**编译原理课程设计**

**——**简单编译器的设计与实现

|  |  |
| --- | --- |
| 姓名 | 分工 |

[摘要 4](#_Toc471663862)

[1.概述 5](#_Toc471663863)

[2.课程设计任务及要求 6](#_Toc471663864)

[2.1 设计任务 6](#_Toc471663865)

[2.2 设计要求 6](#_Toc471663866)

[3.算法及数据结构 7](#_Toc471663867)

[3.1算法的总体思想（流程） 7](#_Toc471663868)

[3.2词法分析模块 8](#_Toc471663869)

[3.2.1功能 8](#_Toc471663870)

[3.2.2数据结构 8](#_Toc471663871)

[3.2.3算法 10](#_Toc471663872)

[3.3语法分析（含语义分析和中间代码生成）模块 11](#_Toc471663873)

[3.3.1功能 11](#_Toc471663874)

[3.3.2数据结构 13](#_Toc471663875)

[3.3.3算法 16](#_Toc471663876)

[3.4中间代码优化模块 19](#_Toc471663877)

[3.4.1功能 19](#_Toc471663878)

[3.4.2数据结构 20](#_Toc471663879)

[3.4.3算法 20](#_Toc471663880)

[3.5目标代码生成模块 23](#_Toc471663881)

[3.5.1功能 23](#_Toc471663882)

[3.5.2数据结构 24](#_Toc471663883)

[3.5.3算法 24](#_Toc471663884)

[4.程序设计与实现 26](#_Toc471663885)

[4.1程序流程图 26](#_Toc471663886)

[4.2 程序说明 26](#_Toc471663887)

[4.3实验结果 32](#_Toc471663888)

[5.系统特色 40](#_Toc471663889)

[6.结论 41](#_Toc471663890)

[7.参考文献 41](#_Toc471663891)

[8.收获、体会和建议 41](#_Toc471663892)

## 摘要

一个编译器所进行的工作一般可以划分为五个阶段：词法分析、语法分析、语义分析和中间代码产生、中间代码的优化、目标代码生成。

我们设计了并且实现了一个简单的类C语言编译器，该编译器拥有完整的前端和后端，能够进行基本的编译功能并产生可执行文件向屏幕输出源程序的运行结果。

该编译器的词法分析器可以识别绝大部分标准C语言支持的词法符号，该词法分析器可以过滤空格、Tab和回车，并且支持注释功能。该词法分析器主要通过有限自动机的状态跳转来实现，根据自动机结束状态来得到该单词的TOKEN值。该模块具有词法错误位置提示功能。

该编译器的语法部分采用了递归下降子程序的文法分析方法，所设计的文法支持了函数、函数类型声明、变量类型声明、变量定义、表达式语句、if条件语句和while循环语句以及简单输出功能。在表达式语句方面，我们设计了支持所有算术运算、关系运算、逻辑运算和位运算功能的语法结构，并且语法上支持一维数组和结构体。该编译器语义分析和生成四元式阶段能够对变量定义和语法错误进行检测，能够识别出未定义的标识符和重复定义标识符，该阶段最终实现生成中间代码——四元式。

该编译器拥有中间代码优化模块，采用DAG优化算法按基本块对四元式进行了优化，该编译器的目标代码是8086汇编语言代码，能够实现将优化后的四元式序列转化生成可执行的汇编语言文件，并且运行执行该文件，向屏幕输出运算结果。

该编译程序的主要特色是能够将最终结果输出显示到屏幕，并且该编译程序能够支持前置++和后置++这种语法，拥有类似C语言的这种简便性。

**关键词**：编译原理，词法分析，语法分析，四元式，DAG算法

### 1.概述

编译原理课程兼有很强的理论性和实践性，是计算机专业的一门非常重要的专业基础课程，在系统软件中占有十分重要的地位。编译原理课程设计是本课程重要的综合实践教学环节，是对平时实验的一个补充。通过编译器相关子系统的设计，使学生能够更好地掌握编译原理的基本理论和编译程序构造的基本方法和技巧，融会贯通本课程所学专业理论知识；培养学生独立分析问题、解决问题的能力，以及系统软件设计的能力；培养学生的创新能力及团队协作精神。

一个编译器所进行的工作一般可以划分为五个阶段：词法分析、语法分析、语义分析和中间代码产生、中间代码的优化、目标代码生成。

首先是词法分析，针对词法分析，我们设计了一个可以识别绝大部分标准C语言支持的词法符号，该词法分析器可以过滤空格、Tab和回车，并且支持注释功能，即过滤掉注释符号$后面的代码。该词法通过有限自动机的状态跳转来实现，根据自动机结束状态来得到该单词的TOKEN值，词法分析器在识别到一个单词后，将该单词记录下来，如果是数据，则会在符号表的相应位置记录它的值，如果是标识符，则会先在符号表上进行查询，若没有则将其记录到符号表上，并将相应TOKEN的指针指向表中该位置。

接下来进行语法分析，在语法分析部分，会对所编写的代码的语法进行检验，看是否合乎我们所设定的语法规则，这里我们采用了递归下降子程序的文法分析方法，所设计的文法支持了函数、函数类型声明、变量类型声明、变量定义、表达式语句、if条件语句和while循环语句以及cout简单输出功能。在表达式语句方面，我们设计了支持所有算术运算、关系运算、逻辑运算和位运算功能的语法结构，并且语法上支持一维数组和结构体。

在语义分析和中间代码产生的阶段。我们在语法分析程序的相应部分加上了语义动作，实现将输入的语句转换成可识别的中间代码——四元式形式。在语义分析部分，主要做的工作是在识别到数据和标识符时将它们压入语义栈，当语法分析到需要生成相应四元式时，执行构建四元式的动作，并将语义栈中的数据作为四元式的数据组成部分。还有就是符号表的构建，当语法分析到变量的定义语句时，需要将被定义的变量记录到符号表内，在记录之前需要检查符号表内该符号是否已经被定义，从而检查重复定义的错误。语义分析和中间代码产生是与语法分析同时进行的，当语法分析结束后，若输入内容的语法正确，则可以获得相应的四元式序列。

然后是优化阶段。在优化阶段，我们主要运用了DAG优化算法。首先将编译器前端生成的四元式划分基本块，然后对每一个基本块执行DAG优化算法，通过该算法可以删除多余的赋值操作和无用的中间变量，从而减少四元式的数量，得到更简洁的四元式序列。

最后一个阶段是目标代码生成。该阶段将四元式进一步翻译生成相应的目标代码，我们所选定的目标代码是8086汇编语言代码，所以该阶段的任务是将优化后的四元式序列转化生成可执行的汇编语言文件，从而进一步运行执行该文件，向屏幕输出运算结果。

### 2.课程设计任务及要求

#### 2.1 设计任务

设计一个简单的文法编译器，该编译器包括完整的前端和后端。该编译器可以划分为词法分析、语法分析和语义动作、中间代码优化和目标代码生成四个模块。

词法分析：

能够识别标准C语言所支持的大部分词法，能够进行简单的词法错误判断，输出错误提示，并且具有注释功能，能够过滤掉无用符号，如空格、Tab和回车等。

语法分析和语义动作：

能够支持基本类似于C语言的基本语法。语法上能够支持声明语句、基本类型的变量定义，如整型、实型和字符型数据；能够支持表达式语句，包括赋值表达式、算术表达式、逻辑表达式、关系表达式和位运算表达式；能够支持if条件语句和while循环语句。

中间代码优化：

采用DAG优化算法，能够对四元式进行优化，从而减少中间代码的数量，提高编译后生成的程序的效率。

目标代码生成：

将中间代码生成汇编语言代码，并且实现目标代码的运行，从而验证编译器编译结果的正确性。

#### 2.2 设计要求

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

2、设计系统的数据结构和程序结构，设计每个模块的处理流程。要求设计合理；

3、编程序实现系统，要求运行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.算法及数据结构

#### 3.1算法的总体思想（流程）

我们将编译器的设计按照功能划分为以下四个部分，通过这四个部分来实现编译器的全部功能。如下图。



图3.1 算法总体思想图

词法分析器用来对输入的字符流进行词法分析，识别每一个单词并生成相应的TOKEN码，并记录它们的数据信息。

语法分析和语义分析以及中间代码的生成被划分为一个模块是因为它们的不可分割性，当某个语法分析通过后，就开始产生语义动作生成相应的四元式，因此该部分用来对词法分析产生的TOKEN序列进行语法分析，从而在符号表中记录相关数据信息，并且产生中间代码。

中间代码优化模块是为了简化中间代码而设计的，该部分通过四元式构建相应的无向图，通过算法产生优化后的四元式序列。

目标代码生成模块用于产生汇编代码，将优化后的四元式进行翻译，生成对应的汇编语言。

该编译器包含了前端和后端的基本功能，能够进行词法错误检测、语法错误检测、标识符定义错误检测，并提示错误行数，且编译后生成可以直接运行的汇编程序，可以算是一个较为完整的简单编译器。

#### 3.2词法分析模块

##### 3.2.1功能

通过将源程序输入该模块，词法分析器能够进行分析，检测词法是否正确，从而生成相应的TOKEN码，并向符号表中记录，如果遇到错误词法，则记录在错误表中。

该词法分析器所能识别的词法包括以下几个部分。

整数：TOKEN值为0

实数：TOKEN值为1

字符：TOKEN值为2

字符串：TOKEN值为3

标识符：TOKEN值为4

关键字：TOKEN值为10-49

现有能够识别的关键字有：

auto,short,int,long,real,double,char,struct,union,enum,typedef,const,unsigned,signed,extern,register,static,volatile,void,if,else,switch,case,for,do,while,goto,continue,break,default,sizeof,return,bool,cout.

界符：TOKEN值为50-69

现有能够识别的界符有：

‘{’，‘}’，‘(’，‘)’，‘[’，‘]’，‘”’，‘’’，‘;’，‘,’，‘.’，‘->’，‘?’，‘#’

运算符：TOKEN值为70-99

现有能够识别的运算符有：

‘+’，‘-’，‘\*’，‘/’，‘%’，‘++’，‘--’，‘=’，‘>’，‘>=’，‘<’，‘<=’，‘==’，‘!=’，‘&&’，‘||’，‘!’，‘&’，‘|’，‘^’，‘~’，‘>>’，‘<<’

##### 3.2.2数据结构

关键字表、界符表和运算符表都采用二维数组的形式进行存储。TOKEN表采用结构体的数据结构进行存储，里面包含该单词的token码，该单词的所在行数，并且会根据该单词的类型指向相应的符号表区域。错误表也采用结构体形式，其中包含错误的行数、错误的类型和错误的内容等信息，用来向用户报告源程序的相关词法、语法和声明错误。

char KT[34][15]={"auto","short","int","long","real",

"double","char","struct","union","enum","typedef",

"const","unsigned","signed","extern","register","static",

"volatile","void","if","else","switch","case","for",

"do","while","goto","continue","break","default",

"sizeof","return","bool","cout"

};//关键字表

char PT[14][3]{"{","}","(",")","[","]",

"\"","\'",";",",",".","->","?","#"

};//界符表

char OT[23][3]{"+","-","\*","/","%","++","--","=",

">",">=","<","<=","==","!=",

"&&","||","!",

"&","|","^","~",">>","<<"

};//运算符表

typedef struct ERRORL{//错误记录表

int line;//错误行数

char type;//错误类型，

char name[100];//错误内容

struct ERRORL\* next;

}ERRORL;

typedef struct TOKEN{//TOKEN记录

char token[3];//token码

int line;//行数记录，系统编码标记

struct INT\* tpI;//整型指针

struct REAL\* tpR;//实型指针

struct CHAR\* tpC;//字符指针

struct SYNBL\* tpIT;//标识符表指针

char\* tpKT;//关键字指针

char\* tpPT;//界符指针

char\* tpOT;//运算符指针

struct TOKEN\* next;

}TOKEN;

typedef struct Identifier{//标识符表

struct SYNBL\* tp;

struct Identifier\* next;

}Identifier;

typedef struct INT{//整数型表

int numi;

}INT;

typedef struct REAL{//实数型表

float numf;

}REAL;

typedef struct CHAR{//字符型表

char numc;

}CHAR;

typedef struct SYNBL{//标识符表

char name[100];//标识符名字

struct TYPEL\* type;//类型表指针

char cat;//种类，目前有：f(函数),a(数组),d(结构),v(变量),n(换名形参),g(赋值形参)

char L;//活跃信息

struct PFINFL\* pfinfl;//函数表指针

struct LENL\* lenl;//长度表指针

struct VALL\* vall;//活动记录表指针

struct SYNBL\* next;

}SYNBL;

##### 3.2.3算法

主要的算法流程图如下所示，该词法模块在读取字符后首先会检验该字符是否是空格、Tab和回车等需要过滤的字符，如果是则不读，如果遇到注释符号$，则该行均不读，直到遇到换行符才开始读取下一行。若不是这些字符则执行下面的算法。



图3.2 词法分析算法

#### 3.3语法分析（含语义分析和中间代码生成）模块

##### 3.3.1功能

该模块接收词法分析模块得到TOKEN序列，通过对TOKEN序列的语法分析，判断该语法是否合法，如果合法则执行相应的语义动作，生成中间代码。

在语法分析模块，我们设计了比较接近于C语言的语法结构，虽然不如C语言语法那么灵活，但是依然能够满足用户基本的需求，源程序的编写与C语言类似，对C语言用户来说没有陌生感。

文法结构如下（这里的{}代表可以循环生成，符号’{’外面有单引号）：

函数体文法：

Start -> Function{Function}

Function -> FType i4 ‘(’ Parameter ‘)’ ‘{‘ Declaration Content ‘}’

类型文法：

FType -> ‘void’|Type

Type -> ‘int’|’real’|’char’

Parameter -> |’void’|Type Id { ‘,’ Type Id }

Id -> i4|i4 ‘[‘ i1 ‘]’

结构体声明文法：

Declaration -> {|’struct’ i4 ‘{‘ Base\_declaration ‘}’ ‘;’|Base\_declaration }

基本类型定义文法：

Base\_declaration -> |{ Type Id { ‘,’ Id } ‘;’ }

语句块文法：

Content -> Structure { Structure }

Structure -> |E ‘;’|If|While|Fun|Cout

函数调用文法：

Fun -> i4 ‘(‘ ‘)’ ‘;’|i4 ‘(‘ Assignment { ‘,’ Assignment } ‘)’ ‘;’

if条件语句文法：

If -> ‘if’ ‘(‘ Expression ‘)’ ‘{‘ Content ‘}’|

‘if’ ‘(‘ Expression ‘)’ ‘{‘ Content ‘}’ ‘else’ ‘{‘ Content ‘}’

while循环语句文法：

While -> ‘while’ ‘(‘ Expression ‘)’ ‘{‘ Content ‘}’

逗号表达式文法：

Expression -> Assignment { ‘,’ Assignment }

赋值表达式文法：

Assignment -> i4 ‘.’ Id ‘=’ Logical\_or | Id ‘=’ Logical\_or | Logical\_or

逻辑表达式文法：

Logical\_or -> Logical\_and { ‘||’ Logical\_and }

Logical\_and -> Inclusive\_or { ‘&&’ Inclusive\_or }

Inclusive\_or ->Exclusive\_or { ‘|’ Exclusive\_or }

Exclusive\_or ->And { ‘^’ And }

And -> Equality { ‘&’ Equality }

关系表达式文法：

Equality -> Relational { w0 Relational }

Relational -> Shift { w1 Shift }

算术表达式文法：

Shift -> Additive { w2 Additive }

Additive -> Multiplicative { w3 Multiplicative }

Multiplicative -> Unary { w4 Unary }

前置算符文法：

Unary -> w5 Postfix | Postfix

后置算符文法：

Postfix ->Primary { w6 }

Primary ->i4|constant| ‘(‘ Expression ‘)’

其中符号表示：

w0 -> ‘==’ | ‘!=’

w1 -> ‘<’ | ‘>’ | ‘<=’ | ‘>=’

w2 -> ‘<<’ | ‘>>’

w3 -> ‘+’ | ‘-‘

w4 -> ‘\*’ | ‘/’ | ‘%’

w5 -> ‘!’ | ‘~’ | ‘++’ | ‘--‘ | ‘sizeof’

w6 -> ‘++’ | ‘--‘ | ‘[‘ Expression ‘]’ | ‘.’ i4

其中constant为整数、实数或者字符类型数据，i4为用户自定义标识符，可以由字母、数字和下划线组成，不能以数字开头，并且不能与关键字相同。

该模块还要执行相应的语义功能以及中间代码的生成。语义功能主要包括变量的定义、表达式四元式的生成、if条件语句和while循环语句的四元式生成等。

变量的定义需要实现与符号表进行信息交换的功能，该变量定义的文法通过检验后，会向符号表中查找该标识符名称，如果没有查到则将该变量的类型和相关信息写入符号表；如果在符号表中查询到该标识符，则说明该变量名已经被定义，于是向错误表中写入标识符重复定义的错误信息，并标注错误行数。

表达式和if、while语句在相应文法检验的过程中，会对操作数进行压栈处理，当文法检验成功后，会弹出操作数生成相应的四元式序列。

当所有的语句都通过了文法检验并且生成了相应的四元式后，该模块会对四元式中的操作数进行检测，如果该操作数是用户自定义的标识符，就去符号表中查询，如果出现符号表中查询不到的标识符，则说明该标识符未被定义，此时向错误表中写入标识符未定义的错误信息，并标注错误行数。

##### 3.3.2数据结构

语法分析过程是对词法分析生成的token序列进行分析，判断是否符合语法标准，因此不需要新的数据结构，只需要获取到token序列即可。

语义分析的过程则需要额外的数据结构来对语义动作和相应操作进行存储。其中在生成四元式的过程中需要用到语义栈来存储当前的token，以便当识别到算符语法结束位置时从语义栈中弹出token生成相应的四元式序列。另外，在语法分析的过程中，还有变量、数组的定义和结构体的声明，这些语法检验通过后也要通过执行相应的语义动作将相关信息填写入符号表内。

因此这里新增了一个语义栈结构体，用来存储生成四元式所需的操作数，还开辟了一个四元式链表，用来存储生成的四元式，四元式中的运算符是通过二维数组的数据结构来保存的。同时符号表中还需要类型表、数组表和结构体表，其中类型表用来存储标识符的类型，如果该标识符是数组或者结构体，则需要额外的数组表或结构体表来存储数据之间的关系。

typedef struct SEM{//语义栈

struct TOKEN\* tpToken;

struct SEM\* fron;

struct SEM\* next;

}SEM;

typedef struct Operand{//操作数

struct TOKEN\* tpToken;

}Operand;

char OperatorL[37][5]{"+","-","\*","/","%","++","--","=",

">",">=","<","<=","==","!=",

"&&","||","!",

"&","|","^","~",">>","<<",

"gt","lb","if","el","ie","wh","do","we","inc","dec","arr","si",".","co"

};

typedef struct Quaternary{//四元式

char\* Operator;//操作符表

struct Operand\* operand[3];//3个操作数

struct Quaternary\* next;

}Quaternary;

typedef struct TYPEL{//类型表

char tval;//类码，类型代码，决定下列指针选择，现有类码为：i(整型),r(实型),c(字符型),a(数组),d(结构体)

struct AINFL\* ainfl;//数组表指针

struct RINFL\* rinfl;//结构体表指针

}TYPEL;

typedef struct AINFL{//数组表

int low;//数组下界

int up;//数组上界

struct TYPEL\* ctp;//成分类型指针，指向该维数组成分的类型的指针

int clen;//成分类型的长度，成分类型数据所占值单元的个数

}AINFL;

typedef struct RINFL{//结构表

char ID[10];//结构域名

int OFF;//(区距)—是idk的值单元首址相对于所在记录值区区头位置

struct TYPEL\* TP;//指针，指向idk域成分类型(在类型表中的信息)

}RINFL;

##### 3.3.3算法



图3.3 赋值语法的算法



图3.4 逻辑或算法



图3.5 数据生成部分算法



图3.6 if条件语句算法



图3.7 while循环语句算法

#### 3.4中间代码优化模块

##### 3.4.1功能

该部分的功能是对从前端得到的四元式进行优化，目的是为了精简代码数量，使得编译后的程序能够更高效的运行，这里我们采用了DAG算法对中间代码进行优化。

首先对前端输出的四元式进行基本块划分，然后对一个基本块执行DAG算法，构造一个无向图，从该图中获取到优化后的四元式序列。之后清空该图，对之后的基本块执行相同的算法，直到将所有基本块的四元式序列都优化为止，最后输出优化后的四元式序列。

##### 3.4.2数据结构

中间代码优化部分需要建立一个无向图，通过相应算法来实现对基本块的优化，因此这里需要一个图的数据结构，该图结点有左右孩子、编号、运算符、标记和前后指针。其中单独创建了一个标记链表的数据结构，首个标记作为主标记，主标记后面所连接的标记作为从标记。运算符是用来记录四元式的运算符的，左右孩子是为了记录单目运算符和双目运算符所指向的操作数，标记是为了记录操作数，而前后指针是为了在DAG算法中更方便地查找删除重复的非用户定义标识符。

此外创建了一个块链表，用来存储分块后的四元式，从而方便将四元式按照基本块进行划分保存，以便进行DAG算法优化。

typedef struct Block{

struct Quaternary\* qua\_block;//分块后的四元式块集合

struct Block\* next;//下一个块

}Block;

typedef struct DAG{//无向图

int num;

char\* oper;

struct Mark\* mark;

struct DAG\* lchild;

struct DAG\* rchild;

struct DAG\* next;

struct DAG\* fron;

}DAG;

typedef struct Mark{//标记

struct TOKEN\* name;

struct Mark\* fron;

struct Mark\* next;

}Mark;

##### 3.4.3算法

划分基本块的算法如下：



图3.8 构建基本块算法

构建DAG优化图的算法如下所示：



图3.9 DAG优化创建算法

从DAG中获取优化后四元式的算法如下图所示：



图3.10 优化后四元式获取算法

#### 3.5目标代码生成模块

##### 3.5.1功能

该模块的功能是通过优化后的四元式序列生成目标代码——汇编语言。该部分首先会对得到的四元式序列进行分析，对四元式序列中的操作数进行判断，如果该操作数是系统生成的中间变量（该中间变量不能在运算化简中被舍弃），那么就会在符号表中查找，看符号表中是否存在该变量，如果不存在该变量，则向符号表中写入该变量，并标注变量类型，从而方便接下来的目标代码生成。

按基本块顺序读取优化后的四元式序列，根据四元式的运算符进行判断，生成相应的汇编代码，当所有基本块都被读完后，汇编代码生成完毕，最后将汇编代码输出到ASM文件中，运行DOS将其编译连接并运行，得到源程序的运行结果。

##### 3.5.2数据结构

该部分需要一个数据结构来存储目标语句，这里采用链表的数据结构，使用一个目标语句链表来保存目标语句的相关信息，其中包含存储的目标代码语句，该语句所处的位置标号以及当前语句的类型。

当前语句的类型目前有4种，一种是0，表示此语句为一般语句，另外3种分别为i、e、d，分别代表if、else、do的四元式产生的目标语句，这些语句因为涉及到跳转，需要当读取到它们跳转到的语句时才能获得地址信息，所以它们所生成的目标语句的跳转地址需要回填，这个语句类型作为回填时查找该语句的标志来使用。

typedef struct Object{//目标语句

char code[30];//存储的目标代码

int lab;//标号

char type;//需要回填的类型,0为不需要回填，i为if的回填，e为else的回填，d为do的回填

struct Object\* fron;

struct Object\* next;

}Object;

##### 3.5.3算法

目标代码的生成算法如下图所示：



图3.11 目标代码生成算法

### 4.程序设计与实现

#### 4.1程序流程图



图4.1 程序运行流程图

#### 4.2 程序说明

程序的编写使用的是C++语言，采用了类的封装，将整个编译器拆分为了四个模块，分别是词法分析器模块、语法分析（含语义分析和中间代码生成）模块、中间代码优化模块和目标代码生成模块，各个模块由上到下依次继承，实现了良好的封装性。

下面按照每个模块类的设计对各个模块所含有的成员和方法进行说明。

词法分析：

class Scanner{//词法分析器模块

private:

char ch,ch\_before;//当前词，前一个词

int state,state\_before;//状态，前状态

int line,warn;//行数，警告

void CreatNewToken(TOKEN\*& t);//创建新的Token结点

void reset(int& state,char\* code,int& i,int& warn);//重置自动机状态

int state\_change(int st,char ch,int& warn);//自动机状态转换

void state\_to\_code(TOKEN\* t, int state\_before, char code[100], int line, int warn);//根据自动机状态生成Token序列

public:

struct TOKEN\* token;//token头指针

struct ERRORL\* error\_head;//错误表头指针

struct ERRORL\* error\_now;//错误表当前位置

Scanner(){token=NULL;error\_head=NULL;error\_now=NULL;}//构造函数

void Scan();//词法分析主函数

int Findexist(char\* code, TOKEN\*& p);//查询符号表函数

void ConvertItoS(int i, string& s);//类型转换，int to string

void ConvertFtoS(float f,string& st);//类型转换，float to string

void ConvertStoC(string st,char\* c);//类型转换，string to char

void ConvertStoI(string st,int& i);//类型转换，string to int

void ConvertStoF(string st,float& f);//类型转换，string to float

void CoutErrorL();//输出错误表

};

语法分析、语义分析和中间代码生成：

class GrammaticalAnalysis:public Scanner{//语法分析和语义动作及

//中间代码生成模块

private:

//语法部分

struct TOKEN\* ch;//当前词

int Start();//语法，开始

int Function();//语法，函数

int Parameter();//语法，参数

int FType();//语法，函数类型

int Type();//语法，变量类型

int Declaration();//语法，声明

int Base\_declaration();//语法，基本类型声明

int Id();//语法，基本类型

int Id\_Expression();//语法，数据类型

int Content();//语法，内容

int Structure();//语法，结构

int Expression();//语法，表达式

int Cout();//语法，输出函数

int Assignment();//语法，赋值

int Logical\_or();//语法，逻辑或

int Logical\_and();//语法，逻辑与

int Inclusive\_or();//语法，或

int Exclusive\_or();//语法，异或

int And();//语法，位与

int Equality();//语法，相等

int Relational();//语法，不等

int Shift();//语法，移位

int Additive();//语法，加法减法

int Multiplicative();//语法，乘法除法

int Unary();//语法，前置算符

int Postfix();//语法，后置算符

int Primary();//语法，标识符、常数生成

int IF();//语法，if条件语句

int While();//语法，while循环语句

int Fun();//语法，函数语句

//语义部分

int d;//结构体标记

struct TOKEN\* type;//类型标记

struct TOKEN\* ch\_sem;//入语义栈的词

struct TOKEN\* operand\_now;//当前操作数

struct SEM\* top;//栈顶

struct SEM\* base;//栈底

struct Quaternary\* quater\_now;//当前四元式指针

void CreateSEM();//生成一个语义栈

void PushStack();//入栈

void DeStack();//出栈

int mark;//标明运算符位置

int counter;//系统变量计数器

void CreateQuaternary();//生成一个四元式

void PushQuaternary();//入四元式队列

void Bi\_oper\_qua();//双目算符的操作数生成

void Unary\_oper\_qua();//单目算符的操作数生成

void Assign\_oper\_qua();//赋值操作数生成

void If\_While\_qua();//IF条件,WHILE循环操作数生成

void GetMark(TOKEN\* ch\_ope);//获取运算符

void GetMark\_front(TOKEN\* ch\_ope);//获取前置运算符

int FindSynbl();//查重

void PushSynbl();//添加进符号表

int CoutSynbl();//输出符号表

void CheckSynbl();//检查标识符是否被定义，未定义则记录到错误表中

public:

void Grammar();

//语法部分

int result;

GrammaticalAnalysis(){token=NULL;counter=1;d=0;error\_head=NULL;error\_now=NULL;}

//语义部分

struct Operand\* operand[3];//语义操作数

struct Quaternary\* quater;//四元式

struct Identifier\* id;//用户定义标识符

struct Identifier\* id\_now;//当前标识符指针

void CoutQuaternary();//输出四元式

};

中间代码优化：

class Optimization:public GrammaticalAnalysis{//中间代码优化模块

private:

struct Quaternary\* optimize\_now;//当前优化四元式

struct Block\* optimize\_block\_now;//优化后的当前块

struct Block\* block\_now;//当前块指针

struct Quaternary\* qua;//四元式

struct DAG\* dag\_head;//DAG头指针

struct DAG\* dag\_tail;//DAG尾指针

struct DAG\* dag\_now;//DAG当前指针

struct Operand\* find\_now\_num1;//当前查找词

struct Operand\* find\_now\_num2;//当前查找词

char\* find\_now\_ope;//当前查找运算符

struct Operand\* add\_now;//当前需要加入的结点

char\* op\_now;//当前运算符

struct DAG\* find\_begin;//查找开始处

struct DAG\* pos;//找到的位置

struct Mark\* mark\_pos;//找到的标记位置

struct DAG\* temp\_dag;//待交换标记所在的dag结点

struct Mark\* temp1;//待交换标记1

struct Mark\* temp2;//待交换标记2

void CreateOptimize();//创建一个优化后四元式头部

void CreateOptimize\_Block();//创建一个优化后块头部

void DivideBlock();//块划分函数

void CoutBlock();//按基本块输出四元式

void CreateDAG();//创建一个DAG

void DeleteDAG();//删除DAG

void AddDAG();//加入一个DAG点

int FindDAG();//检查重复DAG结点

int FindDAG\_unary();//检查单目运算符重复DAG结点

int FindDAG\_Bi();//检查双目运算符重复DAG结点

int MainMark();//检查该标记是不是主标记

void SwopMark();//交换两个标记

void DeleteMark();//删除一个标记

void GetDAG();//得到DAG生成结果

void CoutOptimize();//输出优化后的四元式

public:

struct Quaternary\* optimize;//优化四元式

struct Block\* optimize\_block;//优化后的块

struct Block\* block;//块指针

void Optimize();

Optimization(){dag\_head=NULL;dag\_tail=NULL;dag\_now=NULL;}

};

目标代码生成：

class ObjectCode:public Optimization{//目标代码生成模块

private:

struct Object\* obj\_now;//当前目标代码

struct Object\* obj\_find;//回溯找到的目标代码位置

struct Block\* blo;

struct Quaternary\* qua;

struct Operand\* num;//当前操作数

char numType;//操作数的类型

int find\_num;//需要查找的系统变量名称

int lab;//标号

char type\_find;//需要查找的目标代码类型，i为if，e为else，d为do，用于回溯重填跳转位置

int SystemExist();//查找符号表中是否有必须的中间变量

void UpdateSymbol();//更新符号表，将系统生成中间变量放入

void GetObjCode();//生成目标代码

void AddObject();//加入obj结点

void CoutObj();//输出目标代码

void Judge(string& st);//获取操作数类型，存到numType

int Find\_obj();//从后向前寻找相应类型的首个目标代码语句

int Find\_obj\_head();//从前向后寻找相应类型的首个目标代码语句

//目标代码生成函数（代码段具体语句）

void MOV\_BX\_A();

void MOV\_BX\_B();

void MOV\_CX\_B();

void MOV\_C\_BX();

void MOV\_C\_CX();

void MOV\_A\_BX();

void MOV\_AX\_A();

void MOV\_C\_AX();

void MOV\_DX\_ZERO();

void MOV\_C\_DX();

void MOV\_CX\_ONE();

void MOV\_CX\_ZERO();

void CMP\_AX\_BX();

void CMP\_AX\_ZERO();

void CMP\_BX\_ZERO();

void GetHead();//目标代码头部生成(含数据段)

void GetTail();//目标代码尾部生成

public:

struct Object\* obj;//目标代码指针

void ObjCode();//目标代码主函数

ObjectCode(){lab=0;}//构造函数

};

#### 4.3实验结果

**测试样例1：**

void main(){

int a,b;

struct A{

int c;

int d;

};

a=1;

b=1;

while(b<5){

if(a%2==0){

cout a;

}

else{

b++;

}

++a;

}

}

**词法分析模块：**

输出结果如下所示，显示为结果为行数、token值、单词。

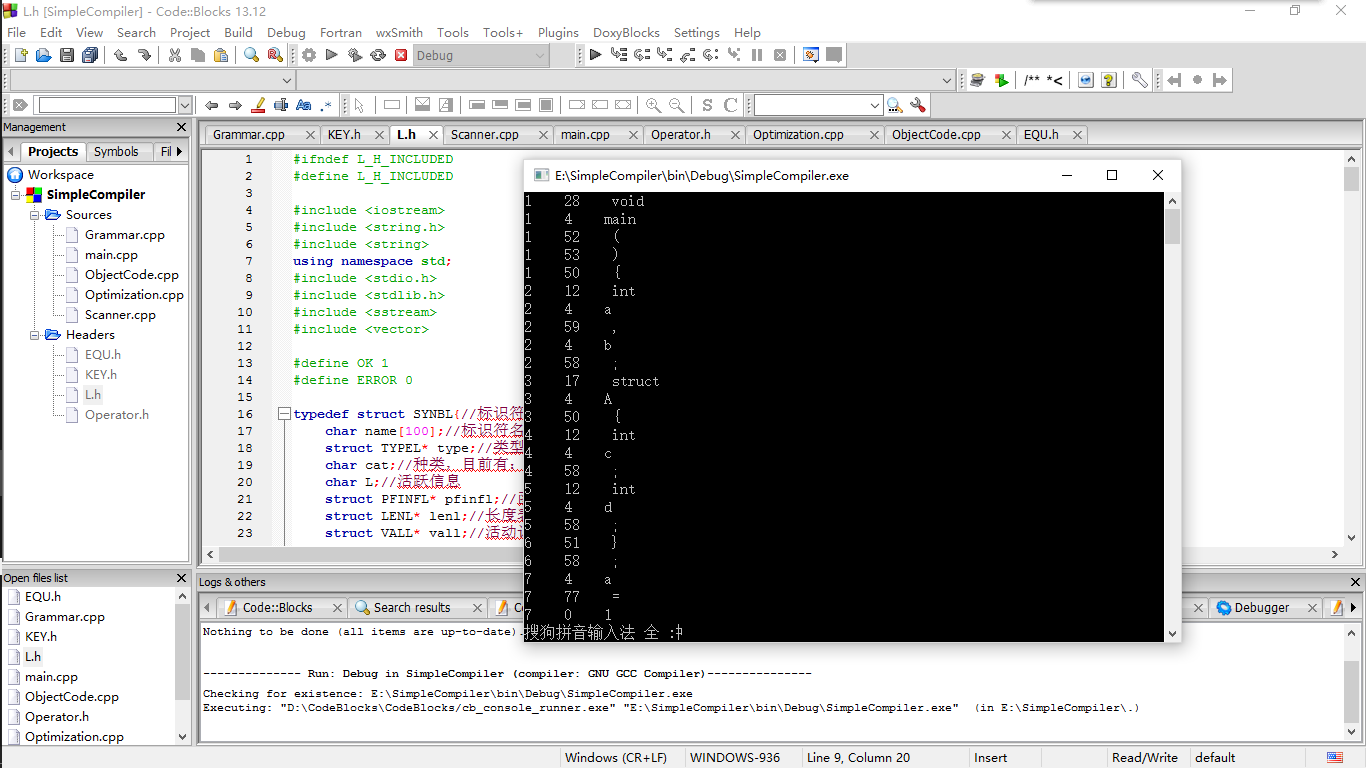
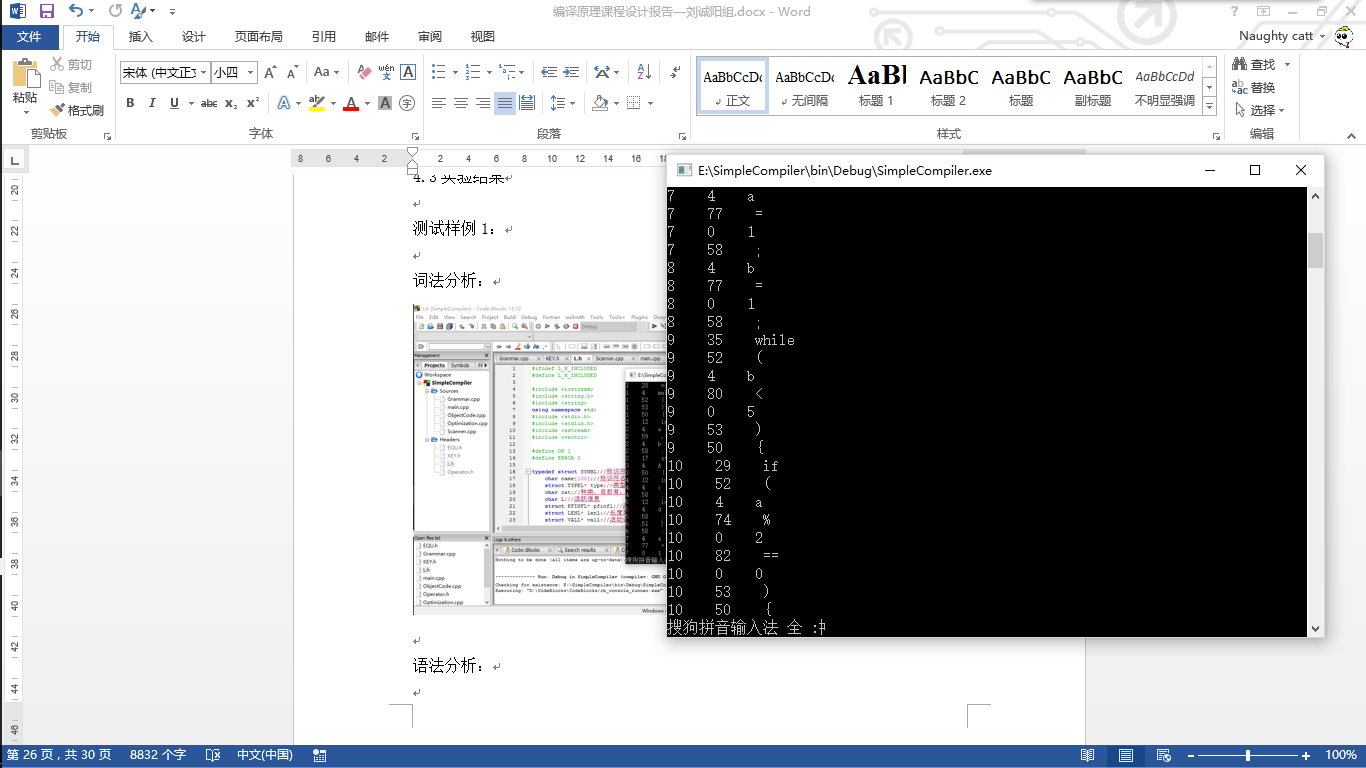
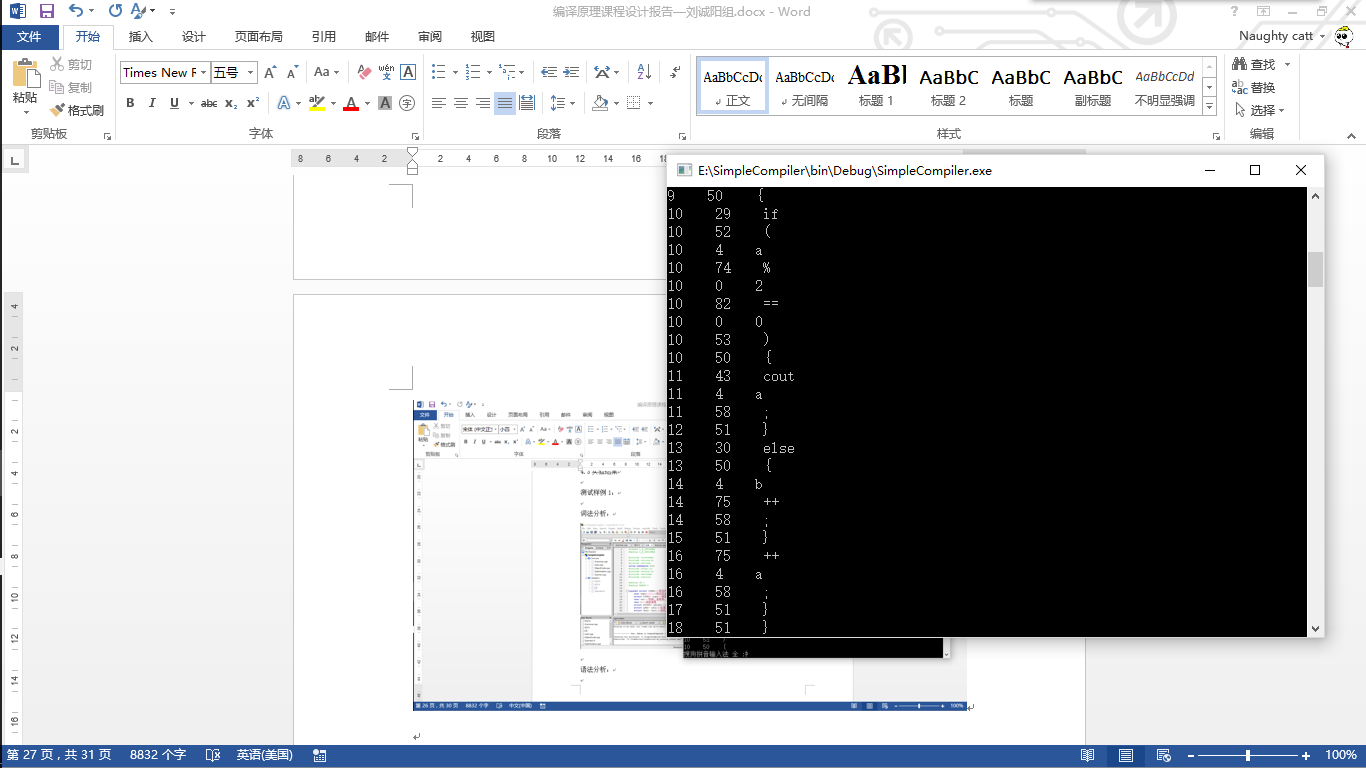
  

图4.2 词法分析结果

**语法分析模块：**

变量的定义会被保存在符号表中，用于之后的标识符检测和最后的目标代码生成阶段，符号表存储结果如下图。

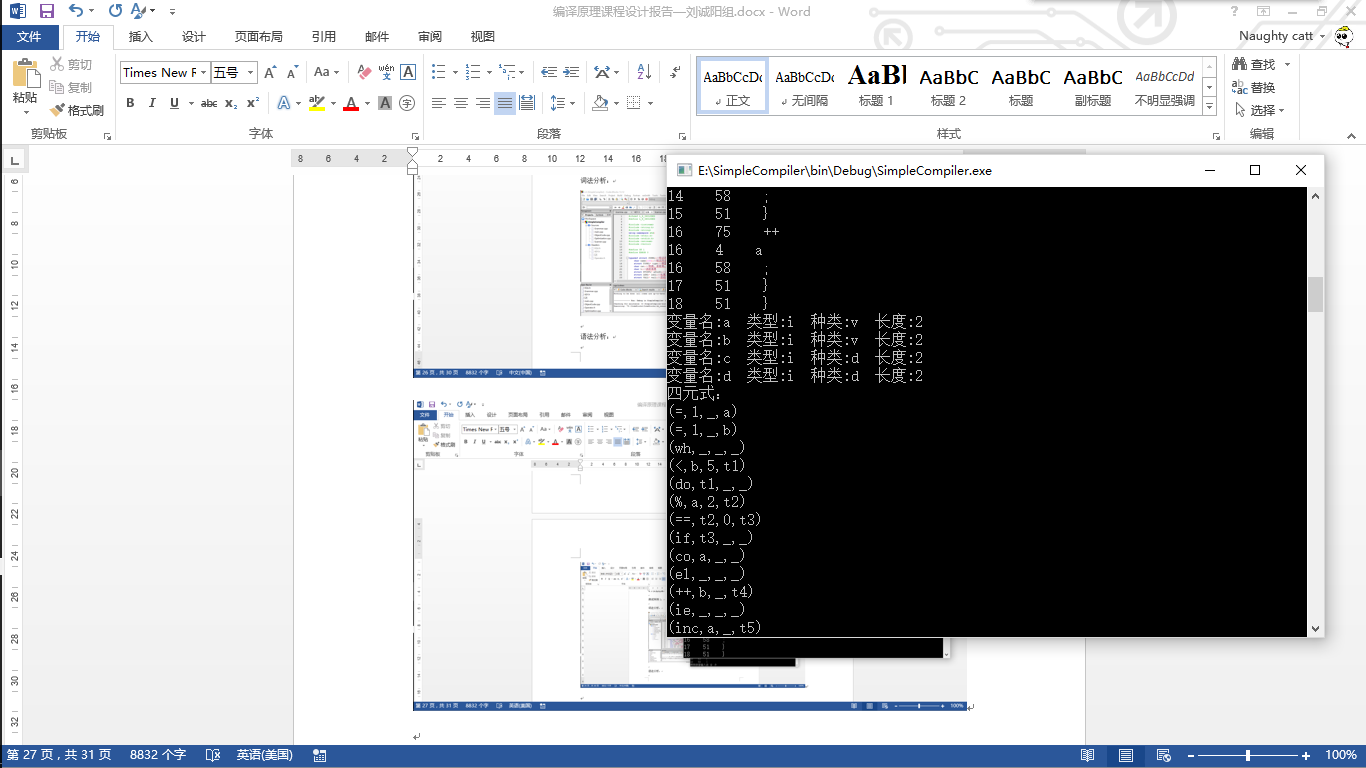


图4.3 符号表显示

该模块会执行相应的语义动作并生成中间代码——四元式，产生的四元式结果如下图所示。

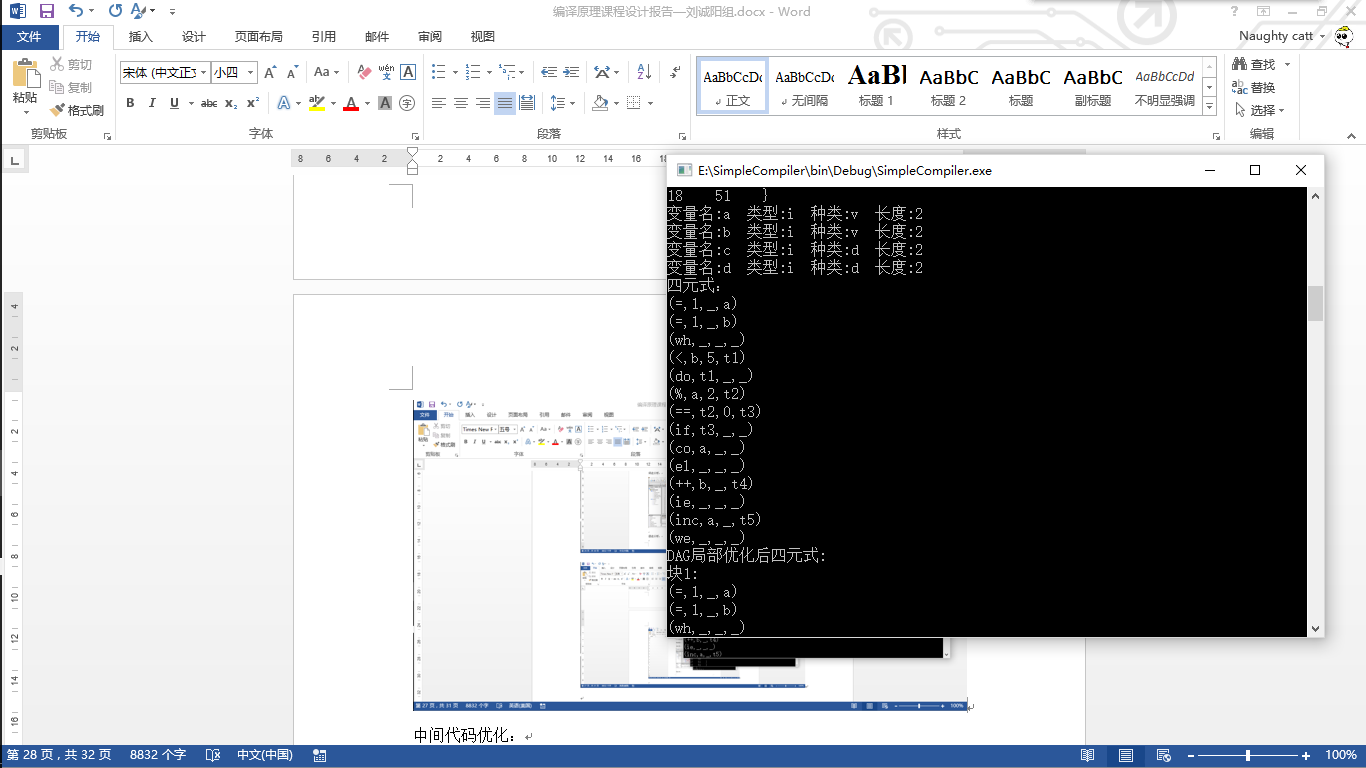


图4.4 四元式显示结果

**中间代码优化：**

中间代码优化采用DAG算法按照基本块进行四元式优化，得到的最后优化四元式结果如下图所示。

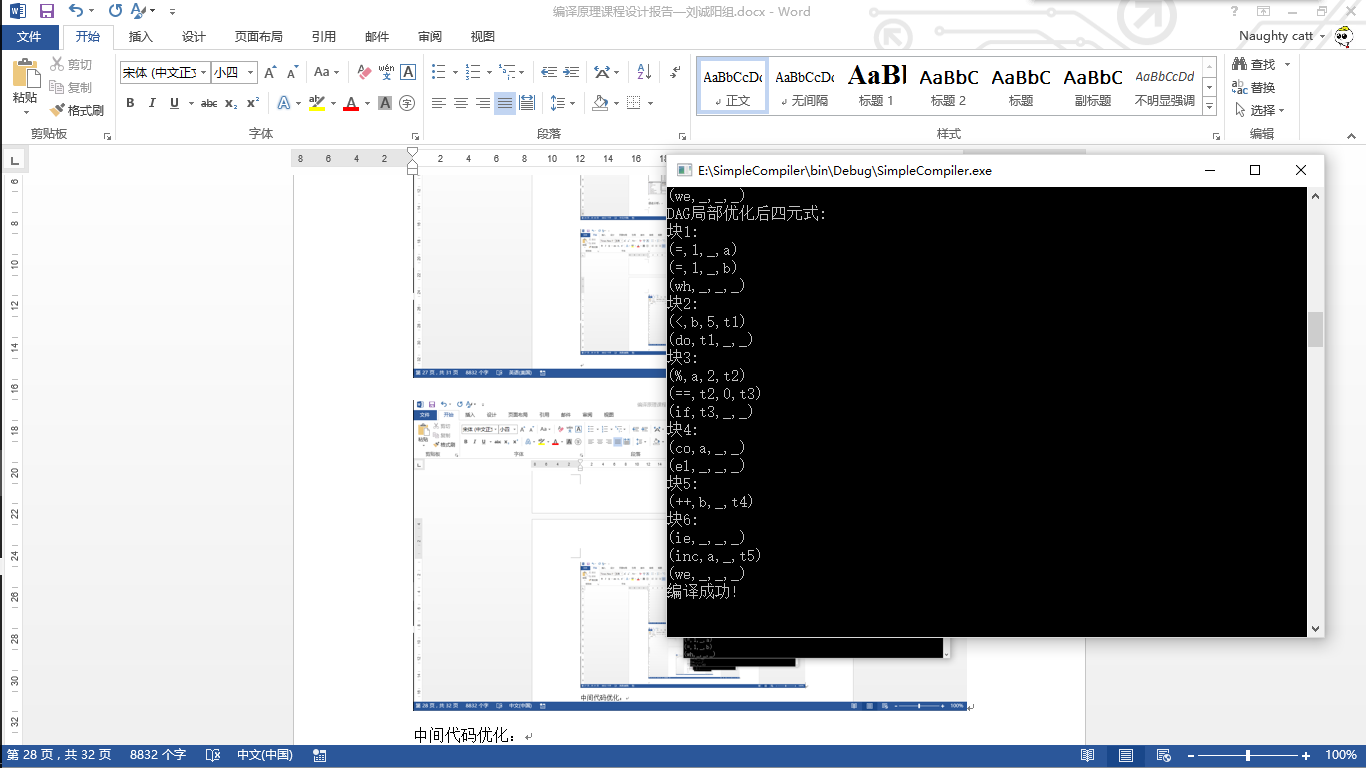


图4.5 优化后四元式结果

**目标代码生成：**

目标代码模块会产生相应的汇编代码，并最终生成可编译的.ASM汇编文件，生成的最终文件结果如下图所示。

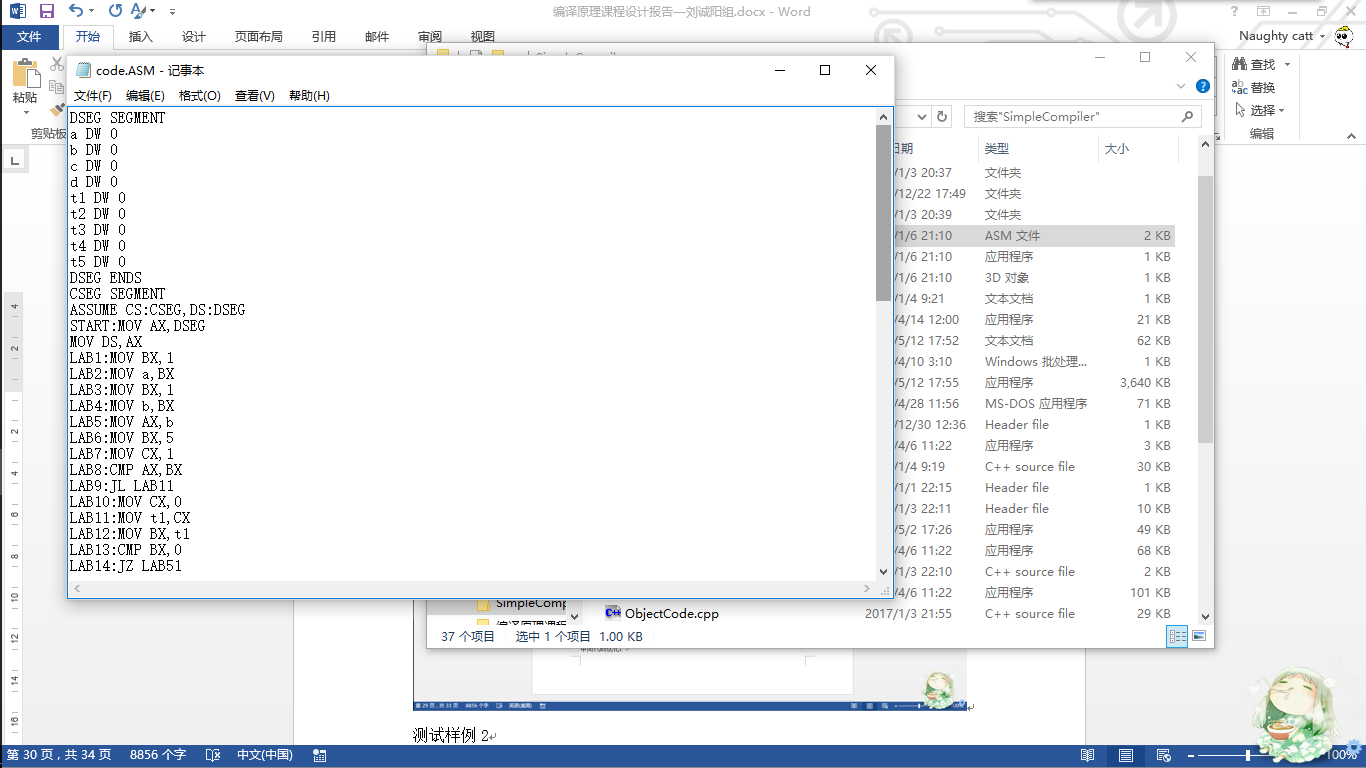
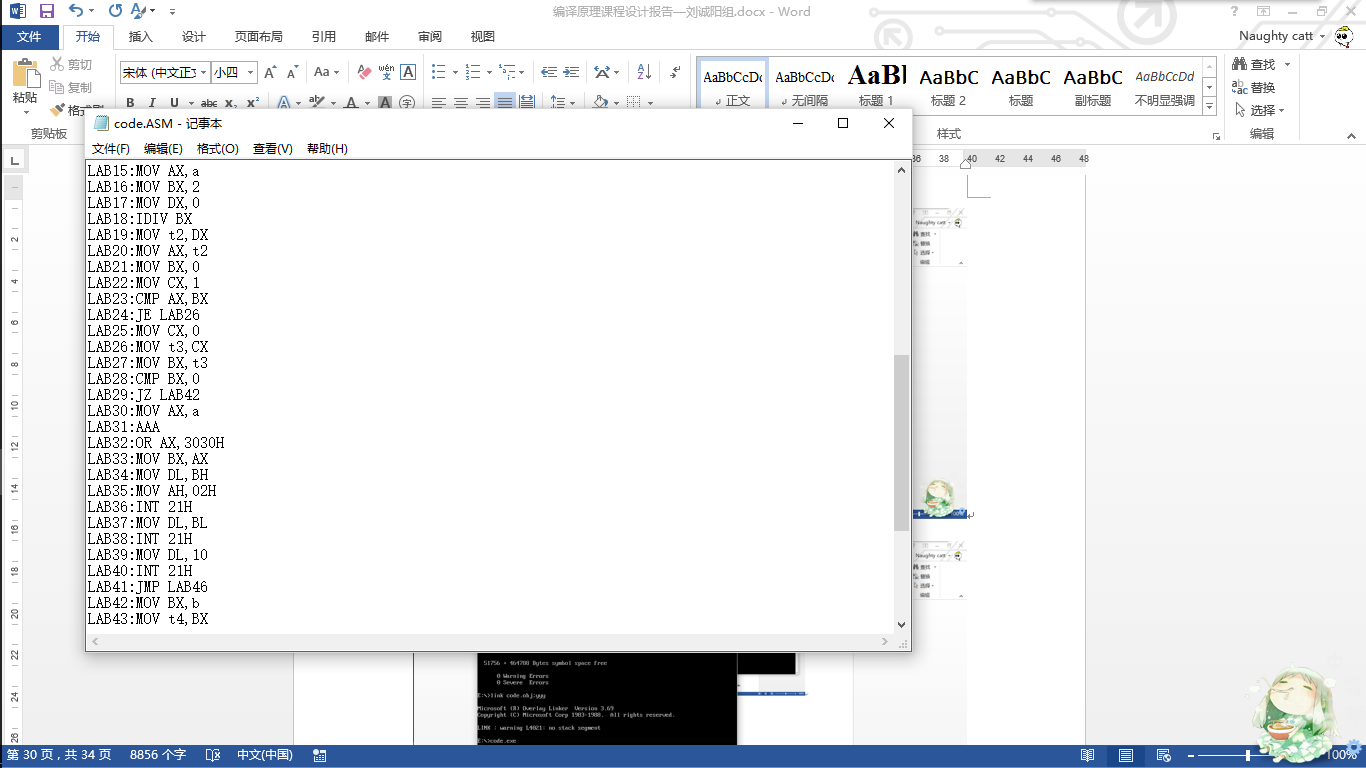
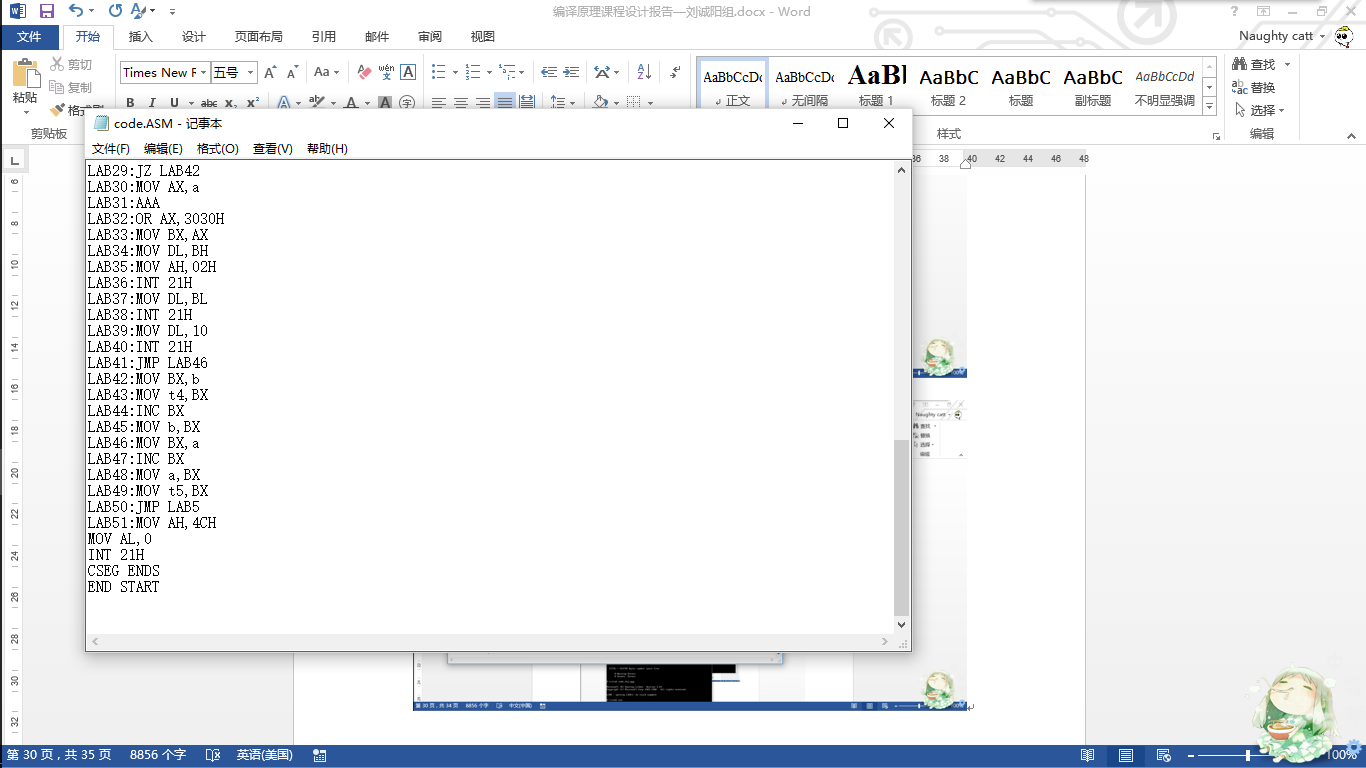
  

图4.6 目标代码生成结果

该模块已经配置好了相应环境，会自动调用汇编的编译文件生成可执行文件，并自动执行该可执行文件，从而向屏幕打印输出源程序的结果，实现完整的编译程序。运行结果如下图所示。

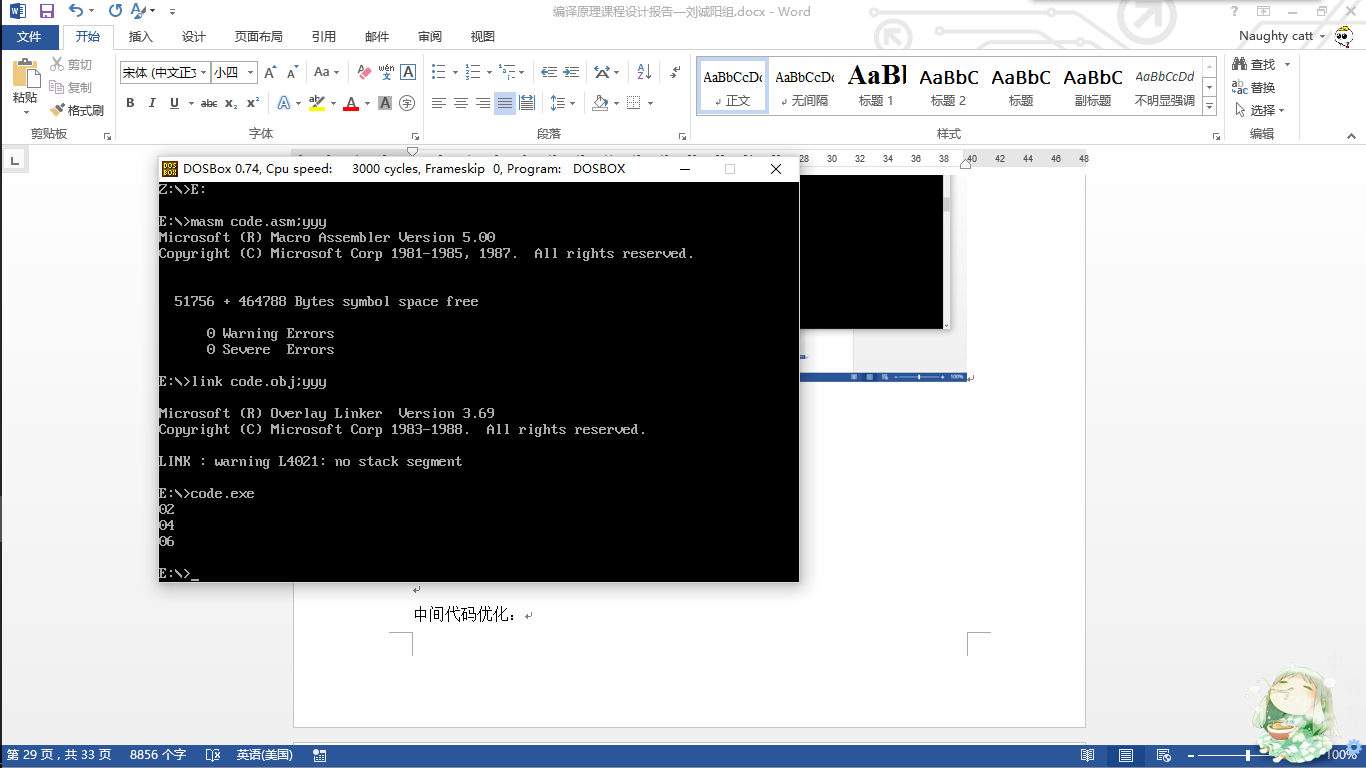


图4.7 源程序结果显示

**测试样例2:**

void main(){

int a,b,c;

a=1;

b=1;

c=15;

while(a<=5){

cout a++;

}

cout c;

while(b<=5){

cout ++b;

}

}

**词法分析：**

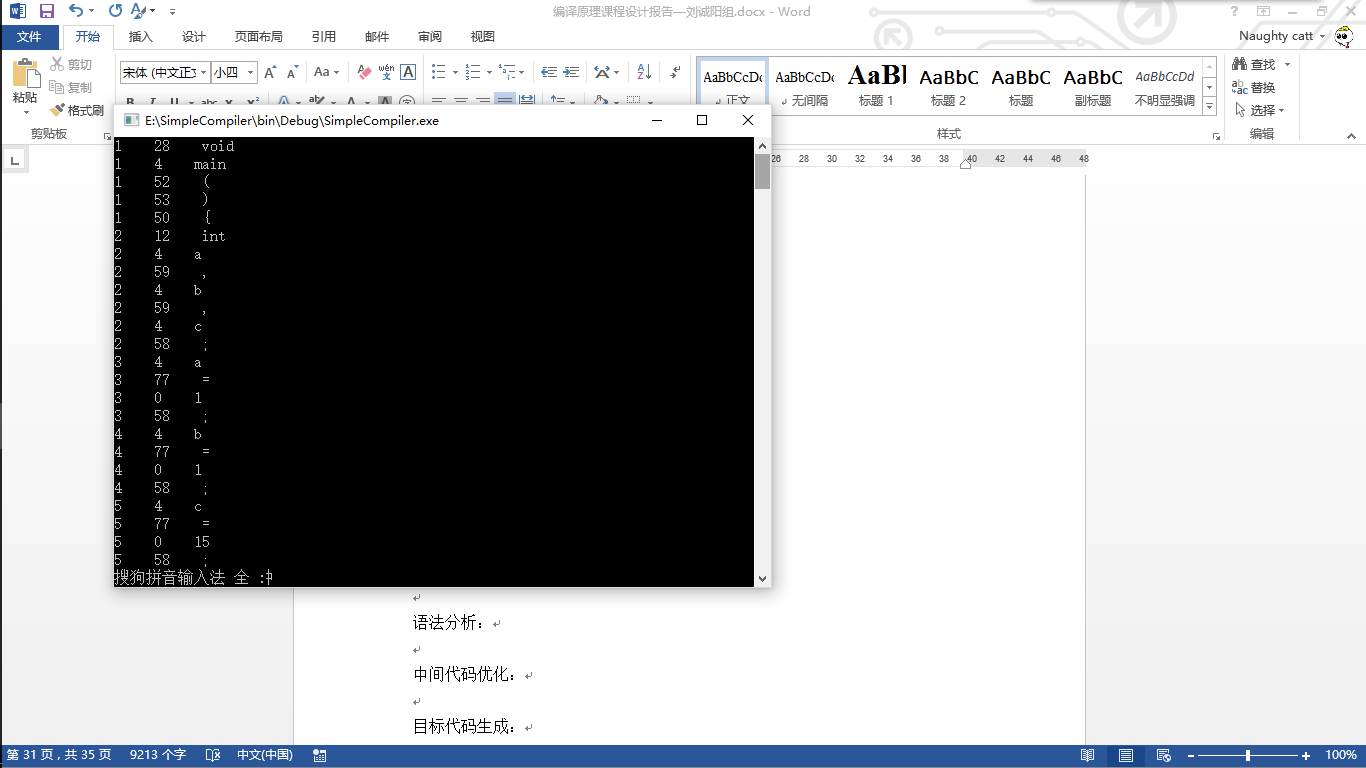
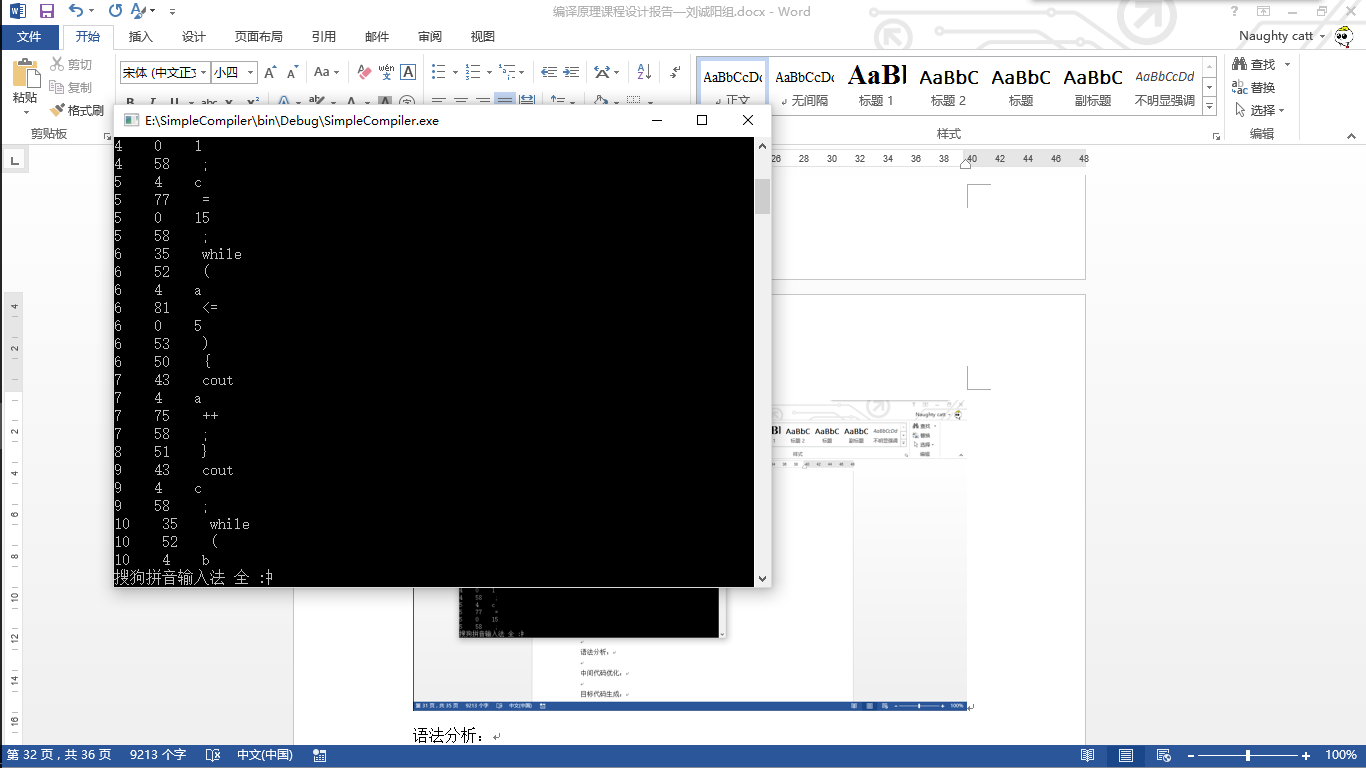
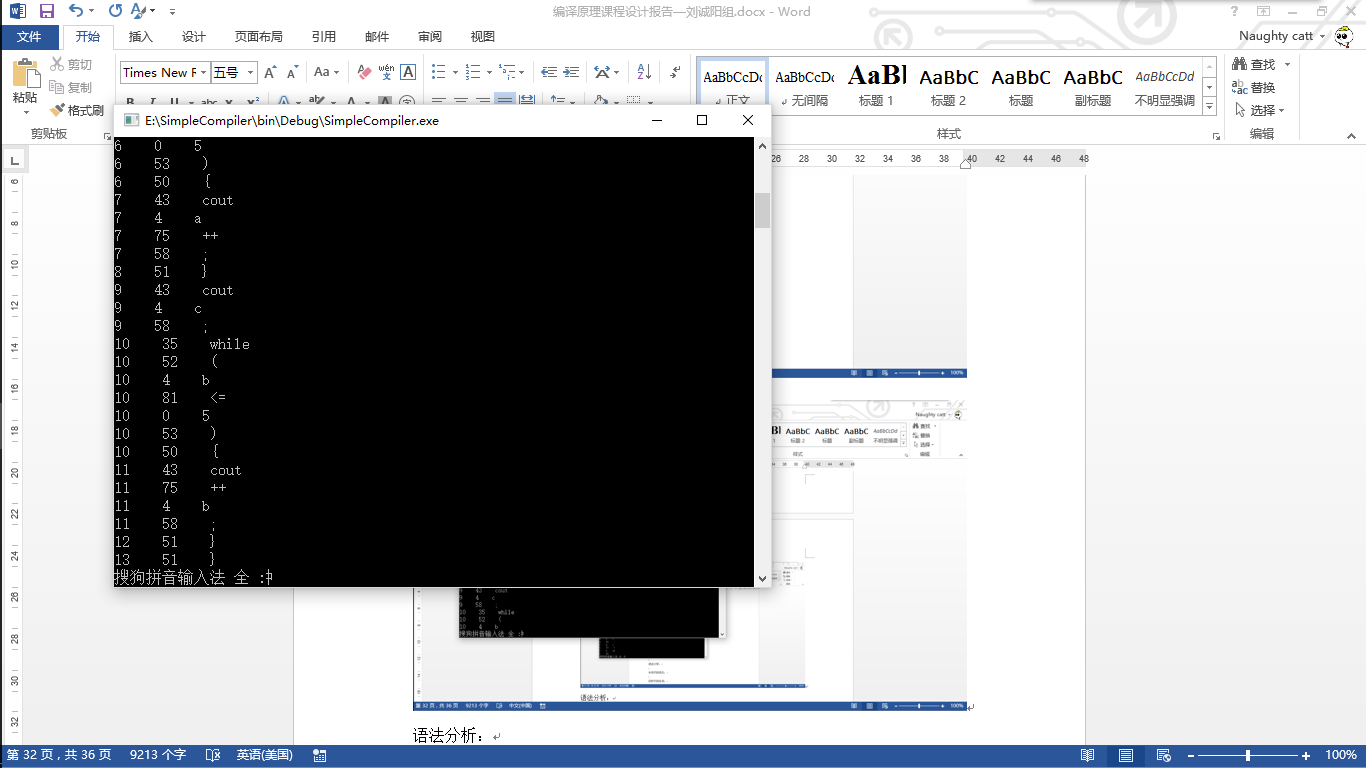
  

图4.8 词法分析结果

**语法分析：**

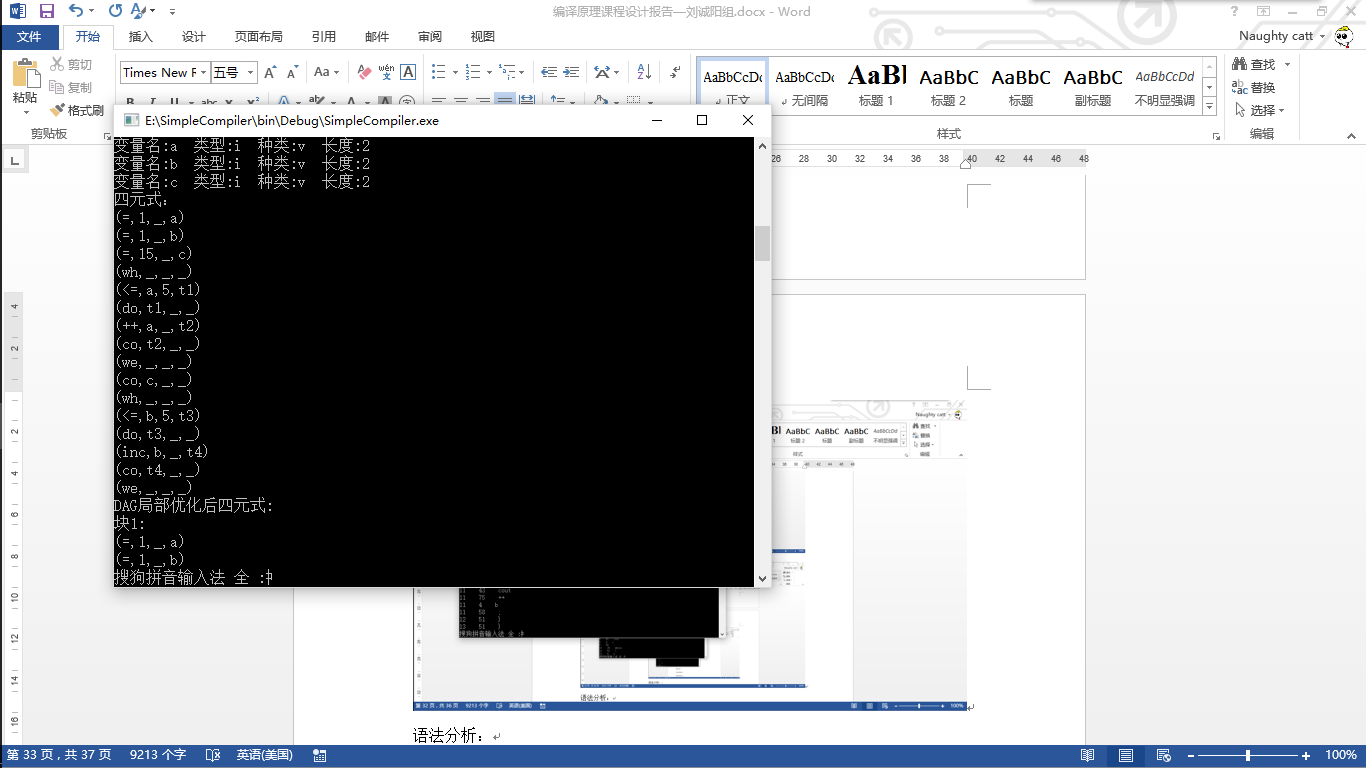


图4.9 四元式生成结果

**中间代码优化：**

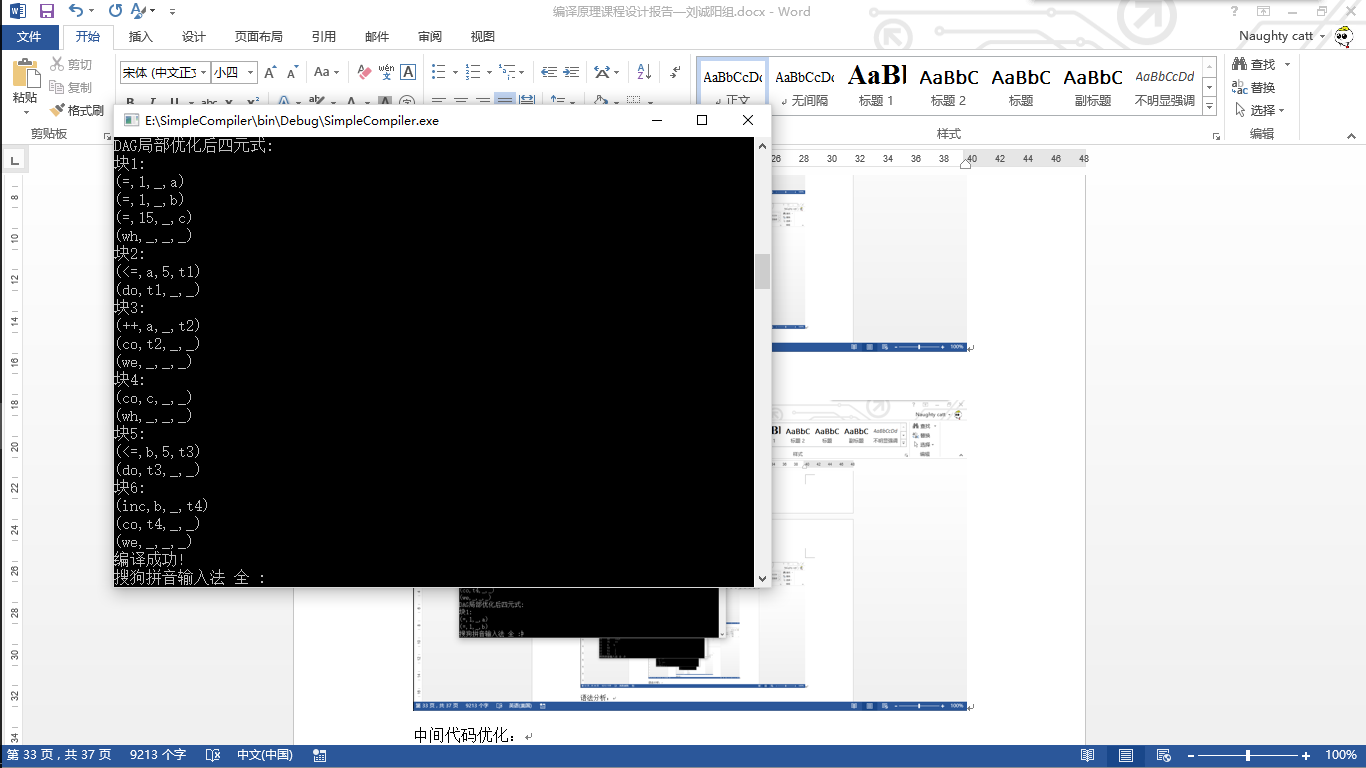


图4.10 中间代码优化结果

**目标代码生成：**

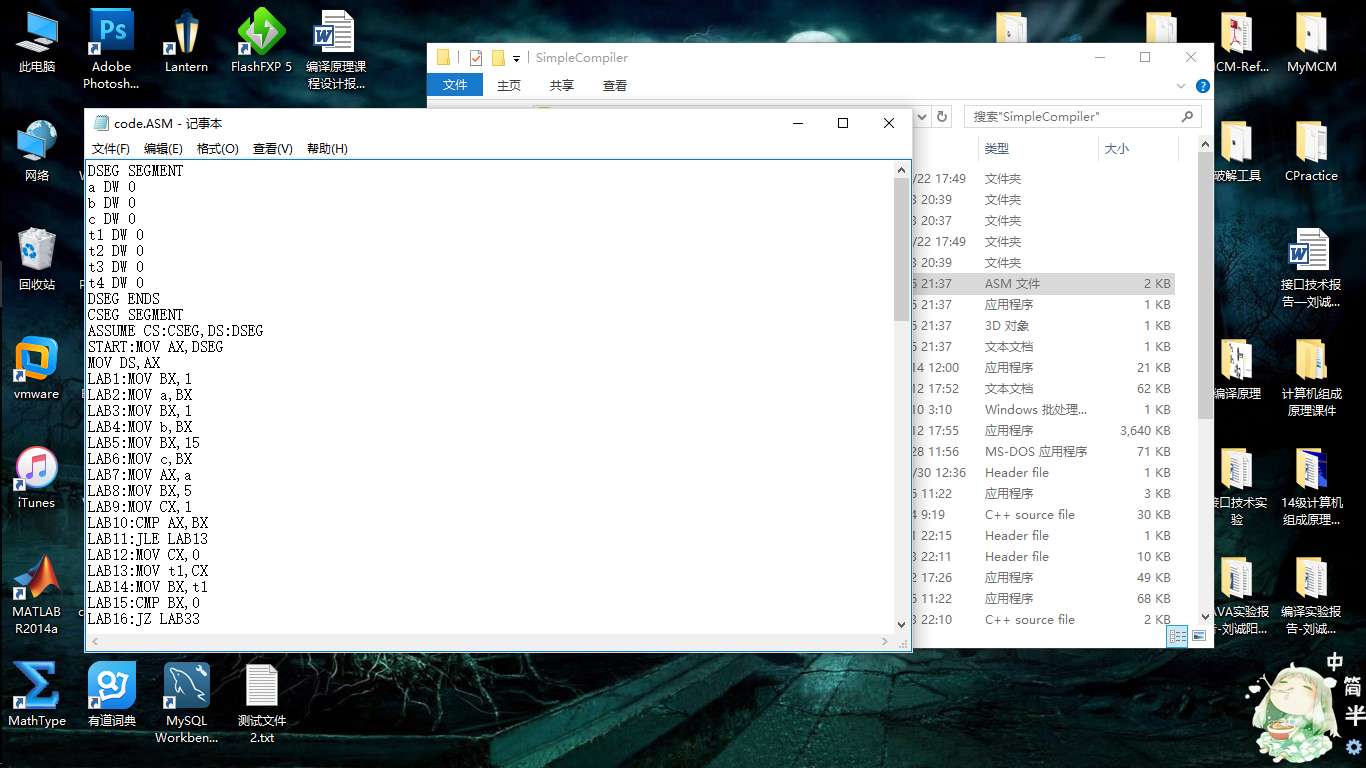
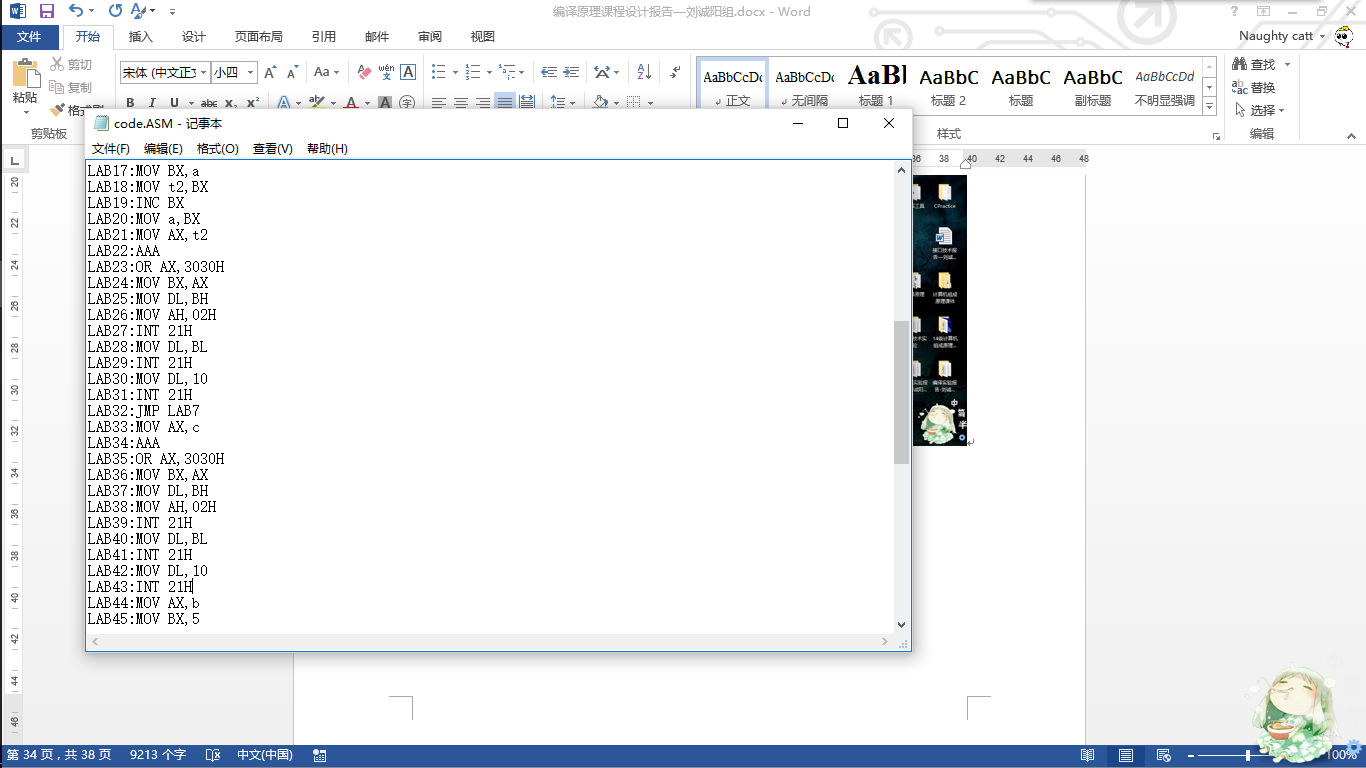
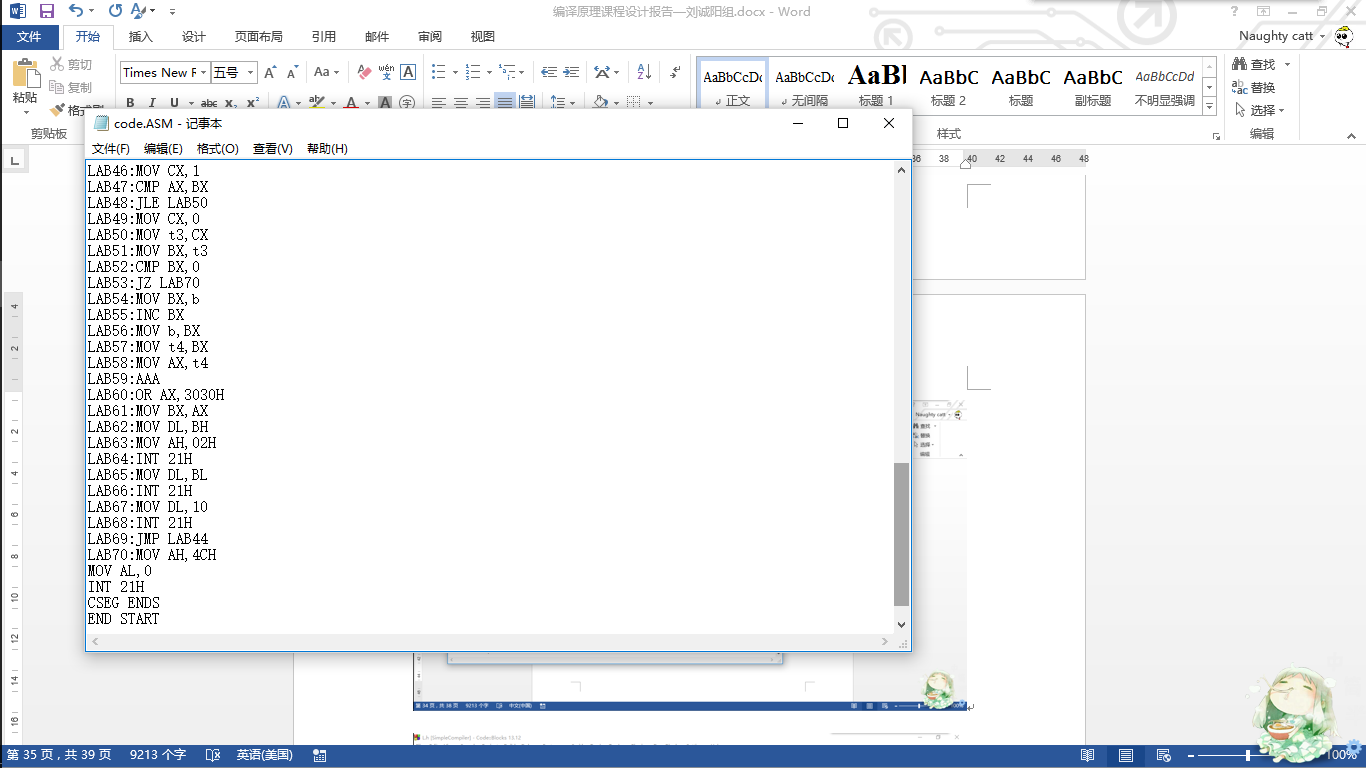
  

图4.11 汇编代码生成结果

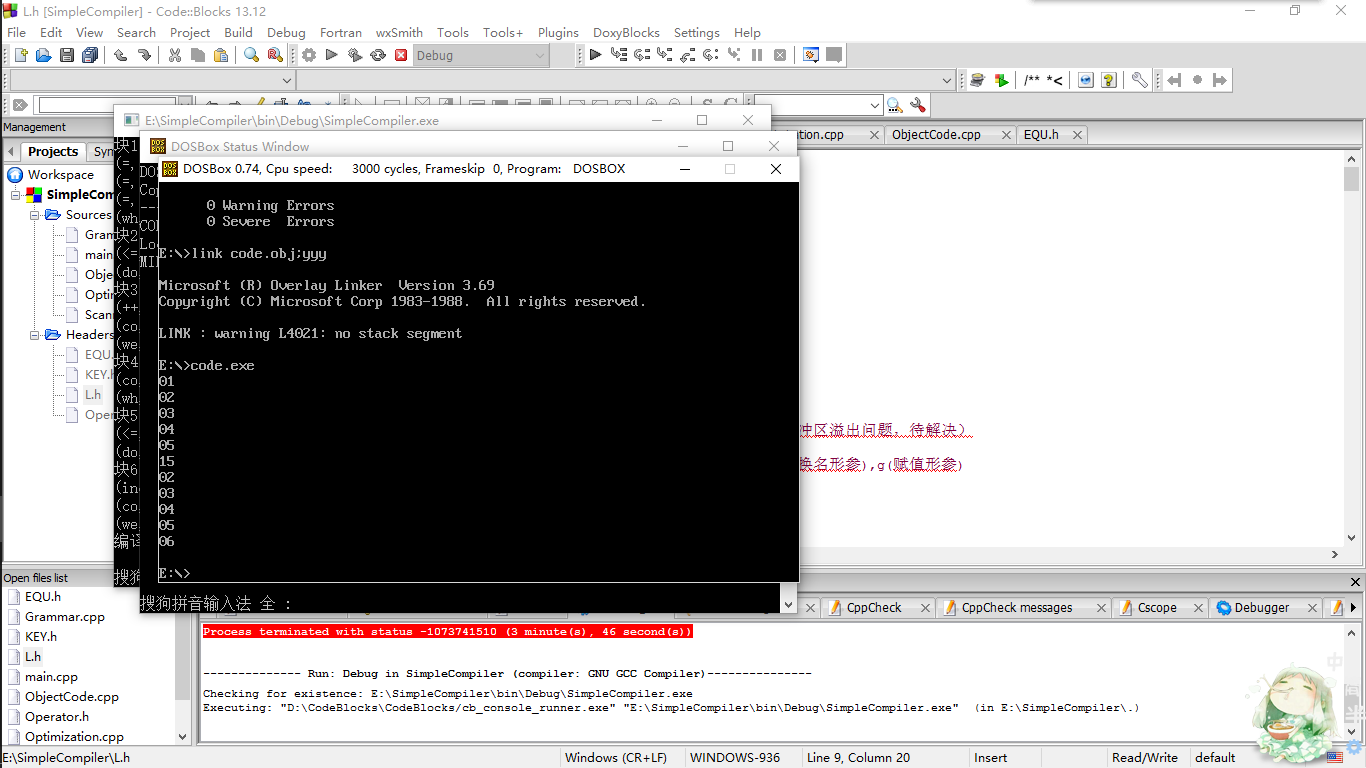


图4.12 程序运行结果

**程序容错性测试：**

**词法错误检测：**

测试样例：

int main(){

real a,b;

char c;

char d[10];

a=1.5;

b=1.;

c='a;

d="hello;

}

该测试用样具有明显的词法错误，如实数小数点后直接跟了分号结束，字符数据缺少右半边单引号，字符串数据缺少右半边双引号。

当输入错误单词形式或者程序无法识别的字符时，词法分析器会检测到该错误单词，并提示错误所在行数，测试结果如下图所示。

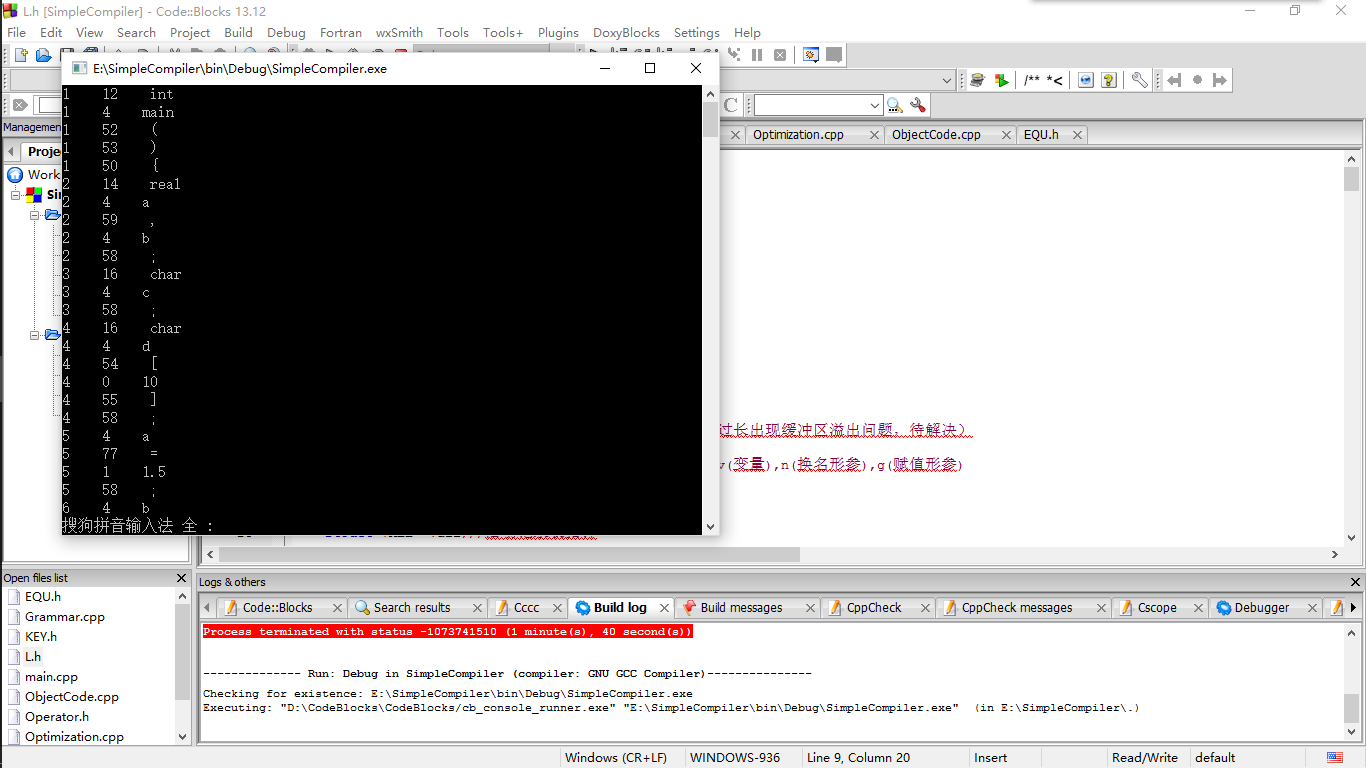
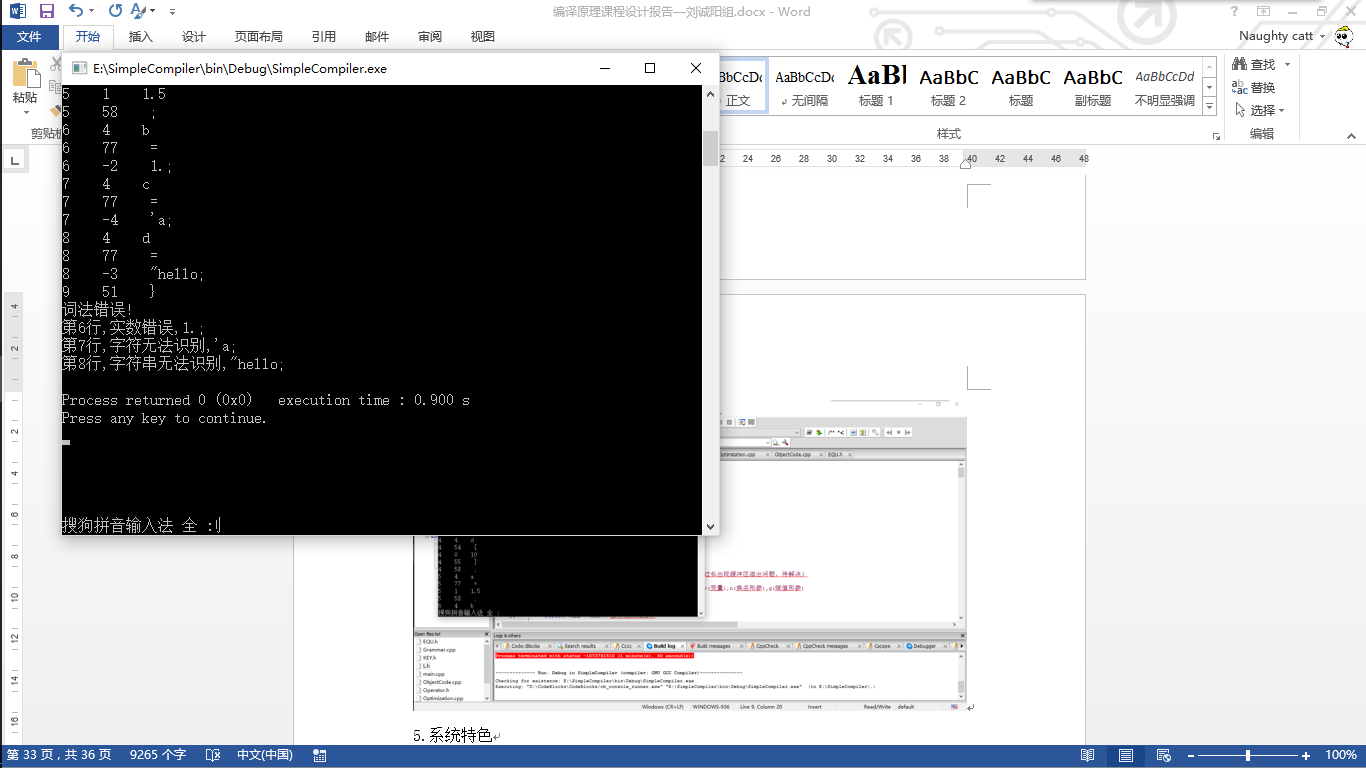
 

图4.13 词法错误检测

**语法和语义错误检测：**

测试样例：

int main(){

real a;

int c;;

a=1.5;

b=1.23;

c=3\*d;

}

该测试样例的变量定义存在错误，可以看到变量b和变量d都是未定义的标识符，并且测试样例存在语法错误，第三行多了一个分号。因此在语法模块中会对其进行检测，将未被定义的错误记录在错误表中。测试结果如下所示。

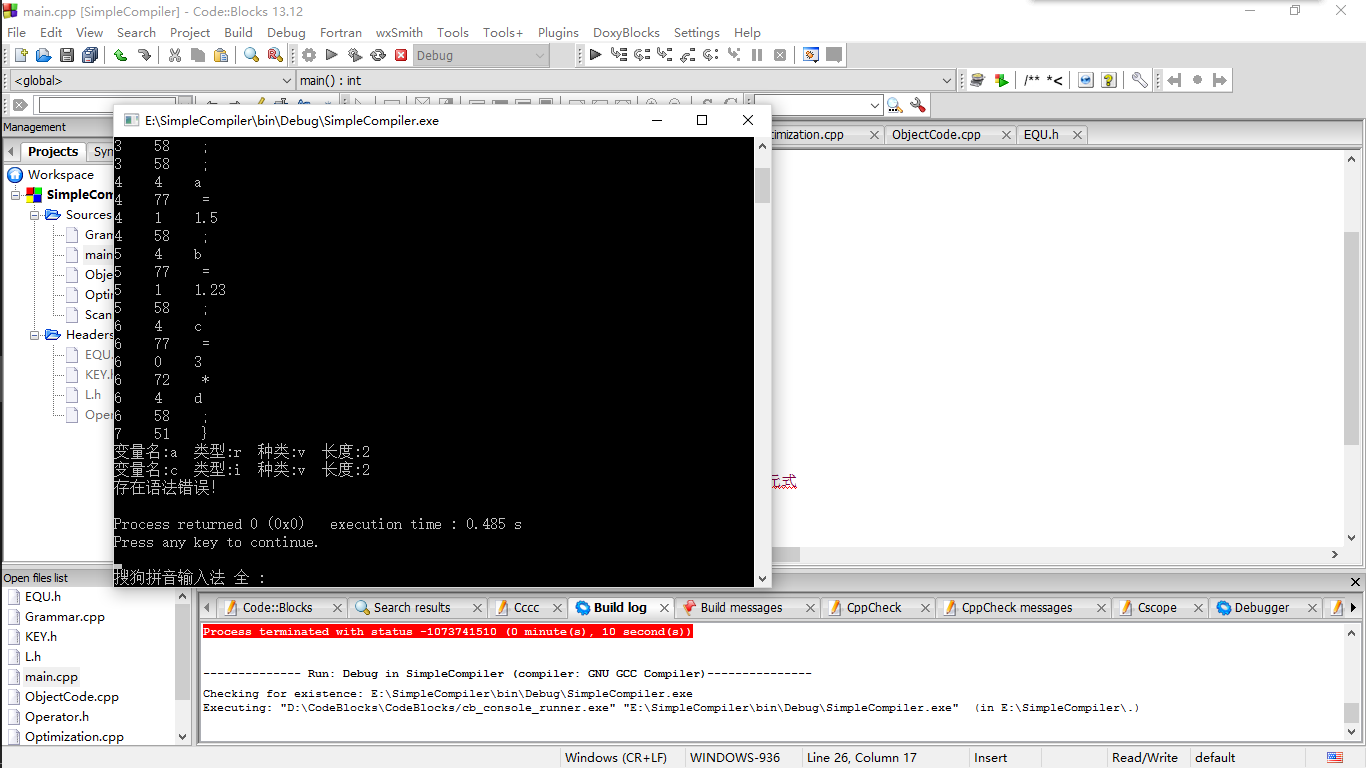


图4.14 语法错误和变量定义错误检测

### 5.系统特色

(1)设计实现了一个完整的编译程序，包括词法分析、语法分析、语义分析和中间代码生成、中间代码优化以及目标代码的生成。

(2)通过程序编译可以得到可执行文件，并设计了一个简单的cout输出功能，可以将程序的运行结果显示到屏幕上，从而验证编译程序的正确性。

(3)语法上设计了前置++（--）和后置++（--）的类C语言功能，可以使用i++或者++i这种语法，并且运算结果与C语言的运算结果一致。

(4)在语法上支持标准C语言的全部算术运算、逻辑运算、关系运算以及位运算，并且支持if-else条件语句和while循环语句。

(5)具有编译错误提示的功能，可以提示词法错误、语法错误、标识符未定义错误、标识符重复定义错误等，并且会提示错误所在行数，方便用户修改源程序，实现了良好的用户体验。

### 6.结论

从总体上来说，我们设计实现了一个较为完整的拥有前端和后端、能够生成可执行文件并且能够程序运行结果的简单编译器，该编译器的最终设计目的得到了实现。

我们的编译器能够识别标准C语言所支持的绝大部分词法符号，并且具有词法错误记录功能，能够提示词法错误的位置。语法部分，我们的编译器支持了变量的定义、数组的定义、函数的声明、各类表达式语句、if-else条件语句和while循环语句以及输出结果的cout语句。在语义动作上可以生成相应语句动作的四元式，从而实现了编译前端。同样，在语法方面的错误也会被记录到错误表内，并最终生成错误记录，提示给用户。在编译器的后端方面，我们的编译器可以对生成的四元式序列进行DAG算法优化，从而简化中间代码，最后我们的编译器可以生成能够执行的汇编语言文件，并将其运行，实现源程序结果的输出显示，实现真正意义上的完整编译运行。

由于时间紧任务重，我们编译器总体设计上的功能得到了基本的实现，美中不足的就是没有来得及实现更多一些的功能，例如结构体和函数调用等功能。尽管我们的语法结构可以支持这些语法，但是语义动作的相关实现并没有来得及做，这算是一个需要改进和努力的地方。但总的来说，我们的编译器设计任务还是取得了较好实现结果。

### 7.参考文献

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.

4、金成植著.《*编译程序构造原理和实现技术*》. 北京：高等教育出版社. 2002.

### 8.收获、体会和建议