

# 中国矿业大学计算机学院

## 系统软件开发实践报告

课程名称 系统软件开发实践

实验名称 实验九 综合实验

学生姓名 胡钧耀

学 号 06192081

专业班级 计算机科学与技术 2019-4 班

任课教师 张博

## 成绩考核

编号	课程教学目标	占比	得分
1	<b>目标 1：</b> 针对编译器中词法分析器软件要求，能够分析系统需求，并采用 FLEX 脚本语言描述单词结构。	15%	
2	<b>目标 2：</b> 针对编译器中语法分析器软件要求，能够分析系统需求，并采用 Bison 脚本语言描述语法结构。	15%	
3	<b>目标 3：</b> 针对计算器需求描述，采用 Flex/Bison 设计实现高级解释器，进行系统设计，形成结构化设计方案。	30%	
4	<b>目标 4：</b> 针对编译器软件前端与后端的需求描述，采用软件工程进行系统分析、设计和实现，形成工程方案。	30%	
5	<b>目标 5：</b> 培养独立解决问题的能力,理解并遵守计算机职业道德和规范，具有良好的法律意识、社会公德和社会责任感。	10%	
总成绩			
指导教师		评阅日期	

# 目 录

实验（九） 目标代码生成实验.....	1
9.1 实验要求与目标.....	1
9.2 实验内容.....	1
9.3 Task 1: 模拟器学习 .....	1
9.3.1 X86 体系结构.....	1
9.3.2 模拟器的选择和安装.....	1
9.3.3 模拟器基本功能学习.....	2
9.4 Task 2: 汇编程序生成 .....	2
9.4.1 生成抽象语法树.....	2
9.4.2 生成四元式.....	3
9.4.3 实验结果.....	6
9.5 Task 3: 生成汇编程序.....	6
9.5.1 程序思路.....	6
9.5.2 代码解析.....	7
9.5.3 实验结果.....	10
9.6 出错处理.....	11
9.6.1 词法分析错误.....	11
9.6.2 语法分析错误.....	12
9.7 可视化软件开发.....	13
9.7.1 前端设计.....	15
9.7.2 后端设计.....	13
9.7.3 结果展示.....	15
9.8 实验总结.....	17
9.8.1 问题.....	17
9.8.2 评价.....	17
9.8.3 收获.....	17

## 实验（九） 目标代码生成实验

### 9.1 实验要求与目标

本实验是在词法分析、语法分析、语义分析和中间代码生成的基础上，将 C 子集源代码翻译为 MIPS32 指令序列在 SPIM Simulator 或翻译为 x86 指令序列并在 x86 Simulator 上运行。

### 9.2 实验内容

1. 确定目标机的体系结构，选择一款模拟器进行安装学习。学习与具体体系结构相关的指令选择、寄存器选择以及存储器分配管理技术。
2. 编写目标代码的汇编生成程序可以将在此之前已经得到的中间代码映射为相应的汇编代码。
3. 实现错误处理程序，包括词法错误、语法错误、编译警告和运行异常。
4. 生成可执行文件，最终获得一个自己的 C 语言编译器，包含基本的前端和后端功能。
5. 实现链接器，生成可运行程序。

### 9.3 Task 1: 模拟器学习

#### 9.3.1 X86 体系结构

X86 架构（The X86 Architecture）是微处理器执行的计算机语言指令集，指 intel 通用计算机系列的标准编号缩写，也标识一套通用的计算机指令集合。8086 CPU 中寄存器总共为 14 个，且均为 16 位，即 AX, BX, CX, DX, SP, BP, SI, DI, IP, FLAG, CS, DS, SS, ES 共 14 个。而这 14 个寄存器按照一定方式又分为了通用寄存器，控制寄存器和段寄存器。

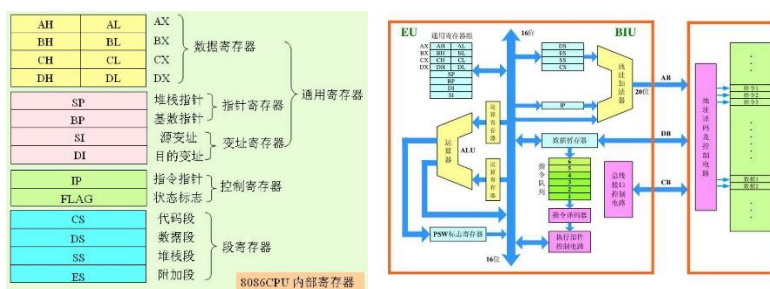


图 1 8086 寄存器组成、分类及其联系

#### 9.3.2 模拟器的选择和安装

首先确定选择哪一种类型的模拟器对生成的目标代码进行检查和调试。由于自己已经先修过《计算机组成原理》《微机原理与接口》《硬件课程设计》等课程，对 X86 的指令系统较为熟悉。综上，选择 X86 体系结构下的 Emu8086 进行尝试。安装 *emu8086v408r.exe* 的过程如下。

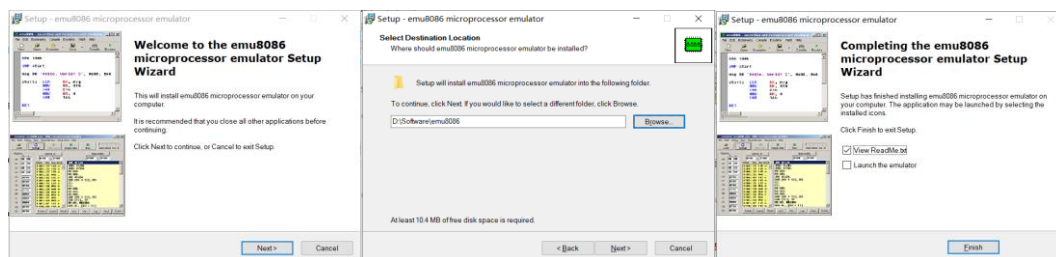


图 2 emu8086 的安装

### 9.3.3 模拟器基本功能学习

Emu8086 是一个结合了原始编辑器、组译器、反组译器、具有除错功能的软件模拟工具（虚拟 PC），可以在 Emu8086 中进行汇编语言的编译和运行。

打开 hello-world 示例项目 *1\_sample.asm*，观察 Emu8086 的整体界面，如下图所示，左上侧工具栏中的工具分别有文件管理（新建、打开、样例文件、保存）、编译、运行、内置工具（计算器、进制转换器）等功能。中间是汇编程序的代码编辑界面。当完成编译语言的编辑后，可以点击左上角的 **emulate** 按钮进行模拟，即对所给文件进行模拟执行，即提供一个 8086 的处理器环境，进行程序的调试、执行、结果的查看等。

文件进行仿真模拟时，将会出现下图所示的两个窗口。右侧窗口为样例文件代码，标黄高亮的一行是正在运行的指令，左侧窗口为寄存器的值等信息。左侧窗口的工具栏功能包括载入、重新载入、撤回、单步步进、运行到程序结束，还可以设置每一步的延迟时间，便于提高调试效率。在 **registers** 部分，还可以观察寄存器状态值的变化，方便追踪程序的运行。

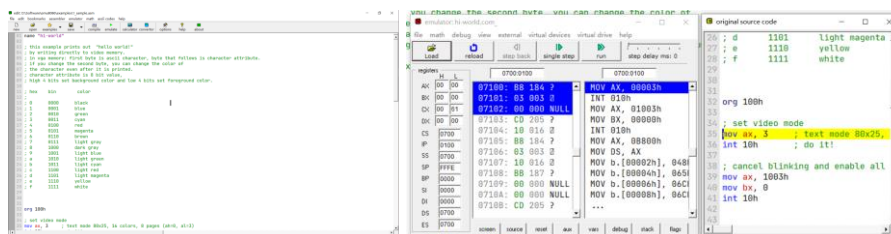
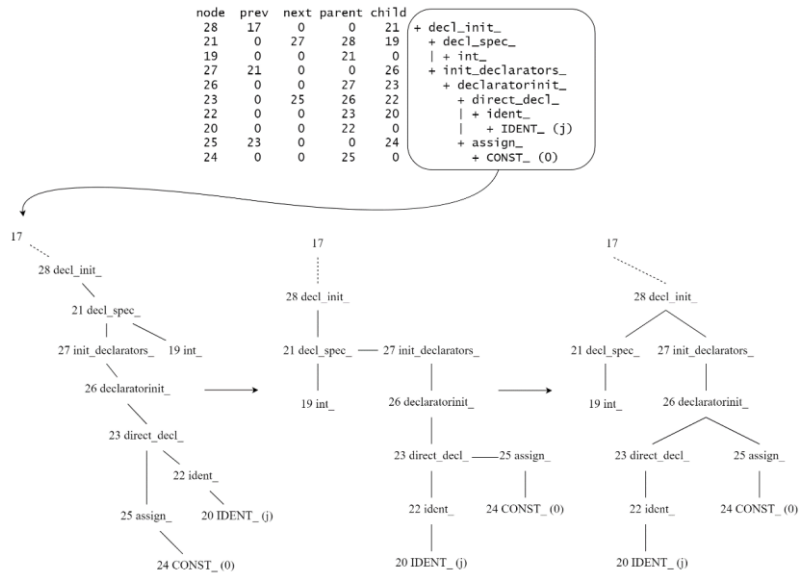


图 3 Emu8086 样例程序运行界面

## 9.4 Task 2: 汇编程序生成

### 9.4.1 生成抽象语法树

在 bison 实验二中，通过 flex 与 bison 的编译，生成最终的 *2-2.exe* 文件可以对源程序进行分析，生成对应的抽象语法树，并将结果保存在 *out.txt* 中，详细过程已经在前面实验中分析。

图 4 根据 *output.txt* 所给子女兄弟链表转换为语法分析树（部分）

#### 9.4.2 生成四元式

四元式作为连接编译器前后端的桥梁，有着十分重要的作用。在语法分析的过程中，得到了遍历之后的抽象语法树 AST，四元式可以在此基础上遍历树节点生成。

核心代码为 *tac.c* 文件的 `start_tac()` 函数，此处的处理思路是：从根节点开始遍历整棵树，判断节点类型，分别处理 `main` 节点、变量定义节点以及算术表达式节点。核心部分代码如下所示。

```
void start_tac(int n)
{
    //暂时只做了简单的 main 函数、变量定义和简单语句的节点处理
    while(n!=0)
    {
        char *start=node_name[node[n].id]; //取出结点 name
        if(strcmp(start,"funcdecl")==0) //main 方法体
        {
            MAIN_K(n);
            break;
        }
        if(strcmp(start,"declarations")==0) //变量定义
        {
            DEFINEPARA_K(n);
            n = node[n].next; //下一个兄弟结点
            continue;
        }
        if(strcmp(start,"statements")==0) //算术表达式
        {
            ASSIGN_K(n);
            n = node[n].next; //下一个兄弟结点
        }
    }
}
```

```

        continue;
    }
    n = node[n].child; //下一个子结点
}
}

```

四元表达式的存储采用的是链表的形式存储。在 *tac.c* 中有多个处理不同结点的函数，如 `MAIN_K()`、`DEFINEPARA_K()`、`ASSIGN_K()`，他们彼此之间结构类似，因此列举其中之一。

`ASSIGN_K()`函数处理赋值结点的主要流程是：取出赋值结点的子结点，子结点包含等号的左边和右边，并建立相应的四元式。其代码如下所示。

```

//处理赋值结点
void ASSIGN_K(int n)
{
    char t1[10],t2[10];
    n = node[n].child;//exp_
    while(n!=0)
    {
        strcpy(t1,newtemp());
        int i=node[n].child;//assignment_
        i=node[i].child;//equals_
        i=node[i].child;//IDENT_
        strcpy(t2,deal_expk(node[i].next));
        get_tac(1,t2,t1,adr);
        int sti = node[i].sti;
        char* q= symbol[sti].name;
        get_tac(1,t1,q,adr);
        n=node[n].next;
    }
}

```

在对节点的处理中，还涉及到四元式的生成函数 `get_tac()`函数，四元式的形式为(*op*, *a*, *b*, *c*)，表示 *a* 和 *b* 经过 *op* 操作，并将结果存入 *c* 中。四元式存储的数据结构为链表，*op* 的存储为数字，在打印时才经过 `op_string()`函数进行映射输出。`get_tac()`函数代码如下。

```

//建立四元式方法
void get_tac(int op,char a[],char b[],char c[])
{
    fourvarcode* t=NULL;
    t=(fourvarcode*)malloc(sizeof(fourvarcode));
    t->op=op;
}

```

```
if(a[0]=='\0')
{
    strcpy(t->addr1.kind,"emptys");
    strcpy(t->addr1.name,"\0");
}
else if((a[0]>='a'&&a[0]<='z')||(a[0]>='A'&&a[0]<='Z'))
{
    strcpy(t->addr2.kind,"strings");
    strcpy(t->addr1.name,a);
}
else
{
    strcpy(t->addr2.kind,"consts");
    strcpy(t->addr1.name,a);
}

if(b[0]=='\0')
{
    strcpy(t->addr2.kind,"emptys");
    strcpy(t->addr2.name,"\0");
}
else if((b[0]>='a'&&b[0]<='z')||(b[0]>='A'&&b[0]<='Z'))
{
    strcpy(t->addr2.kind,"strings");
    strcpy(t->addr2.name,b);
}
else
{
    strcpy(t->addr2.kind,"consts");
    strcpy(t->addr2.name,b);
}

if(c[0]=='\0')
{
    strcpy(t->addr3.kind,"emptys");
    strcpy(t->addr3.name,"\0");
}
else if((c[0]>='a'&&c[0]<='z')||(c[0]>='A'&&c[0]<='Z'))
{
    strcpy(t->addr3.kind,"strings");
    strcpy(t->addr3.name,c);
}
else
{

```



```

        strcpy(t->addr3.kind, "consts");
        strcpy(t->addr3.name, c);
    }

    t->next=NULL;
    tac_temp->next=t;
    tac_temp=t;
}

```

### 9.4.3 实验结果

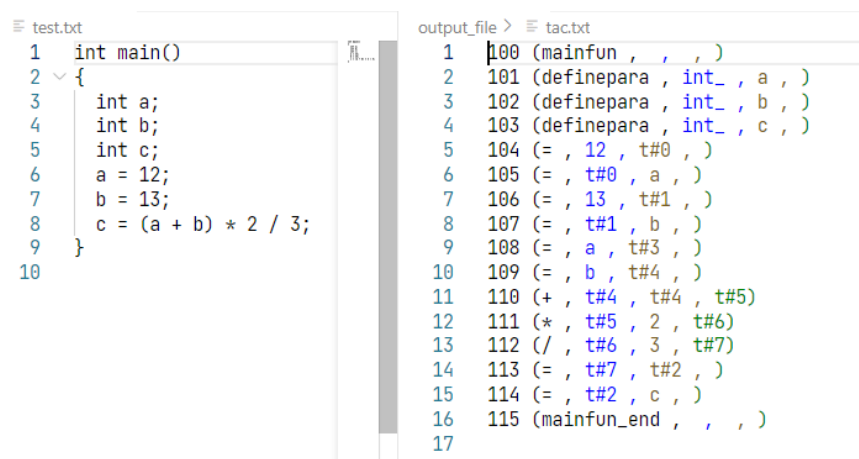
在命令行中输入如下代码。

```

gcc -o generator *.c
generator.exe

```

四元式结果如下。



```

test.txt
1 int main()
2 {
3     int a;
4     int b;
5     int c;
6     a = 12;
7     b = 13;
8     c = (a + b) * 2 / 3;
9 }
10

output_file > tac.txt
1 100 (mainfun , , , )
2 101 (definepara , int_ , a , )
3 102 (definepara , int_ , b , )
4 103 (definepara , int_ , c , )
5 104 (= , 12 , t#0 , )
6 105 (= , t#0 , a , )
7 106 (= , 13 , t#1 , )
8 107 (= , t#1 , b , )
9 108 (= , a , t#3 , )
10 109 (= , b , t#4 , )
11 110 (+ , t#4 , t#4 , t#5)
12 111 (* , t#5 , 2 , t#6)
13 112 (/ , t#6 , 3 , t#7)
14 113 (= , t#7 , t#2 , )
15 114 (= , t#2 , c , )
16 115 (mainfun_end , , , )
17

```

图 5 四元式的生成

## 9.5 Task 3: 生成汇编程序

### 9.5.1 程序思路

本步骤将四元式生成目标代码，需要关注两个点。

第一是汇编语言的指令系统，可以分为存取指令、运算型指令、转移型指令等，分别对应于不同的操作，这就要求在代码的思路，要针对不同的操作进行分类处理；

二是变量空间的分配和寄存器的选择。寄存器是 CPU 内部的元件，寄存器拥有非常高的读写速度，它们可用来暂存指令、数据和位址。在程序的编写过程中要考虑有哪些通用的寄存器，哪些可以作为累加器，哪些可以作为变址寄存器，可以使用某些寄存器专门执行想要的功能。此外，对于变量来说，临时变量与全局变量的生存期不同，其分配策略也应该是不同的。

临时变量只在一部分时间被使用，故可以用寄存器来存储，方便申请和释放；而对全局变量来说，则是不同的。

### 9.5.2 代码解析

代码的编写采用了面向过程的编程思想，根据四元式生成的汇编语句主要是代码段语句，程序的大体框架是预先设定好的。遍历四元式序列，根据四元式算符 `op` 选择相应方法生成汇编语句。

寄存器的使用并不是随意的，而是需要判断和申请。如下代码所示，寄存器包括 `AX`, `BX`, `CX`。在每次使用时，需进行判断标志位是否为 1，如果为 1，则不能进行。在某处使用完毕进行释放后，才可以重新被申请和使用，保证了汇编程序的合法生成。

```
void registerfree(char b[10])
{
    if(strcmp(b,"AX")==0||strcmp(b,"EAX")==0)
    {
        AX_FLAG=0;
    }
    if(strcmp(b,"BX")==0||strcmp(b,"EBX")==0)
    {
        BX_FLAG=0;
    }
    if(strcmp(b,"CX")==0||strcmp(b,"ECX")==0)
    {
        CX_FLAG=0;
    }
}
```

变量分为临时变量和全局变量。临时变量采用寄存器存储，而全局变量需要用到内存来存储，涉及到的代码包括下述两处。涉及到全局定义的变量时首先调用 `define_para` 函数，进行初步的判断和处理，然后调用 `flocal_table` 函数进行内存的分配和符号表的存储。

```
void flocal_table(char type[],char name[],char num[],int n)
{
    symboltable *t;
    t = (symboltable *)malloc(sizeof(symboltable));
    t->next = NULL;
    strcpy(t->type, type);
    strcpy(t->name, name);
    t->location=n;
    lsptempl->next=t;
    lsptempl=t;
}

void define_para(fourvarcode* tac_head)
{
```

```

fourvarcode *t=tac_head->next;
int num=0;
if(t->op!=9&& t->op!=6)
{
    while(t->op!=9 && t->op!=6)
    {
        flocal_table(t->addr1.name,t->addr2.name,t-
>addr3.name,num+1);
        t=t->next;
        num++;
    }
}
registerfree("AX");
registerfree("BX");
registerfree("CX");
}

```

具体的运算，以 op 是乘的四元式为例阐述汇编生成过程为例。将乘数放入 CX 中，被乘数放入 AX 中，最终运算结果放入 AX 中。若四元式结果存放位置为局部变量，则将运算结果移至 BP 段，释放 AX、CX。

```

void build_muls(fourvarcode* t) ///乘，x86 中乘法的被乘数在 AX 里
结果在 AX 中
{
    char a3[10],a1[10],a2[10],axorbx[5];
    strcpy(a1,t->addr1.name);
    strcpy(a2,t->addr2.name);
    strcpy(a3,t->addr3.name);
    if(a1[0]>='0'&&a1[0]<='9'){
        printf("MOV BX,%s\n",a1);
        fprintf(fp,"MOV BX,%s\n",a1);
    }
    if(a2[0]>='0'&&a2[0]<='9'){
        printf("MOV BX,%s\n",a2);
        fprintf(fp,"MOV BX,%s\n",a2);
    }
    if(CX_FLAG==1)//用 cx 寄存器
    {
        printf("MUL BX,CX\n");
        fprintf(fp,"MUL BX,CX\n");
        strcpy(axorbx,"BX");
    }
    else
    {

```

```
printf("MUL BX\n");
fprintf(fp, "MUL BX\n");
AX_FLAG=1;
strcpy(axorbx, "AX");
}
if(a3[1]!='#')//局部变量
{
    int d = 0;
    //d=searchstack(t->addr3.name);
    if(d>-1&& d<0)
    {
        d=-(d+2);
        printf("MOV [BP+%d],%s\n",d,axorbx);
        printf("MOV BP,SP\n");
        fprintf(fp, "MOV [BP+%d],%s\n",d,axorbx);
        fprintf(fp, "MOV BP,SP\n");
    }
    else
    {
        if(stackn>=2)
            ;
        else
            printf("MOV [BP+%d],%s\n",d,axorbx);
            fprintf(fp, "MOV [BP+%d],%s\n",d,axorbx);
    }

    if(strcmp(axorbx, "AX")==0)
    {
        registerfree("AX");
        registerfree("BX");
    }
    else
    {
        registerfree("BX");
        registerfree("CX");
    }
}
else//结果仍然留在 AX 中
{
    if(strcmp(axorbx, "AX")==0)
    {
        registerfree("BX");
    }
    else
```

```

    {
        registerfree("CX");
    }
}
}

```

### 9.5.3 实验结果

在命令行中输入如下代码。

```
gcc -o generator *.c
generator.exe
```

汇编结果如下。

```

test.txt
1 int main()
2 {
3     int a;
4     int b;
5     int c;
6     a = 12;
7     b = 13;
8     c = (a + b) * 2 / 3;
9 }
10

output_file > x86.asm
1 .MODEL SMALL
2 .STACK 100H
3 .DATA
4 .CODE
5 start:mov ax,@data
6 mov ds,ax
7 push bp
8 mov bp,sp
9 jmp alloc
10
11 main:
12 MOV AX,12
13 MOV [BP+2],AX
14 MOV AX,13
15 MOV [BP+4],AX
16 MOV BX,[BP+2]
17 ADD AX,BX
18 MOV BX,2
19 MUL BX
20 MOV BX,3
21 DIV BX
22 MOV [BP+6],AX
23 jmp over
24 alloc:
25 MOV AX,0
26 PUSH AX
27 MOV AX,13
28 PUSH AX
29 MOV AX,12
30 PUSH AX
31 jmp main
32 over:
33 mov ah,4ch
34 int 21h
35 end start
36

```

图 6 汇编语言生成结果

使用 *emu8086.exe* 运行该程序，根据手工计算 $(12+13)*2/3=16.67$ ，转换为十六进制可以认为就是 10，观察汇编程序运行得到的结果，AL=10，说明成功进行了计算，汇编的结果正确。

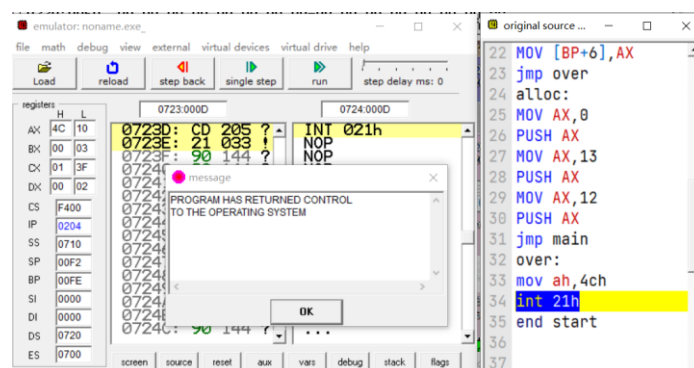


图 7 汇编程序运行结果

## 9.6 出错处理

### 9.6.1 词法分析错误

词法分析是计算机科学中将字符序列转换为单词（Token）序列的过程。在词法分析阶段产生的错误时“不能被识别的单词”的错误，比如未知的标识符、操作符、错误格式之类。故此阶段的错误处理，我们可以在 `lex` 程序中完成。如下图代码处理，表示当识别到未在前文中定义的字符时，做 `OTHER` 处理，此处可以完成对未识别字符的标识。

```
1-词法分析 > lex.l
64  "{"      {return(RELOP);}
65  "}"      {return(RELOP);}
66  ";"      {return(RELOP);}
67  "("      {return(RELOP);}
68  ")"      {return(RELOP);}
69  ".> >"  {yyval = OTHER; return OTHER;}
```

图 8 词法错误处理

然后定义结构体 `err`，存储错误的字符和其具体的位置（行数和列数）。在 `writeout` 函数中，若识别到错误，则将对错误的错误进行存储，待后续输出到指定的文件 `./output_file/scanner_error_output.txt` 中。

```
1-词法分析 > lex.yy.c > err > col
1747
1748 struct err {
1749     char buf[10];
1750     int line;
1751     int col;
1752 };
1753
1754 struct err errbuf[10];
1755 FILE* fp_in;
1756 FILE* fp_err;
1757 FILE* fp;
1758
1759 int pos=0;
1760 void writeout(int c){
1761     switch(c){
1762     case OTHER: {
1763         sprintf(errbuf[pos].buf, "%s", yytext);
1764         errbuf[pos].line = row+1;
1765         errbuf[pos].col = column+1;
1766         pos++;
1767         break;
1768     }
```

图 9 词法错误的输出

接着，定义 `__cifa` 函数完成词法分析以及错误处理。将其分别输出到不同的文件中，分析结果放在 `./output_file/scanner_output.txt`，出错错误处理放在 `./output_file/scanner_error_output.txt`，便于后续的调用和处理。

```

1-词法分析 > C lex.yy.c > quit()
1790 void __cifa(char* filename)
1791 {
1792     int c=0;
1793     fp = fopen("./output_file/scanner_output.txt","w+");
1794     fp_err = fopen("./output_file/scanner_error_output.txt","w+");
1795     fp_in = fopen(filename,"r");
1796     while (c = yylex()){
1797         writeout(c);
1798     }
1799     for(int i=0; i<pos; i++)
1800         fprintf(fp_err,"line(%d): column(%d): error : %s\n", errbuf[i].line, errbuf[i].col, errbuf[i].buf);
1801     fclose(fp);
1802     fclose(fp_in);
1803     fclose(fp_err);
1804     return ;
1805 }
1806

```

图 10 词法错误存储

如下图所示，词法分析程序完成了词法分析以及进行了错误处理。

The screenshot displays three files in a code editor:

- scanner\_output.txt**: A list of tokens and their positions (line, column, token). For example, line 1, column 0 is '[INCLUDE]', line 2, column 0 is '[RELOP]', etc.
- scanner\_error\_output.txt**: Error messages for lines 4, 7, and 8. For example, 'line(5), column(4): error : @', 'line(7), column(7): error : !', and 'line(7), column(8): error : !'.
- input.txt**: The source code being analyzed, which contains several syntax errors. For example, line 1: '#include <iostream>', line 2: 'using namespace std', line 3: 'int main()', line 4: '{', line 5: '@float a = 4.90867e-2;', line 6: 'int b = 0xE124;', line 7: 'int!! c = 0167;', line 8: '// comment1', line 9: 'cout<<"Hello !" <<endl;', line 10: '/\*', line 11: 'comment2', line 12: '123 456 int', line 13: '\*/', line 14: 'cout<<"Welcome to c++!" <<endl;', line 15: 'return 0;', line 16: '}', line 17: ''.

图 11 词法分析与错误处理结果

### 9.6.2 语法分析错误

语法分析的任务是在词法分析的基础上将单词序列组合成各类语法短语，如程序、语句、表达式等等。语法分析程序判断源程序在结构上是否正确。这个任务主要是 **bison** 完成的，故此处的错误处理机制是在 *parse.c* 上完成的。

除了对错误内容和原因的输出之外，在 **yyerror** 中对错误的位置进行输出，对于代码的调试来说也是非常必要的。具体的修改内容如下图所示。在 *parser.c* 中通过关键字 **extern** 声明外部变量行数 **row**，在错误信息报告发出时一并输出。

由于程序中可能出现多条错误，在错误程序报告的文件中会多次追加，因此打开文件的方式必须为 **a+**，以打开一个用于读取和追加文件。

```

2-语法分析 > C parser.c > init_ast(int)
201 ///////////////////////////////////////////////////
288 extern int column;
289 extern int row;
290
291 int yyerror(char *str)
292 {
293
294     fp = fopen("./output_file/parser_error_output.txt","a+");
295     fprintf(fp,"line(%d): Error : %s\n",row+1, str);
296     fclose(fp);
297     return 1;
298 }
299

```

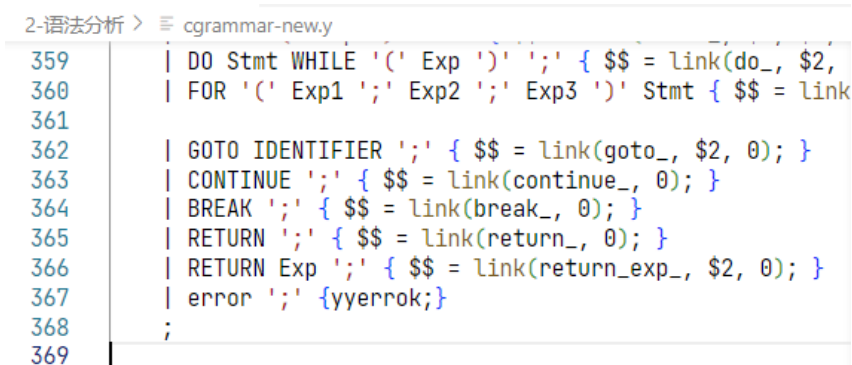
图 12 语法错误位置信息和原因输出

下面介绍错误恢复机制。

在 `bison` 中, `parser` 检测到语法错误的时候, 默认的做法是把所有东西清空, 然后返回让编译器强制停止。但是这种默认行为就只能检测出某一段代码的第一个语法错误, 并不能对代码编写有较好的提示和帮助。

查阅 `bison` 的手册后发现, 如果要让 `parser` 在检测到错误的时候不停止, 而是继续往下匹配, 也就是所谓的错误恢复机制, 那么就应该用 `yyerror`, 这样 `parser` 就会继续根据规则去匹配后面的代码, 而不至于立即退出。具体来说, 可以通过使用宏 `yyerror` 使错误消息立即恢复。如果在错误恢复规则的动作中使用它, 没有任何错误消息会被抑制。这个宏不需要任何参数, `yyerror` 是一个有效的 C 语句。

所以可以在语句的产生式处, 如下图所示, 添加规约 `error` 的产生式, 使语法分析可以持续进行下去。



```

359 | DO Stmt WHILE '(' Exp ')' ';' { $$ = link(do_, $2,
360 | FOR '(' Exp1 ';' Exp2 ';' Exp3 ')' Stmt { $$ = link
361
362 | GOTO IDENTIFIER ';' { $$ = link(goto_, $2, 0); }
363 | CONTINUE ';' { $$ = link(continue_, 0); }
364 | BREAK ';' { $$ = link(break_, 0); }
365 | RETURN ';' { $$ = link(return_, 0); }
366 | RETURN Exp ';' { $$ = link(return_exp_, $2, 0); }
367 | error ';' {yyerror;}
368 ;
369

```

图 13 添加 `yyerror` 方法进行错误恢复

## 9.7 可视化软件开发

可视化设计的任务分为两部分: 后端的链接库生成, 以及可视化桌面应用的设计。下面分别介绍。

### 9.7.1 后端设计

后端设计的任务分为三大模块: 词法分析、语法分析以及中间代码和目标代码的生成。动态链接库的生成在上个实验中已经有所涉及, 故比较顺利。在设计好相应的函数后, 分别通过 `gcc` 命令生成 64 位的 `lib*.dll` 即可。

```
gcc -o lib+[[*]].dll --share -fPIC -m64 *.c
```

三个阶段, 分别生成了 `libscanner.dll`、`libparser.dll`、`libgenerator.dll` 这三个 `dll` 文件, 放入 GUI 程序的 `dll` 文件夹中备用。

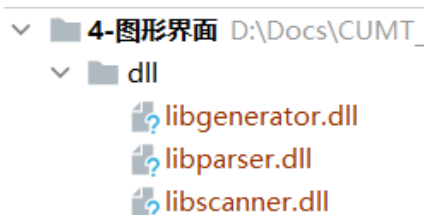


图 14 创建 DLL 文件



然后编写 `bottom_layer.py` 文件作为接口，实现与后端的连接，此处区分为四大功能：词法分析、语法分析、中间代码生成和目标代码生成。具体接口的实现如下，其中 `cifa_handler` 为词法分析接口，`yufa_handler` 为语法分析接口，`middlehandler` 为中间代码生成接口，`asm_handler` 为目标代码生成接口。

```
def cifa_handler(fname):
    del_file(OUTPUT_FILE)
    s = python_string_to_c_string(fname)
    dll = CDLL("./dll/libscanner.dll")
    dll.__cifa(s)
    ret = file_read('./output_file/scanner_output.txt')
    ret_error =
file_read('./output_file/scanner_error_output.txt')
    if ret_error == '':
        ret_error = '无词法错误'
    return ret, ret_error

def yufa_handler(fname):
    del_file(OUTPUT_FILE)
    print(fname)
    s = python_string_to_c_string(fname)
    dll = CDLL("./dll/libparser.dll")
    dll.__yufa(s)

    # 判断 yufa_err 文件是否存在
    if os.path.exists('./output_file/parser_error_output.txt'):
        ret_err =
file_read('./output_file/parser_error_output.txt')
        ret = '出现了语法错误'
    else:
        ret_err = '无语法错误'
        ret = file_read('./output_file/parser_output.txt')
    return ret, ret_err

def middle_handler(fname):
    del_file(OUTPUT_FILE)
    s = python_string_to_c_string(fname)
    dll = CDLL("./dll/libgenerator.dll")
    dll.__asm_x86(s)
    ret = file_read('./output_file/tac.txt')
    print(ret)
    return ret
```

```
def asm_handler(fname):  
    del_file(OUTPUT_FILE)  
    s = python_string_to_c_string(fname)  
    dll = CDLL("./dll/libgenerator.dll")  
    dll.__asm_x86(s)  
    ret = file_read('./output_file/x86.asm')  
    print(ret)  
    return ret
```

### 9.7.2 前端设计

前端可视化界面的实现是要基于 python3 和 pyqt5，选择了 pycharm 作为开发工具，QTdesigner 辅助设计。此款桌面应用可以跨平台使用并在多种尺度下自适应。

前端设计源于目标实现的功能，包括词法分析、语法分析、中间代码生成和目标代码生成。同时在信息展示上，除了源文件的显示，还增加了结果的显示、保存，错误信息的显示与保存，具体的设计页面如下所示。



图 15 编译结果可视化软件界面

### 9.7.3 结果展示

词法分析结果如下图所示。

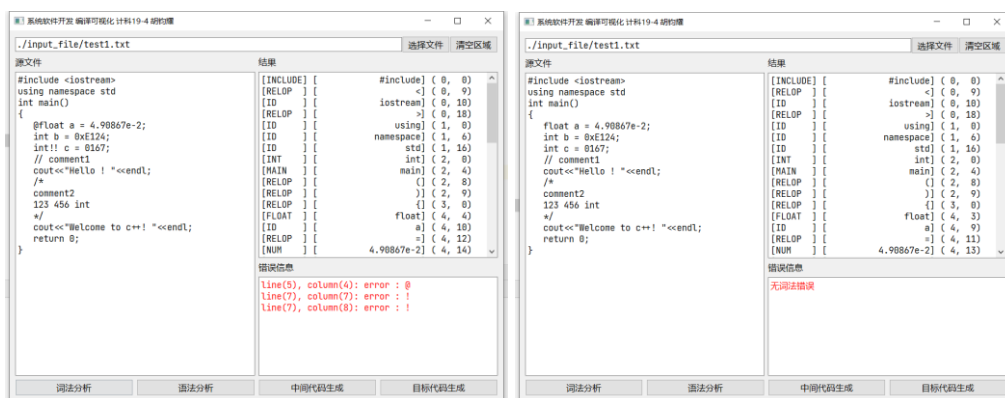


图 16 词法分析结果

语法分析结果如下图所示。

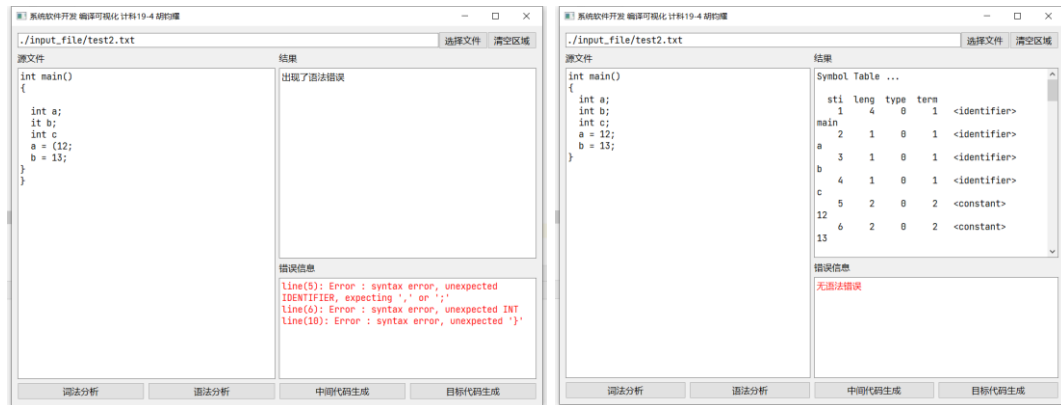


图 17 语法分析结果

中间代码生成结果如下图所示。



图 18 中间代码生成结果

汇编语言生成结果如下图所示。

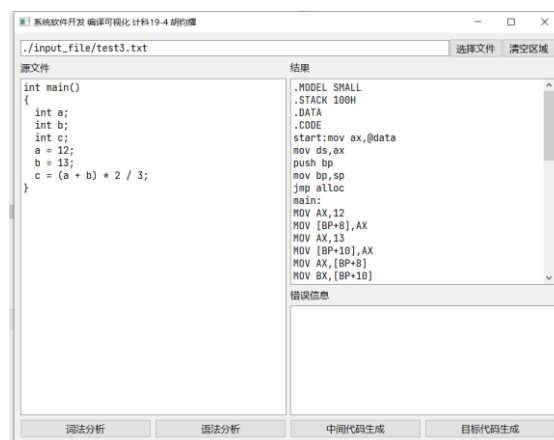


图 19 汇编语言生成结果

## 9.8 实验总结

### 9.8.1 问题

本次实验在修改源代码时，遇到了移入规约冲突，联想到之前实验中遇到的问题，查看实验报告文档就知道了应该如何去修改代码，使用 `bison` 编译时可以使用 `-v` 参数，生成 `.output` 文件以查看冲突产生原因，并通过定义优先级解决冲突。

### 9.8.2 评价

本次实验的难度也比较大，对数据结构和语法树的掌握要求较高。同时，在源代码的阅读过程中，我意识到 `flex` 与 `bison` 中还有非常多的特性是自己不熟悉的，如果能够掌握它们，对实现一个小型的编译器来说大有裨益收获。

再放长远一点看，编译器是世界上第一个电脑软件。在没有编译器之前，只能靠会写汇编的人去编译电脑软件。这需要很大的工作量，而且需要很高的专业技术能力。所以编译器的存在是非常有必要的。有了编译器才会有操作系统，才会有生态。所以如果国内不做好自己的编译器，就很难在这个生态中扎根，想要扎根，编译器是一个开始，需要给予足够的重视。但这是一条很艰辛的路程。

### 9.8.3 收获

这次实验让我了解了 C 编译器后端的基本原理以及相应的操作流程。编译器后端将前期所得抽象语法树转换成四元式，进而生成汇编代码，还了解了如何生成 `DLL` 文件并在其他文件中进行接口调用。虽然过程比较艰难，但还是有了不少的收获。我认为做软件不能浮躁，要静下心来做事。我们还需要长时间去积累经验，保持不断学习、不断思考的精神。