目录

[一．需求说明 3](#_Toc440090714)

[1．文法说明 3](#_Toc440090715)

[2．目标代码说明 9](#_Toc440090716)

[3. 优化方案\* 10](#_Toc440090717)

[二．详细设计 10](#_Toc440090718)

[1．程序结构 10](#_Toc440090719)

[2．类/方法/函数功能 11](#_Toc440090720)

[3．调用依赖关系 16](#_Toc440090721)

[4．符号表管理方案 16](#_Toc440090722)

[5．存储分配方案 21](#_Toc440090723)

[6. 解释执行程序\* 22](#_Toc440090724)

[7. 四元式设计\* 22](#_Toc440090725)

[8. 目标代码生成方案\* 24](#_Toc440090726)

[9. 优化方案\* 24](#_Toc440090727)

[10. 出错处理 26](#_Toc440090728)

[三．操作说明 28](#_Toc440090729)

[1．运行环境 28](#_Toc440090730)

[2．操作步骤 28](#_Toc440090731)

[四．测试报告 30](#_Toc440090732)

[1．测试程序及测试结果 30](#_Toc440090733)

[2．测试结果分析 37](#_Toc440090734)

[五．总结感想 38](#_Toc440090735)

## 一．需求说明

### 1．文法说明

获取的文法为扩充PL0文法。由于文法不存在左递归，因此无需进行改写。

对文法的分类和解读如下：

1. **程序：**

<程序> ::= <分程序>.

作用：规定了程序是由什么组成的

限定条件：至少有一个分程序和.组成



1. **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>

作用：规定了分程序是由什么组成的

限定条件：可由<常量说明部分>、<变量说明部分>、<过程说明部分>、<函数说明部分>以及<复合语句>中的一个或者多个组成，但至少有一个复合语句

<常量说明部分> ::= const<常量定义>{,<常量定义>};

作用:规定了常量的说明部分形式

限定条件：至少有一个const<常量定义>，多条常量定义之间用,隔开，并且以;结束

<常量定义> ::= <标识符>＝ <常量>

作用：规定了常量的定义形式

限定条件:<标识符>和<常量>必须存在，并且用=连接

<变量说明部分> ::= var <变量说明> ; {<变量说明>;}

作用：规定了变量说明部分形式

限定条件：至少有一个var <变量说明>，多个之间用;隔开

<变量说明> ::= <标识符>{, <标识符>} : <类型>

作用：规定了变量说明是由什么组成的

限定条件：至少有一个标识符，多个标识符之间用,隔开，相同类型的标识符与对应的类型之间用:连接

<过程说明部分> ::= <过程首部><分程序>{;<过程首部><分程序>};

作用：规定了过程说明部分的形式

限定条件：至少有一个<过程首部><分程序>，多个之间用;隔开，并且以;结束

<过程首部> ::= procedure<标识符>[<形式参数表>];

作用：规定了过程首部是由什么组成的

限定条件：必须以procedure<标识符>开头，<形式参数表>可有可无，并且以;结束

<函数说明部分> ::= <函数首部><分程序>{;<函数首部><分程序>};

作用：规定了函数说明部分的形式

限定条件：至少有一个<函数首部><分程序>组成，多个之间用;隔开

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

作用：规定了函数首部是由什么组成的

限定条件：必须以function<标识符>开头，<形式参数表>可有可无，同时对函数定义<基本类型>，用:连接，并且最后以;结束

<复合语句> ::= begin<语句>{; <语句>}end

作用：规定了复合语句是由什么组成的

限定条件：begin和end必须存在，至少有一个<语句>，多条<语句>之间用;隔开



1. **语句：**

<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：规定了语句是由什么组成的

限定条件：可由<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>以及<空>当中的任意一个或者多个组成，但至少要有一个

<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：规定了赋值语句是由什么组成的

限定条件：可由<标识符> := <表达式>、<函数标识符> := <表达式>以及<标识符>'['<表达式>']':= <表达式>当中的任意一个构成，但至少要有一个（不能为空）

<函数标识符> ::= <标识符>

作用：规定了函数标识符是由什么组成的

限定条件：无

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

作用：规定了条件语句是由什么组成的

限定条件：if<条件>then<语句>必须存在，else<语句>可有可无

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

作用：规定了情况语句是由什么组成的

限定条件：case <表达式> of <情况表元素>end必须有，<情况元素>可以有多个，用;隔开

<过程调用语句> ::= <标识符>[<实在参数表>]

作用：规定了过程调用语句是由什么组成的

限定条件：<标识符>必须存在，<实在参数表>可有可无

<复合语句> ::= begin<语句>{; <语句>}end

作用：规定了复合语句是由什么组成的

限定条件：begin和end必须存在，至少有一个<语句>，多条<语句>之间用;隔开

<读语句> ::= read'('<标识符>{,<标识符>}')'

作用：规定了读语句是由什么组成的

限定条件：read必须存在，其后的一个或多个<标识符>（用,隔开）被()括起

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

作用：规定了写语句是由什么组成的

限定条件：write必须存在，其后的内容被()括起可以是<字符串>,<表达式>、<字符串>或<表达式>

<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：规定for循环语句是由什么组成的

限定条件：for、downto|to以及do必须存在，根据<语句>，对<标识符>做操作（步长为1），从初始<表达式>到最终<表达式>，做这么多次的循环操作



情况元素

表达式

1. **表达式：**

<表达式> ::= [+|-]<项>{<加法运算符><项>}

作用：规定了表达式是由什么组成的

限定条件：至少有一个<项>，在<项>前面的+或者-可有可无，同时<项>后面可以有多个<加法运算符><项>组成

<加法运算符> ::= +|-

作用：规定了加法运算符是由什么组成的

限定条件：只能是+或-



1. **项：**

<项> ::= <因子>{<乘法运算符><因子>}

作用：规定了项是由什么组成的

限定条件：至少有一个<因子>，并且<因子>后面可以有多个<乘法运算符><因子>组成

<乘法运算符> ::= \*|/

作用：规定了乘法运算符是由什么组成的

限定条件：只能是\*或/



1. **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：规定了因子是由什么组成的

限定条件：可由<标识符> | <无符号整数>|'('<表达式>')' | <函数调用语句> | <标识符>'['<表达式>']'当中的任意一个构成，但至少有一个（不为空）

<函数调用语句> ::= <标识符>[<实在参数表>]

作用：规定了函数调用语句是由什么组成的

限定条件:无

<实在参数表> ::= '('<实在参数> {, <实在参数>}')'

作用：规定了实在参数表是由什么组成的

限定条件：至少有一个<实在参数>，多个之间用,隔开，并且所有<实在参数>用()括起

<实在参数> ::= <表达式>

作用：规定了实在参数是由什么组成的

限定条件：无



1. **条件：**

<条件> ::= <表达式><关系运算符><表达式>

作用：规定了条件是由什么组成的

限定条件：条件语句中的判定条件

<关系运算符> ::= <|<=|>|>= |=|<>

作用：规定了关系运算符是由什么组成的

限定条件：只能是<或<=或>或>=或=或<>



1. **参数：**

<形式参数表> ::= '(' <形式参数段>{; <形式参数段>}')'

作用：规定了形式参数表是由什么组成的

限定条件:至少有一个<形式参数段>，多个之间同:隔开，并且将所有的<形式参数段>用””括起

<形式参数段> ::= [var]<标识符>{, <标识符>}: <基本类型>

作用:规定了形式参数段是由什么组成的

限定条件：至少有一个<标识符>和<基本类型>，两者之间用:连接，若有多个同类型的标识符，则用,隔开，位于句首的var可有可无

<基本类型> ::= integer | char

作用：规定了基本类型是由什么组成的

限定条件：只能是integer或char



1. **类型：**

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

作用：规定了类型是由什么组成的

限定条件：1.由<基本类型>或者array’[’ <无符号整数>’]’of<基本类型>

2.定义数组时应该提前给出数组大小

<基本类型> ::= integer | char

作用：规定了基本类型是由什么组成的

限定条件：只能是integer或char



1. **常量：**

<常量> ::= [+| -]<无符号整数>|<字符>

作用：规定了常量是由什么组成的

限定条件：由整数（有符号或无符号）或者字符组成

<字符> ::= '<字母>' | '<数字>'

作用：规定了字符是由什么组成的

限定条件：由字母或者数字组成，并且用’’括起



1. **无符号整数：**

<无符号整数> ::= <数字>{<数字>}

作用：规定了无符号整数是由什么组成的

限定条件：至少有一位数字（0可以位于首位）



1. **情况元素表：**

<情况表元素> ::= <常量> : <语句>

作用：规定了情况语句是由什么组成的

限定条件：case <表达式> of <情况表元素>end必须有，<情况元素>可以有多个，用;隔开

1. **标识符：**

<标识符> ::= <字母>{<字母>|<数字>}

作用:规定了标识符是由什么组成的

限定条件：至少有一个字母，其后可以有字母和数字的组合，并且标识符的第一个符号一定是字母



1. **其他：**

<字符串> ::= "{十进制编码为32,33,35-126的ASCII字符}"

作用：规定了字符串是由什么组成的

限定条件：至少有一个十进制编码为323335-126的ASCII字符，并且用””括起



<字母> ::= a|b|c|d…x|y|z |A|B…|Z

作用：规定了字母是由什么组成的

限定条件：只能是a-Z中的某一个字母

<数字> ::= 0|1|2|3…8|9

作用:规定了数字是由什么组成的

限定条件：只能是0-9中的某一个数字

### 2．目标代码说明

生成的目标代码为X86汇编。指令集参考X86指令编码。在WINDOWS平台上使用MASM-32仿真器进行模拟运行。

### 3. 优化方案\*

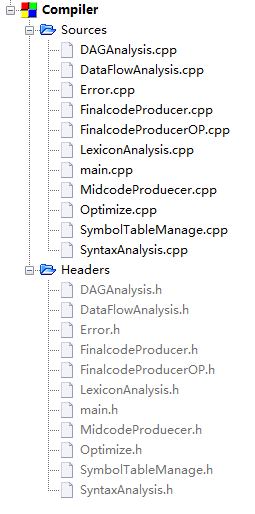
需要完成的优化方案及其要求：

1. 基本块内的公共子表达式删除。要求：对生成的中间代码进行基本块的划分，并对基本块内的中间代码生成DAG图，然后通过启发式算法导出删除了局部公共子表达式的DAG图。
2. 全局寄存器分配。要求：根据生成的冲突图，使用引用计数算法，进行全局寄存器的分配。
3. 数据流分析。要求：完成活跃变量分析。
4. 临时寄存器分配。要求：采用寄存器池对临时寄存器进行分配。

## 二．详细设计

### 1．程序结构

程序的文件组织如下图：



### 2．类/方法/函数功能

##### 2.1 词法分析——LexiconAnalysis



1. **功能**

* 过滤掉源程序中的空格、制表符以及换行符等空白字符和注释；
* 识别各类单词符号；
* 对数字常数完成数字字符串到二进制数值的转换。

1. **核心函数说明**

* void getch()

从文件中获取一个字符

* void clearToken()

清空Token中的内容

* void addToken()

将ch中的内容加入到Token中

* int changeNumber()

将token中的内容转换成数字

* int reserverWords()

判断是否为保留字

* void getsym()

读取单词，并对类型进行判断

##### 2.2语法分析和语义分析——SyntaxAnalysis



1. **功能**

* 按照文法，从源程序符号串中识别出各类语法成分，同时进行语法检查；
* 类型的一致性检查；
* 分析由语法分析所识别出来的语句的意义（即语义）并作相应的语义处理；
* 语义分析程序就要将变量名及有关属性填入符号表中，以备后面使用。

1. **核心函数说明**

* void block()

分程序模块

* void constDeclaration()

常量定义模块

* void varrDeclaration()

变量定义模块

* void procDeclaration()

过程定义模块

* void funcDeclaration()

函数定义模块

* void statement();

语句模块

* operandType expression();

表达式模块

* operandType call();

函数调用模块

* operandType term();

处理项

* operandType factor();

处理因子

* void assignment();

赋值部分

* void compoundStatement();

复合语句部分

* void ifStatement();

if语句部分

* void caseStatement();

case语句部分

* void forStatement();

for语句部分

* void readStatement();

read语句部分

* void writeStatement();

write语句部分

* void program();

总程序模块，处理整个编译过程的开始和结束。

##### 2.3生成中间代码——MidcodeProduecer



1. **功能**

* 生成自定义的中间代码序列

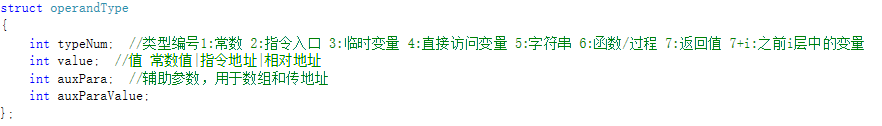
1. **核心函数说明**

* void gen(operationType op, operandType src1, operandType src2, operandType res)

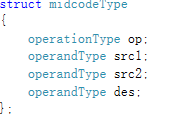
生成中间代码序列

1. **自定义类型说明**

* operandType——操作数类型



* midcodeType——中间代码类型



##### 2.4 代码优化——Optimize



1. **功能**

* 保持语义不变的前提下，提供最优的中间代码。

1. **核心函数说明**

* void Optimize()

调用数据流分析，生成DAG图进行相关优化操作

##### 2.5生成目标代码——FinalcodeProducer



1. **功能**

* 从中间代码生成相应体系结构的最终目标代码

1. **核心函数说明**

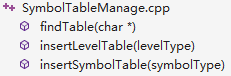
* void FinalcodeProducer()

根据中间代码（四元式）生成X86汇编代码

* int CountTemp(int index, int offset)

统计临时变量的个数，初始化运行栈以适应临时变量

##### 2.6符号表管理——SymbolTableManage



1. **功能**

* 将语义分析得到的信息插入符号表中为以后使用做准备；
* 通过查询符号表获取相应的信息。

1. **核心函数说明**

* bool insertSymbolTable(symbolType tempItem)

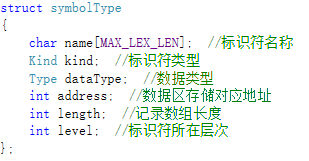
将有关信息插入符号表中，并判断是否成功插入

* findAns findTable(char \*name)

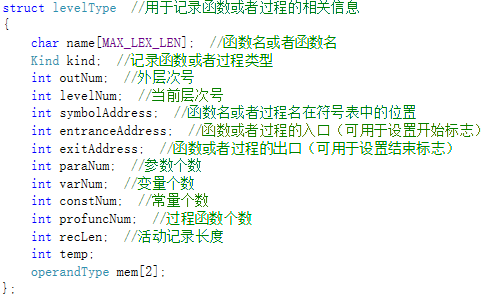
从符号表中查找信息并返回

1. **自定义类型说明**

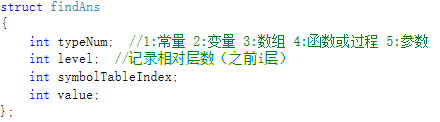
* 标识符表类型



* 层次表类型



* 查询返回类型



##### 2.7出错处理——Error



1. **功能**

* 根据错误编号显示错误信息并且输出具体行数和内容；
* 实现“跳读”功能，使得当出现错误时能够继续向下分析。

1. **核心函数说明**

* void Error(int errorNum)

根据传递进来的错误编号，报告错误信息，输出错误内容，报告错误位置，同时进行适当的“跳读”，继续向下分析

### 3．调用依赖关系



目标代码生成

Final code Producer

FinalcodeProcuder

### 4．符号表管理方案

##### 4.1 符号表说明：

符号表的设计是整个编译器设计中很重要的一部分。一个设计合理的符号表可以极大地减少编程的复杂程度，有利于提升整个编译器的效率。本学期的编译课程设计采用的是PL0文法，该文法支持过程（或函数）间的嵌套，比较繁琐，这也是我们设计中需要特别重视的地方。

课本上介绍的那种支持定位和重定位操作的符号表是一种设计十分合理的符号表结构，推荐使用。这种符号表在查找的时候十分方便，只需要从后向前扫描一遍便可以，并且作用域等问题也在其中解决，十分简单方便。

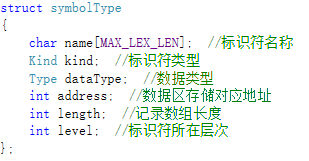
由于在设计符号表的时候轻视课本知识，没有仔细揣摩理解书上那种符号表设计的精髓，于是采用了多表相互组合的方式来存储符号表信息。但是从最终的结果和复杂度看，远远不及课本上所讲的那种。于是在符号表插入过程中判断是否重复定义时，我采用了课本的思路。

所设计的符号表包括一个标识符表和一个层次表。标识符表包含了整个程序中出现的所有出现过的标识符的信息。层次表则是根据不同过程（或函数）来进行划分的，将不同过程（或函数）的变量、常量和参数等区分开来。这种设计的好处在于存储的信息十分全面，并且不会出现丢失情况，但与此同时付出的代价便是浪费了大量的空间，而且查询操作十分的繁琐（但在后面四元式的生成过程中采用新的数据结构得以优化）。

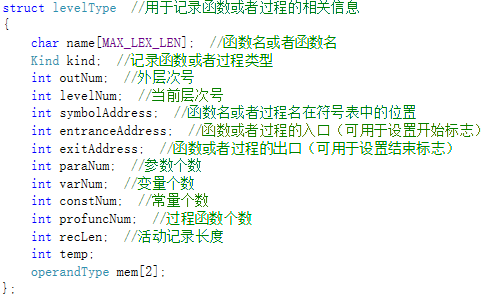
##### 4.2符号表结构：

符号表结构如下：

1. 标识符表：



1. 层次表：



从上面两个表中可以很容易读取出所有的信息，但我们也可以预见所占用的惊人的存储空间。

##### 4.3 符号表操作

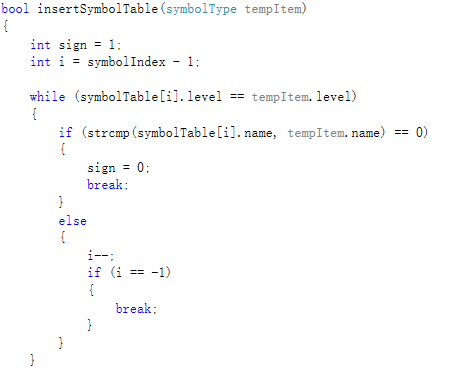
1. 符号表的插入

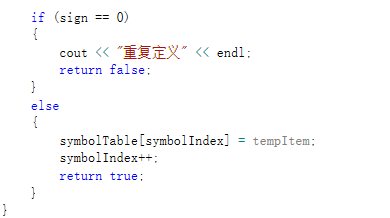
在执行插入操作之前，需要先同该作用域内所有标识符进行比较，判断是否有重复命名的情况出现，若出现重复命名的情况，则转到错误处理程序。这里使用的方法便是书上所讲的反向扫描，直到出现不同层次符号为止。

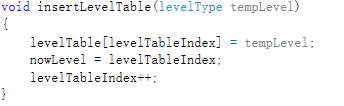
若没有重复命名的情况，则将该标识符及相关的信息插入标识符表中。这里还需要注意一点的是要判断标识符是否为函数名或者过程名，若是，则还需要在层次表中建立新的表项，同时将相应的信息填入到层次表中（可能会出现回填的情况，这也是我们在实际才做当中应当注意的地方）。

成功插入后会返回一个真值（true）作为响应。

相关的代码如下：



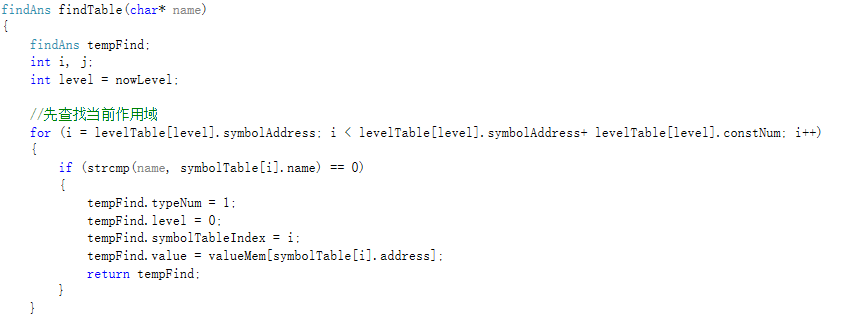


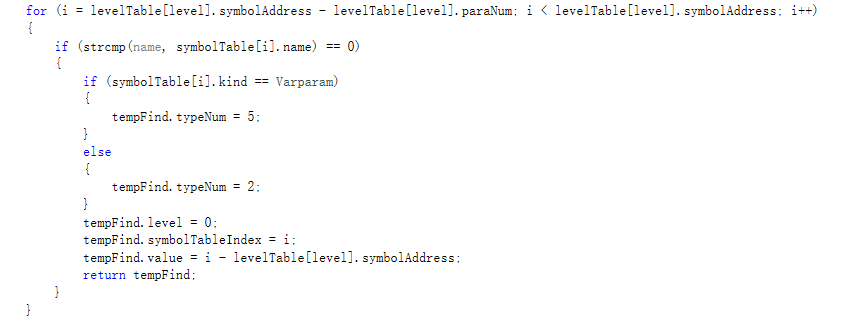


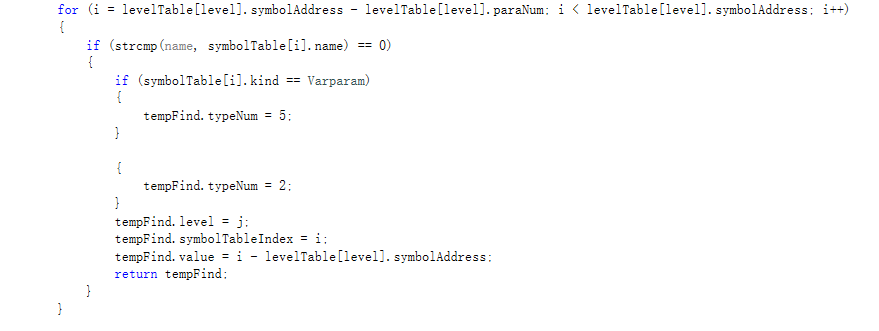
1. 符号表的查询

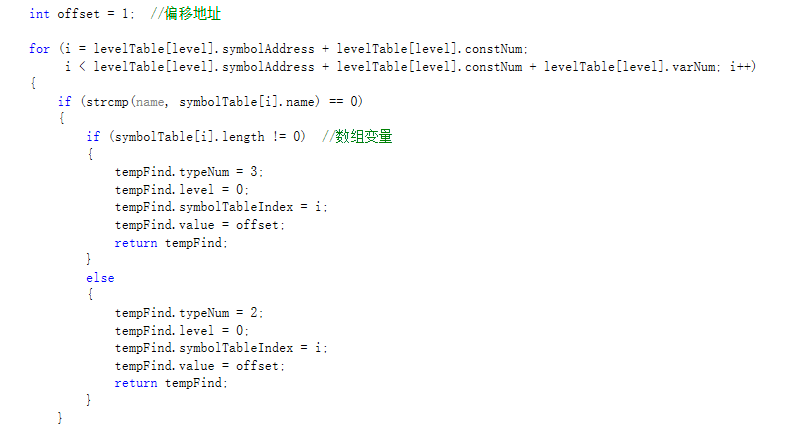
如果要查询一个元素，首先要通过所在的位置来查询层次表中的位置，然后用层次表中保存的标识符表的信息来引导查询标识符表（可是可根据层次表确定该作用域数据在标识符表的上的范围），获得标识符表的内容。

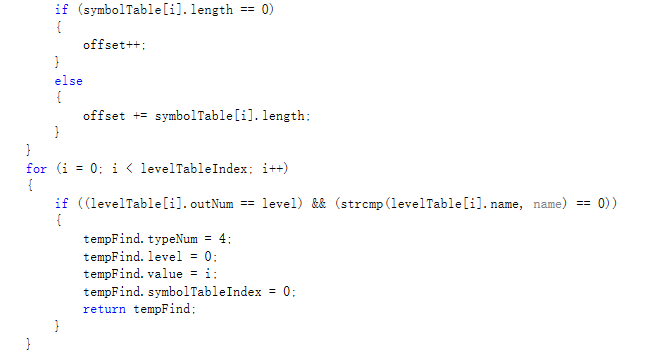
相关的代码如下：

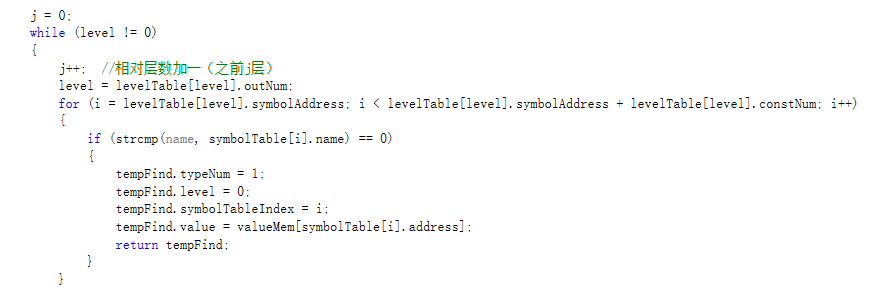


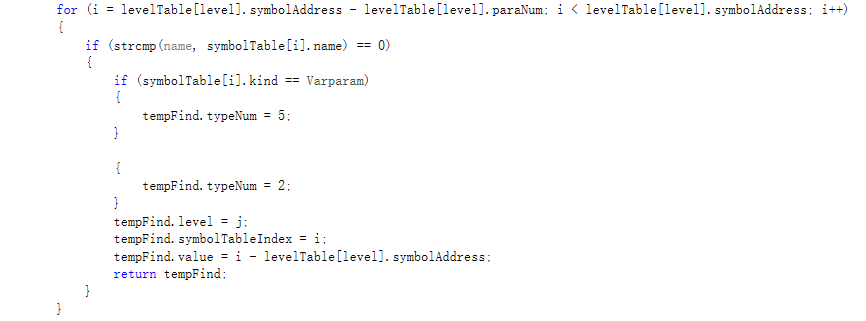


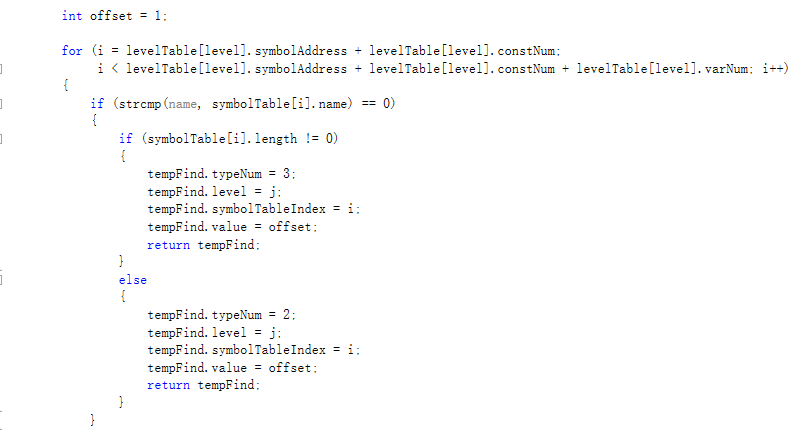


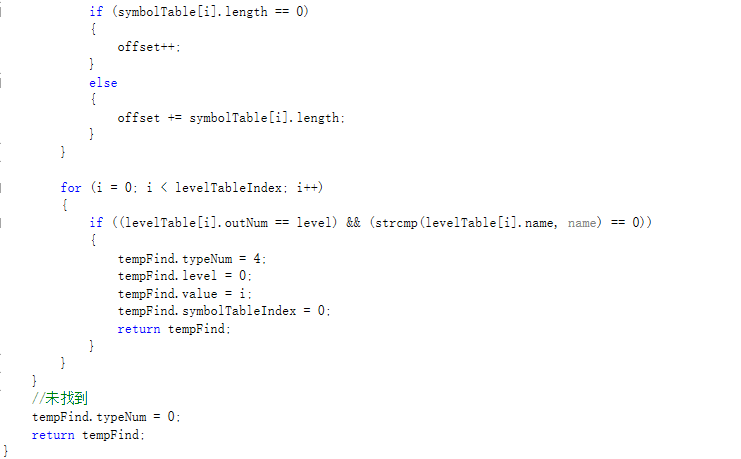












### 5．存储分配方案

采用的是动态存储策略，使用运行栈，并在运行栈中队每个程序模块建立这三个部分：局部数据区、参数区和Display区。

局部参数区用于存储当前所用到的局部数据；

参数区的存储包括隐式参数和显式参数。隐式参数区中包含一个返回地址ret addr，指向前一个活动记录的基地址prev abp和一个存放返回值的位置。

Display区按照层次存放了该过程（或函数）外的各数据区的基地址情况。

在具体的实现过程中，我结合之前设计的层次表和标识符表，加上数据区，便可以实现整个运行栈的功能。其实通过对层次表结构的观察我们可以发现这个表其实就是一个类运行栈的模式。

通过层次表中记录的局部数据区开始位置和每个作用域中变量个数便可以找到所用的局部数据。

在此设计中，将ebp放在了参数和变量之间，因为在函数（或者过程）定义的过程中，我们在一遍扫描中，先扫描到的是参数，然后是所定义的变量。因此，可以通过ebp后的数据的正负来方便的访问参数或者变量，这对于后面中间代码和最终代码的生成有着极大的帮助。

通过层次表中记录的当前层数和外层次号可以容易的访问到函数（或者过程）的外部，并再次通过外部的层次表和相应的标识符表完成对于变量的访问等操作。

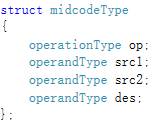
### 6. 解释执行程序\*

选择高难度文法，由于生成最终代码，因此不需要生成解释执行程序。

### 7. 四元式设计\*

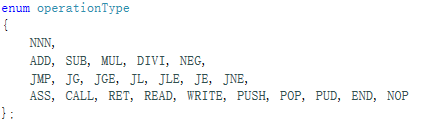
##### 7.1 四元式数据结构定义

1. 四元式结构体：



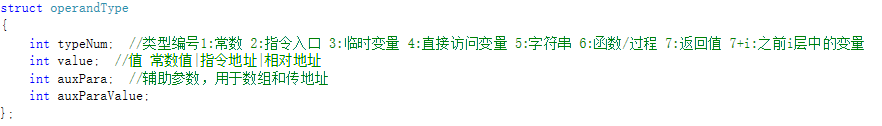
四元式由一个操作符和三个操作数共同组成。其中op记录操作符类型，src1和src2是两个用于运算的操作数，des记录运算结果。具体各个操作数的赋值情况要根据不同的指令进行不同的赋值。

1. 操作符类型：



操作符的命名定义尽量同最终X86汇编代码相似，这将简化后面生成最终代码的工作量，同时也方便检查分析。

1. 操作数结构体：



操作数结构体的设计是本编译器设计中的一个不同点。因为前期符号表的设计偏于复杂，在查询等操作上比较繁琐，效率不高。因此在一遍扫描过程中，选择将有关符号表中的信息存入到操作数当中，这样在生成最终代码时，只需要根据操作数中保存的信息便可以生成最终代码，省去了查找符号表。

同时，生成最终代码时并不需要符号表中所有的信息，采用此结构体也能达到减少存储数据量的目的，提高编译器的效率。

typeNum：用于记录不同类型的标号。

value：根据不同类型，将不同数据对应的值记录（具体值得记录要根据类型决定）。

auxPara：辅助参数，用于记录数组变量、变量传递是否传地址等单纯无法通过typeNum便可区分的类型。

AuxParaValue：将这些变量的值记录存储。

##### 7.2 四元式操作解释

1. 运算操作符：

ADD src1, src2, res 加法操作：res = src1 + src2

SUB src1, src2, des 减法操作：des = src1 - src2

MUL src1, src2, des 乘法操作：des = src1 \* src2

DIVI src1, src2, des 除法操作：des = src1 / src2

NEG src1, nop, des 取反操作：des = - src1

1. 跳转操作符：

JMP nop, nop, des 无条件跳转至des

JG src1, src2, des 如果src1 > src2，则跳转至des

JL src1, src2, des 如果src1 < src2，则跳转至des

JGE src1, src2, des 如果src1 >= src2，则跳转至des

JLE src1, src2, des 如果src1 <= src2，则跳转至des

JE src1, src2, des 如果src1 = src2，则跳转至des

JNE src1, src2, des 如果src1 <> src2，则跳转至des

1. 基本操作符

ASS src1, nop, des 赋值des = src1

CALL src1 函数调用，src1为调用函数名

RET nop, nop, nop 函数返回

READ src1, nop, des 读入，并写入des，src1为写入的数据类型

WRITE src1, src2, nop 将src1输出，src2为输出的数据类型

PUSH src1, src2, nop src1 入栈，src2为参数

POP des 出栈至des

PUD des, nop, nop

NOP nop, nop, nop 空语句

END nop, nop, nop 函数结束标志

### 8. 目标代码生成方案\*

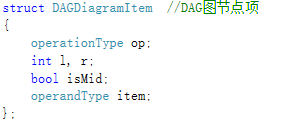
根据之前已经生成的四元式中间代码序列，对应每一条四元式中间代码生成对应的X86汇编代码即可。

因为之前在一遍扫描生成四元式中间代码序列时，已经将各个操作数的类别、属性以及数值等信息存入了每一个操作数结构体中，因此在生成中间代码时我们不再需要查询符号表（这是针对本编译器中设计的符号表过于复杂而导致查询操作繁琐所进行的一点优化），直接使用操作数所存储的信息就可以完成对目标代码的生成。

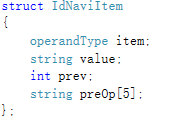
### 9. 优化方案\*

##### 9.1优化方案使用的数据结构

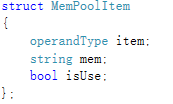
* DAG图节点项：



* 数据流表项：



* 寄存器池项：



##### 9.2优化方案具体算法

* **DAG图消除公共子表达式**

基本块的划分是代码部分重要的基础算法之一，在DAG图优化和数据流分析法分配全局寄存器中都有使用，其代码的实现可以按照书上所讲：

1、首先确定入口语句（每个基本块的第一条语句）的集合

（1）整个语句序列的第一条语句属于入口语句

（2）任何能由条件/无条件跳转语句转移到的第一条语句属于入口语句

（3）紧跟在跳转语句之后的第一条语句属于入口语句

2、每个入口语句直到下一个入口语句，或者程序结束，它们之间的所有语句都属于同一个基本块。

在此要注意，call语句和ret语句都应该算作跳转，而不能仅仅把跳转理解成JMP、JG等。

在划分好基本块后，我们对于每一个基本块构建它自身的DAG图，其算法与书上所讲较为一致，用树形的结构存储DAG图，用一个数组维护节点列表。

而后使用启发式算法生成四元式，需要注意的点有：

1. 四元式的空余部分可以使用NOP操作数补齐。
2. 叶子节点的值要存储，不能随意分配。
3. 启发式算法生成的四元式的顺序尽量要与原四元式的顺序一致，否则会出现错误。可以采用倒序扫描的方式来解决这一问题。

* **数据流分析法分配全局寄存器**

数据流分析中，主要针对的是每个过程中各个基本块间数据流动的过程。

首先要划分基本块，然后根据基本块中各个元素定义和使用的先后来生成Use数组和Def数组。继而可以得到数据流入的集合和数据流出集合。其中和间的关系如下：

通过反复的循环操作，直到In和Out数组的内容不再变化，则得到我们所要的数据流情况。

然后我们可以通过得到的In数组建立冲突图（同一个In数组内的元素相互冲突）。

再由启发式染色算法得到全局寄存器的分配情况。

* **引用计数法分配全局寄存器**

引用计数法也是分配全局寄存器的方式之一，其实现较为简单，只需统计各个元素的使用次数，将寄存器分配给使用最多的两个元素就好。

这里之所以出现两种全局寄存器分配方式，是因为X86汇编只有三个全局寄存器可供使用。因为寄存器数目较少，数据流分析的优点不一定能够体现出来，故实现两种全局寄存器的分配方式，通过实践来决定使用哪种优化方式。

使用X86汇编，较为简单的引用计数法会占据较大的优势。

* **寄存器池分配局部寄存器**

通过寄存器池的方法来分配寄存器可以极大的减少同一数据反复分配寄存器的情况，通过寄存器池的调配令局部寄存器的使用效率大大提高。但实际操作中还需要考虑各种特殊情况，比如除法操作时必须借用EAX与EDX，比如说某些操作需要先取地址，再从地址中取值，一下使用多个寄存器的情况。其实现起来还是有一定的难度，且效果并不理想，与想象中具有一定的差距。

### 10. 出错处理

##### 10.1出错处理方案

出错处理贯穿在词法分析、语法分析和语义分析过程中，是一个综合性很强的模块。并且由于错误信息的多样性以及出错的不确定等等因素导致出错处理的难度增加。而衡量一个编译器的好坏一个很重要的因素便是编译器是否能够正确的处理出错情况，因此，好的出错处理方案尤为关键。

出错处理的主要关注点有两个，一个是报错的准确性，能够准确的报出错误信息和错误所在位置，这是最基本的一点，也是我们很容易实现的一点；另一个是错误局部化处理，尽可能设法将错误限制在一个局部的范围内，避免错误扩散和影响程序其他部分的分析和检查。相应的方法有很多，但是实现起来有一定的难度。在这里我们采取的是“跳读”的方法，即当我们检测到错误时，向后跳跃读取字符，直到我们读到某一终结符或者某一特定符号为止，例如:分号、右括号、有中括号、关键字end等等，这时候再开始正常的向后分析。

这一方法实现起来相对容易，但是也存在着明显的缺点，便是对“跳”多少的把握不准确。“跳”多了便会导致分析的不准确，“跳”少了便会受到错误的影响。因此，把握“跳”的度是关键。在这个问题的处理上本人做的也不是很好，仍有待改进。目前是对不同的错误采用不同的“跳读”方式。

##### 10.2错误信息定义

1：字符缺少 '

2：非法字符

3：字符串缺少 "

4：程序未以 . 结束

5：在句子之前缺少 ;

6：缺少 ;

7：重复定义，符号表插入失败

8：数组定义数据类型不合法

9：缺少保留字 of

10：缺少 ]

11：数组长度类型错误

12：缺少 [

13：缺少保留字 array

14：缺少 :

15：常量赋值符号不合法，非 :=

16：常量数据类型不合法

17：常量赋值符号不合法

18：标识符定义不合法

19：函数未以 ; 结束

20：过程未以 ; 结束

21：函数数据类型不合法

22：缺少保留字 end

23：缺少保留字 begin

24：不合法语句

25：赋值语句标识符类型不合法

26：缺少 :=

27：缺少保留字 then

28：缺少保留字 if

29：case语句情况元素表中常量类型不合法

30：缺少保留字 case

31：查询符号表失败，未定义

32：缺少 )

33：缺少 (

34：缺少保留字 read

35：缺少保留字 write

36：缺少保留字 downto 或 to

37：缺少保留字 do

38：缺少保留字 for

39：var类型和参数类型不匹配，需要使用变量

40：调用语句缺少 , 或 ;

41：参数数据类型不合法

42：变量数据类型不合法

43：参数个数不匹配

44：参数个数为0，调用过程或函数无参数

45：参数个数为0，不应有括号

46：因子类型错误

47：write语句格式不合法

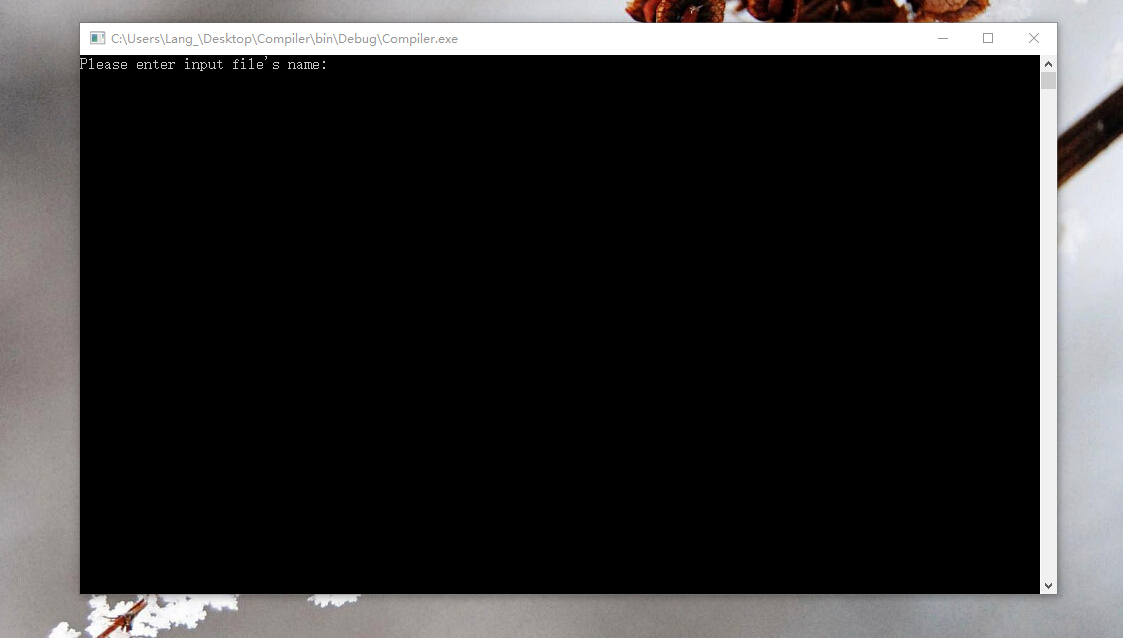
## 三．操作说明

### 1．运行环境

* 开发语言：C++
* 开发环境：Windows 7及其以上、CodeBlocks 13.12
* 汇编环境：MASM-32

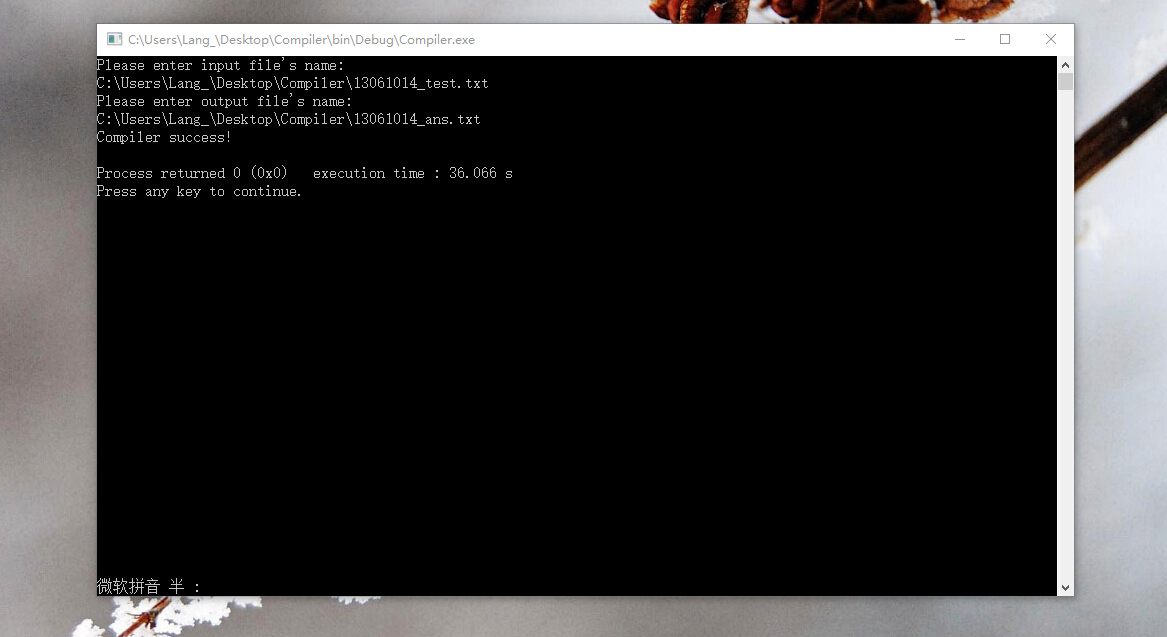
### 2．操作步骤

1. 打开程序，编译通过后运行程序，输入源程序的路径和目标程序的输出路径。

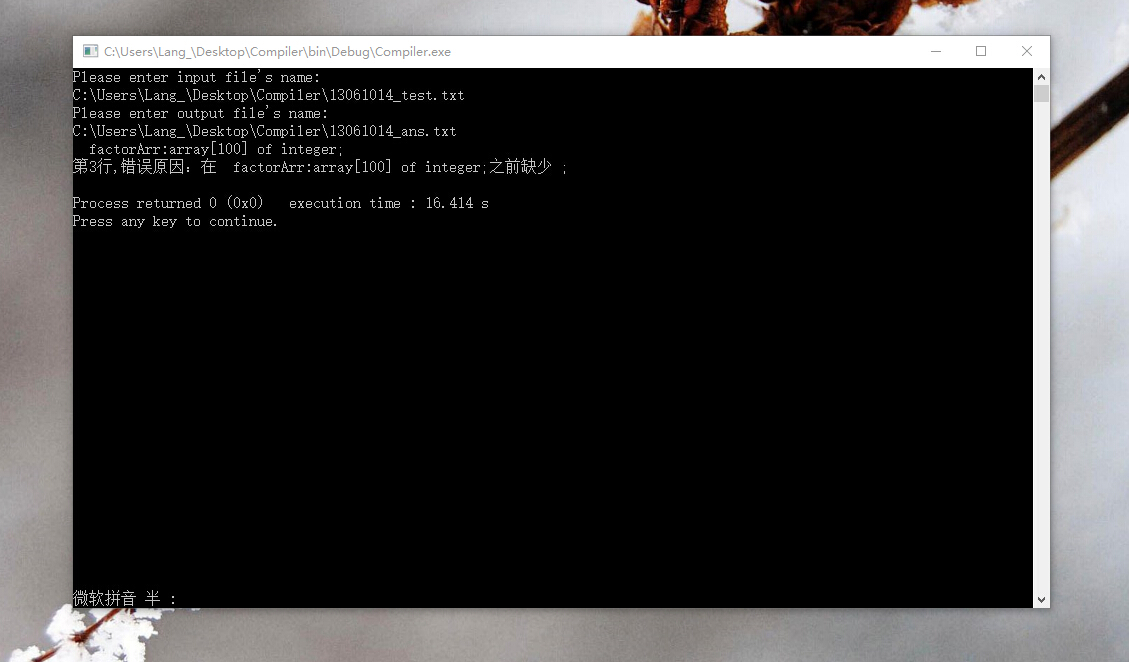


1. 如果编译成功，则会显示：Compiler success！如果编译失败，则会显示错误信息和错误所在行号。

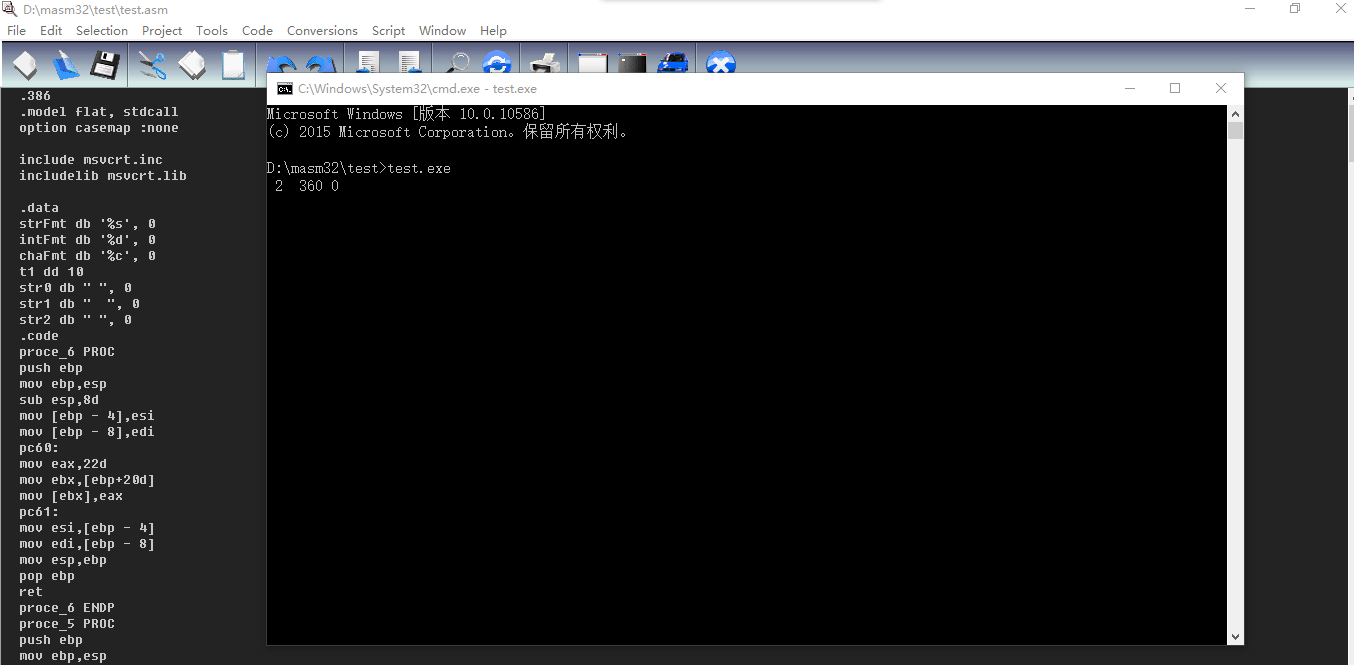
编译成功：



编译失败：



1. 在MASM-32中模拟运行，查看运行结果。



## 四．测试报告

### 1．测试程序及测试结果

1. var a,b,c:integer;

ch:char;

begin

read(a);

write(a);

write("The number is ",a);

read(ch);

write(ch);

write("The char is ",ch);

read(a,b,c);

write("The area of this picture",(a + b - c) \* c / 2);

end.

1. const z = 1;

var m,n,i:integer;

a:array[8] of integer;

function gcd(m,n:integer):integer;

begin

if n=0

then gcd:=m

else

begin

m := m - m / n \* n;

gcd:=gcd(n,m)

end

end;

begin

n:=3;

m:=15;

read(n,m);

write(gcd(m,n))

end.

1. var i:integer;

begin

read(i);

case i of

1: i:=i+1;

2: ;

3: i:=i+3;

100:

end;

write(i)

end.

1. const OP1='a',OP2='b',ASCIIZERO=+48,ERRORCODE=-1;

var d1,d2,r1,r2,rop,asci,i:integer;

a:array[100] of integer;

op:char;

string:array[2]of char;

procedure sort(var para1,para2:integer);

var i,j,max,temp:integer;

procedure sortPrint;

begin

write("after swap")

end;

begin

if para1>=para2 then

begin

temp:=para1;

para1:=para2;

para2:=temp ;

sortPrint

end

else

end

;

function timesByR(para3:integer):integer;

begin

if para3>1 then

timesByR:=para3\*timesByR(para3-1)

else

timesByR:=para3

end

;

function timesByFor(para4:integer):integer;

var k,l:integer;

begin

l:=1;

for k:=1 to para4 do

l:=l\*k

;

timesByFor:=l

end

;

begin

read(i);

a[i]:=330;

write(a[i]);

d1:=4;

d2:=3;

op:=97;

string[0]:=48;

ascii:=string[0]+ASCIIZERO;

sort(d1,d2);

r1:=timesByR(d1)\*1;

r2:=timesByFor(d1)\*1;

if r1<>r2 then

write("error\n",ERRORCODE)

else

write("correct",r1)

;

case op of

'a':rop:=d1+d2;

'b':rop:=d1-d2

end

;

write(ascii);

write(rop);

write("end of main")

end

.

1. const const1=001,const2=3,const3=-3,const4=+3,const5='a'

var

factorArr:array[100] of integer;

alphaArr:array[26] of char;

i,j,n,ans:integer;

a,b:char;

function multi(n:integer):integer;

begin

if n<=1 then

multi:=1

else

multi:=n\*multi(n-1)

end

;

function multi1(n:integer):integer;

var ans,i:integer;

begin

i:=0;

ans:=i+1;

if n>1 then

for i:=2 to n do

ans:=ans\*i

;

multi1:=ans

end

;

function multi2:integer;

begin

multi2:=multi(4)-multi1(3)

end

;

procedure swap(var a,b:char);

var temp:char;

begin

temp:=a;

a:=b;

b:=temp

end

;

procedure add(var i:integer;step:integer);

begin

i:=i+step;

end

;

procedure sub(var i:integer;step:integer);

begin

i:=i-step;

end

;

procedure output;

begin

write("Hello World!")

end

;

begin

j:=1;

for i:=0 to 25 do

alphaArr[i]:=const5+i;

for i:=25 downto 0 do

write(alphaArr[i]);

output;

write("Please enter a number between 1 and 10.");

read(n);

if n<=10 then

if n>=1 then

begin

factorArr[j]:=multi(n);

add(j,1);

factorArr[j]:=multi1(n);

sub(j,2);

factorArr[j]:=multi2;

write("The answer is ",factorArr[3]/factorArr[1])

end

else

write("Your input is less than 1")

else

write("Your input is greater than 10");

read(ans);

case ans of

-1:write("first input is less than second input",ans);

0:write("first input is equal to second input",ans);

1:write("first input is greater than second input",ans)

end;

read(a,b);

swap(a,b);

write(a,b)

end.

1. var ch:char;

begin

read(a);

write(a);

write("this is the ch: ");

read(ch);

write(ch);

write("thie is the num we read :",a);

end.

1. const a=12345678;

var

c:char;

begin

c=a;

write(c);

end.

1. const a=45,b=27;

var x,y,g,m :integer;

q:array[5] of integer;

p:arry[10] of char;

begin

q[1]:=a/b;

p[1]:=355;

write("q[1]=",q[1]);

write(" p[1]=",p[1])

end.

1. const x=3

var a, b , c ,d , e :integer;

begin

a:=2 + (2-3);

b:=3 + 2 \* 3 \* 0 + (2-3);

c:=3 + (2-3) \* (2\*3);

d:=-2 + 2 \* 3 ;

if a>b then begin

c:=(a+b)\*(a-b)/(a+b)-(a-b);

d:=(a-b)+(a-b)\*(a+b);

e:= c -d;

end else begin

c:=(a+b)\*(a-b)/(a+b)-(a-b);

d:=(a-b)+(a-b)\*(a+b);

e:= c + d;

end;

write(" a= ",a);

write(" b= ",b);

write(" c= ",c);

write(" d= ",d);

write(" e= ",e);

end.

1. var i,j:integer;

procedure h1(var i:integer);

procedure h2(var i:integer);

procedure h3(var i:integer);

begin

i:=9

end;

begin

h3(i)

end;

begin

h2(i)

end;

begin

h1(i,j);

write(i)

end.

### 2．测试结果分析

1. 覆盖了所有的读语句和写语句的情况；

覆盖了字符串；

覆盖了加减乘除运算；

覆盖了括号；

覆盖了变量的定义；

覆盖了分程序。

1. 覆盖了常量的定义；

覆盖了变量的定义，包括普通变量和数组变量；

覆盖了函数定义；

覆盖了条件语句；

覆盖了参数传递；

覆盖了运算；

覆盖了赋值语句。

1. 覆盖了case语句；

覆盖了空语句。

1. 覆盖了常量定义；

覆盖了变量（基本变量和数组变量）定义；

覆盖了标识符；

覆盖了关键字和保留字；

覆盖了基本运算；

覆盖了比较运算；

覆盖了函数定义；

覆盖了函数调用；

覆盖了过程定义；

覆盖了过程调用；

覆盖了参数传递；

覆盖了赋值语句；

覆盖了读语句；

覆盖了写语句；

覆盖了for语句；

覆盖了条件语句；

覆盖了case语句；

覆盖了表达式；

覆盖了项；

覆盖了因子；

覆盖了分程序；

1. 覆盖了常量定义；

覆盖了变量（基本变量和数组变量）定义；

覆盖了标识符；

覆盖了关键字和保留字；

覆盖了基本运算；

覆盖了比较运算；

覆盖了函数定义；

覆盖了函数调用；

覆盖了过程定义；

覆盖了过程调用；

覆盖了参数传递；

覆盖了赋值语句；

覆盖了读语句；

覆盖了写语句；

覆盖了for语句；

覆盖了条件语句；

覆盖了case语句；

覆盖了表达式；

覆盖了项；

覆盖了因子；

覆盖了分程序；

1. 错误原因：标识符未定义。

1. 错误原因：缺少:=

1. 错误原因：定义变量的数据类型不合法

1. 错误原因：缺少;

1. 错误原因：参数个数不匹配