|<读语句>|<写语句>|<for 循环语句>|<空>

作用:说明<语句>的构成形式。

特别注意：语句可以为空！

<赋值语句>::= <标识符> := <表达式>| <函数标识符> := <表达式>|<标识符>['<表达式>']:= <表达式>

作用：说明<赋值语句>的构成。

限定条件：<标识符> := <表达式>表示为变量赋值，<标识符>必须是一个变量。<函数标识符> := <表达式>表示返回值，但后面的语句可以继续执行。

示例：

a:=123

x:=mod(x,y)

a[i]=x*y

<函数标识符>::= <标识符>

作用：说明<函数标识符>

限定条件：<标识符>必须是本函数体的函数名。（语法可以简化）

<表达式>::= [+|-]<项>{<加法运算符><项>}

<项>::= <因子>{<乘法运算符><因子>}

<因子>::= <标识符>|<无符号整数>|'(<表达式>)'|<函数调用语句>|<标识符>'(<表达式>)'

作用：说明<表达式><项><因子>的构成。

限定条件：<标识符>必须是已经声明的变量或常量，<标识符>'(<表达式>)'中<标识符>必须是已经声明的数组变量。

（具体请见前面的语法图）

<函数调用语句>::= <标识符>'(<实在参数表>)'

作用：说明<函数调用语句>的构成

限定条件：<标识符>必须是声明的函数名，而且只可以是本层或其相应外层已经声明的函数名。

特殊情况：fun1(fun2(a,b),c)

<实在参数表>::= <实在参数>{<实在参数>}

作用：说明<实在参数表>的构成。

限定条件：实在参数的个数必须与相应函数或过程的形参表的个数保持一致。对应类型也应一致，不然需按情况报错。

<实在参数>::= <表达式>|<标识符>|<函数调用语句>

作用：说明<实在参数>的构成。

限定条件：对应变形参的实在参数必须是一个已经声明的变量标识符。

<加法运算符>::= +|-

<乘法运算符>::= */

<条件>::= <表达式><关系运算符><表达式>

作用：说明<条件>的构成。

限定条件: <条件>只可以出现在<条件语句>中

<关系运算符>::= <|<=|>|>=|=|<>

<条件语句>::= if<条件>then<语句> | if<条件>then<语句>else<语句>

作用:说明<条件语句>的组成。

限定条件: <条件语句>以"if"开始。

句子示例:

```
if t>0 then
    write(1);
else
    write("0")
```

<情况语句>::= case <表达式> of <情况表元素>{<情况表元素>}end

作用:说明<情况语句>的构成

限定条件: 可以有一个或多个<情况元素表>, <情况元素表>之间以';'分隔, 情况语句以"case"开始, 以"end"结尾。

句子示例:

```
case g of
    1: write("good ");
    2: write("better ");
    3: write("best ")
End
```

<情况表元素>::= <情况常量表> : <语句>

<情况常量表>::= <常量>{<常量>}

<for 循环语句>::= for <标识符> := <表达式> (downto | to) <表达式> do <语句> //步长为1

作用:说明<for 循环语句>的组成。

限定条件: <for 循环语句>以"for"开始。

示例:

```
for i:=3 downto 1 do
    begin
        write("input x");
        read(x);
        write("input y");
        read(y);
        x:=mod(x,y);
        write("x mod y = ",x);
        write("choice 1 2 3: ");
        read(g);
    end
```


<过程调用语句> ::= <标识符>'(['<实在参数表>])'

限定条件:

<标识符>必须是已经声明的过程名, 而且只可以是本层或其相应外层已经声明的过程名。

<复合语句> ::= begin<语句>{<语句>}end

限定条件:

<复合语句>以 begin 开始, 以 end 结束。中间可以有一条或多条<语句>, <语句>之间用';'分隔。

<读语句> ::= read('<标识符>{<标识符>}')

作用:说明<读语句>的构成。

限定条件: <读语句>以 read 开始, 以')'结束, <标识符>必须是已经声明的变量。

示例:

```
read(g);
```

<写语句> ::= write('<字符串>,<表达式> ')|write('<字符串>')|write('<表达式>')

作用:说明<写语句>的构成。

示例:

```
write("x mod y = ",x);  
write("choice 1 2 3: ");
```

<字母> ::= a|b|c|d...x|y|z |A|B...|Z

<数字> ::= 0|1|2|3...8|9

<程序>的完整示例:

```
var x,y,g,m:integer;  
    i:integer;  
    a,b:integer;  
procedure swap;  
    var temp:integer;  
begin  
    temp:=x;  
    x:=y;  
    y:=temp  
end;  
function mod(var fArg1,fArg2:integer):integer;
```

```

begin
    fArg1:=fArg1-fArg1/fArg2*fArg2;
    mod:=fArg1
end;
begin
    for i:=3 downto 1 do
        begin
            write("input x: ");
            read(x);
            write("input y: ");
            read(y);

            x:=mod(x,y);
            write("x mod y = ",x);
            write("choice 1 2 3: ");
            read(g);
            case g of
                1: write("good ");
                2: write("better ");
                3: write("best ")
            end
        end
    end
end.

```

附加说明：

- (1) **char** 类型的表达式，用字符的 **ASCII** 码对应的整数参加运算，在写语句中输出字符
- (2) 标识符区分大小写字母
- (3) 赋值语句中<函数标识符> := <表达式> 作为函数的返回值，其类型应与返回类型一致，此语句后面的语句可继续执行
- (4) 写语句中的字符串原样输出
- (5) 情况语句中，**case** 后面的表达式和情况常量表里面的常量只允许出现 **integer** 和 **char** 类型
- (6) 数组的下标从 0 开始

4.文法变更

对文档进行部分改写，使其符合递归下降子程序法可以分析的要求，并尽量使其达到 LL(1)文法要求：

1. 消除左递归
2. 使有 ‘|’ 存在的规则中 **first** 集合不相交。

初步处理结果如下：

<程序> ::= <分程序>

<分程序> ::= [~~<常量说明部分>~~][<变量说明部分>][<过程说明部分>] [<函数说明部分>]<复合语句>

<常量说明部分> ::= const<常量定义>{<常量定义>;}

<常量定义> ::= <标识符>= <常量>

<常量> ::= [+|-]<无符号整数>|<字符>

<字符> ::= '<字母>|<数字>'

<字符串> ::= "{(<字母>|<数字>|'')}"

<无符号整数> ::= <数字>{<数字>}

<标识符> ::= <字母>{<字母>|<数字>}

<变量说明部分> ::= var <变量说明>;{<变量说明>;}

<变量说明> ::= <标识符>{<标识符>}:<类型>

<类型> ::= <基本类型>|array '['<无符号整数>']' of <基本类型>

<基本类型> ::= integer | char

<过程说明部分> ::= <过程首部><分程序>;~~<过程说明部分>;~~(与上面的<分程序>说明有重复)

<函数说明部分> ::= <函数首部><分程序>;~~<函数说明部分>;~~(与上面的<分程序>说明有重复)

<过程首部> ::= procedure<标识符>'(['<形式参数表>])';

<函数首部> ::= function <函数标识符>'(['<形式参数表>])':<基本类型>;

<函数首部> ::= function <函数标识符>'(['<形式参数表>])':<基本类型>;(加函数两个字以示区别)

<形式参数表> ::= <形式参数段>{<形式参数段>}

<形式参数段> ::= [var]<标识符>{<标识符>}:<基本类型>

<语句> ::= <赋值语句>|<条件语句>|<情况语句>|<过程调用语句>|<复合语句>|<读语句>|<写语句>|<for 循环语句>|<空>

(First 集合一样，在程序处理中采用毋哲学长的做法，词法分析时进行跳读 1 个字符判断)

<赋值语句> ::= <标识符>:=<表达式>|<函数标识符>:=<表达式>|<标识符>'['<表达式>']':=<表达式>

<函数标识符> ::= <标识符>

<表达式> ::= [+|-]<项>{<加法运算符><项>}

<项> ::= <因子>{<乘法运算符><因子>}

<因子> ::= <标识符>|<无符号整数>|'('<表达式>')'|<函数调用语句>|<标识符>'['<表达式>']'

<函数调用语句> ::= <标识符>'(['<实在参数表>])'

<实在参数表> ::= <实在参数>{<实在参数>}

<实在参数> ::= <表达式>|<函数调用语句>(删去<标识符>，因为可由<表达式>推出)

<加法运算符> ::= +|-

<乘法运算符> ::= *|/

<条件> ::= <表达式><关系运算符><表达式>

<关系运算符>	::= < <= > = <>
<条件语句>	::= if<条件>then<语句>[else<语句>](更符合前面的 BNF 规范)
<情况语句>	::= case <表达式> of <情况表元素>; <情况表元素>}end
<情况表元素>	::= <情况常量表> : <语句>
<情况常量表>	::= <常量>{, <常量>}
<for 循环语句>	::= for <标识符> := <表达式> (downto to) <表达式> do <语句> //步长为 1
<过程调用语句>	::= <标识符>'('(<实在参数表>)'') (删去此句, 因为可用<函数调用语句> ::= <标识符>'('(<实在参数表>)'')'同样处理。程序中将 procedure 和 function 可以 一起处理, 只是一个有返回值 PROCINT 或者 PROCCHAR, 另一个是 PROCNONE)
<复合语句>	::= begin<语句>; <语句>}end
<读语句>	::= read('<标识符>{,<标识符>}')
<写语句>	::= write('<字符串>,<表达式> ') write('<字符串>') write('<表达式>')
<字母>	::= a b c d...x y z A B... Z
<数字>	::= 0 1 2 3...8 9

文法新增:

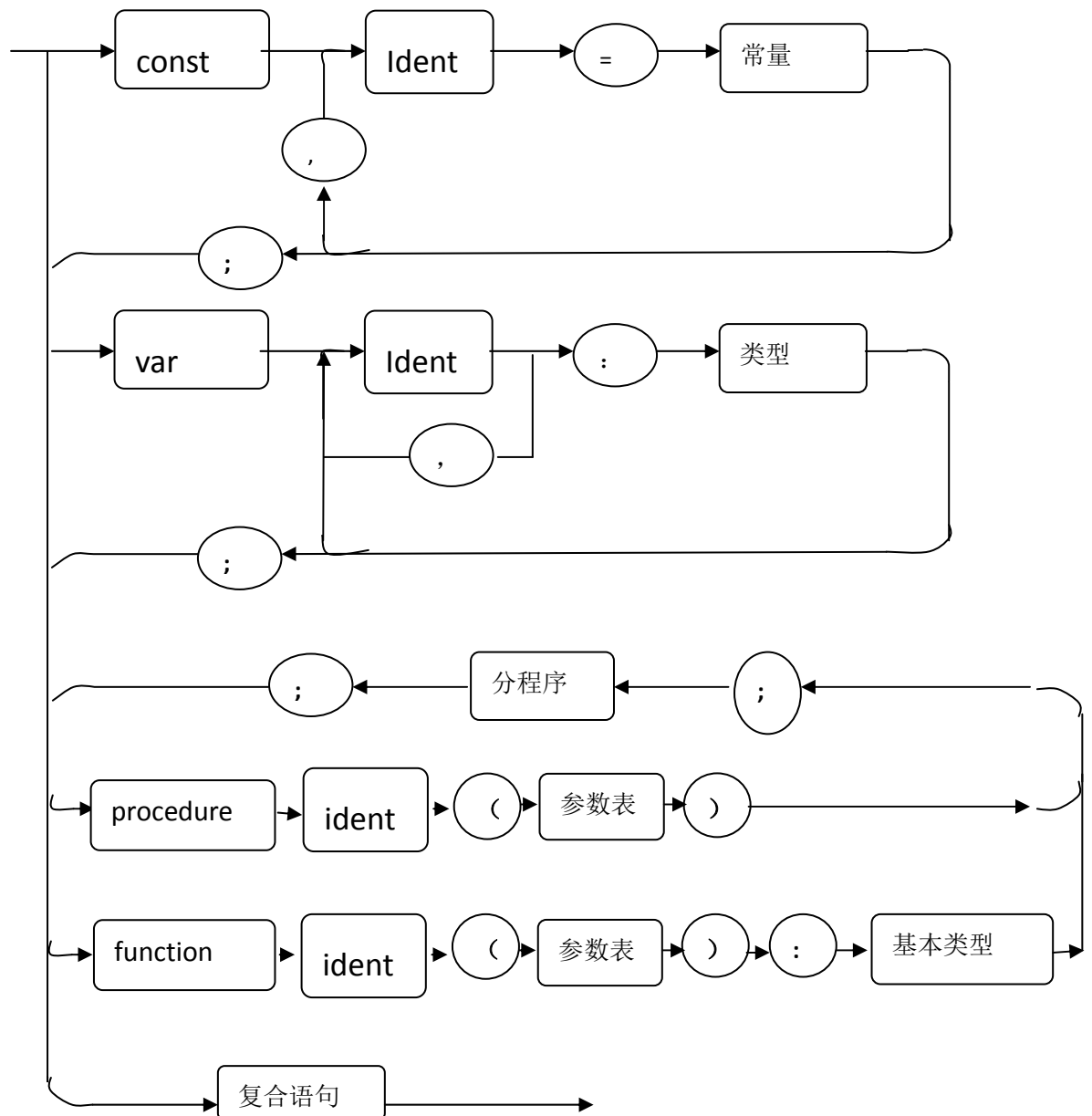
- 1.表达式中可以有字符(但出现后会报错, 生成的汇编码无误)
- 2.字符串中可以有其他字符(但出现后会报错, 生成的汇编码无误)。
- 3.对于一些常见的错误, 比如:=写成=, 能实现报错同时生成正确的汇编码。容错机制相对完善。

5.语法图

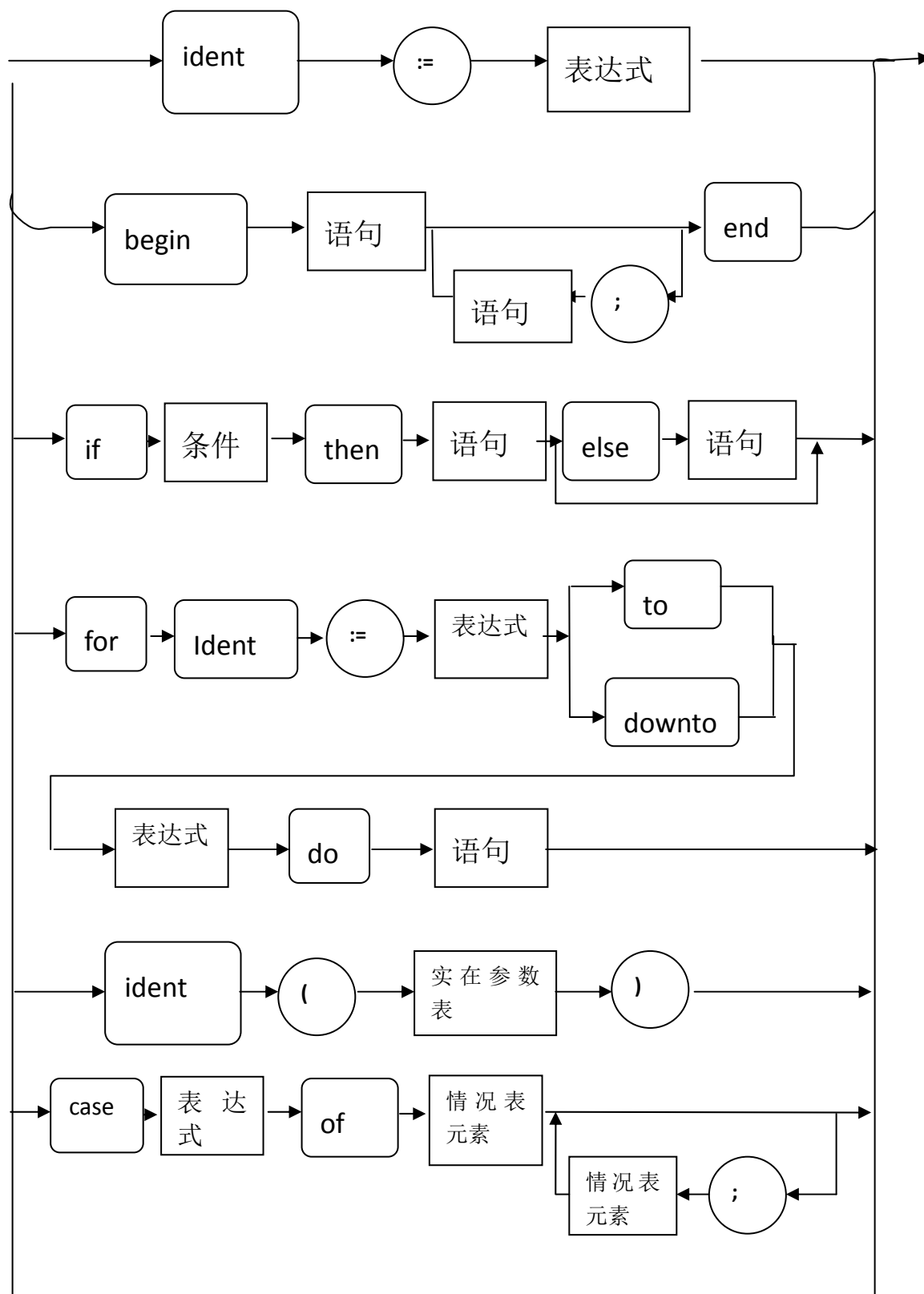
5.1 程序



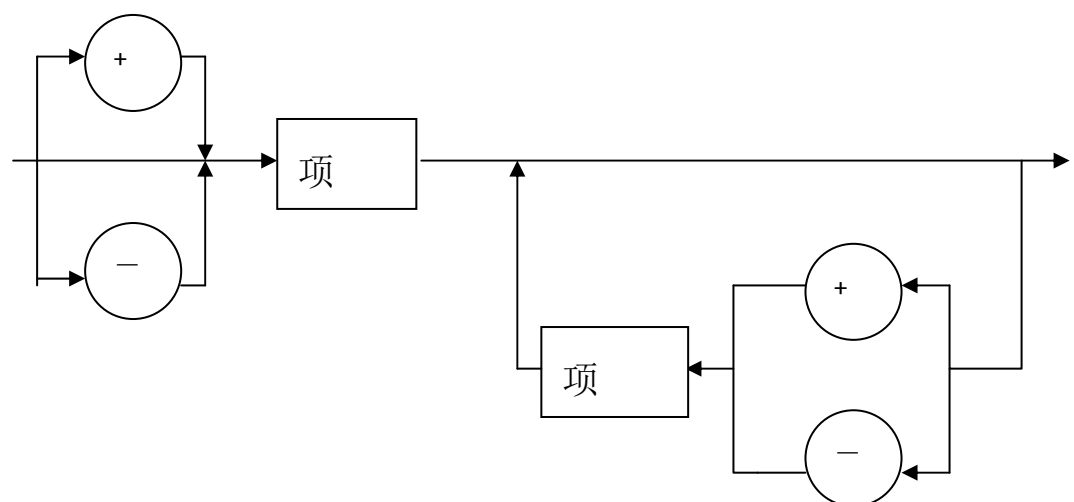
5.2 分程序



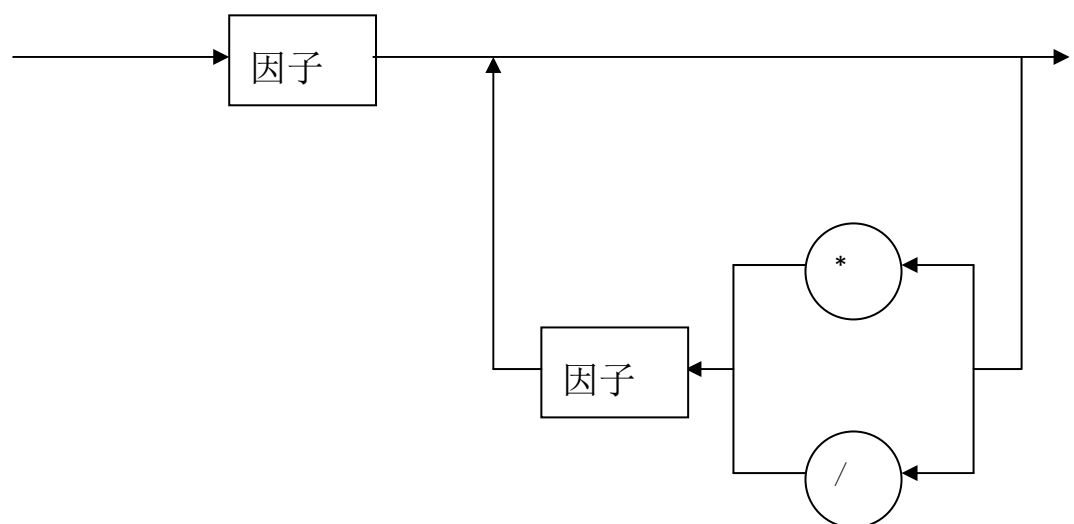
5.3 语句（因页面限制，读语句与写语句没有画出）



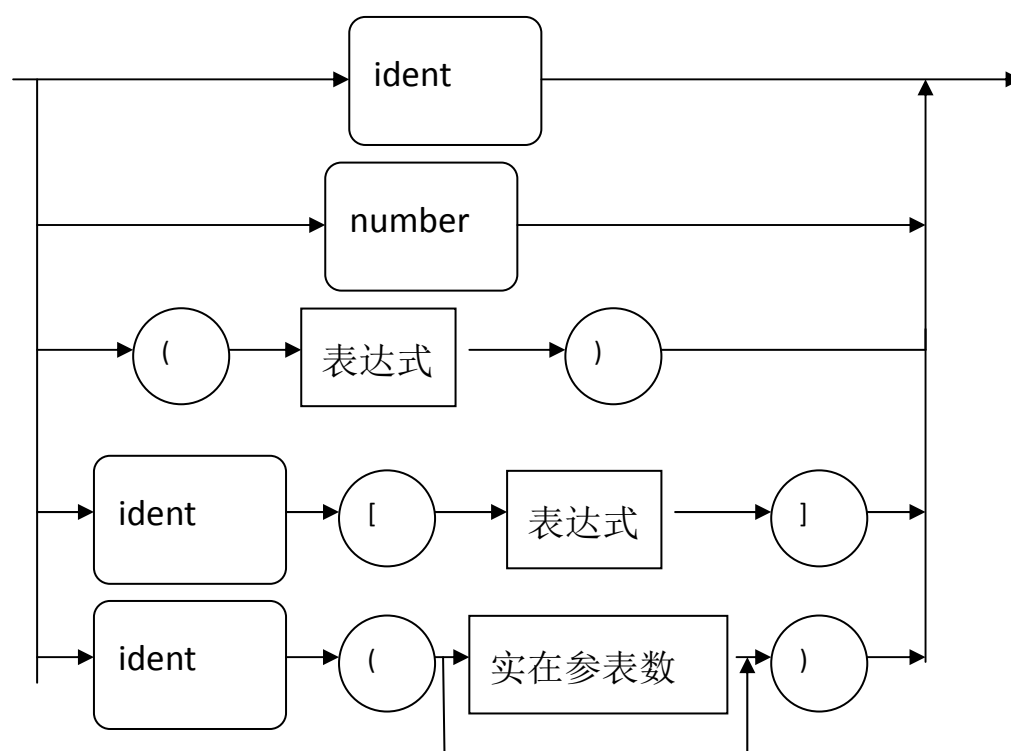
5.4 表达式



5.5 项



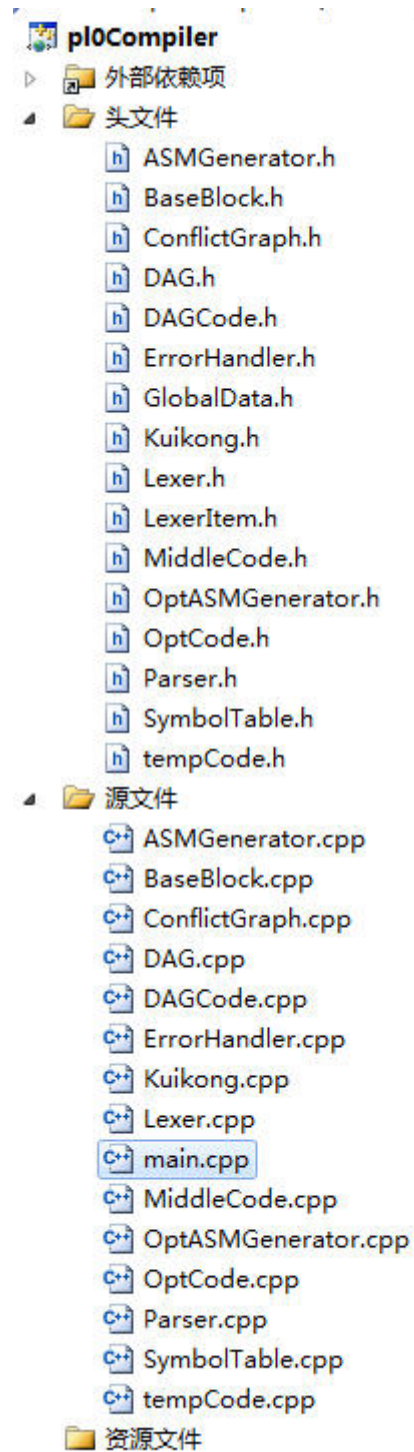
5.6 因子

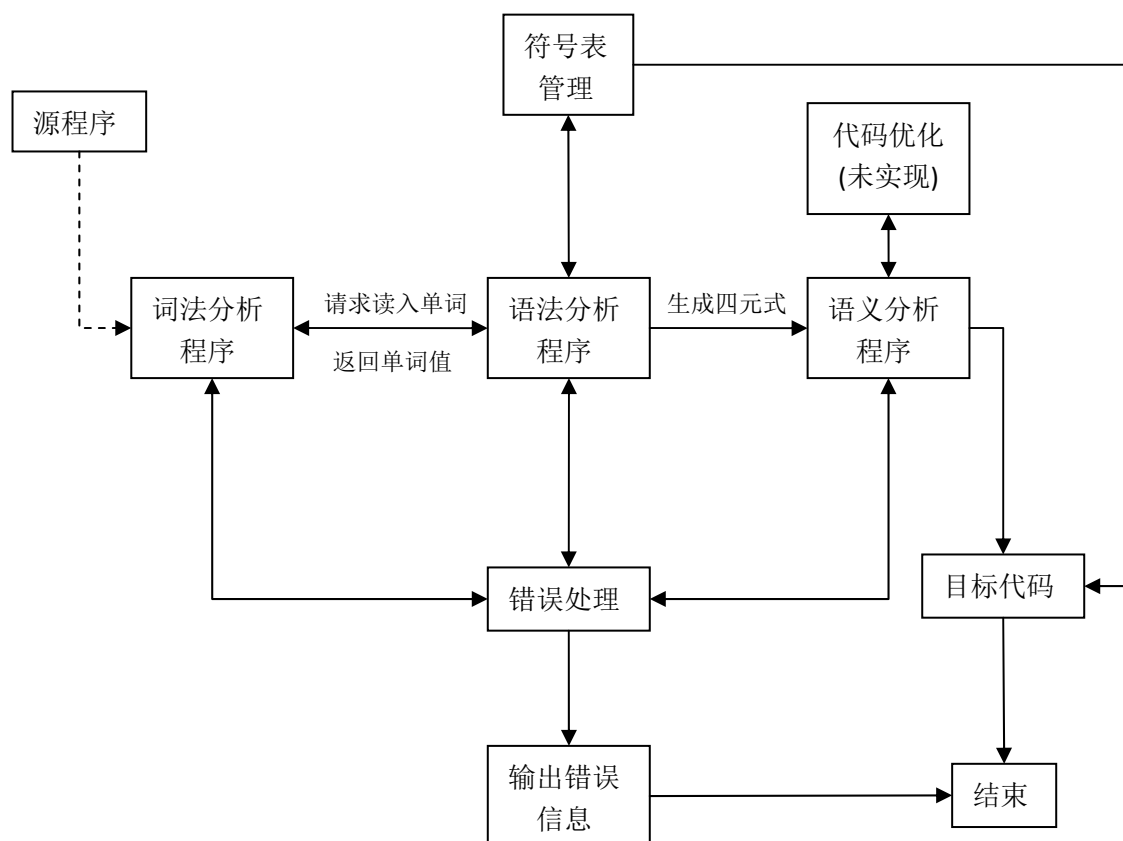


二. 详细设计

1. 程序结构

(截止到 2012 年 1 月 1 日, 已完成部分截图如下):





2. 类/方法/函数功能

【描述各类/方法或函数的功能，以及关键算法】

2.1 词法分析

```

class LexerItem //记录词法分析得到的每个单词信息
{
public:
    int kind;
    string symbol;
};
class Lexer //词法分析
{
public:
    Lexer(); //构造函数，初始化词法分析 Lexer 类
    LexerItem getSym(int mode=0); //读下一个单词
  
```

```

    LexerItem getNextSym(int i); //用于跳读第 i 单词
    void testLexer();           //测试
    int x();                    //返回行号
    int y();                    //返回列号
    static char ch;             //ch 为当前字符
    static int NowNo,tot;       //NowNo 为字符编号，tot 为该行总字符数
    static int NoOfLine; //行数
    static char line[1000];    //该行字符串
    static deque<LexerItem>SymbolList; //用双端队列存取读到的单词，便于从后面前面加单词(push_back)，以及从前面取单词(pop_front)
    int getch();               //0 正常返回，-1 异常
    int isBlank(char c); //判断是否是空格
    int isLetter(char c); //判断是否是字符
    int isNumber(char c); //判断是否是数字
    int isKeyWord(string s); //判断是否是保留字，返回保留字的序号或者是标志符的序号
};

```

2.2 语法分析与语义分析

```

class Parser //语法分析与语义分析
{
public:
    static LexerItem sym; //词法分析得到的单词信息
    static int nowProcNum; //过程编号
    static kindType t; //语法制导 类型变量 t
    static string idName; //标识名
    static int tempNum; //临时变量 No
    static int flagNum; //跳转点的 No
    static int stringNum; //字符串的 No
    static int level; //第 level 层
    static bool functionFlag;

    void initParser(); //相对于构造函数，用于初始化
    int isbaseType(string s); //是否是基本类型
    int isCompare(string s); //是否是关系运算符
    int isAdd(string s); //是否是加减运算符
    int isMul(string s); //是否是乘除运算符
    void program(); //程序
    void block(); //分程序
    void constDes(); //常量说明部分
    void constDefine(); //常量定义

```

```

string constValue();//<常量>
void varDes();//变量说明部分
void varDefine();//变量说明
void function(int lastProcNum);//函数说明部分
void functionStart(int lastProcNum);//函数首部
void procedure(int lastProcNum);//过程说明部分
void procedureStart(int lastProcNum);//过程首部
void argTable();//形式参数表
void arg(bool tempflag);//形式参数段
void complexSentence(int lastProcNum);//复合语句
void statement(int lastProcNum);//语句
void becomesState(int lastProcNum);//赋值语句
string expression(int lastProcNum);//表达式
string term(int lastProcNum);//项
string factor(int lastProcNum);//因子
string callFunction(int lastProcNum);//函数调用语句
string callProcedure(int lastProcNum);//过程调用语句
int valueTable(int lastProcNum,int ProcIndex,string procName);//实在参数表
void ifState(int lastProcNum);//条件语句
string condition(int lastProcNum);//条件
void caseState(int lastProcNum);//情况语句
void Parser::situationTable(int lastProcNum,string target,string flag,string smallflag);//情况
表元素
void forState(int lastProcNum);//for 循环语句
void readState(int lastProcNum);//读语句
void writeState(int lastProcNum);//写语句
string getNewTemp(int lastProcNum,bool flag=true);//获取新的临时变量,flag 用于记录
isnotadr 是否是地址
string getNowTemp();//得到当前临时变量
string getNewFlag();//获取新的跳转点
string getNowFlag();//得到当前跳转点
string getNewString();//获取新的字符串标记
string getNowString();//得到当前字符串标记
};

```

2.3 中间代码（四元式）

```

class MiddleCodeItem//中间代码的结构（四元式）
{
public:
    string opr;
    string target1;

```

```

        string target2;
        string result;
};

class MiddleCode//记录中间代码
{
public:
    static deque<MiddleCodeItem>codeList;//中间代码
    void addMiddleCode(string opr,string target1,string target2,string result);//增加中间代码
    void showMiddleCode();//展示中间代码
};

```

2.4 目标代码生成

```

class ASMGGenerator//汇编代码生成
{
public:
    void setObjectfileName(char s[]);
    void procStart(int index);
    void procEnd(int index);
    void WriteString(string s);
    void WriteExpression(int proclIndex,string s);
    void Read(int proclIndex,string s);
    void arrayEqual(int proclIndex,string a,string b,string c);
    void equalArray(int proclIndex,string a,string b,string c);
    void Becomes(int proclIndex,string a,string b);
    void Add(int proclIndex,string a,string b,string c);
    void Sub(int proclIndex,string a,string b,string c);
    void Mul(int proclIndex,string a ,string b,string c);
    void Div(int proclIndex,string a,string b,string c);
    void Compare(int proclIndex,string op,string a,string b,string c);
    void IfFalse(int proclIndex,string a,string b);
    string opCMP(string op);
    void SetFlag(int proclIndex,string a);
    void JumpTo(int proclIndex,string a);
    void ValueEqual(int proclIndex,string a,string b,string c);
    void Display(int proclIndex,string a,string b);
    void Call(int proclIndex,string a);
    void CallValue(int proclIndex,string a,string b);
    int isCompare(string s);
    string getNewJudge();
    void symbol(int proclIndex,string s,string reg);//结果在 reg 寄存器里面

```

```

void symbolAdr(int proclIndex,string s,string reg);
void symbolAdrEqual(int proclIndex,string s,string tempreg,string reslutreg);
static int codePoint1;
static int codePoint2;
static int judgeFlagNum;
void init();//构造函数，初始化
void generate();//整体汇编生成，调用 init();data();code();
void data();//数据区代码生成
void code();//对中间四元式解析，生成代码区
};

```

2.5 错误信息处理

```

class ErrorHandler//错误信息处理类
{
public:
    ErrorHandler();//构造函数，用于初始化错误处理信息
    int errorTot;//错误总数
    int isType(string s);//记录是否是类型
    void lexerError(int x,int y,string message);//词法分析错误处理程序
    void parserError(int x,int y,string message);//语法分析错误处理程序
    void declareError(int x,int y,string message);//语义分析错误处理程序
    void sentenceError(int x,int y,string message);//声明（变量或常量重复定义）错误处理程
序
};

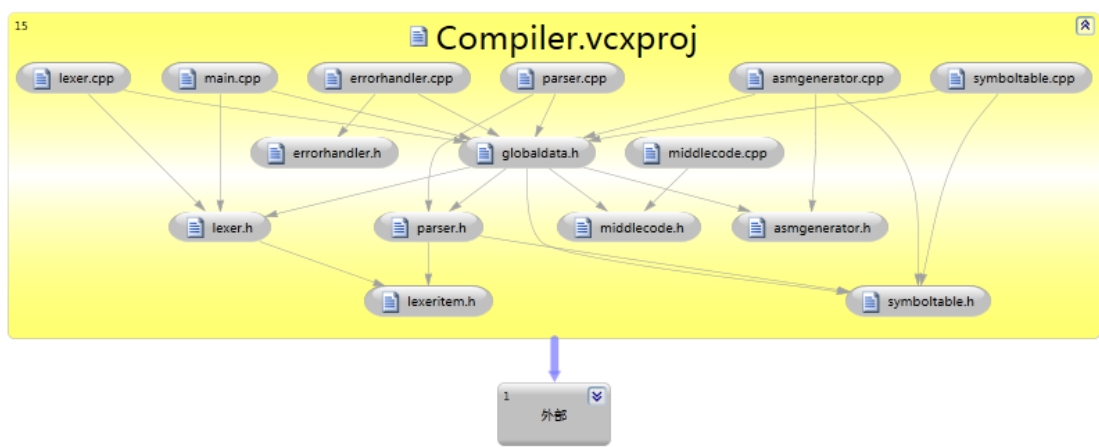
```

2.6 代码优化

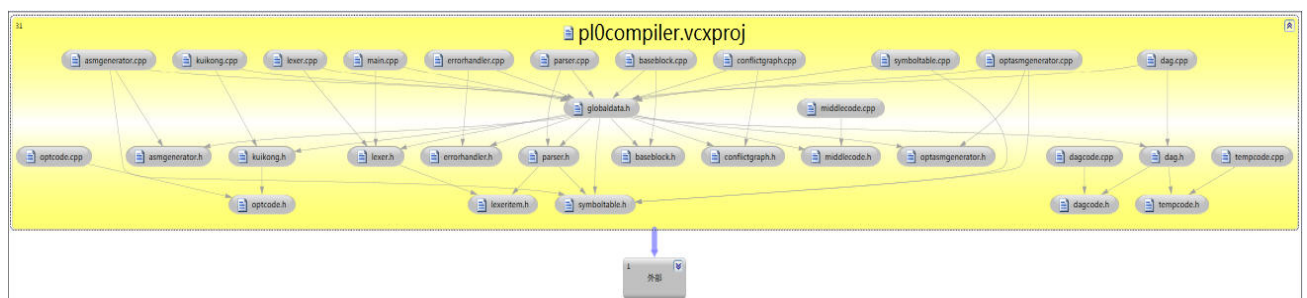
代码优化请见申优文档，里面有详细说明。

3. 调用依赖关系

【说明各类之间的关系，方法/函数之间的调用关系】
（未加优化源码）



(加优化源码)



4. 符号表结构

符号表采用多层嵌套（3 级结构）的方式（具体图示请见本节最后）。

class SymbolTableItem//记录声明后的每个标识符的信息

```
{
public:
    string name;//名字
    kindType    kind;//    类    型    （    定    义    为
enum{CONSTINT,CONSTCHAR,INT,CHAR,ARRAYINT,ARRAYCHAR,PROCINT,PROCCHAR,PROCNONE,
ERROR}类型）
    //int arrayref;//数组时指向对应 ArrayTable 中的序号，否则指向 0;
    int arraydim;//有数组时表示数组维数;否则不定义    //最后试试用构造函数初始化为 1;
    bool isnotadr;//true 表示不是地址，false 表示是地址
    //bool isinreg;
    //bool isinmem;
    int size;//表示所占的大小，一般变量为 1，数组为相应元素个数。//这样冗余甚至能去掉。
    int adr;//变量（形参）：相对起始的位移量    //对于函数或过程，填写入口
Procnum
    int value;//对于常量存储其值
    bool isnotconst;//true 表示不是常数，false 表示是常数。
```



```
};
```

class SymTableContext//记录每个函数/过程体的信息，其中的 itemList 包含 SymbolTableItem 类

```
{
public:
    deque<SymbolTableItem>itemList;//记录函数/过程里的变量、常量信息
    int lev;//表示 proc/func 所在层数
    int outproc;//记录外层的函数编号
    string procName;//记录函数名
    int procArgNum;//记录函数的参数个数
    kindType procKind;//记录函数的种类
    int startProcId;//记录一维表开始的 id，用作索引
    bool functionReturnFlag;//记录函数有无返回值
};
```

class SymbolTable//符号表，其中的 tableContext 包含 SymTableContext 类

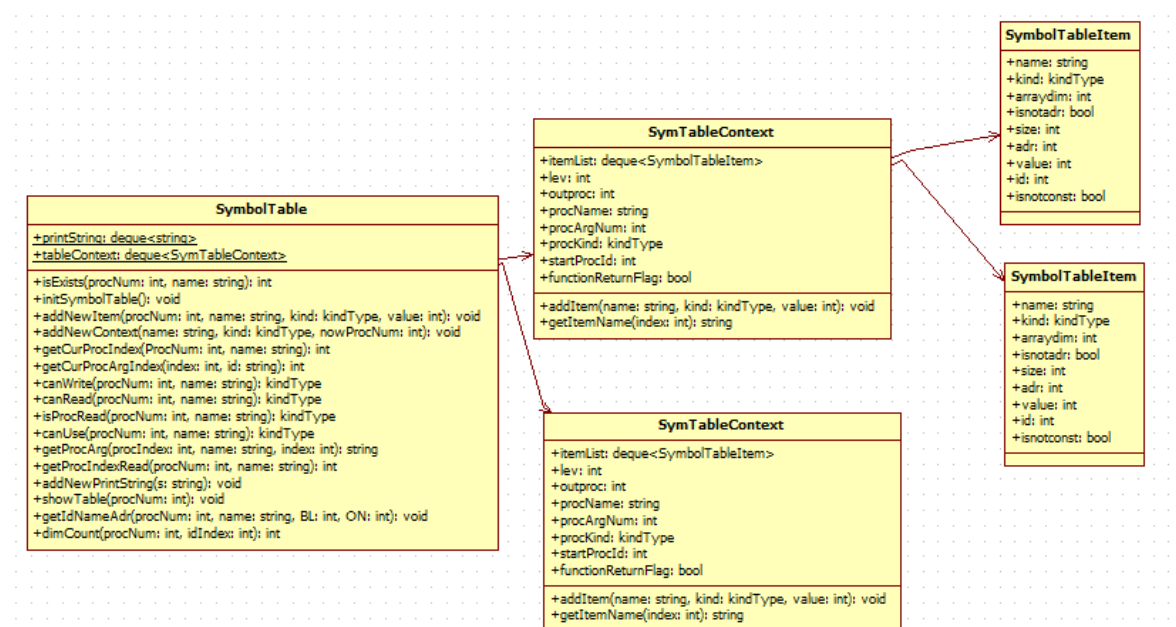
```
{
public:
    int isExists(int procNum,string name);//判断声明的标识符在符号表中是否曾经出现

    static deque<string> printString;//记录字符串
    static deque<SymTableContext>tableContext;//记录函数/过程，每个元素是上面的 SymTableContext 类
    //static deque<int>arrayList;
    //static int arrayNum;
    //void addArrayItem(string name,kindType kind,int value);
    //int getArrayItemDim(int index);
    //static int number;
    void initSymbolTable();//初始化符号表
    void addNewItem(int procNum,string name,kindType kind,int value);//在符号表 procNum 的函数/过程中增加新的符号（标识符）
    void SymbolTable::addNewContext(string name,kindType kind,int nowProcNum);//增加新的函数/过程，即新建 SymTableContext 类
    //int SymbolTable::getProcIndex(string name);
    //int SymbolTable::getProcArgIndex(int index,string id);
    int SymbolTable::getCurProcIndex(int ProcNum,string name);//获取当前函数/过程的索引（即位置）
    //int SymbolTable::getCurProcArgIndex(int index,string id);//获取当前函数的参数
    kindType SymbolTable::canWrite(int procNum,string name);//获取标识符是否可以赋值以及类型的信息
    kindType SymbolTable::canRead(int procNum,string name);//获取标识符是否可以读取值以及类型的信息
};
```

```

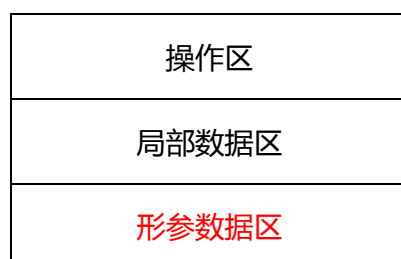
kindType SymbolTable::isProcRead(int procNum,string name);//获取函数/过程是否可以调用
kindType SymbolTable::canUse(int procNum,string name);//获取标识符是否可以使用以及类型的信息
string SymbolTable::getProcArg(int procIndex,string name,int index);//获取函数的形式参数名
int SymbolTable::getProcIndexRead(int procNum,string name);//获取可调用函数的位置
void SymbolTable::addNewPrintString(string s);//增加打印的字符串
void SymbolTable::showTable(int procNum);//打印符号表
void SymbolTable::getIdNameAdr(int procNum,string name,int &BL,int &ON);//从当前函数/过程获取可以使用的符号（本层或者外层）的位置信息(二维)
int SymbolTable::dimCount(int procNum,int idIndex);//获取符号表中某位置之前已使用空间(汇编中计算[ebp+?]使用)
};

```



符号表数据结构（3 级）图示

5. 运行栈结构



prev abp
ret addr
传入参数区
Display 区

运行栈 S

其中，传入参数区是调用函数（caller）压入运行栈的参数，形参数据区是直接将传入参数区的参数复制到被调用函数（callee）的参数。

6. 出错信息编号及描述

```
class ErrorHandler//错误信息处理类
{
public:
    ErrorHandler();//构造函数，用于初始化错误处理信息
    int errorTot;//错误总数
    int isType(string s);//记录是否是类型
    void lexerError(int x,int y,string message);//词法分析错误处理程序
    void parserError(int x,int y,string message);//语法分析错误处理程序
    void declareError(int x,int y,string message);//语义分析错误处理程序
    void sentenceError(int x,int y,string message);//声明（变量或常量重复定义）错误处理程序
};
```

编号	错误类型	错误字符
1	词法分析错误	字符中含有非法字符
2	词法分析错误	字符最后缺少匹配'
3	词法分析错误	字符串最后缺少匹配\"
4	词法分析错误	不能识别的字符
5	词法分析错误	字符中含有非法字符
6	语法分析错误	函数没有返回值
7	语法分析错误	分程序中没有复合语句
8	语法分析错误	常量说明部分开始没有 const
9	语法分析错误	常量说明部分最后缺少;
10	语法分析错误	常量定义开始不是标识符
11	语法分析错误	常量定义没有中间的=

12	语法分析错误	常量出现不符合定义的符号
13	语法分析错误	变量说明部分开始没有 var
14	语法分析错误	变量说明部分缺少;
15	语法分析错误	变量说明开始不是标识符
16	语法分析错误	变量说明缺少:
17	语法分析错误	变量说明没有类型
18	语法分析错误	变量说明定义数组缺少[
19	语法分析错误	变量说明定义数组缺少]
20	语法分析错误	变量说明定义数组缺少 of
21	语法分析错误	变量说明数组定义没有基本类型
22	语法分析错误	函数说明部分缺少;
23	语法分析错误	函数首部缺少 function
24	语法分析错误	函数首部开始缺少标识符
25	语法分析错误	函数首部缺少 (
26	语法分析错误	函数首部缺少)
27	语法分析错误	函数首部缺少:
28	语法分析错误	函数首部没有设置基本类型
29	语法分析错误	函数首部缺少;
30	语法分析错误	过程说明部分缺少;
31	语法分析错误	过程首部缺少 function
32	语法分析错误	过程首部开始缺少标识符
33	语法分析错误	过程首部缺少 (
34	语法分析错误	过程首部缺少)
35	语法分析错误	过程首部缺少;
36	语法分析错误	形式参数段缺少标志符
37	语法分析错误	形式参数段开始不是标识符
38	语法分析错误	形式参数段缺少:
39	语法分析错误	形式参数段缺少基本类型定义
40	语法分析错误	复合语句开始没有 begin
41	语法分析错误	复合语句结尾没有 end
42	语法分析错误	语句以标识符开头,却不是赋值语句与过程调用语句
43	语法分析错误	赋值语句开始不是标识符
44	语法分析错误	数组变量后面缺少]
45	语法分析错误	赋值语句或 for 循环语句缺少:=
46	语法分析错误	赋值语句缺少:=或者数组没有[]标号
47	语法分析错误	因子 (<表达式>) 缺少)
48	语法分析错误	不能识别的因子
49	语法分析错误	因子中不应有字符
50	语法分析错误	函数调用缺少标识符
51	语法分析错误	函数调用缺少 (
52	语法分析错误	过程调用缺少标识符
53	语法分析错误	过程调用缺少 (
54	语法分析错误	过程调用缺少)

55	语法分析错误	条件语句条件多了(
56	语法分析错误	条件语句条件多了)
57	语法分析错误	条件语句缺少 then
58	语法分析错误	条件中没有关系运算符
59	语法分析错误	情况语句开始缺少 case
60	语法分析错误	情况语句开始缺少 of
61	语法分析错误	情况语句缺少 end
62	语法分析错误	情况子语句缺少:
63	语法分析错误	for 循环语句开始缺少 for
64	语法分析错误	for 循环语句开始缺少标识符
65	语法分析错误	for 循环语句缺少 to/downto
66	语法分析错误	for 循环语句缺少 do
67	语法分析错误	读语句开始缺少 read
68	语法分析错误	读语句缺少(
69	语法分析错误	未声明或是常量或是数组但不能赋值
70	语法分析错误	读语句缺少标识符
71	语法分析错误	读语句缺少)
72	语法分析错误	写语句开始缺少 write
73	语法分析错误	写语句缺少(
74	语法分析错误	写语句缺少)
75	语法分析错误	程序中没有结尾.
76	语义分析错误	赋值语句中标识符后面有]但不是数组元素
77	语义分析错误	赋值语句中等式左边变量未定义 不是变量 是过程变量 是数组元素但后面没有[
78	语义分析错误	因子使用没有定义、不能读取的标识符
79	语义分析错误	函数使用错误
80	语义分析错误	因子使用没有定义的数组
81	语义分析错误	函数名不正确 应该是函数，而这里却是过程
82	语义分析错误	name+"函数名/过程不正确"
83	语义分析错误	参数数目不正确
84	语义分析错误	函数调用缺少)
85	语义分析错误	过程名不正确 应该是过程，而这里却是函数
86	语义分析错误	for 循环语句中标识符未定义或者不是变量或者是数组元素
87	语义分析错误	name+"重复定义"
88	语义分析错误	传地址形参不能是复杂表达式
89	语义分析错误	除数不能是 0

此外，按照杨海燕老师的要求，

数组越界检查：动静态都要做，静态做常量下标判断;动态即生成越界检查的目标码

除数为 0 判断：只做静态常量判断

integer 与 char 类型一致性检查是否要做：生成类型转换代码。

7. 保留字、标识符定义

编号	符号	编号	符号	编号	符号
0		20	write	40	<>
1	const	21	identifier	41	:=
2	int	22	number	42	:
3	char	23	character	43	.
4	var	24	string	44	'
5	array	25	(45	"
6	of	26)		
7	integer	27	[
8	procedure	28]		
9	function	29	,		
10	if	30	;		
11	then	31	+		
12	case	32	-		
13	for	33	*		
14	to	34	/		
15	downto	35	<		
16	do	36	<=		
17	begin	37	>		
18	end	38	>=		
19	read	39	=		

8. 原始四元式

Begin	function***			函数/过程开始
End	function***			函数/过程结束
Read	dell			读
WriteString	string1			写字符串
WriteExpression	t1 (值)			写表达式
value=	vr1	a		传值或传地址
CallValue	function***	#t4		调用函数
call	function***			调用过程
=	#t4		dell	赋值语句
+	var1	var2	var3	代数运算符

-	var1	var2	var3	代数运算符
*	var1	var2	var3	代数运算符
/	var1	var2	var3	代数运算符
<>	choice	1	#t1	比较运算符
>	choice	2	#t1	比较运算符
>=	choice	3	#t1	比较运算符
<	choice	4	#t1	比较运算符
<=	choice	5	#t1	比较运算符
=	choice	6	#t1	比较运算符
IfFalse	#t1	flag		如果 false 则跳转
JumpTo	flag_flag3			跳转
SetFlag	flag_flag4			设置跳转点
[]=	B	k	#t1	取数组地址

9. 优化后四元式

Begin	function***			函数/过程开始
End	function***			函数/过程结束
Read	dell			读
WriteString	string1			写字符串
WriteExpression	t1 (值)			写表达式
value=	vr1	a		传值或传地址
CallValue	function***	#t4		调用函数
call	function***			调用过程
=	#t4		dell	赋值语句
+	var1	var2	var3	代数运算符
-	var1	var2	var3	代数运算符
*	var1	var2	var3	代数运算符
/	var1	var2	var3	代数运算符
<>false	choice	1	#flag1	比较跳转运算符
>false	choice	2	#flag1	比较跳转运算符
>=false	choice	3	#flag1	比较跳转运算符
<false	choice	4	#flag1	比较跳转运算符
<=false	choice	5	#flag1	比较跳转运算符
=false	choice	6	#flag1	比较跳转运算符
JumpTo	flag_flag3			跳转
SetFlag	flag_flag4			设置跳转点
[]=	B	k	#t1	取数组地址

(注：将 IfFalse 与比较运算符合并，构成新的四元式如上表所示)

10. 优化种类及优化模式

模式编号	模式说明
-1	不做优化
0	窥孔优化
1	+DAG 图(消除公共子表达式), 局部寄存器池优化
2	+全局数据流分析
3	+构建冲突图, 图着色算法, 全局寄存器池优化

11. 中间结果显示

BlockDivide.txt	基本块划分
BlockFlow.txt	基本块流向
BlockVarInOut.txt	基本块活跃变量分析
ColorPrint.txt	冲突图
ConflictGraph.txt	图着色算法分配全局寄存器结果
DAG.txt	DAG 图和节点表
DefineUse.txt	定义使用表
MiddleCode(DAG).txt	中间代码 (DAG 图优化后)
MiddleCode(Kuikong).txt	中间代码 (窥孔优化后)
MiddleCode(Origin).txt	中间代码 (原始)
ReverseColor.txt	图着色算法逆序输出结果
symbolTable(DAG).txt	符号表 (DAG 图优化后)
symbolTable(Origin).txt	符号表 (原始)

三. 操作说明

1. 运行环境

【说明搭建运行环境的步骤】

1.1 程序安装

用 VC++6.0 打开 pl0Compiler.dsw 即可打开功能, 如果能够运行及安装正常。

在 Win32 平台下 (Windows xp 系统或者 Window7 32 位版本) 安装 VC6.0, 可以直接运行源程序。在 GNU C++标准环境下都可以运行本程序源码。

在 Win32 平台下 (Windows xp 系统或者 Window7 32 位版本) 安装 MASM32 软件, 可以运行本编译程序输出的.asm32 位汇编程序。

1.2 环境配置

安装好 VC++6.0 和 MASM32 软件后，不需额外环境配置。

2. 操作步骤

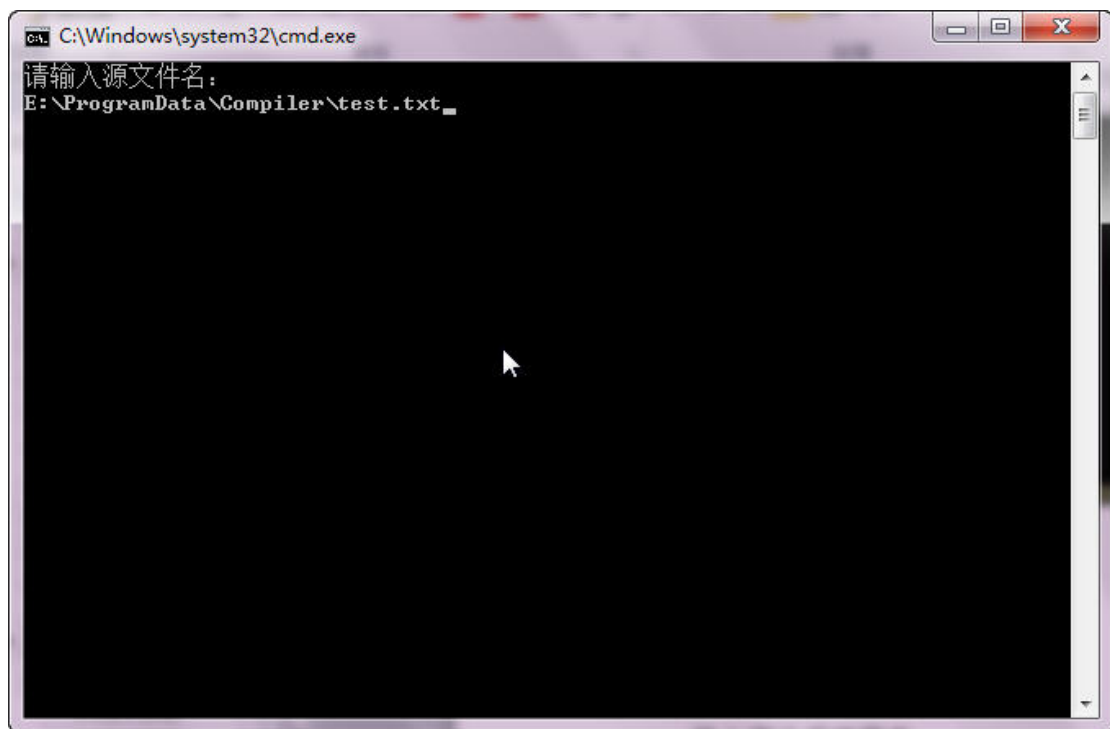
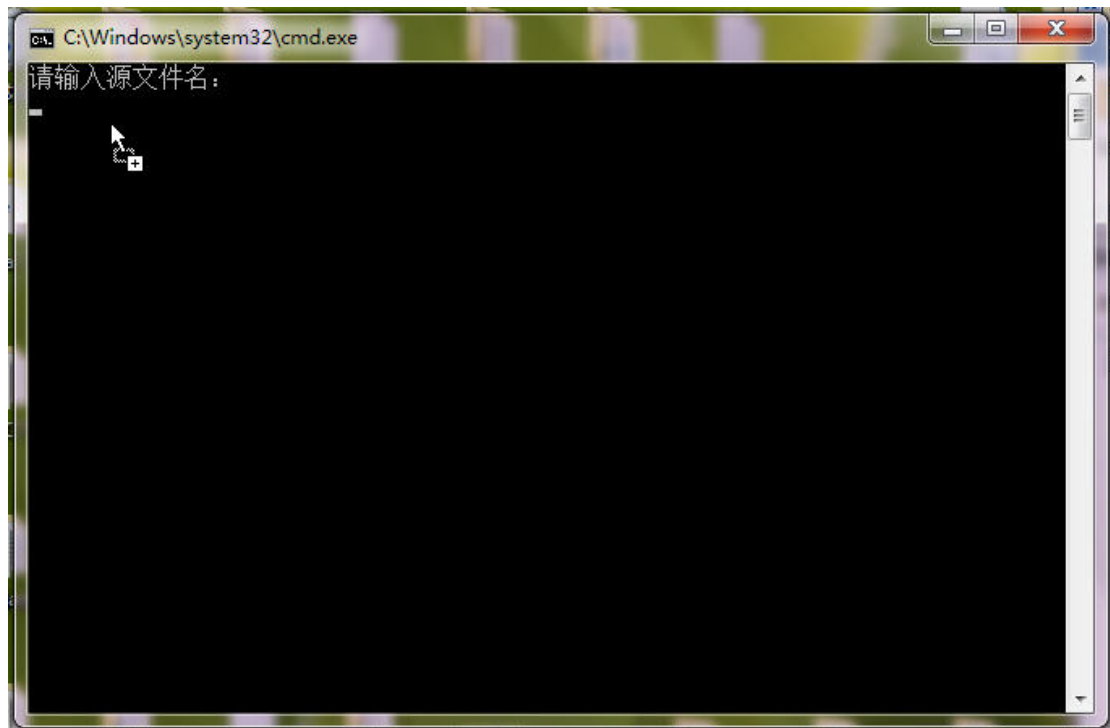
【详细说明操作步骤或以图示说明】

2.1 源程序准备

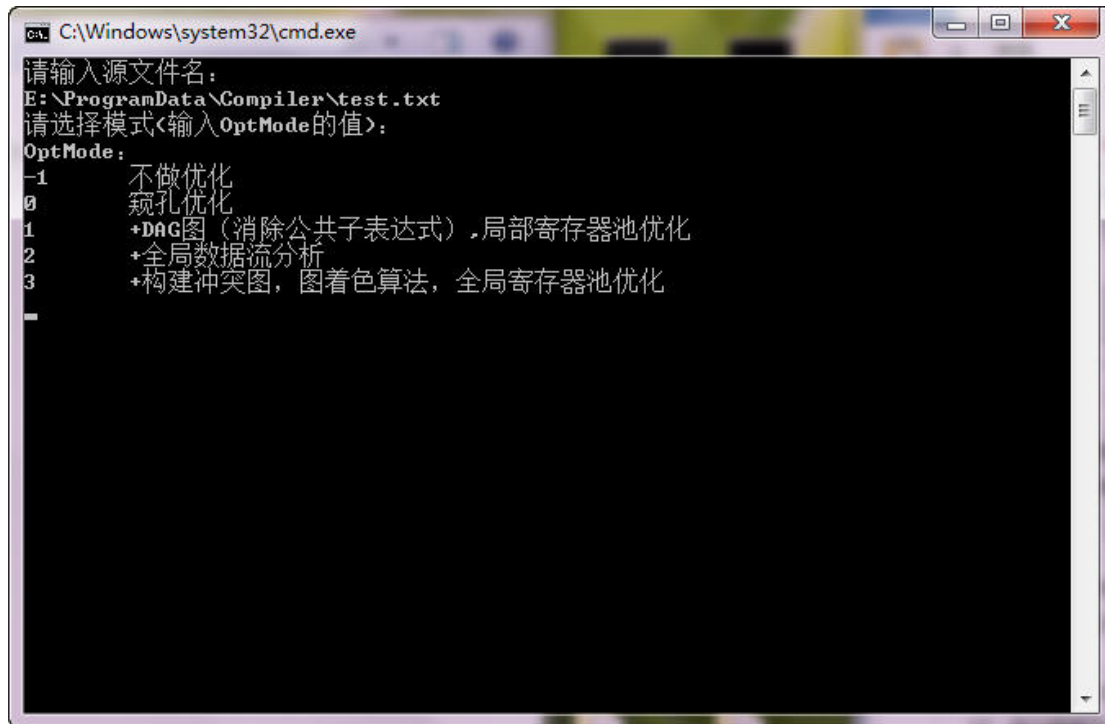
直接准备好任意后缀（.txt、.pl0、.pas 等等）的文本形式源程序放在系统任意位置，以下以 test.txt 为例。

2.2 编译、运行、结果显示过程

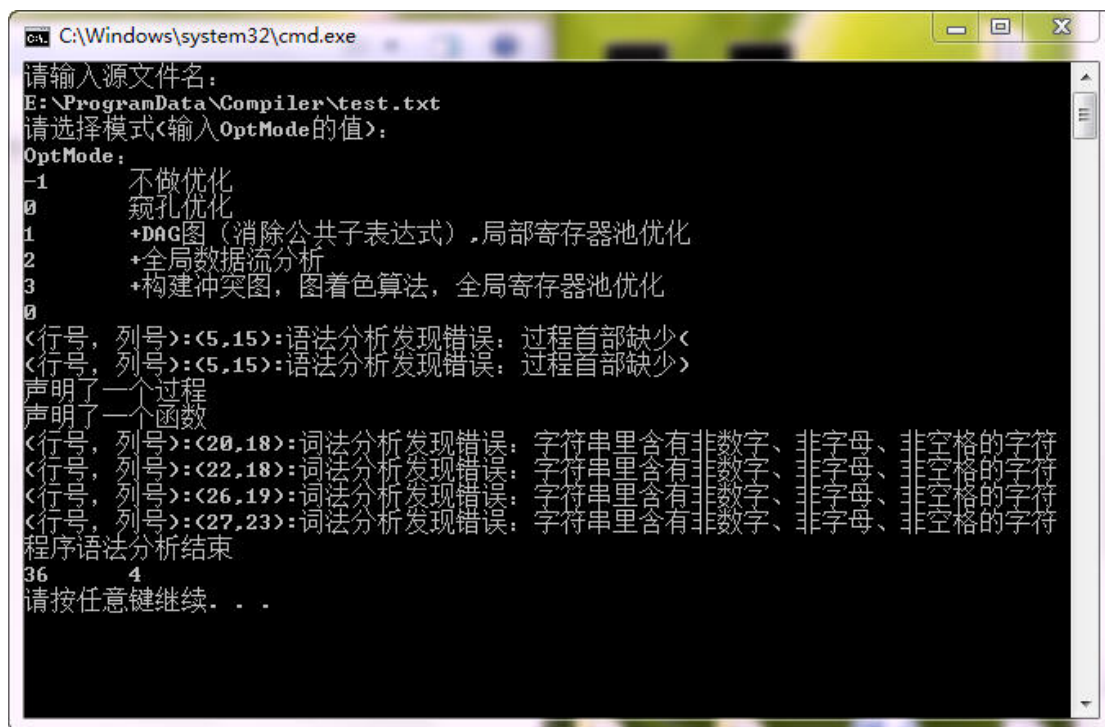
- 1.运行 pl0Compiler.exe 或者在 VC++6.0 中按下快捷键“Ctrl+F5”，运行本程序。
- 2.运行时出现“请输入源文件名”字样，此时输入文件名（包括文件路径），可以直接将源文件用鼠标拖拉至控制台 cmd 窗口处，系统自动填入文件名（包括文件路径）。



3.输入完之后按回车，弹出优化模式选择的提示。



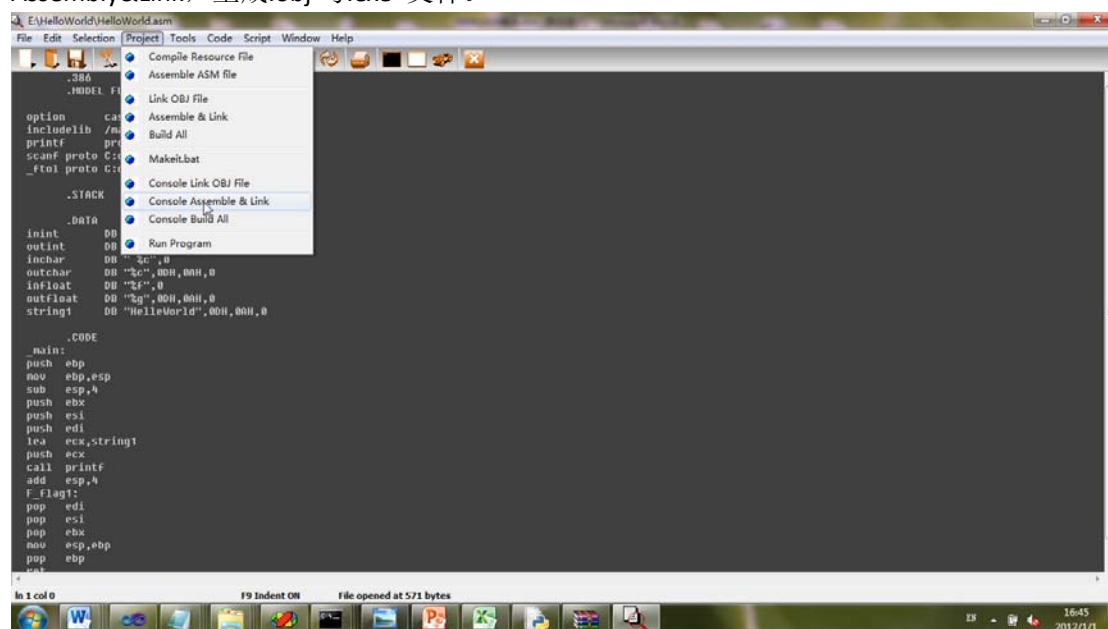
4.以输入 0 为例（优化模式是窥孔优化），按回车。



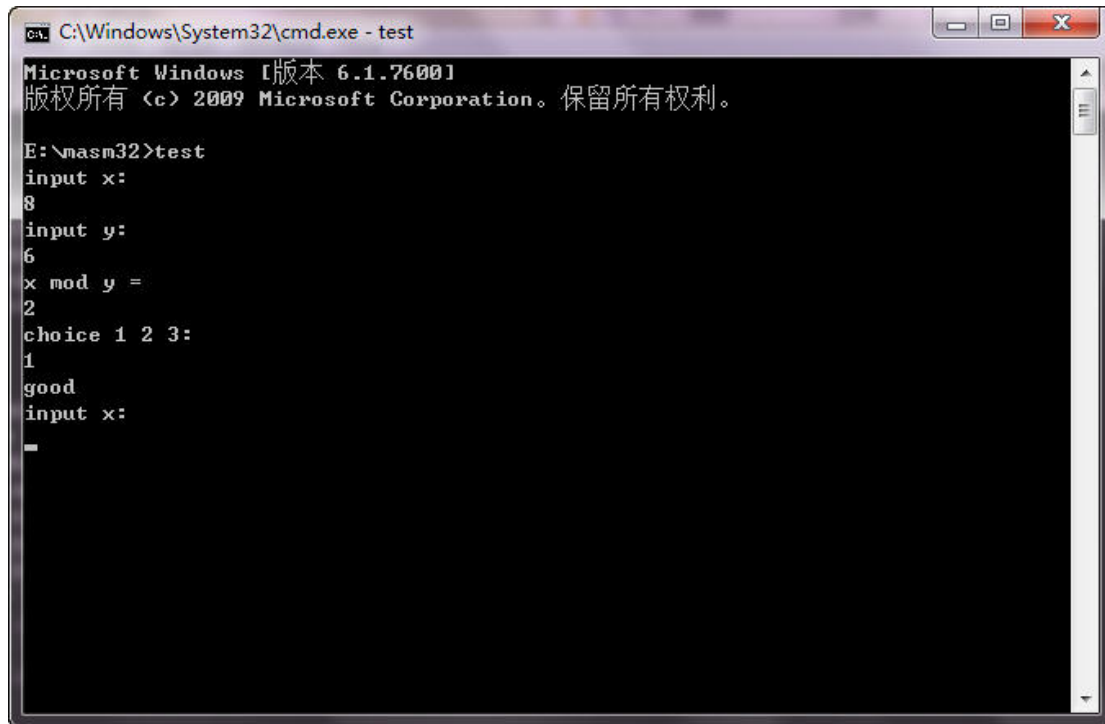
5.程序输出处理进度以及错误提示等信息，编译程序执行完毕，输出中间运行代码在本程序目录中，输出.asm 汇编文件在源程序所在目录中。

BlockDivide.txt	2012/1/1 16:18	文本文档	1 KB
BlockFlow.txt	2012/1/1 16:18	文本文档	1 KB
BlockVarInOut.txt	2012/1/1 16:18	文本文档	1 KB
ColorPrint.txt	2012/1/1 16:18	文本文档	1 KB
ConflictGraph.txt	2012/1/1 16:18	文本文档	1 KB
DAG.txt	2012/1/1 16:18	文本文档	1 KB
DefineUse.txt	2012/1/1 16:18	文本文档	1 KB
MiddleCode(DAG).txt	2012/1/1 16:18	文本文档	1 KB
MiddleCode(Kuikong).txt	2012/1/1 16:18	文本文档	1 KB
MiddleCode(Origin).txt	2012/1/1 16:18	文本文档	1 KB
ReverseColor.txt	2012/1/1 16:18	文本文档	1 KB
symbolTable(DAG).txt	2012/1/1 16:18	文本文档	1 KB
symbolTable(Origin).txt	2012/1/1 16:18	文本文档	1 KB

6. 打开 MASM32 Editor，打开刚才生成的汇编文件，并在下图所示位置点击 Console Assembly&Link，生成.obj 与.exe 文件。



7. 在控制台中运行刚才生成的 test.exe 文件，即可运行。



```
C:\Windows\System32\cmd.exe - test
Microsoft Windows [版本 6.1.7600]
版权所有 (c) 2009 Microsoft Corporation。保留所有权利。

E:\masm32>test
input x:
8
input y:
6
x mod y =
2
choice 1 2 3:
1
good
input x:
-
```

四. 测试报告

4.1.正确程序演示

4.1.1 测试 1

4.1.1.a 测试程序

(斐波那契数列的计算)

```
var result,n:integer;
function fib(k:integer):integer;
begin
    if k=0 then fib:=0
    else if k=1 then fib:=1
    else fib:=fib(k-1)+fib(k-2)
end;
begin
    write("Please enter n");
    read(n);
    write("Fib is ");
```

```
    result:=fib(n);  
    write(result)  
end.
```

4.1.1.b 测试结果

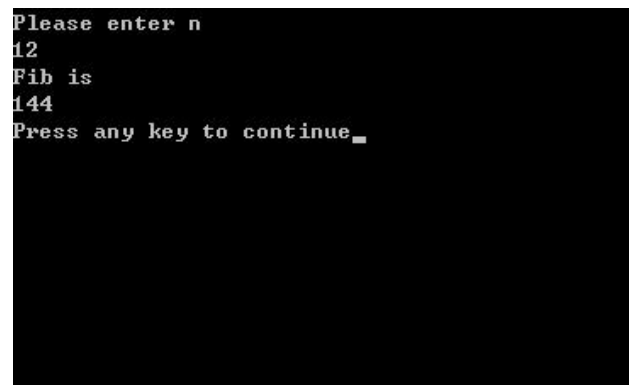
```
Please enter n  
12  
Fib is  
144  
Press any key to continue
```

4.1.1.c 测试结果分析

语义：递归 条件语句多重嵌套

语法：程序 分程序 字符串 无符号整数 标识符 变量说明部分 变量说明 类型标识符 过程说明部分 过程首部 形式参数表 形式参数段 语句 赋值语句 读语句表达式 项 因子 实在参数表 实在参数 加法运算符 条件 关系运算符(=) 条件语句 for 循环语句 过程调用语句 复合语句 读语句 写语句 字母 数字

4.1.1.d 截图



```
Please enter n  
12  
Fib is  
144  
Press any key to continue_
```

4.1.2 测试 2

4.1.2.a 测试程序

```
const m='r';
var james,c,e:integer;
function proc1(var a,b:integer):integer;
    procedure proc2();
        var d:integer;
        begin
            d:=m;
            c:=2;
            write(d)
        end;
    begin
        proc2();
        proc1:=3;
        b:=4
    end;
begin
    c:=1;
    james:=1-proc1(c,e)+3;
    write(m);
    write(c);
    write(e);
    write(james)
end.
```

4.1.2.b 测试结果

114

r

2

4

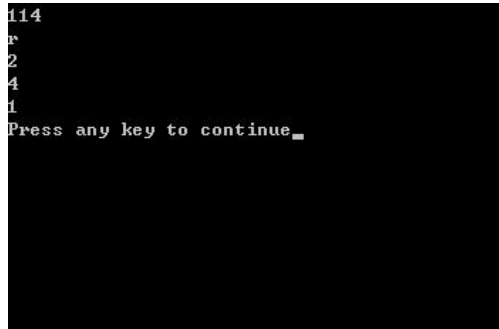
1

4.1.2.c 测试结果分析

语义：常数字符定义 地址传递 调用外层变量 内层改变外层变量值 函数返回值

语法: 程序 分程序 常量说明部分 常量定义 常量 字符串 无符号整数 标识符 变量说明部分 变量说明 类型标识符 过程说明部分 函数说明部分 过程首部（无形参） 函数首部 形式参数表 形式参数段 语句 赋值语句 函数标识符 表达式 项 因子 函数调用语句 实在参数表 实在参数 加法运算符 过程调用语句 复合语句 写语句 字母 数字

4.1.2.d 截图



4.1.3 测试 3

4.1.3.a 测试程序

（数组排序）

```
var i,j,n:integer;a:array [10] of integer;
procedure swap(var x,y:integer);
  var temp:integer;
  begin
    temp:=x;
    x:=y;
    y:=temp
  end;
begin
  n:=4;
  a[0]:=1;
  a[1]:=3;
  a[2]:=5;
  a[3]:=4;
  for i:=0 to n-1 do
  begin
    for j:=i+1 to n-1 do
    begin
      if a[i]>a[j] then swap(a[i],a[j])
```



```
        end
    end;
    write(a[0]);
    write(a[1]);
    write(a[2]);
    write(a[3]);
end.
```

4.1.3.b 测试结果

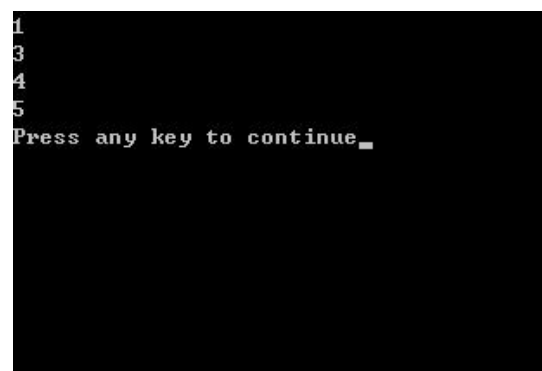
```
1
3
4
5
Press any key to continue
```

4.1.3.c 测试结果分析

语义：地址传递 双层 for 嵌套

语法：程序 分程序 常量说明部分 常量定义 常量 字符串 无符号整数 标识符 变量说明部分 变量说明 类型标识符 过程说明部分 函数说明部分 过程首部（无形参） 函数首部 形式参数表 形式参数段 语句 赋值语句 函数标识符 表达式 项 因子 函数调用语句 实在参数表 实在参数 加法运算符 过程调用语句 复合语句 写语句 字母 数字 数组

4.1.3.d 截图



```
1
3
4
5
Press any key to continue_
```

4.1.4 测试 4

4.1.4.a 测试程序

（打印 100 以内素数）

```
const true=1,false=0;
var x,y,m,n,pf:integer;

procedure prime();
  var i,f:integer;
  procedure mod();
  begin
    x:=x-x/y*y
  end;
  begin
    f:=true;
    for i:=3 to m-1 do
      begin
        x:=m;
        y:=i;
        mod();
        if x=0 then f:=false;
        i:=i+1
      end;
    end;
    if f=true then
      begin
        write(m);
        pf:=true
      end
    end;
  end;
begin
  pf:=false;
  n:=100;
  begin
    if n=2 then
      begin pf:=true;write(n) end;
    for m:=3 to n do
      begin
        prime();
        m:=m+1;
      end;
    end;
  end;
end;
```

```
        if pf=false then write(0)
end.
```


4.1.4.b 测试结果

```
3
5
7
11
13
17
19
23
29
31
37
41
43
47
53
59
61
67
71
73
79
83
89
97
Press any key to continue
```

4.1.4.c 测试结果分析

语法: 程序 分程序 常量说明部分 常量定义 常量 字符串 无符号整数 标识符 变量说明部分 变量说明 类型标识符 (只有 **integer**, 无 **char** 和 **real**) 过程说明部分 函数说明部分 过程首部 函数首部 形式参数表 形式参数段 语句 赋值语句 函数标识符 表达式 项 因子 函数调用语句 实在参数表 实在参数 加法运算符乘法运算符 条件 关系运算符 (=) 当循环语句 过程调用语句 复合语句 写语句 字母 数字 **for** 循环语句

4.1.4.d 截图



```
3
5
7
11
13
17
19
23
29
31
37
41
43
47
53
59
61
67
71
73
79
83
89
97
Press any key to continue_
```

4.1.5 测试 5

4.1.5.a 测试程序

```
var a,b : array[10] of integer;
    i,temp: integer;
begin
    for i:=0 to 9 do
    begin
        a[i]:=i;
        temp:=a[i];
        write(temp)
    end;
    for i:=0 to 9 do
    begin
        b[a[i]]:=10-i;
    end;

    for i:= 0 to 9 do
    begin
```

```

        temp:=b[i];
        write(temp)
    end
end.

```

4.1.5.b 测试结果

```

0
1
2
3
4
5
6
7
8
9
10
9
8
7
6
5
4
3
2
1
Press any key to continue

```

4.1.5.c 测试结果分析

语义：双层数组嵌套

语法：程序 分程序 常量说明部分 常量定义 常量 字符串 无符号整数 标识符 变量说明部分 变量说明 类型标识符 过程说明部分 函数说明部分 语句 赋值语句 函数标识符 表达式 项 因子 函数调用语句 实在参数表 实在参数 加法运算符 复合语句 写语句 字母 数字 数组

4.1.5.d 截图



4.2.错误程序演示

4.2.1 测试 6

4.2.1.a 测试程序

（同时报多个错误）

```
var a,b,c:integer;
function fun1(x,y:integer)
begin
    fun1=x+y
end
function fun2(x,y:integer)
begin
    fun2:=x+y
end
begin
    a:=1;
    b:=2;
    c:=3;
    write(fun1(fun2(a,b),c))
```

end

4.2.1.b 测试结果

请输入源文件名:

test11.txt

(行号, 列号):(3,7):语法分析发现错误: 函数首部缺少:

(行号, 列号):(3,7):语法分析发现错误: 函数首部没有设置基本类型

(行号, 列号):(3,7):语法分析发现错误: 函数首部缺少;

(行号, 列号):(4,7):语义发现错误:赋值语句中等式左边变量未定义|不是变量|是过程变量
|是数组元素但后面没有[

(行号, 列号):(4,7):语法分析发现错误: 赋值语句或 for 循环语句缺少:=
:=错写成=,汇编中已做更正

(行号, 列号):(6,9):语法分析发现错误: 函数说明部分缺少;
声明了一个函数

(行号, 列号):(7,7):语法分析发现错误: 函数首部缺少:

(行号, 列号):(7,7):语法分析发现错误: 函数首部没有设置基本类型

(行号, 列号):(7,7):语法分析发现错误: 函数首部缺少;

(行号, 列号):(8,8):语义发现错误:赋值语句中等式左边变量未定义|不是变量|是过程变量
|是数组元素但后面没有[

(行号, 列号):(10,6):语法分析发现错误: 函数说明部分缺少;
声明了一个函数

(行号, 列号):(14,12):语义发现错误:因子使用没有定义、不能读取的标识符

(行号, 列号):(14,12):语义发现错误:函数使用错误

(行号, 列号):(14,12):语义发现错误:fun1 函数名不正确|应该是函数, 而这里却是过程

(行号, 列号):(14,17):语义发现错误:fun1 函数名/过程不正确

(行号, 列号):(14,17):语义发现错误:因子使用没有定义、不能读取的标识符

(行号, 列号):(14,17):语义发现错误:函数使用错误

(行号, 列号):(14,17):语义发现错误:fun2 函数名不正确|应该是函数, 而这里却是过程

(行号, 列号):(14,19):语义发现错误:fun2 函数名/过程不正确

(行号, 列号):(14,21):语义发现错误:参数数目不正确

(行号, 列号):(14,24):语义发现错误:参数数目不正确

program incomplete

(行号, 列号):(15,4):语法分析发现错误: 程序中没有结尾.

程序语法分析结束

15 4

请按任意键继续...

4.2.1.c 测试结果分析

本测试程序反映本编译程序具有一定的跳读错误能力。

错误说明:

(以下错误一次性报出)

函数没有说明返回类型; 函数首部结尾没有; 中间的赋值号:=写成了=; 函数说明部分的结尾end 没有写“;”; 最后的end 结尾没有写.

4.2.1.d 截图

```
请输入源文件名:
test11
<行号, 列号>:(3,7):语法分析发现错误: 函数首部缺少:
<行号, 列号>:(3,7):语法分析发现错误: 函数首部没有设置基本类型
<行号, 列号>:(3,7):语法分析发现错误: 函数首部缺少;
<行号, 列号>:(4,7):语义发现错误:赋值语句中等式左边变量未定义!不是变量!是过程变量
!是数组元素但后面没有[
<行号, 列号>:(4,7):语法分析发现错误: 赋值语句或for循环语句缺少:=
:=错写成=, 汇编中已做更正
<行号, 列号>:(6,9):语法分析发现错误: 函数说明部分缺少;
声明了一个函数
<行号, 列号>:(7,7):语法分析发现错误: 函数首部缺少:
<行号, 列号>:(7,7):语法分析发现错误: 函数首部没有设置基本类型
<行号, 列号>:(7,7):语法分析发现错误: 函数首部缺少;
<行号, 列号>:(8,8):语义发现错误:赋值语句中等式左边变量未定义!不是变量!是过程变量
!是数组元素但后面没有[
<行号, 列号>:(10,6):语法分析发现错误: 函数说明部分缺少;
声明了一个函数
<行号, 列号>:(14,12):语义发现错误:因子使用没有定义、不能读取的标识符
<行号, 列号>:(14,12):语义发现错误:函数使用错误
<行号, 列号>:(14,12):语义发现错误:fun1函数名不正确!应该是函数, 而这里却是过程
<行号, 列号>:(14,17):语义发现错误:fun1函数名/过程不正确
<行号, 列号>:(14,17):语义发现错误:因子使用没有定义、不能读取的标识符
<行号, 列号>:(14,17):语义发现错误:函数使用错误
<行号, 列号>:(14,17):语义发现错误:fun2函数名不正确!应该是函数, 而这里却是过程
<行号, 列号>:(14,19):语义发现错误:fun2函数名/过程不正确
<行号, 列号>:(14,21):语义发现错误:参数数目不正确
<行号, 列号>:(14,24):语义发现错误:参数数目不正确
program incomplete
<行号, 列号>:(15,4):语法分析发现错误: 程序中没有结尾.
程序语法分析结束
15      4
请按任意键继续. . .
```

修正错误, 运行结果如下:

(正确程序)

```
var a,b,c:integer;
function fun1(x,y:integer):integer;
begin
    fun1:=x+y
end;
function fun2(x,y:integer):integer;
begin
    fun2:=x+y
end;
begin
    a:=1;
```



```

    b:=2;
    c:=3;
    write(fun1(fun2(a,b),c))
end.

```

输出结果:

6

4.2.2 测试 7

4.2.2.a 测试程序

```

var result,n:integer;
function a():integer;
begin
end;
function fib(var k:integer):integer;
begin
    if k=0 then fib:=0
    else if k=1 then fib:=1
    else fib:=fib(k-1)+fib(k-2)
end;
begin
    n:=5;
    result:=fib(n);
    write(result)
end.

```

4.2.2.b 测试结果

请输入源文件名:

test07

(行号, 列号):(4,4):语义发现错误:函数没有返回值

声明了一个函数

(行号, 列号):(9,19):语义发现错误:传地址形参不能是复杂表达式

(行号, 列号):(9,28):语义发现错误:传地址形参不能是复杂表达式

声明了一个函数

程序语法分析结束

15 4

请按任意键继续...

4.2.2.c 测试结果分析

函数无返回值、var 传地址形参不能是复杂表达式

4.2.2.d 截图

```
请输入源文件名:
test11
<行号, 列号>:(3,7):语法分析发现错误: 函数首部缺少:
<行号, 列号>:(3,7):语法分析发现错误: 函数首部没有设置基本类型
<行号, 列号>:(3,7):语法分析发现错误: 函数首部缺少;
<行号, 列号>:(4,7):语义发现错误:赋值语句中等式左边变量未定义!不是变量!是过程变量!
是数组元素但后面没有[
<行号, 列号>:(4,7):语法分析发现错误: 赋值语句或for循环语句缺少:=
:=错写成=, 汇编中已做更正
<行号, 列号>:(6,9):语法分析发现错误: 函数说明部分缺少;
声明了一个函数
<行号, 列号>:(7,7):语法分析发现错误: 函数首部缺少:
<行号, 列号>:(7,7):语法分析发现错误: 函数首部没有设置基本类型
<行号, 列号>:(7,7):语法分析发现错误: 函数首部缺少;
<行号, 列号>:(8,8):语义发现错误:赋值语句中等式左边变量未定义!不是变量!是过程变量!
是数组元素但后面没有[
<行号, 列号>:(10,6):语法分析发现错误: 函数说明部分缺少;
声明了一个函数
<行号, 列号>:(14,12):语义发现错误:因子使用没有定义、不能读取的标识符
<行号, 列号>:(14,12):语义发现错误:函数使用错误
<行号, 列号>:(14,12):语义发现错误:fun1函数名不正确!应该是函数, 而这里却是过程
<行号, 列号>:(14,17):语义发现错误:fun1函数名/过程不正确
<行号, 列号>:(14,17):语义发现错误:因子使用没有定义、不能读取的标识符
<行号, 列号>:(14,17):语义发现错误:函数使用错误
<行号, 列号>:(14,17):语义发现错误:fun2函数名不正确!应该是函数, 而这里却是过程
<行号, 列号>:(14,19):语义发现错误:fun2函数名/过程不正确
<行号, 列号>:(14,21):语义发现错误:参数数目不正确
<行号, 列号>:(14,24):语义发现错误:参数数目不正确
program incomplete
<行号, 列号>:(15,4):语法分析发现错误: 程序中没有结尾.
程序语法分析结束
15      4
请按任意键继续. . .
```

4.2.3 测试 8

4.2.3.a 测试程序

(数组下标越界)

```

var a:array[5] of integer;
begin
    a[5]:=8;
    a[3]:=a[5];
    write(a[3])
end.

```

4.2.3.b 测试结果

请输入源文件名：

test08

(行号，列号):(3,5):语义发现错误:数组下标越界

(行号，列号):(4,11):语义发现错误:数组下标越界

程序语法分析结束

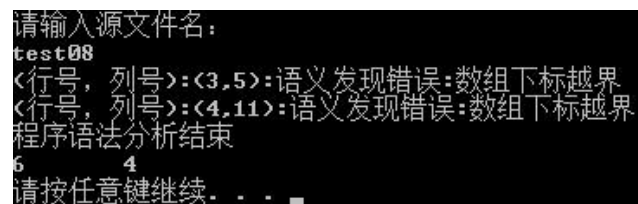
6 4

请按任意键继续...

4.2.3.c 测试结果分析

数组下标越界（静态检查）

4.2.3.d 截图



```

请输入源文件名：
test08
<行号，列号>:(3,5):语义发现错误:数组下标越界
<行号，列号>:(4,11):语义发现错误:数组下标越界
程序语法分析结束
6            4
请按任意键继续. . .

```

4.2.4 测试 9

4.2.4.a 测试程序

```

var a:array[8] of integer;
begin
    a[2]:=3;

```

```
write(a[2]);  
a[5+6]:=8  
end.
```

4.2.4.b 测试结果

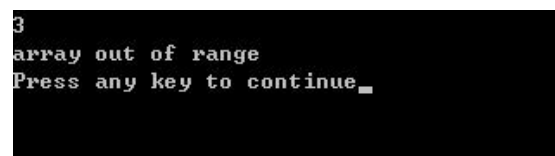
(汇编时生成)

```
3  
array out of range  
Press any key to continue
```

4.2.4.c 测试结果分析

数组下标越界（动态检查）

4.2.4.d 截图

A screenshot of a terminal window with a black background and white text. The text displayed is: 3, array out of range, and Press any key to continue. The cursor is at the end of the last line.

4.2.5 测试 10

4.2.5.a 测试程序

```
var a:integer;  
begin  
    a=3/0;  
end.
```

4.2.5.b 测试结果

请输入源文件名:

test10

(行号, 列号):(3,3):语法分析发现错误: 赋值语句或 for 循环语句缺少:=
:=错写成=,汇编中已做更正

(行号, 列号):(3,7):语义发现错误:除数不能是 0

程序语法分析结束

4 4

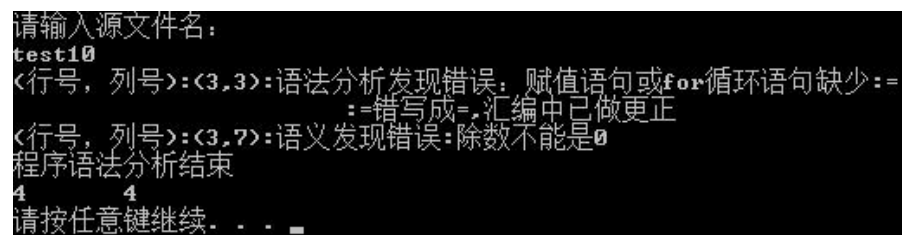
请按任意键继续...

4.2.5.c 测试结果分析

赋值号:=写成= (常见错误)

除数为 0 (静态检查)

4.2.5.d 截图



```
请输入源文件名:
test10
<行号, 列号>:<3,3>:语法分析发现错误:赋值语句或for循环语句缺少:=
:=错写成=,汇编中已做更正
<行号, 列号>:<3,7>:语义发现错误:除数不能是0
程序语法分析结束
4 4
请按任意键继续. . . _
```

4.3.优化程序演示

4.3.1 窥孔优化

4.3.1.a 测试程序

```
var x,y,g,m:integer;
    i:integer;
    a,b:integer;
procedure swap();
var temp:integer;
begin
    temp:=x;
    x:=y;
    y:=temp
end;
function mod(var fArg1,fArg2:integer):integer;
```

```

begin
    fArg1:=fArg1-fArg1/fArg2*fArg2;
    mod:=fArg1
end;
begin
    for i:=3 downto 1 do
    begin
        write("input x ");
        read(x);
        write("input y ");
        read(y);

        x:=mod(x,y);
        write("x mod y equal ",x);
        write("choice 1 2 3 ");
        read(g);
        case g of
            1: write("good ");
            2: write("better ");
            3: write("best ")
        end

    end

end
end.

```

4.3.1.b 测试结果

MiddleCode(Origin).txt

(56 行)

```

Begin    1
:=   x      temp
:=   y      x
:=   temp   y
End 1
Begin    2
/   fArg1   fArg2   #t1
*   #t1 fArg2   #t2
-   fArg1   #t2 #t3
:=   #t3     fArg1
:=   fArg1   mod
End 2
Begin    0
:=   3      i

```

```

SetFlag @flag_flag1
>= i 1 #t4
IfFalse #t4 @flag_flag2
WriteString string1
Read x
WriteString string2
Read y
Display 0 2
value= x fArg1 2
value= y fArg2 2
CallValue 2 #t5
:= #t5 x
WriteString string3
WriteExpression x
WriteString string4
Read g
= g 1 #t6
IfFalse #t6 @flag_flag5
JumpTo @flag_flag4
SetFlag @flag_flag4
WriteString string5
JumpTo @flag_flag3
SetFlag @flag_flag5
= g 2 #t7
IfFalse #t7 @flag_flag7
JumpTo @flag_flag6
SetFlag @flag_flag6
WriteString string6
JumpTo @flag_flag3
SetFlag @flag_flag7
= g 3 #t8
IfFalse #t8 @flag_flag9
JumpTo @flag_flag8
SetFlag @flag_flag8
WriteString string7
JumpTo @flag_flag3
SetFlag @flag_flag9
SetFlag @flag_flag3
- i 1 i
JumpTo @flag_flag1
SetFlag @flag_flag2
End 0

```

MiddleCode(Kuikong).txt

(45 行)

窥孔优化后的中间代码如下：

```
Begin    1
:=  x      temp
:=  y      x
:=  temp    y
End 1
Begin    2
/  fArg1  fArg2  #t1
*  #t1  fArg2  #t2
-  fArg1  #t2  #t3
:=  #t3    fArg1
:=  fArg1    mod
End 2
Begin    0
:=  3      i
SetFlag  @flag_flag1
>=false  i  1  @flag_flag2
WriteString  string1
Read  x
WriteString  string2
Read  y
Display  0  2
value=  x  fArg1  2
value=  y  fArg2  2
CallValue 2      #t5
:=  #t5    x
WriteString  string3
WriteExpression  x
WriteString  string4
Read  g
=false  g  1  @flag_flag5
WriteString  string5
JumpTo  @flag_flag9
SetFlag  @flag_flag5
=false  g  2  @flag_flag7
WriteString  string6
JumpTo  @flag_flag9
SetFlag  @flag_flag7
=false  g  3  @flag_flag9
WriteString  string7
SetFlag  @flag_flag9
-  i  1  i
JumpTo  @flag_flag1
```



```
SetFlag @flag_flag2
End 0
```

4.3.1.c 测试结果分析

本测试程序反映窥孔优化对中间代码的显著优化效果。

4.3.2 局部公共子表达式删除（DAG 图优化）

4.3.2.a 测试程序

```
var a,b,c:integer;
begin
a:=b*(-c)+b*(-c);
end.
```

4.3.2.b 测试结果

窥孔优化后的中间代码如下：

```
Begin 0
- 0 c #t1
* b #t1 #t2
- 0 c #t3
* b #t3 #t4
+ #t2 #t4 #t5
:= #t5 a
End 0
```

DAG 图

Name	Left	Right
0	-1	-1
c_0	-1	-1
-	0	1
b_0	-1	-1
*	3	2
+	4	4

节点表

Name	ID
0	0
c	1
#t1	2
b	3
#t2	4
#t3	2
#t4	4
#t5	5
a	5

符号表 (DAG)

procID:0 procName:Main procKind:8

ArgNum:0 lev:0Outfunc:-1

tableContext:

id	name	kind	value	isnotadr	dim
0	a	2	0	1	1
1	b	2	0	1	1
2	c	2	0	1	1
3	#t1	2	0	1	1
4	#t2	2	0	1	1
5	#t3	2	0	1	1
6	#t4	2	0	1	1
7	#t5	2	0	1	1
8	c_0	2	0	1	1
9	b_0	2	0	1	1

DAGCode (中间代码)

Begin 0

:= c c_0

:= b b_0

- 0 c #t1

* b #t1 #t2

+ #t2 #t2 a

End 0

4.3.2.c 测试结果分析

经过 DAG 图优化后的代码

4.3.3 活跃变量分析

4.3.3.a 测试程序

```
var x,y,z,i,a,b:integer;
begin
  x:=a;
  y:=b;
  for i:=0 to 100 do
    begin
      z:=a*10;
      x:=x+y;
      if x>z then x:=x-y;
      y:=y+1;
    end
  end.
end.
```

4.3.3.b 测试结果

BlockDivide:

```
1  Begin    0
1  :=  a      a_0
1  :=  b      b_0
1  :=  0      i
1  :=  b_0    y
1  :=  a_0    x
2  SetFlag  @flag_flag1
2  <=false  i    100 @flag_flag2
3  :=  a      a_0
3  :=  x      x_0
3  :=  y      y_0
3  +  x_0 y    x
3  *  a  10  z
3  >false   x  z    @flag_flag3
4  :=  x      x_0
4  :=  y      y_0
4  -  x_0 y    x
5  SetFlag  @flag_flag3
5  :=  y      y_0
5  :=  i      i_0
5  +  i_0 1    i
5  +  y_0 1    y
```

```

5  JumpTo  @flag_flag1
6  SetFlag @flag_flag2
6  End 0

```

BlockFlow:

BaseBlock1:

```

in
out 2

```

BaseBlock2:

```

in 1 5
out 3 6

```

BaseBlock3:

```

in 2
out 4 5

```

BaseBlock4:

```

in 3
out 5

```

BaseBlock5:

```

in 3 4
out 2

```

BaseBlock6:

```

in 2
out

```

DefineUse:

Block1:

```

use a b
define a_0 b_0 i y x

```

Block2:

```

use i
define

```

Block3:

```

use a x y
define a_0 x_0 y_0 z

```

Block4:

```

use x y
define x_0 y_0

```

Block5:

```

use y i
define y_0 i_0

```

Block6:

```

use
define

```

BlockVarInOut:

Block1:

In a b
Out i a x y

Block2:

In i a x y
Out a x y i

Block3:

In a x y i
Out x y i a

Block4:

In x y i a
Out y i a x

Block5:

In y i a x
Out i a x y

Block6:

In
Out

所有跨越基本块的活跃变量

a b i x y

4.3.3.c 测试结果分析

以上通过全局数据流分析做出流图、并得到活跃变量信息

4.3.4 构建冲突图,图着色算法

4.3.4.a 测试程序

(同 4.3.4)

```
var x,y,z,i,a,b:integer;
```

```
begin
```

```
  x:=a;
```

```
  y:=b;
```

```
  for i:=0 to 100 do
```

```
    begin
```

```
      z:=a*10;
```

```
      x:=x+y;
```

```
      if x>z then x:=x-y;
```

```

        y:=y+1;
    end
end.

```

4.3.4.b 测试结果

冲突图（邻接矩阵）

ID	ProcIndex	Id	Name	ConflictNo
0	0	0	x:2 3	1
1	0	1	y:2 3	0
2	0	3	i:3 0	1
3	0	4	a:4 2	0 1
4	0	5	b:3	

图着色算法的结果

ProcIndex	Id	Name	RegNum
0	0	x:-2	
0	1	y:-2	
0	3	i:1	
0	4	a:0	
0	5	b:1	

4.3.4.c 测试结果分析

由活跃变量分析的结果得到冲突图，然后由冲突图根据图着色算法分配全局寄存器

4.3.5 局部寄存器与全局寄存器优化混合

4.3.5.a 测试程序

```

var x,y,z,i,a,b,ok:integer;
begin
    ok:=1;
    if ok=1 then
        begin
            a:=1;b:=2
        end;
    x:=a;

```

```

y:=b;
for i:=0 to 10 do
begin
    z:=a*10;
    x:=x+y;
    if x>z then x:=x-y;
    y:=y+1;
end;
write(x);
write(y);
end.

```

4.3.5.b 测试结果

生成汇编代码:

```

.386
.model flat,stdcall

option    casemap:none
includelib C:\masm32\lib\msvcrt.lib
printf    proto C:dword,.:vararg
scanf     proto C:dword,.:vararg
.STACK    4096*16

.DATA
inint     DB "%d",0
outint    DB "%d",0DH,0AH,0
inchar    DB " %c",0
outchar   DB "%c",0DH,0AH,0
arrayerror DB "array out of range",0DH,0AH,0

.CODE

;Begin    0

start:
pushebp
mov ebp,esp
sub esp,80
pushebx
pushedi

```

```

;:= 1      ok
mov eax,1

;:=false   ok   1   @flag_flag1
mov ecx,eax
cmp ecx,1
mov dword ptr [ebp-28],eax
jne @flag_flag1

;:= 2      b
mov ebx,2

;:= 1      a
mov edi,1

;SetFlag @flag_flag1
@flag_flag1:

;:= a      a_0
mov eax,edi

;:= b      b_0
mov ecx,ebx

;:= 0      i
mov ebx,0

;:= b_0    y
mov edx,ecx

;:= a_0    x
mov dword ptr [ebp-64],ecx
mov ecx,eax
mov dword ptr [ebp-60],eax
mov dword ptr [ebp-4],ecx
mov dword ptr [ebp-8],edx

;SetFlag @flag_flag3
@flag_flag3:

;:=false   i    10   @flag_flag4
mov eax,ebx
cmp eax,10

```



```

jg  @flag_flag4

;:=  a      a_0
mov eax,edi

;:=  x      x_0
mov ecx,dword ptr [ebp-4]

;:=  y      y_0
mov edx,dword ptr [ebp-8]

;+  x_0 y    x
mov dword ptr [ebp-60],eax
mov eax,ecx
add  eax,dword ptr [ebp-8]

;*  a  10  z
mov dword ptr [ebp-4],eax
mov eax,edi
imul eax,10

;>false  x  z  @flag_flag5
mov dword ptr [ebp-68],ecx
mov ecx,dword ptr [ebp-4]
cmp ecx,eax
mov dword ptr [ebp-12],eax
mov dword ptr [ebp-72],edx
jle  @flag_flag5

;:=  x      x_0
mov eax,dword ptr [ebp-4]

;:=  y      y_0
mov ecx,dword ptr [ebp-8]

;-  x_0 y    x
mov edx,eax
sub  edx,dword ptr [ebp-8]
mov dword ptr [ebp-68],eax
mov dword ptr [ebp-72],ecx
mov dword ptr [ebp-4],edx

;SetFlag  @flag_flag5
@flag_flag5:

```

```

;:= y      y_0
mov eax,dword ptr [ebp-8]

```

```

;:= i      i_0
mov ecx,ebx

```

```

;+ i_0 1 i
mov ebx,ecx
add ebx,1

```

```

;+ y_0 1 y
mov edx,eax
add edx,1

```

```

;JumpTo @flag_flag3
mov dword ptr [ebp-72],eax
mov dword ptr [ebp-76],ecx
mov dword ptr [ebp-8],edx
jmp @flag_flag3

```

```

;SetFlag @flag_flag4
@flag_flag4:

```

```

;WriteExpression x
pushdword ptr [ebp-4]
lea esi,outint
pushesi
call printf
add esp,8

```

```

;WriteExpression y
pushdword ptr [ebp-8]
lea esi,outint
pushesi
call printf
add esp,8

```

```

;End 0
pop edi
pop ebx
mov esp,ebp
pop ebp
ret

```

end start

4.3.5.c 测试结果分析

在前面所有优化的基础上进行进一步机器相关的优化。

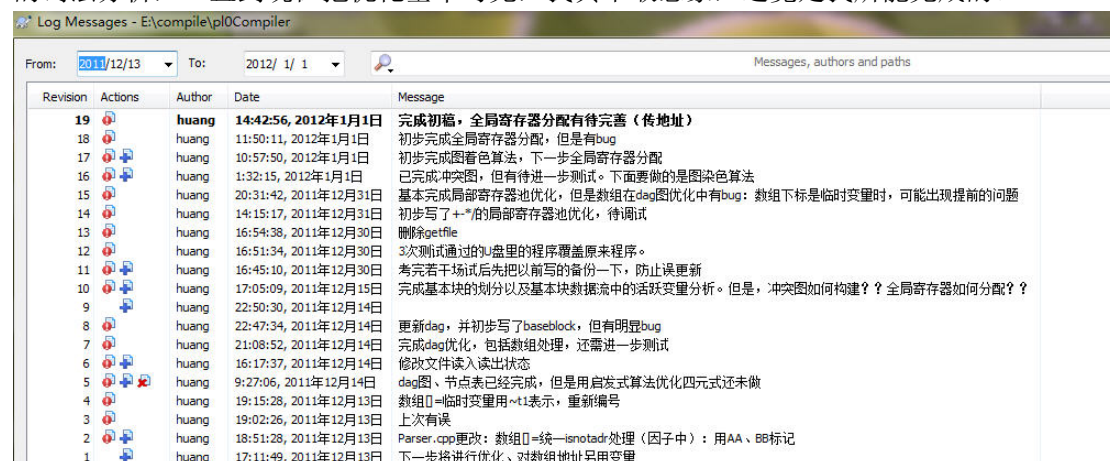
五. 总结感想

【说明在完成课程设计中的收获、认识和感想】

感谢杨海燕老师不厌其烦的答疑解惑，感谢史晓华老师在编译课堂上的循循善诱、谆谆教诲以及上学期 C++ 课上的细致讲解，感谢各位助教在我们上机时耐心地等候我们一次又一次改正 bug。

这是我第一次写这么长的程序，以前接受过的编程训练，最长的恐怕就是 C2 课程，一次编一个 100~200 行的程序。而本次课程设计，让我感受到刻骨铭心的成长。

多少日日夜夜抱着本在自习室独自一人写着程序，码着代码。从开始很少的几百行代码的词法分析，一直到现在把优化基本写完，我真不敢想象，这竟是我所能完成的。



Revision	Actions	Author	Date	Message
19		huang	14:42:56, 2012年1月1日	完成初稿，全局寄存器分配有待完善（传地址）
18		huang	11:50:11, 2012年1月1日	初步完成全局寄存器分配，但是有bug
17		huang	10:57:50, 2012年1月1日	初步完成图着色算法，下一步全局寄存器分配
16		huang	1:32:15, 2012年1月1日	已完成冲突图，但有待进一步测试。下面要做的是图染色算法
15		huang	20:31:42, 2011年12月31日	基本完成局部寄存器优化，但是数组在dag图优化中有bug：数组下标是临时变量时，可能出现提前的问题
14		huang	14:15:17, 2011年12月31日	初步写了+*/的局部寄存器优化，待调试
13		huang	16:54:38, 2011年12月30日	删除getfile
12		huang	16:51:34, 2011年12月30日	3次测试通过的j盘里的程序覆盖原来程序。
11		huang	16:45:10, 2011年12月30日	考完若干场试后先把以前写的备份一下，防止误更新
10		huang	17:05:09, 2011年12月15日	完成基本块的划分以及基本块数据流中的活跃变量分析。但是，冲突图如何构建？？全局寄存器如何分配？？
9		huang	22:50:30, 2011年12月14日	
8		huang	22:47:34, 2011年12月14日	更新dag，并初步写了baseblock，但有明显bug
7		huang	21:08:52, 2011年12月14日	完成dag优化，包括数组处理，还需进一步测试
6		huang	16:17:37, 2011年12月14日	修改文件读入读出状态
5		huang	9:27:06, 2011年12月14日	dag图、节点表已经完成，但是用启发式算法优化四元式还未做
4		huang	19:15:28, 2011年12月13日	数组[]=临时变量用~t1表示，重新编号
3		huang	19:02:26, 2011年12月13日	上次有误
2		huang	18:51:28, 2011年12月13日	Parser.cpp更改：数组[]→统一isnotadr处理（因子中）：用AA、BB标记
1		huang	17:11:49, 2011年12月13日	下一步将进行优化、对数组地址另用变量

最后写完程序 Demo 后我对代码的管理混乱，不得不转用 svn 管理。上图便是 log 的截图。中间又忙着各种考试，使得很长时间内一行代码没写。但回过头来看着自己更新的每一行代码，每一行注释，感受到的确是自己的每一天的成长。

这次课设最大的感受便是理论与实践的差距。在编译课上学到的东西，感觉都能懂。但是真正到了实际应用的时候，发现自己对很多基本的概念都不理解。比方那种种优化，在纸上写，我可以写的很熟练，但是到了真正编程要用的时候，却发现自己大脑一片茫然，不知所措。

我觉得这门课程的开设对我太有意义了。北航的编译课程安排的也很合理，老师都是以循循善诱的方式来教学，而且讲授内容灵活多变，并不死板的跟随课本的章节，并合理照顾到了课程设计的进度。

衷心祝愿北航编译越来越好，让更多的人受益！