编译原理课程设计

学 院（系）：电子信息与电气工程学部

学 生 姓 名：胡文博

学　　号：201474009

班　　级：电计 1402

同 组 人：祝磊（电计 1402）

大连理工大学

Dalian University of Technology

目　　录

1　源语言定义 1
2　词法分析程序的实现 1
　2.1　词法分析器的数据结构 1
　2.2　词法分析器的数学模型-DFA 2
　2.3　词法分析器的测试 4
3 递归下降法实现语法分析程序 4
4 虚拟机的设计与实现 4
　4.1 虚拟机架构选择 4
　4.2 虚拟机指令详述 4
　4.3 虚拟机测试 6
5 编辑器的实现 6
　5.1 界面设计和语法高亮显示 7
　5.2 基于字典树的自动补全算法 8
　5.3 利用进程调用实现编辑器和编译器以及虚拟机的对接 10
感　　想 11

1 源语言定义

2 词法分析程序的实现

2.1 词法分析器的数据结构

对于词法分析器而言，其输入是源代码文件，输出是有一定数据结构的 token 流。对于 token 我们设计了如图 2.1.1 所示的数据结构，其中 Term 是一个 enum 类型，其结构如图 2.1.2 所示。我们并没有按常规的那样将 token 的类型分为关键字，标识符，数

```
class Token
{
public:
    Token()
    {
        term = _OVER;
        value = "";
    }
    Term term;
    string value;
    void disp();
};
```

图 2.1.1

```
enum Term
{
    _MAIN,
    _INT,
    _WHILE,
    _IF,
    _ELSE,
    _RETURN,
    _VOID,

    _PRINT,
    _READ,
    _FOR,

    _NOT,                   // !
    _ADD,                   // +
    _SUB,                   // -
    _MUL,                   // *
    _DIV,                   // /
    _MOD,                   // %
    _BIGGER,                // >
    _SMALLLER,              // <
    _COMMA,                 // ,
    _SEMICOLON,             // ;
    _BRACE_L,               // {
    _BRACE_R,               // }
    _PARENTHESE_L,          // (
    _PARENTHESE_R,          // )
    _BRACKET_L,             // [
    _BRACKET_R,             // ]
    _ASSIGN,                // =
    _EQUAL,                 // ==
    _BIGGEROREQUAL,         //>=
    _SIMMALLEROREQUAL,      //<=
    _NOTEQUAL,              //!=
    _ID,
    _NUM,
    _OVER                   //the flag of source over
};
```

图 2.1.2

字以及符号，而是使用了更细的分类，如图 2.1.2 中，我们将每一个关键字每一种符号都分为单独的一类，这样做的目的是方便后续语法分析器的开发如果按照常规分法的话当语法分析器读到一个 token 的类型是关键字的时候还需要读 token 的值来确定它是什么关键字，而我们这样的设计将这些细分的种类作为 enum 类型则在语法分析时只需要读取该 token 的类型便可以确定关键字是什么，提高编译器的执行效率。另外一个值得注意的地方是我们在 token 类型里添加了一个 token 结束的标志“_OVER”，引入这个标志的原因是我们的编译器为了高效采用了一遍扫描的结构，而一遍扫描的结构需要其

作用是告诉语法分析器源代码的结束位置，我们用这个标志来告诉语法分析器不能再向词法分析器请求下一个 token 了。

2.2 词法分析器的数学模型-DFA

词法分析器在数学上等价于一个确定的有穷状态机，为了使 DFA 看起来清晰，我们将我们将上文所说的 token 细分类型又在画 DFA 的阶段合并在了一起，DFA 如图 2.2.1 所示。在 DFA 中我们将关键字和标识符归为了一类，但在实际的词法分析器程序里这两类是分离的，每当识别到一个单词后会判断其是否属于关键字，是什么关键字然后将具体的类型赋值给 token。另外一个在 DFA 中没有体现出来的部分是“-”号是应该识别为负号还是减号的问题，比如在读入源代码段“a=-1”时词法分析器应该输出”a”，“=”，“-1”而不是”a”，“=”，“-”，“1”，但在读入源代码段“a=a-1”时就应该输出“a”，“=”，“a”，“-”，“1”，这一部分理论上是语法分析的部分，DFA 无法将两种情况区分开，需要借助前文语义才能区分开。经分析发现，只有在上一个 token 的类型是“ID”和“NUMBER”时“-”号才应该识别为减号，其他情况下都应该识别为负号，基于此我们在词法分析器里维护了一个名为“minusOrNegtiveFlag”的标志位，在对 token 类型赋值时更新该标志位，然后在遇到“-”时根据该标志位决定其为减号还是负号。

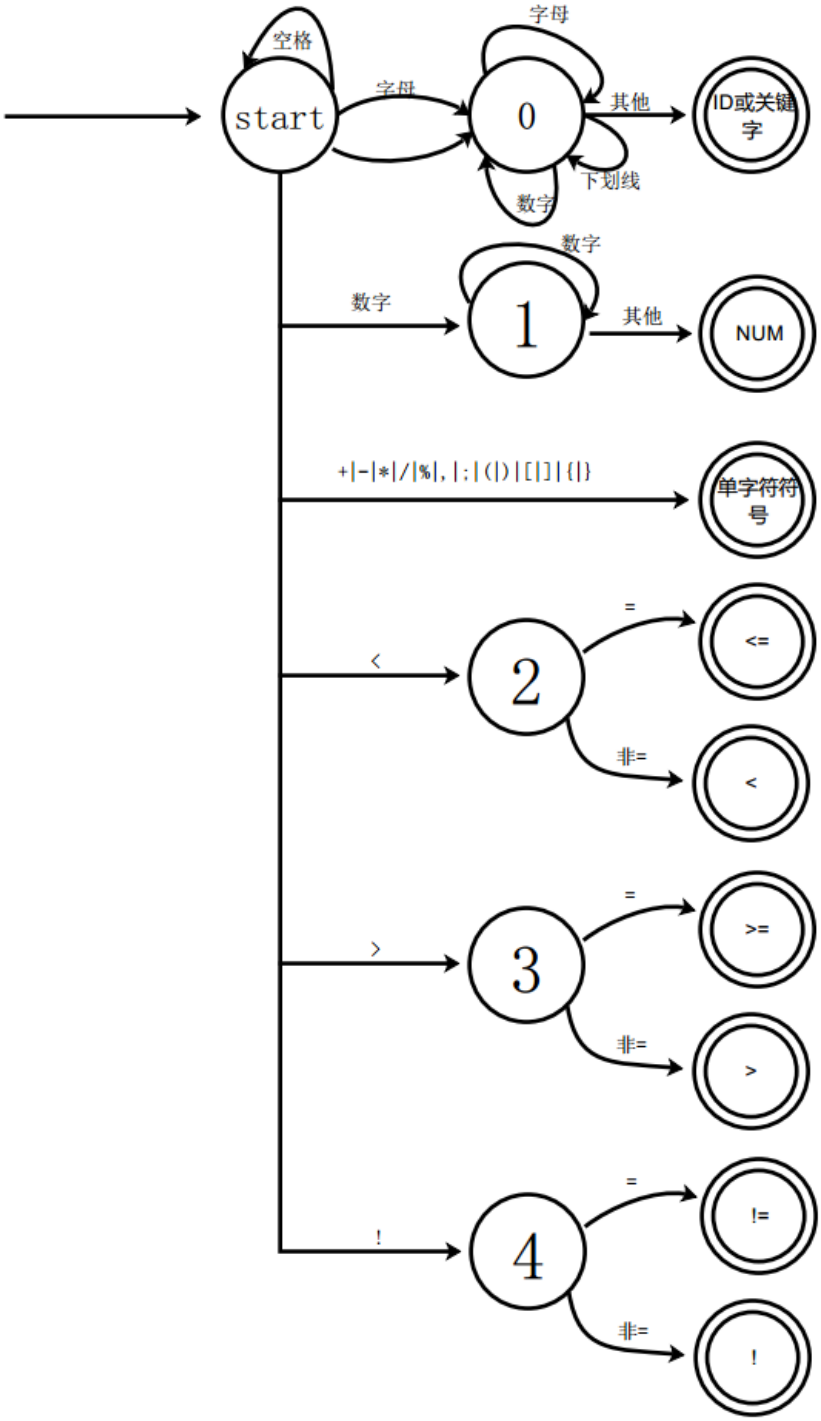


图 2.2.1

2.3 词法分析器的测试

3 递归下降法实现语法分析程序

4 虚拟机的设计与实现

4.1 虚拟机架构选择

虚拟机是借助操作系统对实际物理机器的一种模拟，本报告所述的虚拟机是指某种高级语言的虚拟机（例如 JAVA 语言的虚拟机 JVM，Python 语言的虚拟机 CPython，以及 Lua 语言的 Lua VM 等）。虚拟机的功能是对高级语言编译生成的字节码进行解释执行。需要实现以下几点功能：

- 取指令，指令来源于高级语言编译器的输出
- 译码，决定执行何种操作，以及从内存中取操作数
- 执行，指令译码后被虚拟机执行
- 存储计算结果

目前主流的虚拟机的实现方式有两种，一种是基于寄存器的虚拟机比如 Dalvik 以及 Lua5.0 的虚拟机，另一种是基于栈的寄存器比如 JAVA 的 JVM 以及 Python 的 CPython 虚拟机。其中基于寄存器的虚拟机架构中存储操作数的内存数据结构是 CPU 寄存器，而基于栈的虚拟机架构的操作数的存储数据结构是栈，基于寄存器的虚拟机架构通常可以实现更快的速度，但是其指令长度通常大于基于栈的虚拟机架构，鉴于本次课程设计实现的编译器支持的语法种类有限且并以速度为追求目标而是以理解编译器及虚拟机原理为主，故本课程设计选用基于栈的虚拟机架构，使虚拟机指令清晰易懂。

4.2 虚拟机指令详述

基于前述实现的 naïve C 的语言特性，我们设计了如下指令集：（为了叙述方便，以下内容将操作数栈简称为栈，将程序计数器称之为 PC）

- LOAD addr --将地址 addr 内存单元中的数据入栈
- IMM num --将立即数 num 入栈
- STO addr --将栈顶数据存入地址 addr 所对应的内存单元
- POP --栈顶数据出栈
- ADD --将栈顶和次栈顶数据出栈，并将其和入栈
- SUB --将栈顶和次栈顶数据出栈，并将次栈顶和栈顶数据的差入栈
- MULT --将栈顶和次栈顶数据出栈，并将其积入栈
- DIV --将栈顶和次栈顶数据出栈，并将次栈顶和栈顶数据的商入栈
- JMP label --将 PC 无条件跳转到 label 所对应的代码段地址
- JZ label --首先将栈顶数据出栈，并判定此数据的真假，为真就将 PC 跳转到 label 所对应的代码段地址，否则不跳
- EQ --将栈顶和次栈顶数据出栈，若次栈顶和栈顶数据相等则将真值入栈，否则将假值入栈
- NOTEQ --将栈顶和次栈顶数据出栈，若次栈顶和栈顶数据不等则将真值入栈，否则将假值入栈
- GT --将栈顶和次栈顶数据出栈，若次栈顶数据大于栈顶数据则将真值入栈，否则将假值入栈
- LES --将栈顶和次栈顶数据出栈，若次栈顶数据小于栈顶数据则将真值入栈，否则将假值入栈
- GE --将栈顶和次栈顶数据出栈，若次栈顶数据大于或等于栈顶数据则将真值入栈，否则将假值入栈
- LE --将栈顶和次栈顶数据出栈，若次栈顶数据小于或等于栈顶数据则将真值入栈，否则将假值入栈
- AND --将栈顶和次栈顶数据出栈，并将其相与的结果入栈
- OR --将栈顶和次栈顶数据出栈，并将其相或的结果入栈
- NOT --将栈顶数据出栈，若该数据为真则将假值入栈，否则将真值入栈
- IN --输入指令，用于从标准输入设备获取一个数据并将该数据入栈
- OUT --输出指令，用于向标准输出设备输出栈顶数据并将该数据出栈

该虚拟机的部分实现代码如图 4.2.1 所示，其中名为 code 的 vector 充当了代码区的角色，PC 是程序计数器，data 代表内存，calculateStack 是操作数栈。

```
else if (code[pc] == "STO")
{
        data[stoi(code[++pc])] = calculateStack.top();
}
else if (code[pc] == "POP")
{
        calculateStack.pop();
}
else if (code[pc] == "ADD")
{
        int tmp1 = calculateStack.top();
        calculateStack.pop();
        int tmp2 = calculateStack.top();
        calculateStack.pop();
        calculateStack.push(tmp1 + tmp2);
}
else if (code[pc] == "SUB")
{
        int tmp1 = calculateStack.top();
        calculateStack.pop();
        int tmp2 = calculateStack.top();
        calculateStack.pop();
        calculateStack.push(tmp2 - tmp1);
}
else if (code[pc] == "MULT")
{
        int tmp1 = calculateStack.top();
        calculateStack.pop();
        int tmp2 = calculateStack.top();
        calculateStack.pop();
        calculateStack.push(tmp1 * tmp2);
}
else if (code[pc] == "DIV")
{
        int tmp1 = calculateStack.top();
        calculateStack.pop();
        int tmp2 = calculateStack.top();
        calculateStack.pop();
        calculateStack.push( tmp2 / tmp1);
}
```

图 4.2.1

4.3 虚拟机测试

5 编辑器的实现

对于我们设计的 naive C 语言来说，编辑器并不是必要的，有了前述的编译器和虚拟机，naive C 语言编写的程序已经可以在计算机上运行，但是为了程序员能方便地使

用 naive C 进行程序设计我们设计了一款编辑器来实现 naive C 的语法高亮，自动补全，以及图形化地调用编译器和虚拟机运行编写的 naive C 程序。

5.1 界面设计和语法高亮显示

我们的编译器和虚拟机为了高效执行选择使用 C++进行开发，但在编辑器上执行速度并没有非常高的要求，所以我们采用了更加方便的 python 语言进行编辑器的开发。我们采用了 python 自带的 Tkinter 库进行 GUI 的开发。

如图 5.1.1 所示界面主要分为三个区域，菜单栏区，代码编辑区，以及自动补全提示区。代码区以 tab 形式组织，可以开启多个不同的 tab 编辑不同的源代码。菜单栏区包含 File, Edit， Run, Language 四个功能按钮，File 下含的子菜单功能主要与文件操作相关，比如开启新的 tab，打开文件，保存文件，以及文件另存为的功能。Edit 子菜单包括了 Undo 和 Redo 的功能，方便用户撤销错误的修改，同时为了方便操作，给这两个子菜单添加了快捷键的操作方式，“ctrl + z” 可以撤销更改，“ctrl + r”可以重做上一次的修改。Run 子菜单包含了调用编译器编译源代码的按钮和调用虚拟机进行执行程序的按钮，同时可以利用快捷键进行编译和运行，“F7”是编译快捷键，“F5”是运行快捷键。而 Language 菜单栏下包含了源代码语言选择的子菜单，目前我们的编辑器支持 naive C 高级语言和 naive 汇编低级语言，在 Language 菜单栏下可以切换这两种语言。

为了方便用户使用该编辑器进行 naive C 语言的程序设计，我们设计了语法高亮功能（见图 5.1.1），使代码看起来更清晰美观。基于 naive C 的语言特性，我们设计了五类颜色来高亮不同的语法元素

- ●函数名及系统函数（main，print 等） --用红色高亮显示
- ●数据类型（int 等） --用兰色高亮显示
- ●循环，条件及分支关键字（if，while 等） --用绿色高亮显示
- ●运算符号（<,*等） --用橘黄色高亮显示
- ●数字 --用紫色高亮显示

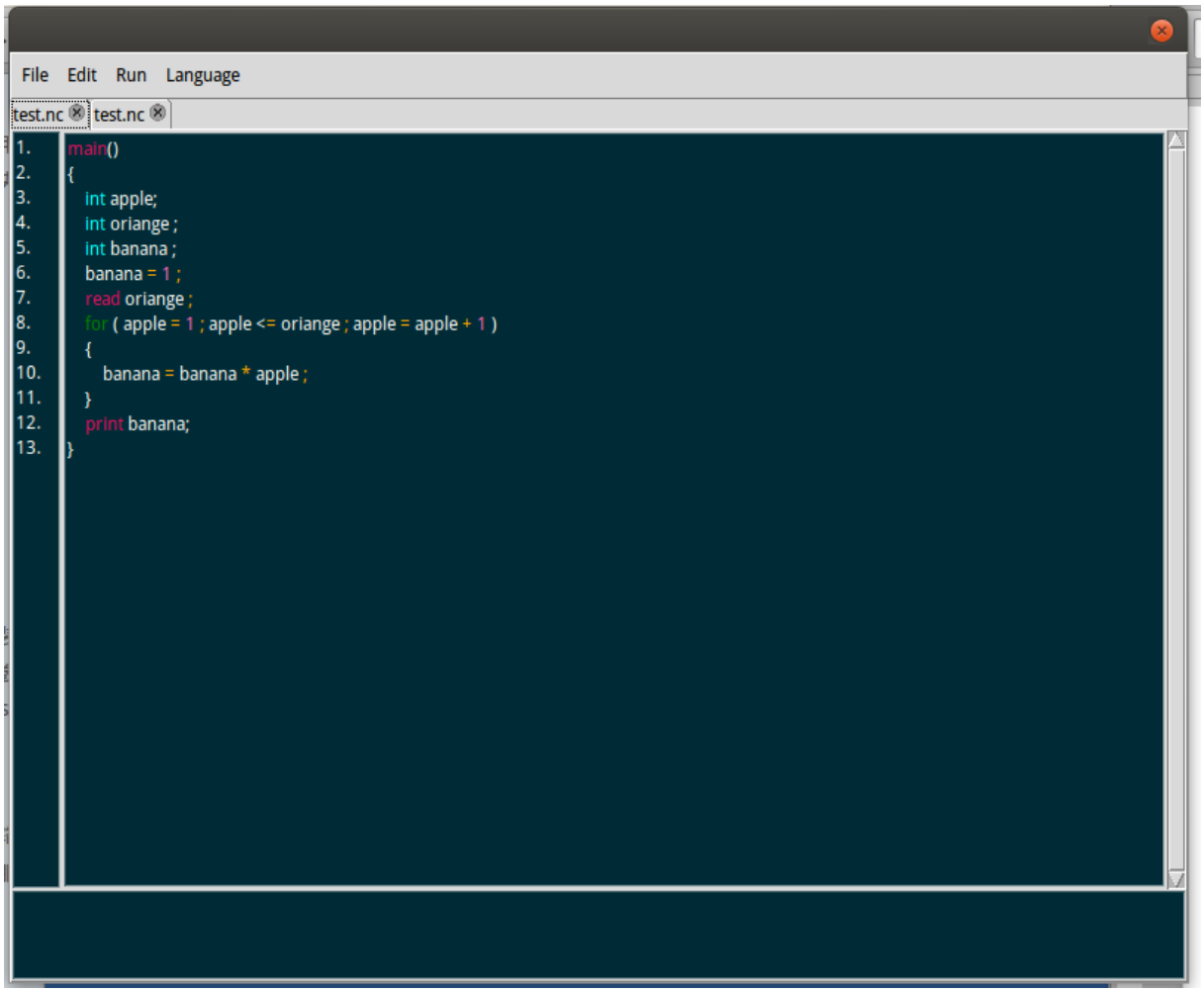


图 5.1.1

5.2 基于字典树的自动补全算法

尽管编辑器并不需要非常高的执行效率，但是自动补全一定要能非常快速地检索出用户可能想要输入的单词，如果自动补全的检索效率低于用户的打字速度则此自动补全算法则没有了存在的意义。为了实现快速自动补全，编辑器经常食用一种字典树的数据结构来实现字符串快速查找以及快速前缀匹配。

字典树又称 trie 树，是一种哈希树的变种，它的优点是：最大限度地减少无谓的字符串比较，查询效率比哈希表高。Trie 是一颗存储多个字符串的树。相邻节点间的边代表一个字符，这样树的每条分支代表一则子串，而树的叶节点则代表完整的字符串。和普通树不同的地方是，相同的字符串前缀共享同一条分支。例如，给出一组单词 inn, int, at, age, adv, ant, 我们可以得到如图 5.2.1 的字典树。

在我们的 naïve C 语言的编辑器中，我们将源代码的关键字以及虚拟机的汇编指令的关键字组织为字典树形式，在用户编写源代码时，根据前缀匹配预存的关键字，在下方显示出用户可能想要输入的关键字，并且用户可以使用下方向键选择自动补全的单词完成输入，使用效果如图 5.2.2 所示。

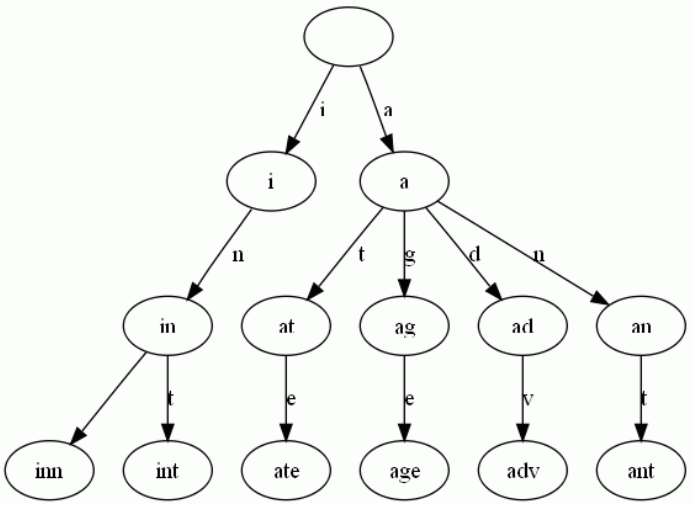


图 5.2.1

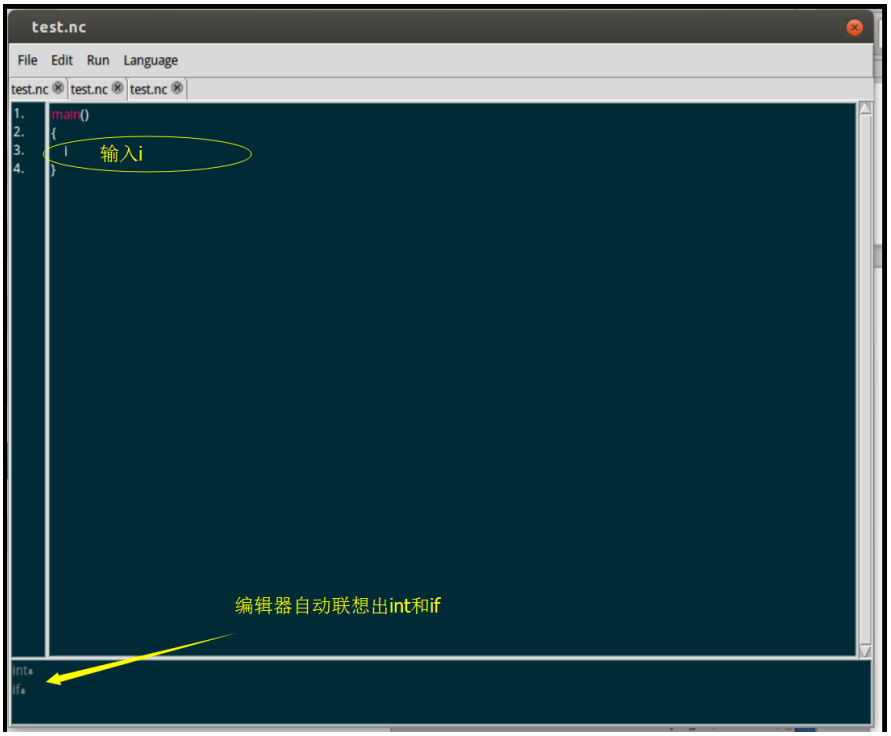


图 5.2.2

5.3 利用进程调用实现编辑器和编译器以及虚拟机的对接

为了使用户在编辑器中编写完 naive C 源码后能够方便地调用我们的编译器和编辑器，我们在编辑器中设计了调用的按钮，在编辑器的实现中，我们利用操作系统提供的系统调用函数来实现编辑器的进程调用编译器和虚拟机的进程，效果如图 5.3.1 所示。

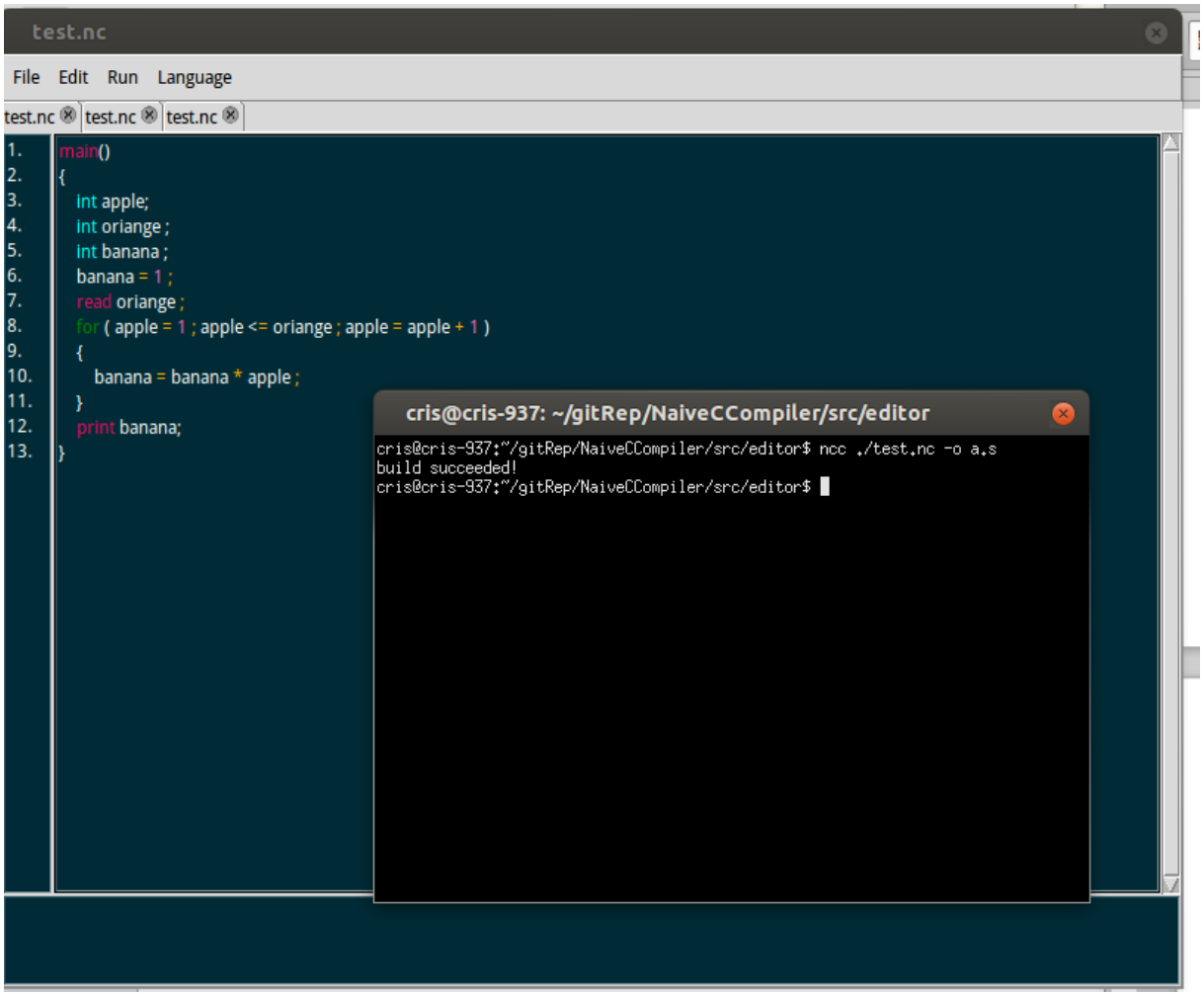


图 5.3.1

感　　想

俗话说知易行难，对于计算机学科的基础工具编译器来说更是这样，理论性的东西在上学期的编译原理课上都能学得非常明白，然而当开始真正动手做的时候又会发现困难重重，比如存储分配，比如虚拟机指令设计，但是万事只是开头难，一旦真正开始投入进去开始去思考去实现，一切困难就都是纸老虎了。

对于一个编译器这样的课程设计而言，我觉得最重要的是前期设计好所要实现的语言特性，要根据剩余的时间来合理的舍弃一些语言特性，比如我们刚开始的时候设计的语言也行比较丰富，支持函数调用，递归调用，字符串处理等导致系统过于复杂，而留给我们的时间又不太多，因此迟迟没有实质性的进展。后来我们决定一步一步地走，先实现基本语言特性，于是我们去掉了函数调用的功能，又重新规划了一下系统各个部分之间的接口设计，接下来就开发地非常顺利了。在实现了基本语言特性后我们又实现了带自动补全和语法高亮功能的编辑器来配合使用。

做完这个课程设计最大的收获是把编译原理理论的东西和具体实践联系起来了。这次课程设计中，我们必须把许多抽象的原理用代码直观地实现，实现的过程中更加深了我们对原理的理解。这正是理论指导实践，实践反过来促进理解理论。

另外一个很大的收获是，通过这次课程设计我和我的队友祝磊实践了以 git 版本控制工具进行协作开发的方式。git 是一个分布式的版本控制和分支管理工具，可以和最大的开源代码托管平台的 github 无缝连接。我们将我们的编译器代码开源并且托管于 github 上，项目地址：https://github.com/crisb-DUT/NaiveCCompiler 。协作开发的一个难点是如何恰当地分工合作从而尽可能地降低两个人的沟通成本以及重复代码，我们为了使合作接口尽可能清晰，选择了这样的分工，祝磊负责语法和语义分析以及中间代码生成，我负责词法分析，虚拟机以及编辑器，这样两部分程序的接口就十分明确，我的编辑器用来将 naïve C 源代码以文件的方式传给我的词法分析器，然后我的词法分析器提供一个 next()方法供他的语法分析器调用来获得 token，最后他的语义分析程序将生成的中间代码（伪汇编）传递给我的虚拟机，同时虚拟机提供调用接口 run()来供编辑器以图形化的方式调用。

最后，感谢一起通力合作的队友祝磊，感谢老师不厌其烦地和我们交流编译器应该如何设计。

附录

课程设计组员完成模块如下：

我：　词法分析器，虚拟机，编辑器

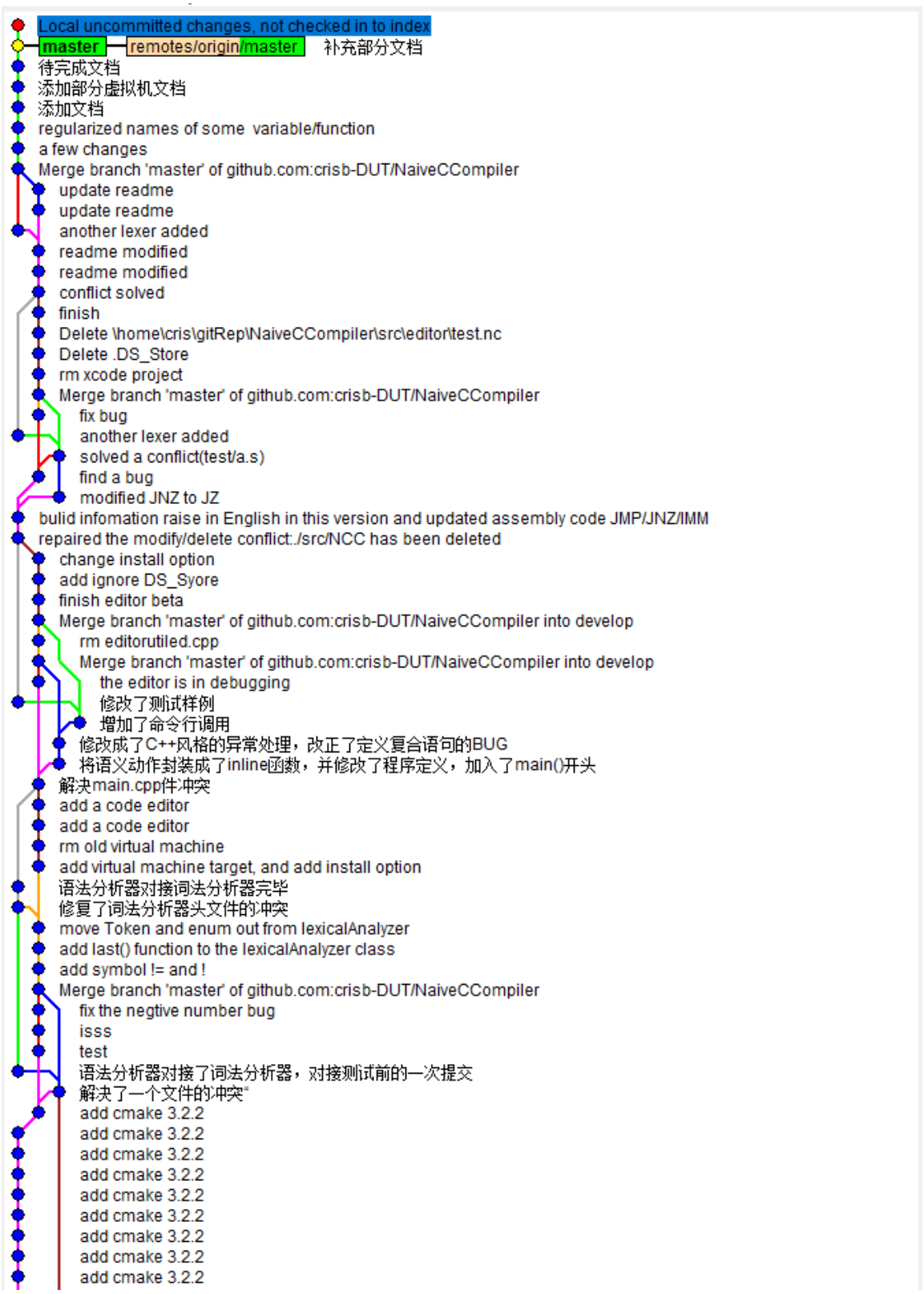
祝磊：语法和语义分析，中间代码生成，词法分析器

（注：我们两个人分别做了两个词法分析器，但最终编译器中使用的是我的词法分析器）

各组员的代码量如下图所示：

可见我们两个的代码量都在 1600-1900 行之间，贡献比例大致五五开。

另我们的项目开发流程如下图所示：


Local uncommitted changes, not checked in to index
master
remotes/origin/master
补充部分文档
待完成文档
添加部分虚拟机文档
添加文档
regularized names of some variable/function
a few changes
Merge branch 'master' of github.com:crisb-DUT/NaiveCCompiler
update readme
update readme
another lexer added
readme modified
readme modified
conflict solved
finish
Delete \home\cris\gitRep\NaiveCCompiler\src\editor\test.nc
Delete .DS_Store
rm xcode project
Merge branch 'master' of github.com:crisb-DUT/NaiveCCompiler
fix bug
another lexer added
solved a conflict(test/a.s)
find a bug
modified JNZ to JZ
bulid infomation raise in English in this version and updated assembly code JMP/JNZ/IMM
repaired the modify/delete conflict:./src/NCC has been deleted
change install option
add ignore DS_Syore
finish editor beta
Merge branch 'master' of github.com:crisb-DUT/NaiveCCompiler into develop
rm editorutiled.cpp
Merge branch 'master' of github.com:crisb-DUT/NaiveCCompiler into develop
the editor is in debugging
修改了测试样例
增加了命令行调用
修改成了C++风格的异常处理，改正了定义复合语句的BUG
将语义动作封装成了inline函数，并修改了程序定义，加入了main()开头
解决main.cpp件冲突
add a code editor
add a code editor
rm old virtual machine
add virtual machine target, and add install option
语法分析器对接词法分析器完毕
修复了词法分析器头文件的冲突
move Token and enum out from lexicalAnalyzer
add last() function to the lexicalAnalyzer class
add symbol != and !
Merge branch 'master' of github.com:crisb-DUT/NaiveCCompiler
fix the negtive number bug
isss
test
语法分析器对接了词法分析器，对接测试前的一次提交
解决了一个文件的冲突"
add cmake 3.2.2
add cmake 3.2.2
add cmake 3.2.2
add cmake 3.2.2
add cmake 3.2.2
add cmake 3.2.2
add cmake 3.2.2
add cmake 3.2.2
add cmake 3.2.2