

中国矿业大学计算机学院

系统软件开发实践报告九

课程名称: 系统软件开发实践
实验名称: C 编译器后端设计
学生姓名: 石 涛
学生学号: 02160390
专业班级: 计算机科学与技术 2016-7 班
任课教师: 张博老师

2019 年 4 月 18 日

目 录

1 实验目的.....	1
2 实验内容.....	1
3 实验原理.....	1
4 实验步骤.....	1
4.1 搭建汇编开发环境.....	2
4.2 生成测试用例 test.c 的抽象语法树.....	7
4.3 生成四元式.....	8
4.4 生成汇编文件.....	15
4.5 生成目标文件和可执行文件.....	21
5 实验总结.....	23
5.1 编程中的困难.....	23
5.2 程序评价.....	23
5.3 实验收获.....	23

1 实验目的

将前四周得到的编译器前端结果（抽象语法树）转换为四元式，最后生成 X86 汇编（例如：386）。

2 实验内容

- (1) 在 DOSBox 环境下生成汇编程序；
- (2) 将生成的汇编程序在 masm 汇编生成可执行程序。

3 实验原理

- (1) c 语言源文件经过词法分析器得到 tokens；
- (2) tokens 送入语法分析器，利用抽象语法树生成四元式；
- (3) 四元式生成汇编文件；
- (4) 汇编文件在 DOSBox 中通过 masm 进行编译得到目标文件；
- (5) 目标文件通过链接再得到 exe 二进制可执行文件。

4 实验步骤

本次实验的思路如下所示：

- (1) 搭建汇编开发环境；
- (2) 生成测试用例 test.c 文件的抽象语法树；
- (3) 利用抽象语法树生成四元式；
- (4) 将四元式转换成汇编文件；
- (5) 生成目标文件和可执行文件。

具体的实验步骤如下所示。

4.1 搭建汇编开发环境

为了能够在 DOSBox 环境下生成汇编程序，首先我需要安装 DOSBox。具体步骤如下：访问网址 <https://www.dosbox.com/download.php?main=1>，点击 DOWNLOAD 进行 DOSBox 的下载，然后按照安装指示完成 DOSBox 的安装。具体如图 4.1 至图 4.3 所示。

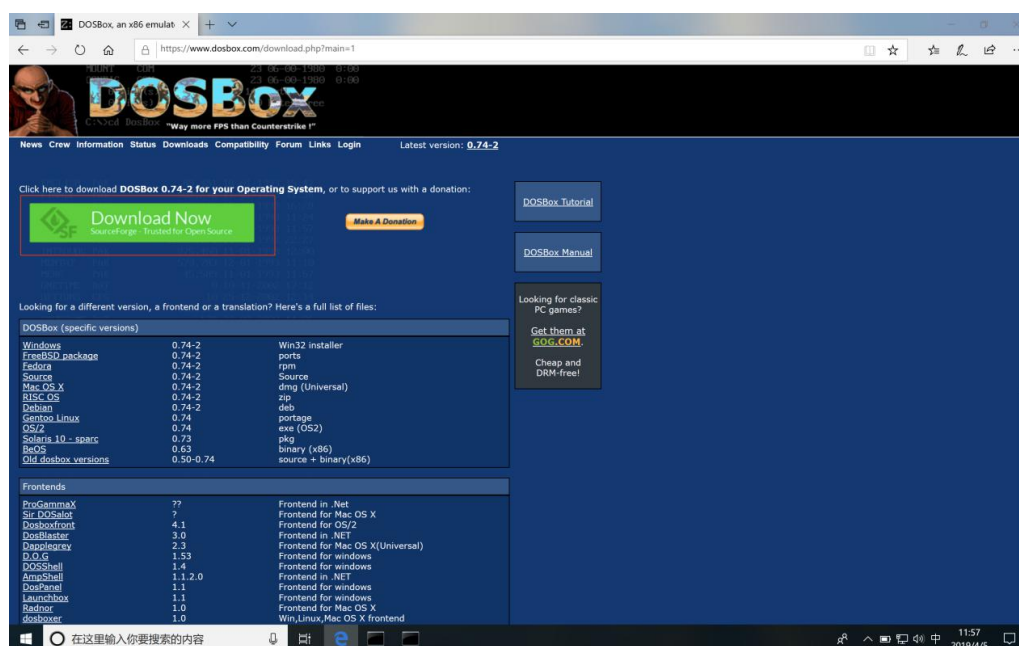


图 4.1 下载 DOSBox

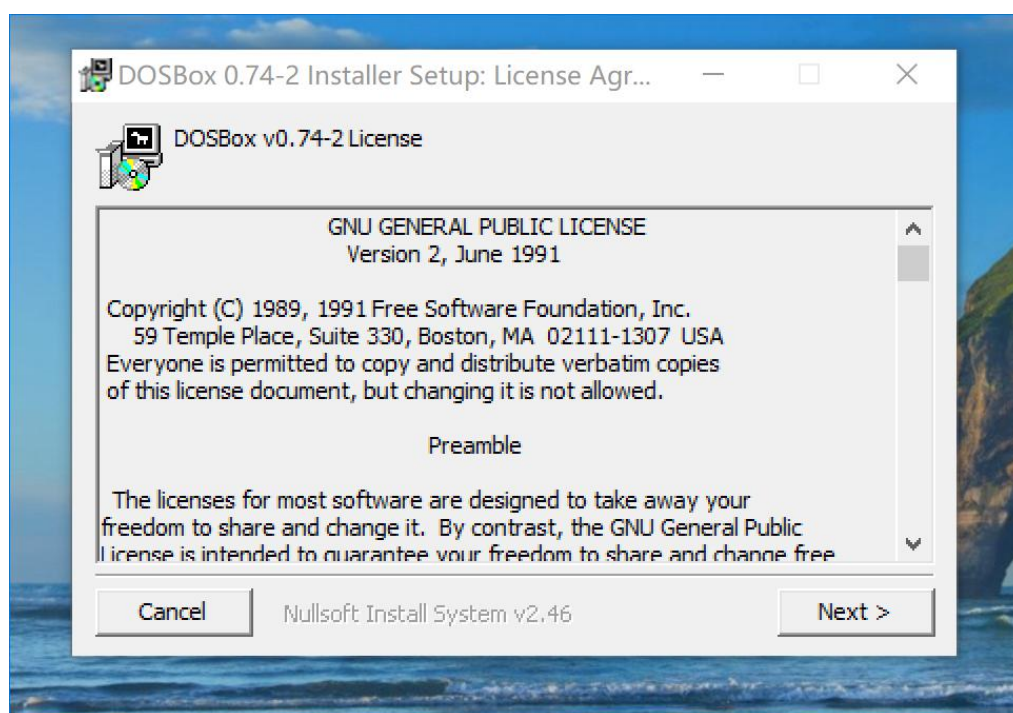


图 4.2 安装 DOSBox 步骤 1

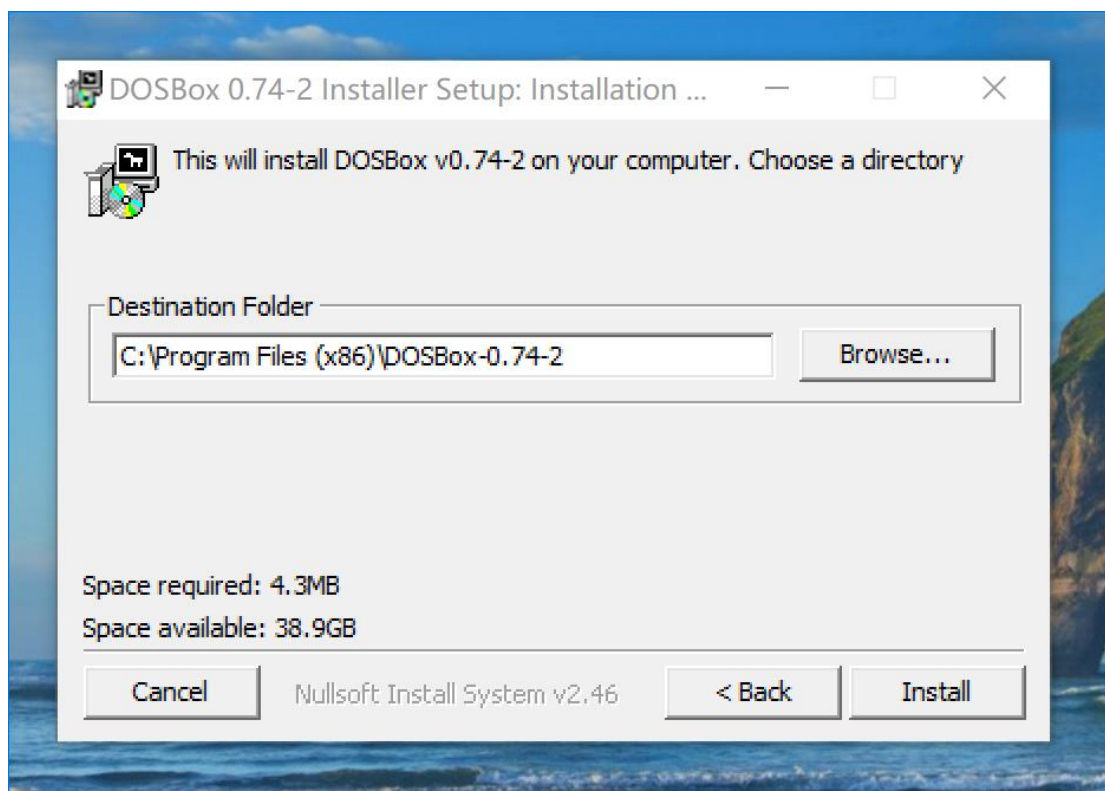


图 4.3 安装 DOSBox 步骤 2

安装 DOSBox 完成后，我需要创建一个用于保存汇编工具（MASM、Link、DEBUG 等工具）和汇编文件 (*.asm) 的目录：C:\asm，然后把 MASM 文件夹中的文件都拷贝到该目录中。如图 4.4 所示。

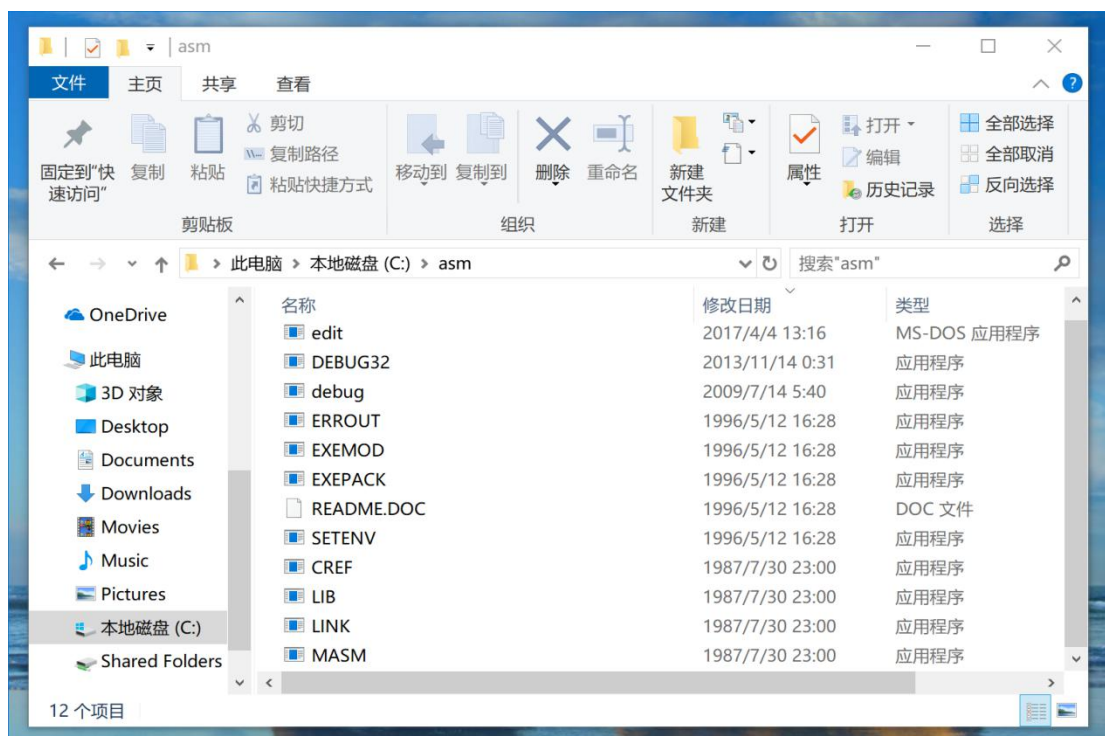


图 4.4 创建目录保存汇编工具和汇编文件

然后，当我打开 DOSBox 程序，会发现如图 4.5 所示的两个界面，我只需要对小的界面进行操作，如图 4.6 所示。

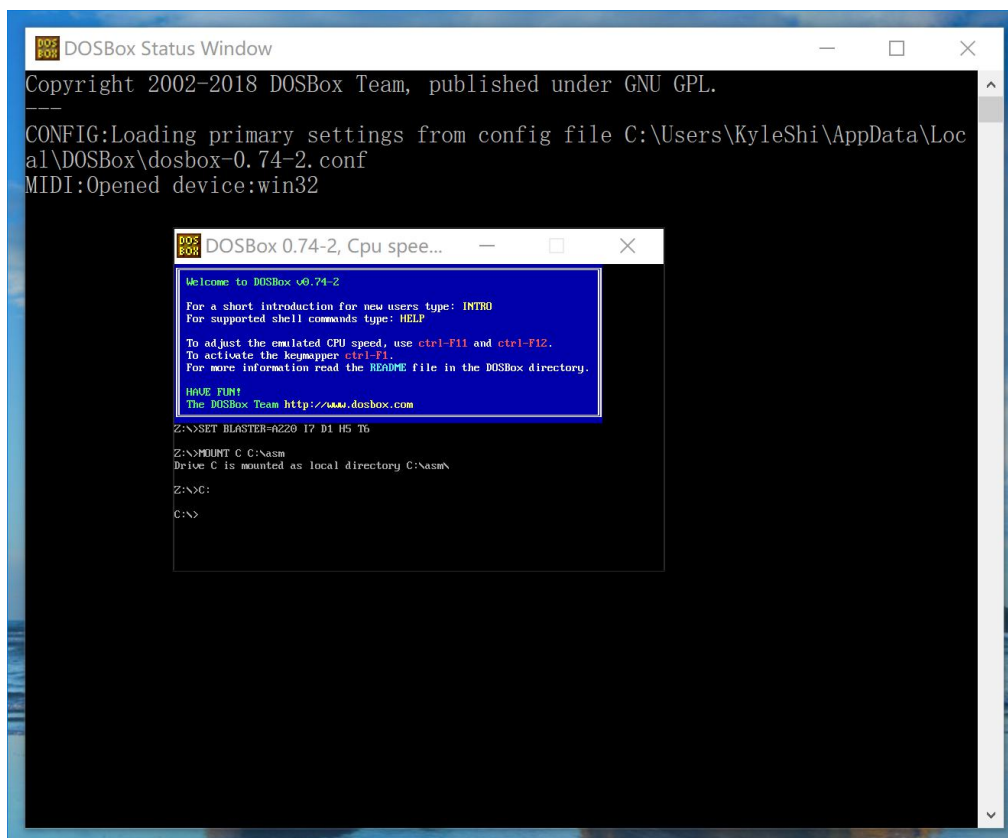


图 4.5 DOSBox 的两个界面

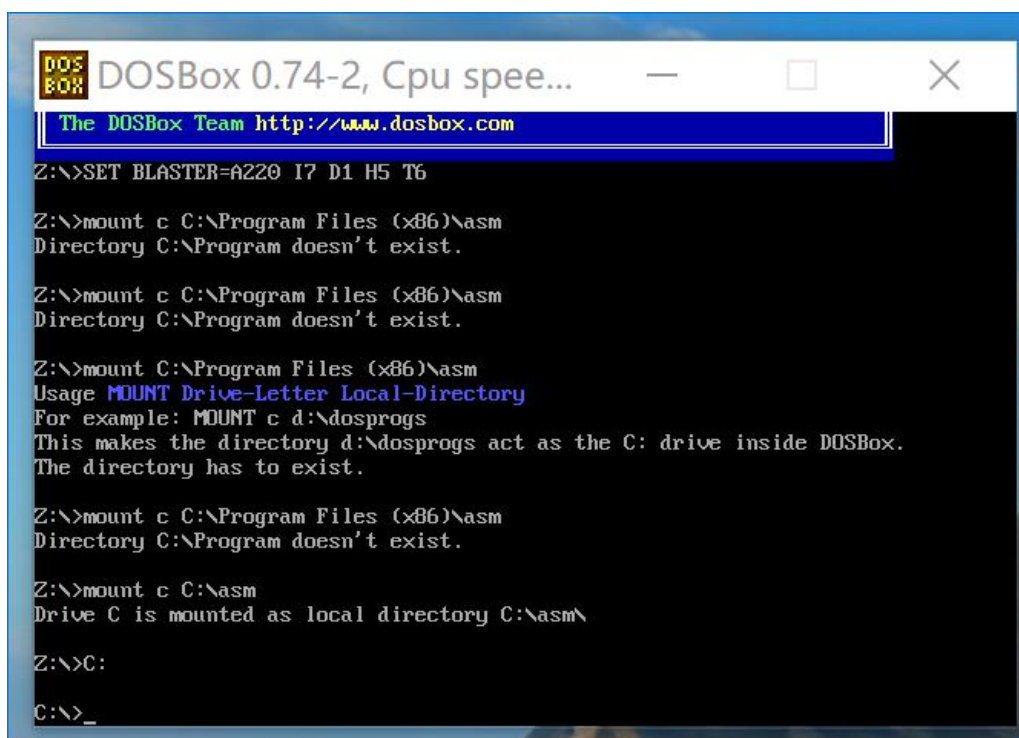
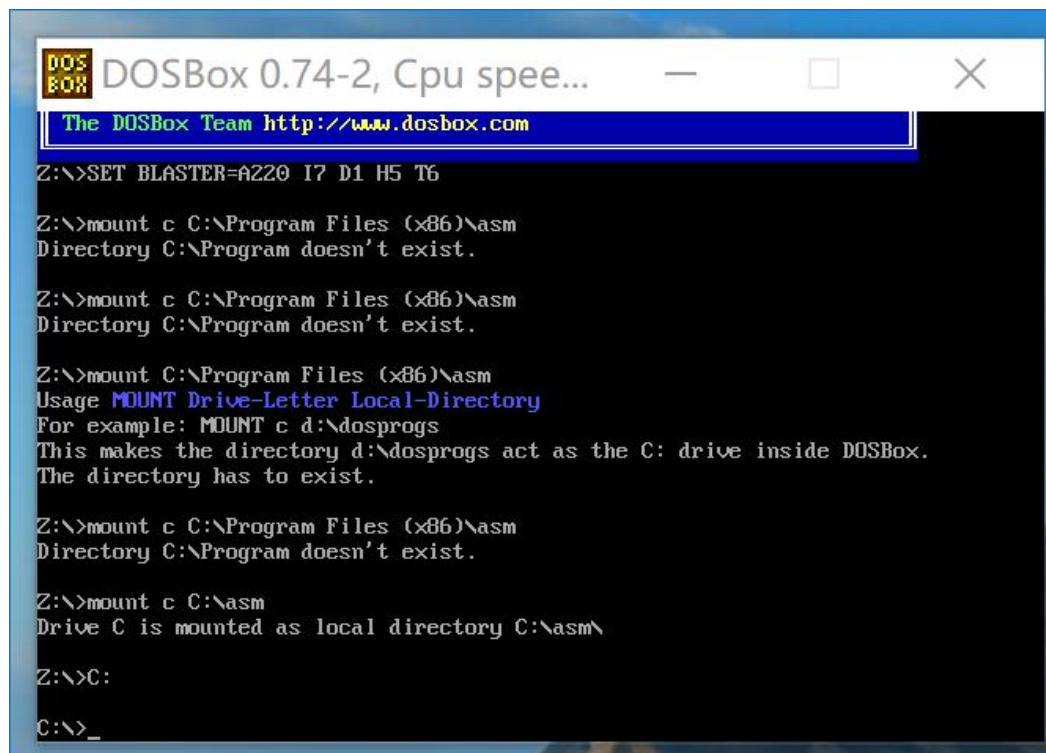


图 4.6 需要操作的 DOSBox 界面

然后，我需要进入 DOS 环境。首先，输入命令：MOUNT C C:\asm（将目录 C:\asm 挂载为 DOSBox 下的 C:）；然后，输入命令：C:，将 C:\asm 写入环境变量 PATH 中，执行结果如图 4.7 所示。之后我就进入了 DOS 的环境，在这里就可以编写汇编程序了。



```
DOS BOX DOSBox 0.74-2, Cpu spee...
The DOSBox Team http://www.dosbox.com

Z:\>SET BLASTER=A220 I7 D1 H5 T6

Z:\>mount c C:\Program Files (x86)\asm
Directory C:\Program doesn't exist.

Z:\>mount c C:\Program Files (x86)\asm
Directory C:\Program doesn't exist.

Z:\>mount C:\Program Files (x86)\asm
Usage MOUNT Drive-Letter Local-Directory
For example: MOUNT c d:\dosprogs
This makes the directory d:\dosprogs act as the C: drive inside DOSBox.
The directory has to exist.

Z:\>mount c C:\Program Files (x86)\asm
Directory C:\Program doesn't exist.

Z:\>mount c C:\asm
Drive C is mounted as local directory C:\asm\

Z:\>C:
C:\>_
```

图 4.7 进入 DOS 环境

为了避免每一次进入界面都要输入上面的命令，我可以进行如下所示的简单配置，这样以后使用时可以直接进入我们想要的目录：打开 DOSBox 的安装根目录 C:\Program Files (x86)\DOSBox-0.74，双击文件 DOSBox 0.74 Options.bat，运行该批处理文件后，系统会用文本文件 Notepad 打开配置文件 dosbox-0.74.conf。将光标定位到 dosbox-0.74.conf 文件的[autoexec]节点（在该文件末尾），在文件中添加以下内容：

```
MOUNT C D:\DEBUG
```

```
C:
```

修改配置文件的过程如图 4.8 所示。

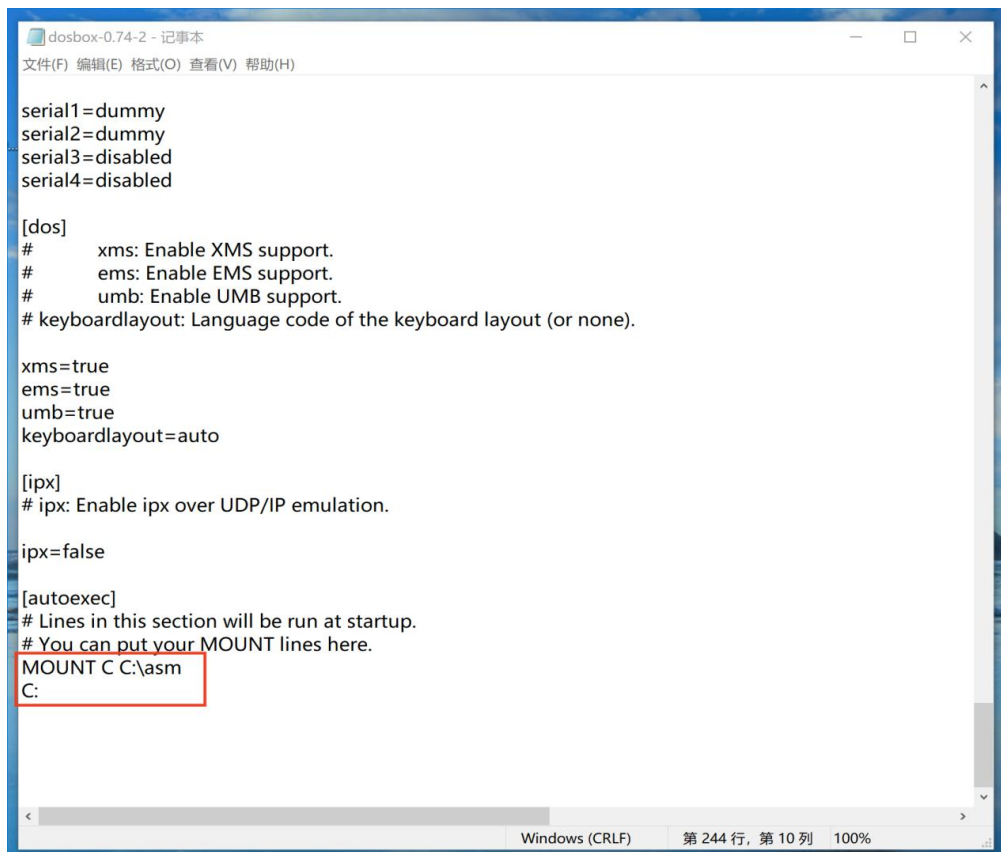


图 4.8 修改配置文件 dosbox-0.74.conf

保存文件后，再次运行 DOSBox 之时，就可以看到直接进入了 DOS 环境，如图 4.9 所示。

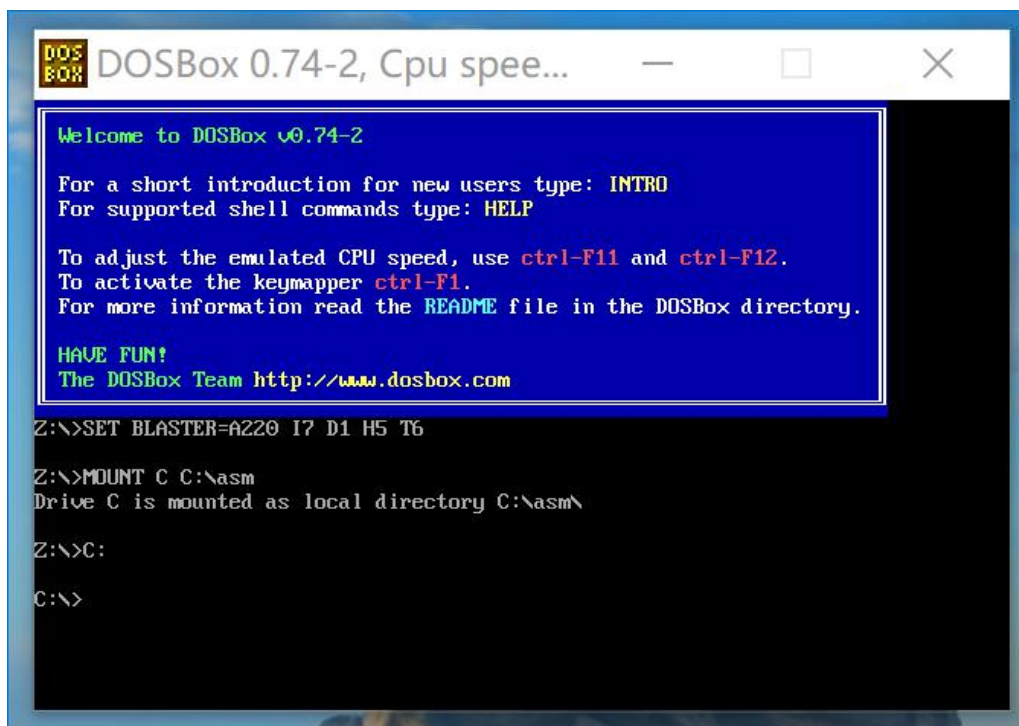
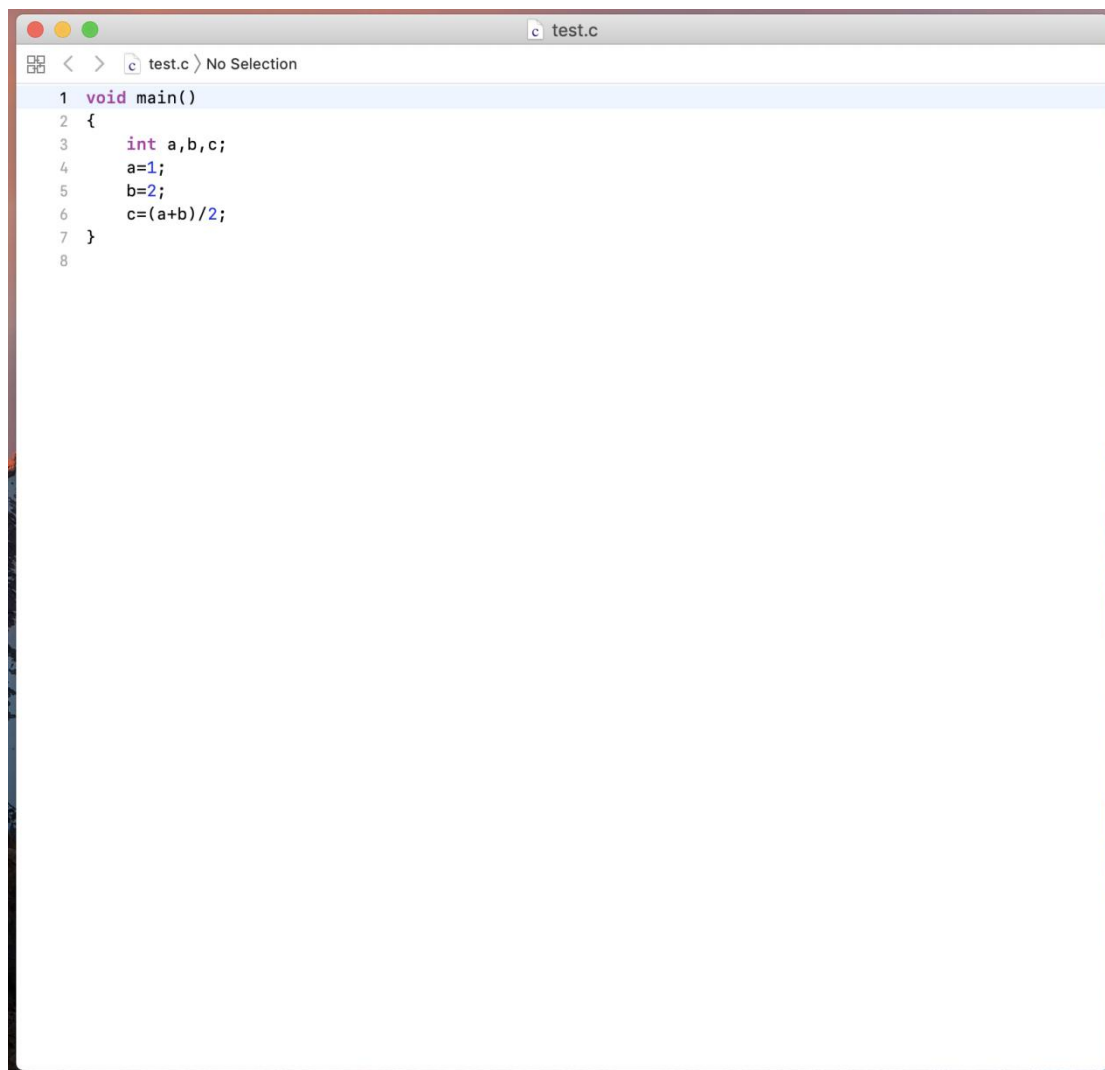


图 4.9 修改配置文件完成后，DOSBox 将直接进入 DOS 界面

4.2 生成测试用例 test.c 的抽象语法树

汇编开发环境搭建完成后，我需要生成 test.c 文件的抽象语法树，具体过程如下所示：打开 Visual Studio 2010 命令行，依次输入以下命令：flex -l input.lex, bison -d cgrammar-new.y, cl lex.yy.c cgrammar-new.tab.c main.c parser.c, cl lex.yy.c cgrammar-new.tab.c main.c parser.c -o"test.exe", test.exe < test.c, 命令执行完成后 out 文件中将被写入对 test.c 进行语法分析的符号表和抽象语法树。由于生成抽象语法树的具体过程在之前的实验四中完成过，在此不再赘述。test.c 文件中的代码如图 4.10 所示，test.c 的抽象语法树如图 4.11 所示。



```
1 void main()
2 {
3     int a,b,c;
4     a=1;
5     b=2;
6     c=(a+b)/2;
7 }
8
```

图 4.10 测试用例 test.c 文件

Abstract Syntax Tree ...

```

+ goal_
+ extdef_
+ funcdef_
+ funcdecl_
+ decl_spec_
+ void_
+ direct_decl_
+ funcdecl_
+ ident_
+ IDENT_ (main)
+ funcbody_
+ compound_stmt_
+ declarations_
+ decl_init_
+ decl_spec_
+ int_
+ init_declarators_
+ declarator_
+ direct_decl_
+ ident_
+ IDENT_ (a)
+ declarator_
+ direct_decl_
+ ident_
+ IDENT_ (b)
+ declarator_
+ direct_decl_
+ ident_
+ IDENT_ (c)
+ statements_
+ exp_
+ assignment_
+ equals_
+ IDENT_ (a)
+ CONST_ (1)
+ exp_
+ assignment_
+ equals_
+ IDENT_ (b)
+ CONST_ (2)
+ exp_
+ assignment_
+ equals_
+ IDENT_ (c)
+ div_
+ exp_
+ assignment_
+ add_
+ IDENT_ (a)
+ IDENT_ (b)
+ CONST_ (2)

```

图 4.11 test.c 对应的语法分析树

4.3 生成四元式

接下来，我需要利用抽象语法树生成四元式。由编译原理的知识可知，四元式代码中的临时变量对应抽象语法树的内部节点，因此我可以通过遍历抽象语法树生成对应的四元式。在查阅了相关资料后，我的大致思路为：从叶节点开始遍历抽象语法树，若该节点的父节点为赋值节点，则先后为该父节点的右子节点和左子节点创建四元式；若该节点的父节点不是赋值节点，则只为该父节点创建四元式。遍历抽象语法树生成四元式的 C++程序 `QuadrupleGenerate.cpp` 如下所示：

```

namespace translate {    /* 翻译 */
    using namespace grammar;
    ofstream fout("/Users/apple/Desktop/实验 9_编译器代码/Quadruple.txt");

    vector<vector<string>> table; /* 符号表 */
    int temp_cnt = 0;           /* 变量计数 */
    int nextquad = 100;        /* 四元式标号 */

    void mktable() {          /* 新建符号表 */
        table.push_back(vector<string>());
    }

    void rmtable() {          /* 删除符号表 */
        table.pop_back();
    }

    void enter(string name) { /* 声明变量 */
        table.back().push_back(name);
    }

    string lookup(string name) { /* 查看变量是否存在 */
        per(i, 0, table.size()) rep(j, 0, table[i].size())
            if (table[i][j] == name) return name;
        return "nil";
    }

    string newtemp() {         /* 新建一个变量 */
        return "T" + to_string(++temp_cnt);
    }

    pair<int, vector<string>> gen(string a, string b, string c, string d) { /* 生成
四元式 */
        vector<string> vs{ a,b,c,d };
        return make_pair(nextquad++, vs);
    }
}

```

```

    }

void dfs(int u) { /* 遍历语法树 */
    if (G[u].empty()) return; /* 如果为空直接 return */
    if (symbol[G[u].front()] == "{") mktable(); /* 如果第一个儿子为
{, 新建符号表 */
    rep(i, 0, G[u].size()) dfs(G[u][i]); /* 遍历所有的儿子节点 */

    E &e = attr[u]; /* e 为 attr[u]的引用, 便于直接修改 */
    e = attr[G[u][0]]; /* 简写, 把第一个儿子的直接赋值给 attr[u] */
    attr[u].code.clear(); /* 先清空 code 四元式 */
    rep(i, 0, G[u].size()) { /* 把所有儿子的四元式加进来 */
        for (auto it : attr[G[u][i]].code) {
            e.code.push_back(it);
        }
    }

    if (symbol[u] == "variable_definition") { /* 函数定义 */
        string name = attr[G[u][1]].name;
        enter(name);
    }
    else if (symbol[u] == "assignment_expression" && symbol[G[u][0]] ==
"ID") { /* 赋值语句 */
        string p = lookup(attr[G[u][0]].name); /* 查符号表 */
        if (p == "nil") {
            cerr << "变量未声明" << endl;
            return;
        }
        e.place = p;
        e.code.push_back(gen("=", attr[G[u][2]].place, "-", e.place));
    }
    else if (symbol[u] == "primary_expression" && symbol[G[u][0]] ==
"NUM") { /* 规约 NUM */
        e.place = newtemp();
    }
}

```

```

        e.value = attr[G[u][0]].value;
        e.code.push_back(gen("=", e.value, "-", e.place));
    }
    else if (symbol[u] == "primary_expression" && symbol[G[u][0]] ==
"ID") { /* 规约 ID */
        string p = lookup(attr[G[u][0]].name);
        if (p == "nil") {
            cerr << "变量未声明" << endl;
            return;
        }
        e.place = p;
        e.name = attr[G[u][0]].name;
    }
    else if (symbol[u] == "additive_expression" && G[u].size() > 1 &&
symbol[G[u][1]] == "+") { /* 加法表达式 */
        E e1 = attr[G[u][0]];
        E e2 = attr[G[u][2]];
        e.place = newtemp();
        e.code.push_back(gen("+", e1.place, e2.place, e.place));
    }
    else if (symbol[u] == "subtract_expression" && G[u].size() > 1 &&
symbol[G[u][1]] == "-") { /* 减法表达式 */
        E e1 = attr[G[u][0]];
        E e2 = attr[G[u][2]];
        e.place = newtemp();
        e.code.push_back(gen("-", e1.place, e2.place, e.place));
    }
    else if (symbol[u] == "multiplicative_expression" && G[u].size() > 1
&& symbol[G[u][1]] == "*") { /* 乘法表达式 */
        E e1 = attr[G[u][0]];
        E e2 = attr[G[u][2]];
        e.place = newtemp();
        e.code.push_back(gen("*", e1.place, e2.place, e.place));
    }
}

```

```

else if (symbol[u] == "multiplicative_expression" && G[u].size() > 1
&& symbol[G[u][1]] == "/") {  /* 除法表达式 */
    E e1 = attr[G[u][0]];
    E e2 = attr[G[u][2]];
    e.place = newtemp();
    e.code.push_back(gen("/", e1.place, e2.place, e.place));
}
else if (symbol[u] == "relational_expression" && G[u].size() > 1) {  /*
关系表达式 */
    E id1 = attr[G[u][0]];
    E id2 = attr[G[u][2]];
    e.code.push_back(gen("j" + symbol[G[u][1]], id1.place, id2.place,
"0"));

    e.True = e.code.back().first;
    e.code.back().second[3] = to_string(e.code.back().first + 2);
    e.code.push_back(gen("j", "-", "-", "0"));
    e.False = e.code.back().first;
}
else if (symbol[G[u][0]] == "WHILE") {  /* WHILE 语句 */
    e = attr[G[u][0]];
    attr[u].code.clear();
    for (auto it : attr[G[u][2]].code)
        e.code.push_back(it);
    for (auto it : attr[G[u][4]].code)
        e.code.push_back(it);
    e.code.push_back(gen("j", "-", "-", to_string(e.code.front().first)));

    for (auto &it : e.code) {
        if (it.first == attr[G[u][2]].False) {
            it.second[3] = to_string(e.code.back().first + 1);
            break;
        }
    }
    e.code.back().second[3] = to_string(e.code.front().first);
}

```



```

    }
else if (symbol[G[u][0]] == "IF" && G[u].size() == 5) { /* IF 语句 */
    e = attr[G[u][0]];
    attr[u].code.clear();
    for (auto it : attr[G[u][2]].code)
        e.code.push_back(it);
    for (auto it : attr[G[u][4]].code)
        e.code.push_back(it);
    for (auto &it : e.code) {
        if (it.first == attr[G[u][2]].False) {
            it.second[3] = to_string(e.code.back().first + 1);
            break;
        }
    }
}
else if (symbol[G[u][0]] == "IF" && G[u].size() == 7) { /* IF ELSE 语
句 */
    e = attr[G[u][0]];
    attr[u].code.clear();
    for (auto it : attr[G[u][2]].code)
        e.code.push_back(it);
    for (auto it : attr[G[u][4]].code)
        e.code.push_back(it);
    e.code.push_back(gen("j", "-", "-", to_string(e.code.back().first +
attr[G[u][6]].code.size() + 2)));
    e.code.back().first = e.code.front().first + e.code.size() - 1;
    for (auto &it : e.code) {
        if (it.first == attr[G[u][2]].False) {
            it.second[3] = to_string(e.code.back().first + 1);
            break;
        }
    }
    for (auto it : attr[G[u][6]].code) {
        it.first++;
    }
}

```

```

        if (it.second[0][0] == 'j') {
            int num = atoi(it.second[3].c_str());
            it.second[3] = to_string(num + 1);
        }
        e.code.push_back(it);
    }
}

if (symbol[G[u].back()] == "{") mktable();    /* 如果最后一个儿子
为 “}” ， 则删除符号表 */
}

void main() {
    int rt = cnt - 1;    /* rt 为根节点 */
    dfs(rt);
    for (auto it : attr[rt].code)
        fout << it.first << " (" << it.second[0] << ", " << it.second[1]
        << ", " << it.second[2] << ", " << it.second[3] << ")" << endl;
    fout << nextquad << endl;    /* 多输出一行 */
}
}

```

生成的四元式文件 Quadruple.txt 如图 4.12 所示。

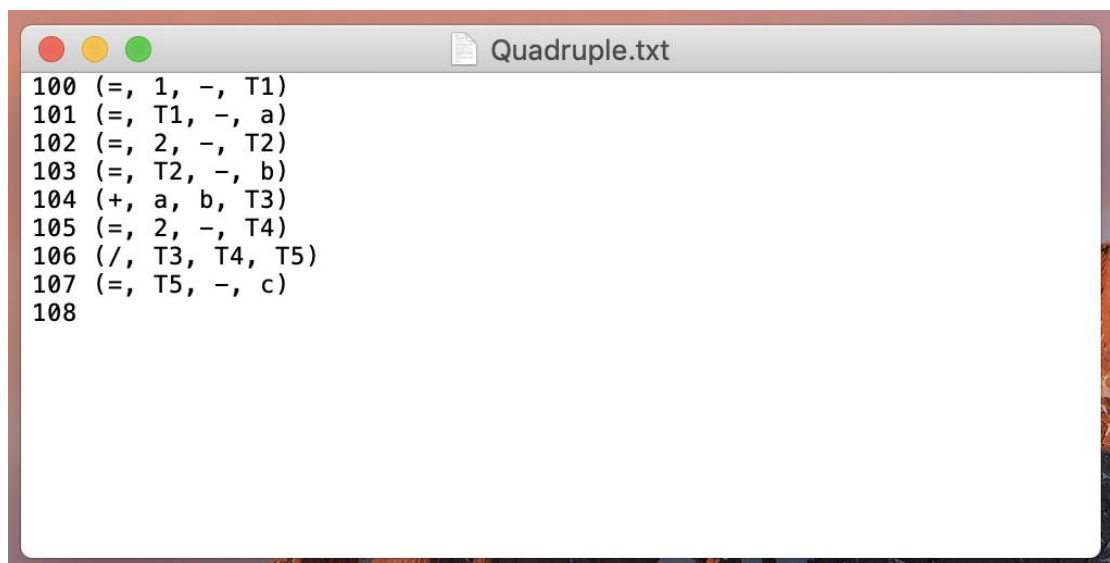


图 4.12 通过遍历 test.c 的语法树生成的四元式

4.4 生成汇编文件

生成四元式后，我需要将四元式转换成汇编文件。以上一步中生成的四元式作为输入，输出汇编语言程序 result.asm。基本思路为：通过查看语义分析所产生的结果，分析四元式，输出相应的汇编语句。本次目标代码的翻译采用了 8086 汇编指令集，数据均为 16 位无符号整型数据。将四元式转换成汇编文件的程序 CodeGenerate.cpp 中的核心代码如下所示。

```
namespace translate {    /* 翻译 */
    using namespace grammar;
    ofstream fout("/Users/apple/Desktop/实验 9_编译器代码/Quadruple.txt");

    vector<vector<string>> table;    /* 符号表 */
    int temp_cnt = 0;                /* 变量计数 */
    int nextquad = 100;              /* 四元式标号 */

    void mktable() {                /* 新建符号表 */
        table.push_back(vector<string>());
    }

    void rmtable() {                /* 删除符号表 */
        table.pop_back();
    }

    void enter(string name) {        /* 声明变量 */
        table.back().push_back(name);
    }

    string lookup(string name) {     /* 查看变量是否存在 */
        per(i, 0, table.size()) rep(j, 0, table[i].size())
            if (table[i][j] == name) return name;
        return "nil";
    }

    string newtemp() {              /* 新建一个变量 */
```

```

        return "T" + to_string(++temp_cnt);
    }

    pair<int, vector<string>> gen(string a, string b, string c, string d) { /* 生成四元式 */
        vector<string> vs{ a,b,c,d };
        return make_pair(nextquad++, vs);
    }

    void dfs(int u) { /* 遍历语法树 */
        if (G[u].empty()) return; /* 如果为空直接 return */
        if (symbol[G[u].front()] == "{") mktable(); /* 如果第一个儿子为{, 新建符号表 */
        rep(i, 0, G[u].size()) dfs(G[u][i]); /* 遍历所有的儿子节点 */

        E &e = attr[u]; /* e 为 attr[u]的引用, 便于直接修改 */
        e = attr[G[u][0]]; /* 简写, 把第一个儿子的直接赋值给 attr[u] */
        attr[u].code.clear(); /* 先清空四元式 */
        rep(i, 0, G[u].size()) { /* 把所有儿子的四元式加进来 */
            for (auto it : attr[G[u][i]].code) {
                e.code.push_back(it);
            }
        }

        if (symbol[u] == "variable_definition") { /* 函数定义 */
            string name = attr[G[u][1]].name;
            enter(name);
        }
        else if (symbol[u] == "assignment_expression" && symbol[G[u][0]] == "ID") { /* 赋值语句 */
            string p = lookup(attr[G[u][0]].name); /* 查符号表 */
            if (p == "nil") {
                cerr << "变量未声明" << endl;
                return;
            }
        }
    }

```

```

    }
    e.place = p;
    e.code.push_back(gen("=", attr[G[u][2]].place, "-", e.place));
}
else if (symbol[u] == "primary_expression" && symbol[G[u][0]] ==
"NUM") { /* 规约 NUM */
    e.place = newtemp();
    e.value = attr[G[u][0]].value;
    e.code.push_back(gen("=", e.value, "-", e.place));
}
else if (symbol[u] == "primary_expression" && symbol[G[u][0]] ==
"ID") { /* 规约 ID */
    string p = lookup(attr[G[u][0]].name);
    if (p == "nil") {
        cerr << "变量未声明" << endl;
        return;
    }
    e.place = p;
    e.name = attr[G[u][0]].name;
}
else if (symbol[u] == "multiplicative_expression" && G[u].size() > 1
&& symbol[G[u][1]] == "*") { /* 乘法表达式 */
    E e1 = attr[G[u][0]];
    E e2 = attr[G[u][2]];
    e.place = newtemp();
    e.code.push_back(gen("*", e1.place, e2.place, e.place));
}
else if (symbol[u] == "additive_expression" && G[u].size() > 1 &&
symbol[G[u][1]] == "+") { /* 加法表达式 */
    E e1 = attr[G[u][0]];
    E e2 = attr[G[u][2]];
    e.place = newtemp();
    e.code.push_back(gen("+", e1.place, e2.place, e.place));
}
}

```

```

        else if (symbol[u] == "relational_expression" && G[u].size() > 1)
{   /* 关系表达式 */
    E id1 = attr[G[u][0]];
    E id2 = attr[G[u][2]];
    e.code.push_back(gen("j" + symbol[G[u][1]], id1.place, id2.place,
"0"));

    e.True = e.code.back().first;
    e.code.back().second[3] = to_string(e.code.back().first + 2);
    e.code.push_back(gen("j", "-", "-", "0"));
    e.False = e.code.back().first;
}
else if (symbol[G[u][0]] == "WHILE") {   /* WHILE 语句 */
    e = attr[G[u][0]];
    attr[u].code.clear();
    for (auto it : attr[G[u][2]].code)
        e.code.push_back(it);
    for (auto it : attr[G[u][4]].code)
        e.code.push_back(it);
    e.code.push_back(gen("j", "-", "-", to_string(e.code.front().first)));

    for (auto &it : e.code) {
        if (it.first == attr[G[u][2]].False) {
            it.second[3] = to_string(e.code.back().first + 1);
            break;
        }
    }
    e.code.back().second[3] = to_string(e.code.front().first);
}
else if (symbol[G[u][0]] == "IF" && G[u].size() == 5) {   /* IF 语句
*/

    e = attr[G[u][0]];
    attr[u].code.clear();
    for (auto it : attr[G[u][2]].code)
        e.code.push_back(it);

```



```

for (auto it : attr[G[u][4]].code)
    e.code.push_back(it);
for (auto &it : e.code) {
    if (it.first == attr[G[u][2]].False) {
        it.second[3] = to_string(e.code.back().first + 1);
        break;
    }
}
}
else if (symbol[G[u][0]] == "IF" && G[u].size() == 7) { /* IF ELSE
语句 */

    e = attr[G[u][0]];
    attr[u].code.clear();
    for (auto it : attr[G[u][2]].code)
        e.code.push_back(it);
    for (auto it : attr[G[u][4]].code)
        e.code.push_back(it);
    e.code.push_back(gen("j", "-", "-", to_string(e.code.back().first +
attr[G[u][6]].code.size() + 2)));
    e.code.back().first = e.code.front().first + e.code.size() - 1;
    for (auto &it : e.code) {
        if (it.first == attr[G[u][2]].False) {
            it.second[3] = to_string(e.code.back().first + 1);
            break;
        }
    }
}
for (auto it : attr[G[u][6]].code) {
    it.first++;
    if (it.second[0][0] == 'j') {
        int num = atoi(it.second[3].c_str());
        it.second[3] = to_string(num + 1);
    }
    e.code.push_back(it);
}
}

```

```

    }

    if (symbol[G[u].back()] == "{") mktable(); /* 如果最后一个儿子
为}, 删除符号表 */
    }

void main() {
    int rt = cnt - 1; /* rt 为根节点 */
    dfs(rt);
    for (auto it : attr[rt].code)
        fout << it.first << " (" << it.second[0] << ", " << it.second[1]
        << ", " << it.second[2] << ", " << it.second[3] << ")" <<
endl;

    fout << nextquad << endl; /* 多输出一行 */
}
}

```

生成的汇编文件 result.asm 如图 4.13 所示。

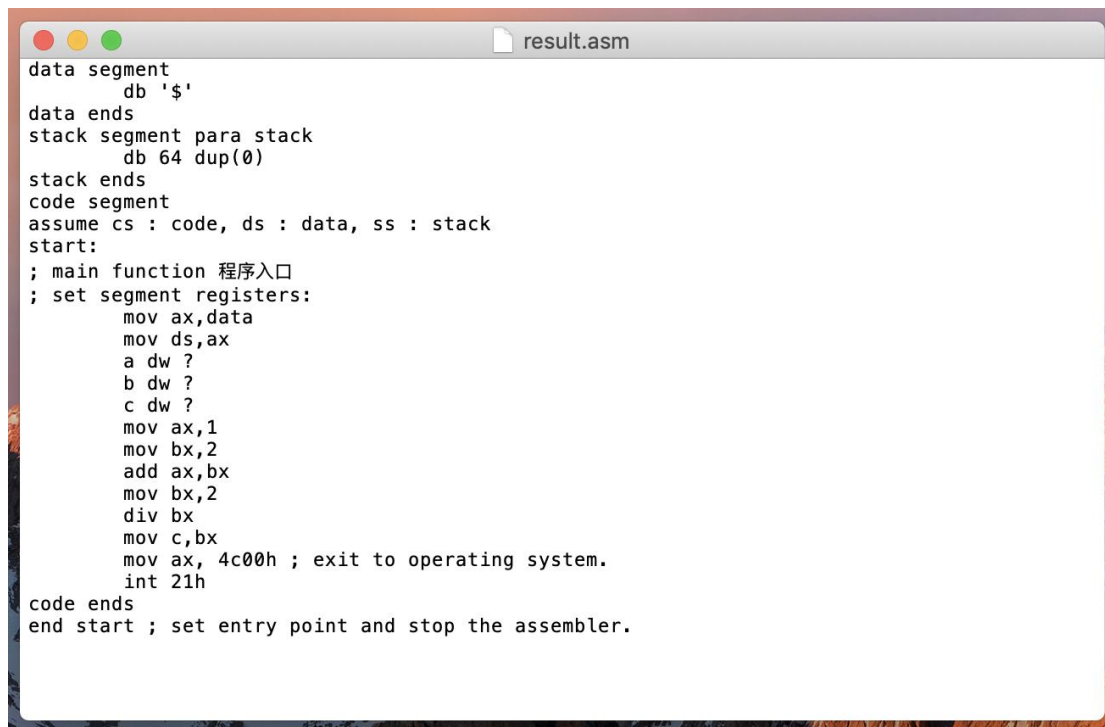
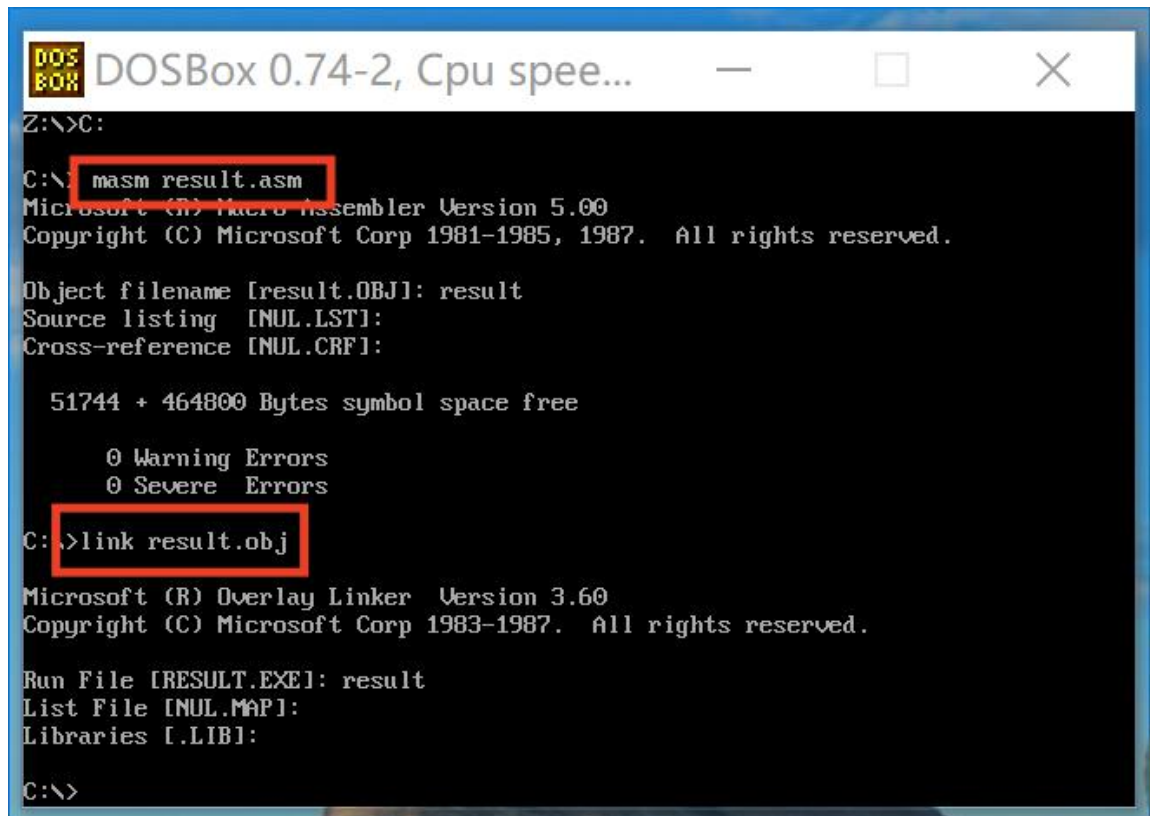


图 4.13 生成的汇编文件 result.asm

4.5 生成目标文件和可执行文件

生成汇编文件 result.asm 后，我需要利用 DOSBox 和 masm 生成目标文件和可执行文件。具体步骤为：将 result.asm 放入之前创建的汇编文件目录 C:\asm，然后打开 DOSBox，输入命令：masm result.asm，生成目标文件 result.obj，然后输入命令：link result.obj，链接生成可执行文件 result.exe。命令执行过程如图 4.14 所示，生成的目标文件 result.obj 和可执行文件 result.exe 如图 4.15 所示。



```
DOS
BOX DOSBox 0.74-2, Cpu spee...
Z:\>C:
C:\> masm result.asm
Microsoft (R) Macro Assembler Version 5.00
Copyright (C) Microsoft Corp 1981-1985, 1987. All rights reserved.

Object filename [result.OBJ]: result
Source listing [NUL.LST]:
Cross-reference [NUL.CRF]:

51744 + 464800 Bytes symbol space free

0 Warning Errors
0 Severe Errors

C:\>link result.obj
Microsoft (R) Overlay Linker Version 3.60
Copyright (C) Microsoft Corp 1983-1987. All rights reserved.

Run File [RESULT.EXE]: result
List File [NUL.MAP]:
Libraries [.LIB]:

C:\>
```

图 4.14 执行命令生成 result.obj 和 result.exe

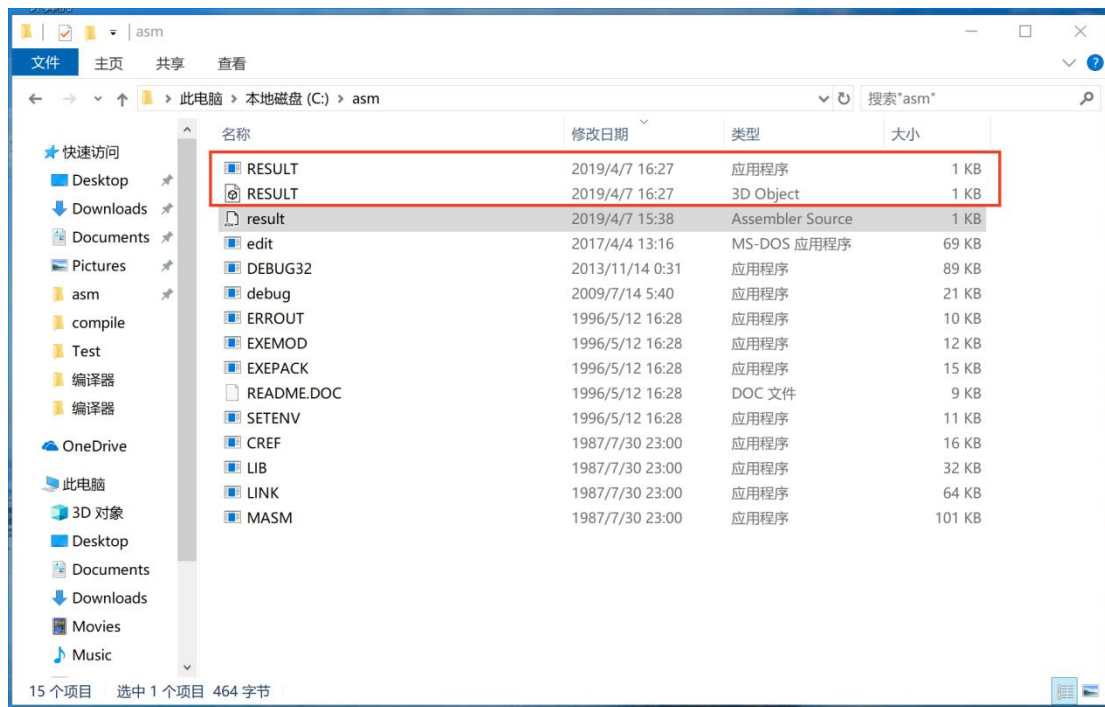


图 4.15 生成的目标文件和可执行文件

然后，依次输入以下命令：debug result.exe，u(u 命令作用：对机器代码反汇编显示)，对机器代码反汇编显示，如图 4.16 所示。

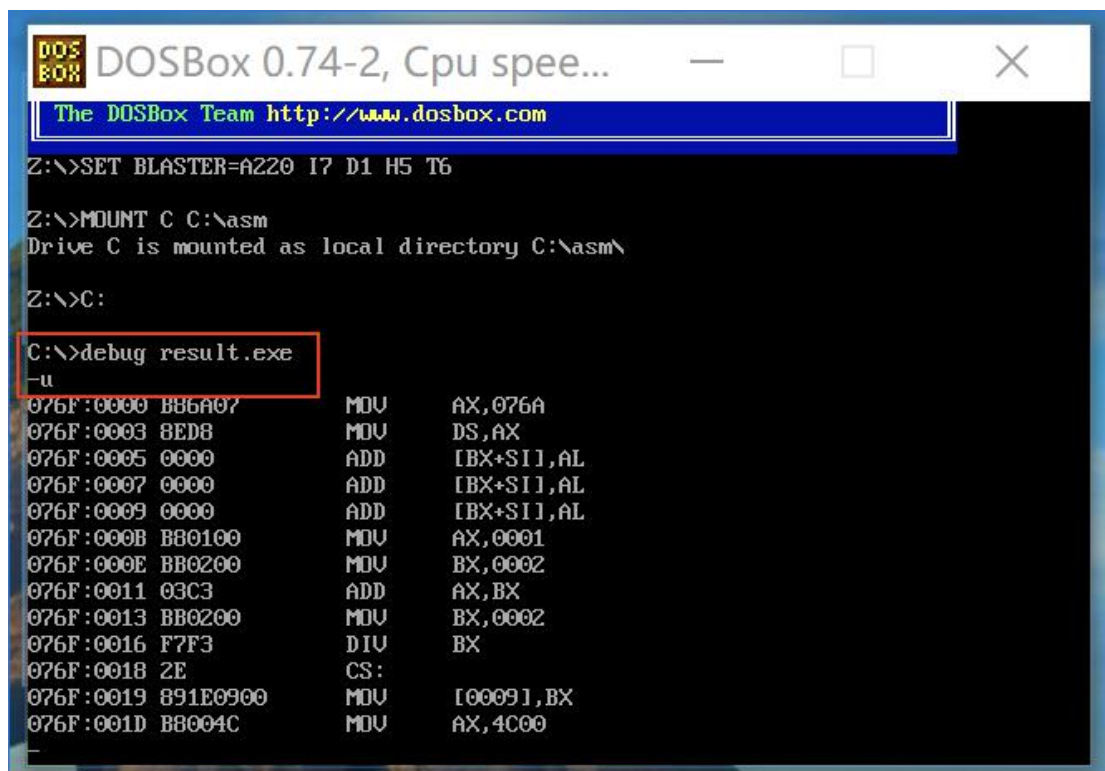


图 4.16 执行反汇编命令

5 实验总结

5.1 编程中的困难

本次实验是系统软件开发实践课程的最后一次实验，难度较大，需要我自主地完成 C 编译器后端的设计，刚开始时，我对利用抽象语法树生成四元式并将四元式翻译成汇编文件的原理不是十分理解，不知道从何下手。但通过我不断地查阅资料，我最终设计出了恰当的程序实现了 C 语言编译器后端的设计。

5.2 程序评价

在本次实验中，我主要使用了两个程序 `QuadrupleGenerate.cpp` 和 `CodeGenerate.cpp` 生成四元式和目标代码，这两个程序较好地完成了利用抽象语法树生成四元式以及将四元式翻译成汇编代码的任务，但程序还存在以下不足：

(1) 在错误处理时，程序只能够粗略输出错误产生原因，并无法输出错误所在行数；

(2) 编译器必须在前一阶段全部结束后，才能开始下一阶段的分析，工作效率不是很高。

5.3 实验收获

在本次实验中，我进行了 C 编译器后端设计，将测试用例 `test.c` 成功转换成了可执行文件 `result.exe`。我的实验收获主要有以下几点：

(1) 万事开头难。在编写程序时，一定要先理清思路，否则将事半功倍；

(2) 程序的模块封装工作十分重要。各模块之间需要保持相对独立，这样无论是在测试还是修改时，都不会产生各模块之间大规模的影响；

(3) 对于数据结构的选取要谨慎，否则会加重程序的编写负担，应该尽量采用编译器已经封装好的类，函数，算法；

(4) 编程很重要，文档的撰写也同样重要，譬如分析表、符号表等；

(5) 加深对知识的理解，对代码的编写有着巨大的帮助。在实验开始时，我做了很多无用功，究其根本是对编译器的原理理解地不够透彻，不知道如何实现。只有真正理解了原理，才能编写出正确高效的代码。