**中国矿业大学计算机学院**

**2018 级本科生课程设计报告**

课程名称 系统软件开发实践

实验名称 Bison理论与练习实验2

报告时间 2021年3月13日

学生姓名 丁旭行

学 号 14184501

专 业 计算机科学与技术

任课教师 徐东红

**目 录**

[1 实验目的 0](#_Toc66636211)

[2 实验内容 0](#_Toc66636212)

[3 实验要求 1](#_Toc66636213)

[4 Bison解决移进-规约冲突的原理 1](#_Toc66636214)

[5 Lex源程序结构描述 2](#_Toc66636215)

[6 Bison源程序结构描述 5](#_Toc66636216)

[7 实验步骤 8](#_Toc66636217)

[7.1 Windows环境 8](#_Toc66636218)

[7.2 Linux环境 16](#_Toc66636219)

[8 分析输出结果 22](#_Toc66636220)

[9 实验总结 22](#_Toc66636221)

# 1 实验目的

利用附录提供的C语言文法的相关参考资料，利用Yacc/Bison编写一个C语言分析器。

# 2 实验内容

利用语法分析器生成工具Bison编写一个C语言的语法分析程序，与词法分析器结合， 能够根据语言的上下文无关文法，识别输入的单词序列是否文法的句子。

# 3 实验要求

(1) 阅读Flex源文件input.lex、Bison源文件cgrammar-new.y；

(2) 实现C语言的语法分析功能，最后上机调试；

(3) 生成语法分析程序2\_2.exe，以给定的测试文件test.c作为输入，输出运行结果到输出文件out.txt中。

# 4 Bison解决移进-规约冲突的原理

当一个输入字符串有两种可能的分析时，而且其中一个分析完成了一个规则(归约选项)，而另一个却没有(移进选项)，此时对于移进或者归约都是合法的，这种情况就被称为移进-归约冲突(shift-reduce conflict)。Bison的设计是，用移进来解决冲突，除非有操作符优先级声明的指令。

冲突的存在是因为文法有二义性，而Bison通过声明文法其中的终结符的优先级和结合规则来指定何时移进何时归约。考虑下面的二义文法片断(其二义性体现在'1 – 2 \* 3'可以用两种不同的方式进行分析)：

expr:

expr - expr

| expr \* expr

| - expr

;

假定分析器已经看到了终结符'1'，'-'和'2'；那么应该对它们归约到减法运算 规则吗？这取决于下一个终结符。当然，若下一个终结符是')'，就必须归约；此时移进是非法的，因为没有任何规则可以对序列'- 2 )'进行归约，也没有以这个序列开始的什么东西。但是如果下一个终结符是'\*'或者'<'，那么就需要做一个选择：移进或者归约，都可以让分析得以完成，但是却有不同的结果。

为了决定Bison应该怎么做，必须考虑这两个结果。若下一个终结符即操作符op被移进，那么必然是op首先做归约，然后才有机会让前面的减法操作符做归约。其结果就是有效的'1 – (2 op 3)'。另一方面，若在移进op之前先对减法做归约，那结果就是'(1 – 2) op 3'。很显然，这里移进或者规约的选择取决于减法操作符'-'与下一个操作符op之间的优先级：若op是乘法操作符'\*'，那么就选择移进；若是关系运算符'<'则应该选择规约。

那么诸如'1 – 2 – 5'这样的输入又如何呢？是应该作为'(1 – 2) – 5'还是应该作为'1 – (2 – 5)'？对于大多数的操作符，我们倾向于前一种形式，称作左关联(left association)。后一种形式称作右关联(right association)，对于赋值操作符来说是比较理想的。当堆栈中已经有'1 – 2'且预读终结符是'-'，此时分析器选择移进还是归约与选择左关联还是右关联是一回事：移进将会进行右关联，否则左关联。

Bison允许通过声明%left和%right来指定操作符优先级。每个这样的声明都包含一列终结符，这些终结符都是操作符，它们的优先级和关联性都被声明了。%left声明让所有这些操作符左关联，而%right声明让它们右关联。第三种方案是%noassoc，表示不可结合(即它定义的终结符不能连续出现：例如文法中不允许出现形如’a < b < c’的句子，则<就是不可结合的)。注意：产生式的优先级和结合性与产生式右部最右面的终结符(非终结符不用考虑)的优先级和结合性相同,当两个产生式优先级一致时，可以用%prec强制定义产生式的优先级和结合性。

# 5 Lex源程序结构描述

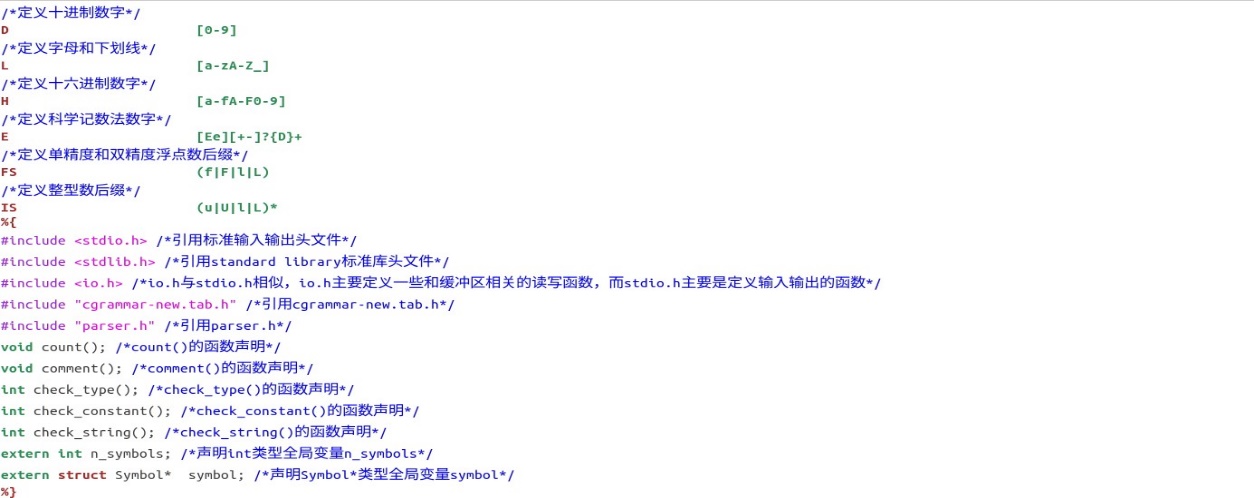


图1 Lex定义段



图2 Lex规则段1

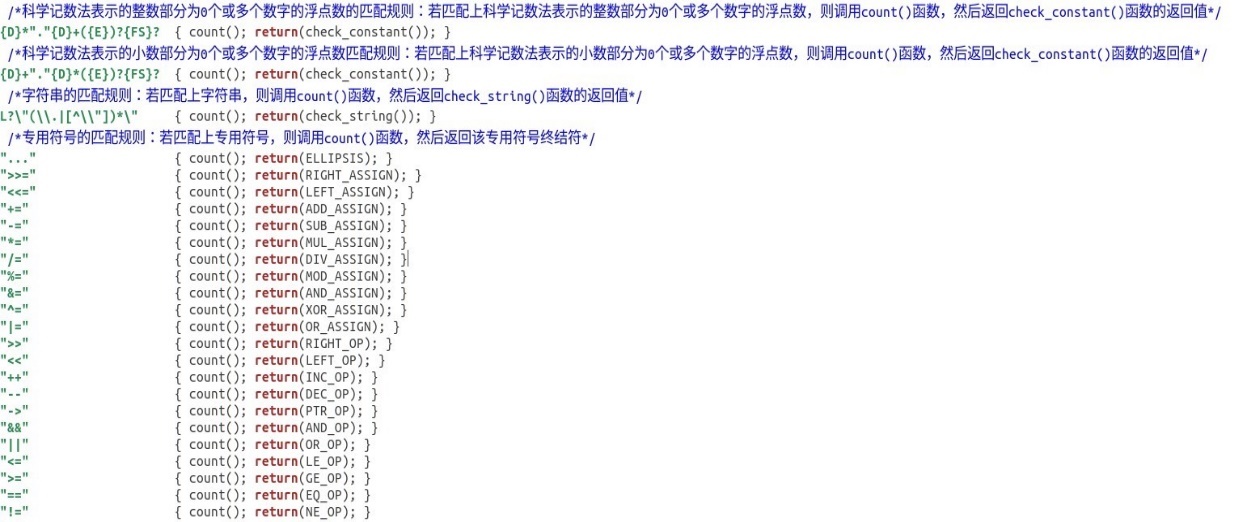


图3 Lex规则段2



图4 Lex规则段3

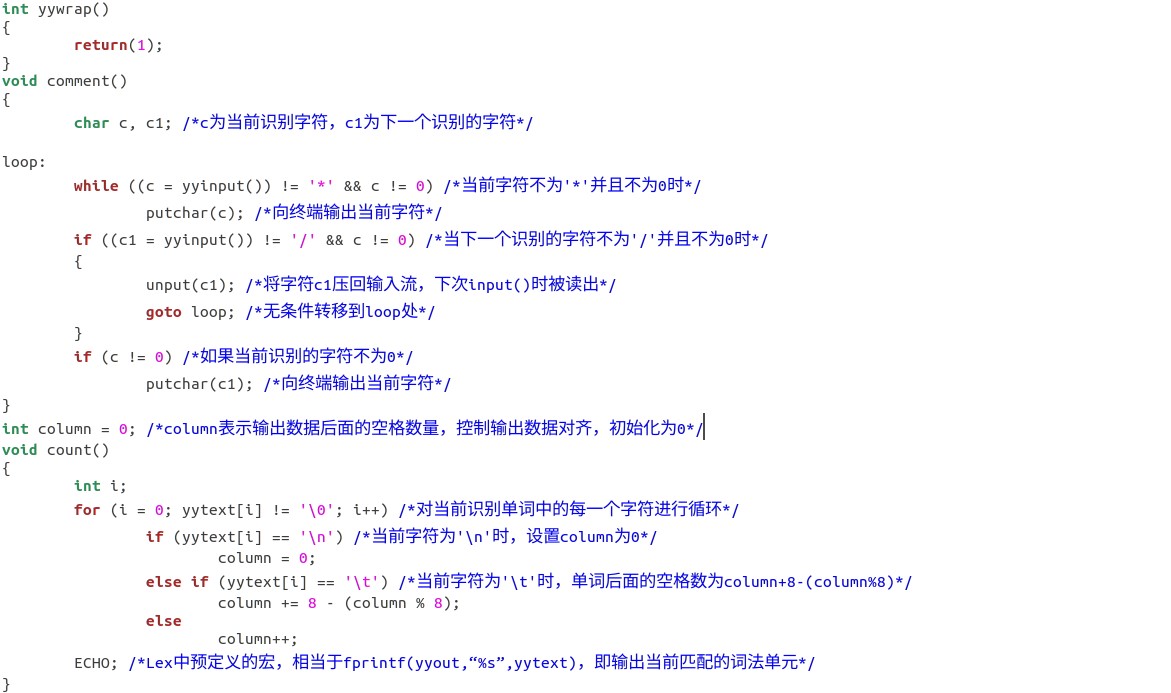


图5 Lex用户代码段1



图6 Lex用户代码段2

# 6 Bison源程序结构描述

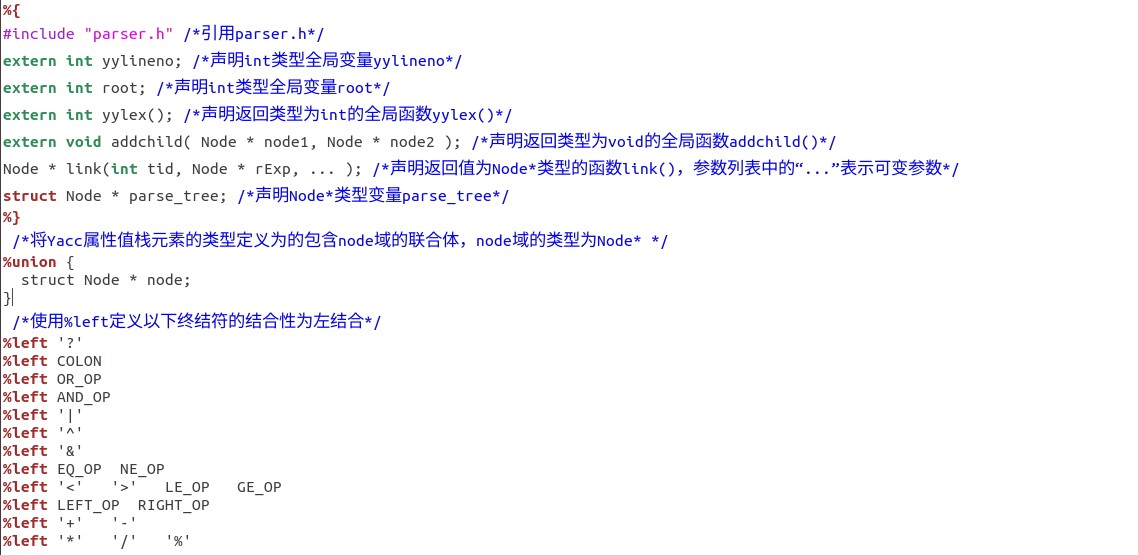


图7 Bison定义段1

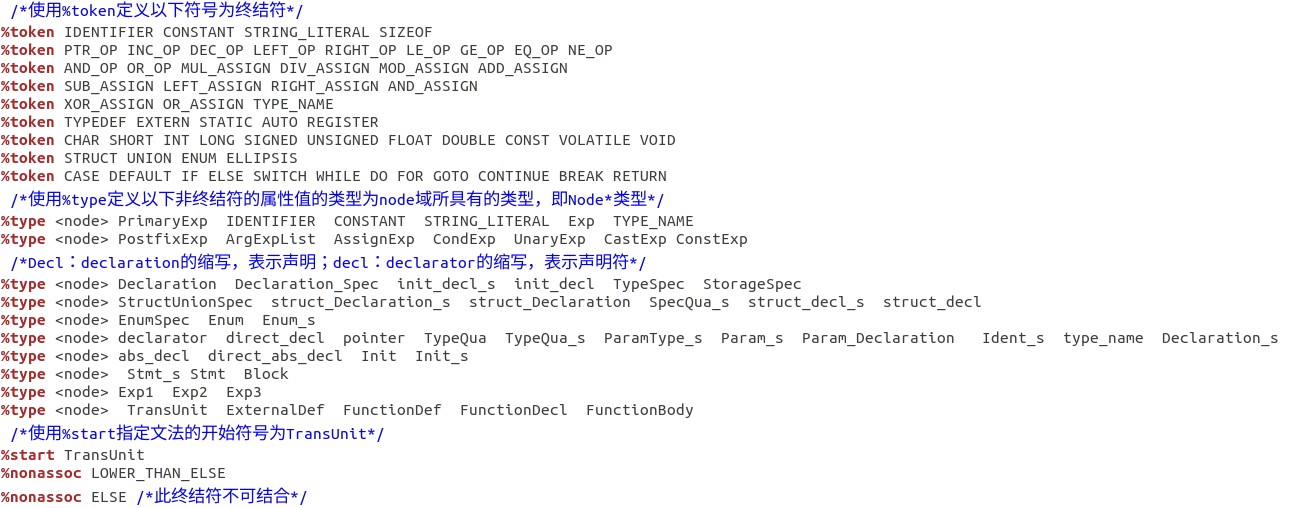


图8 Bison定义段2

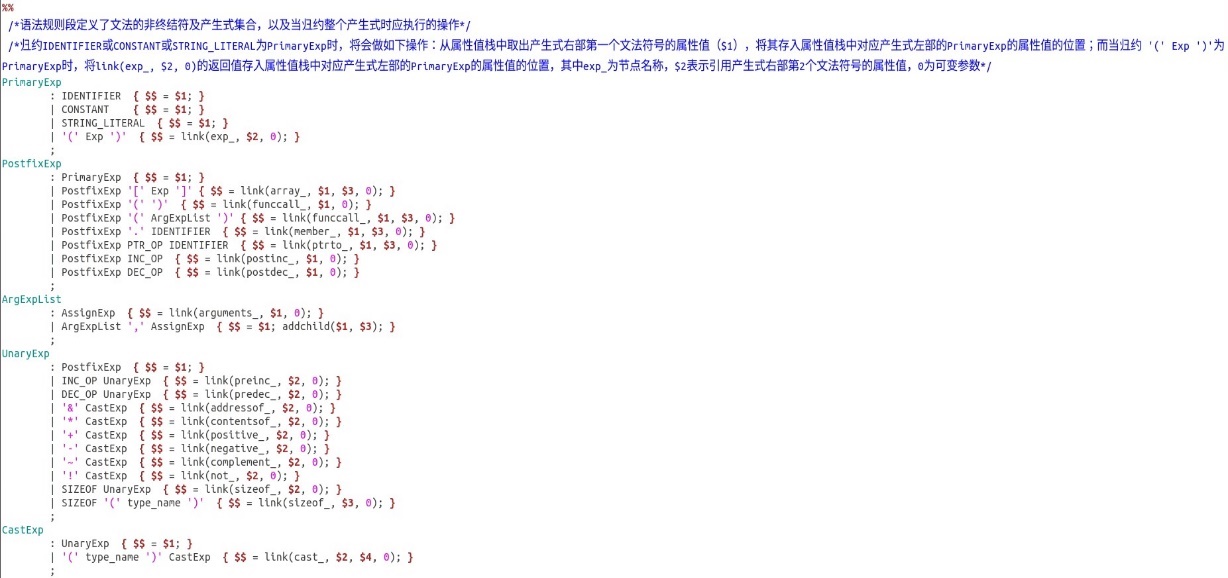


图9 Bison规则段1

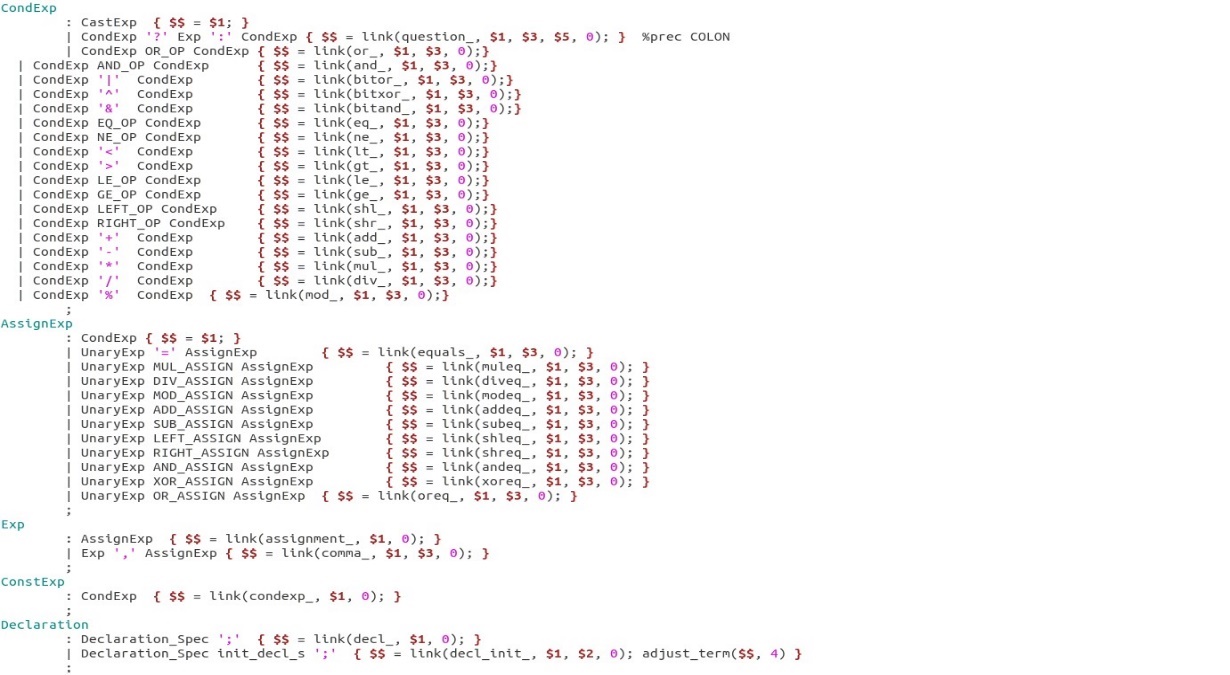


图10 Bison规则段2

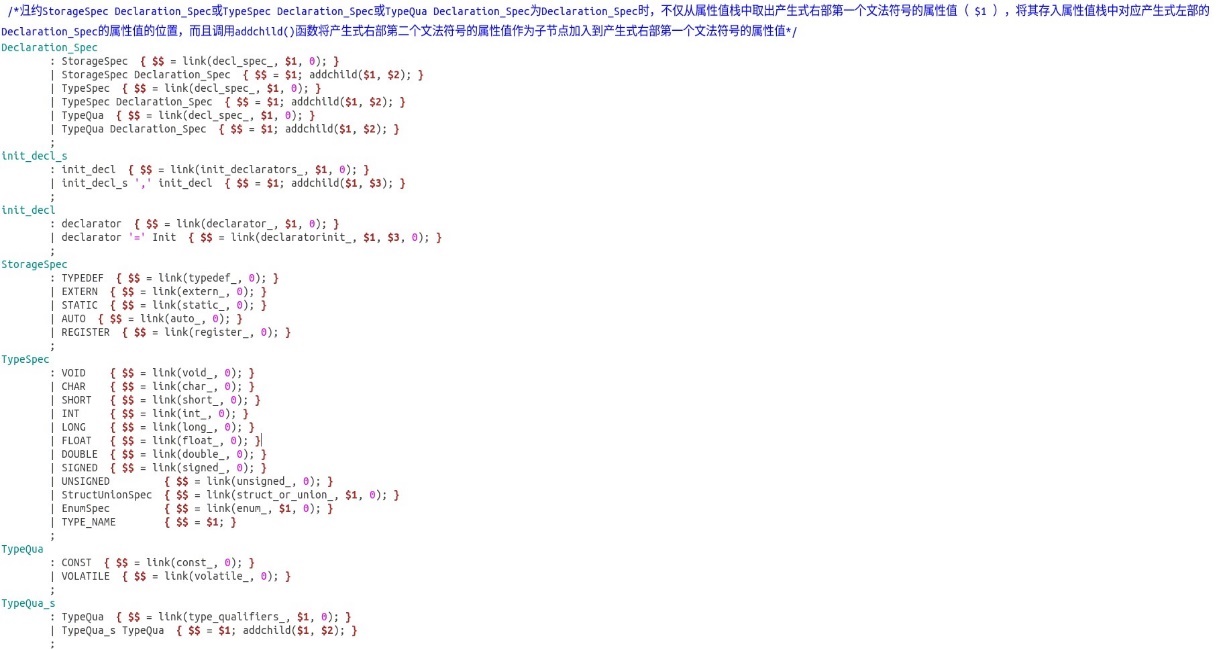


图11 Bison规则段3

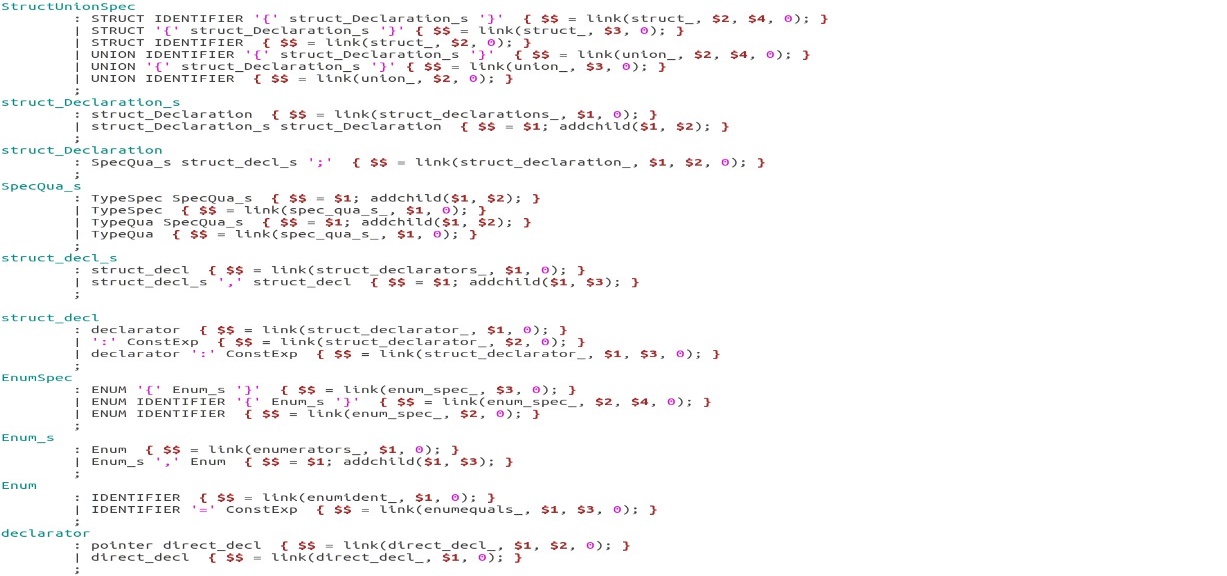


图12 Bison规则段4

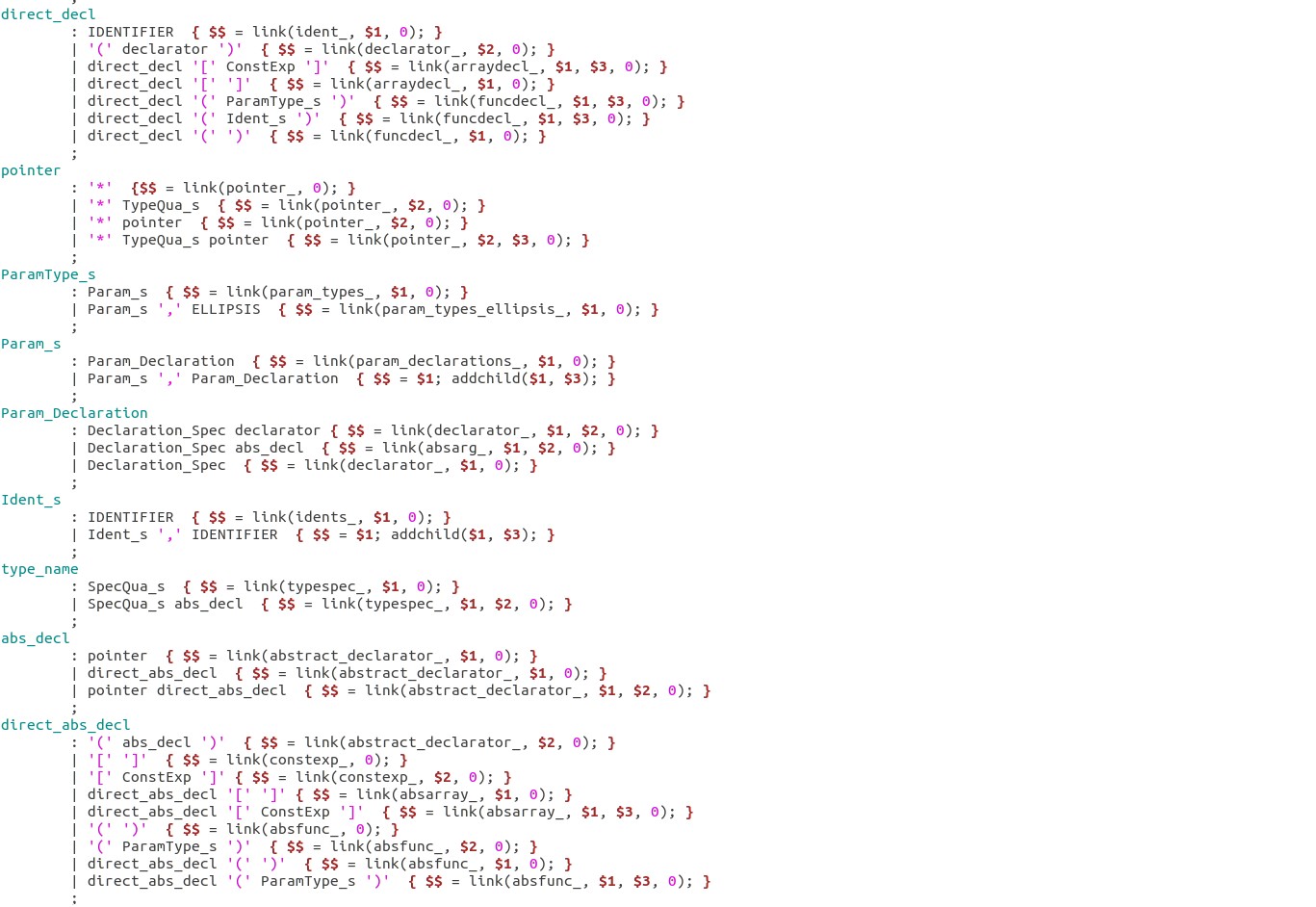


图13 Bison规则段5

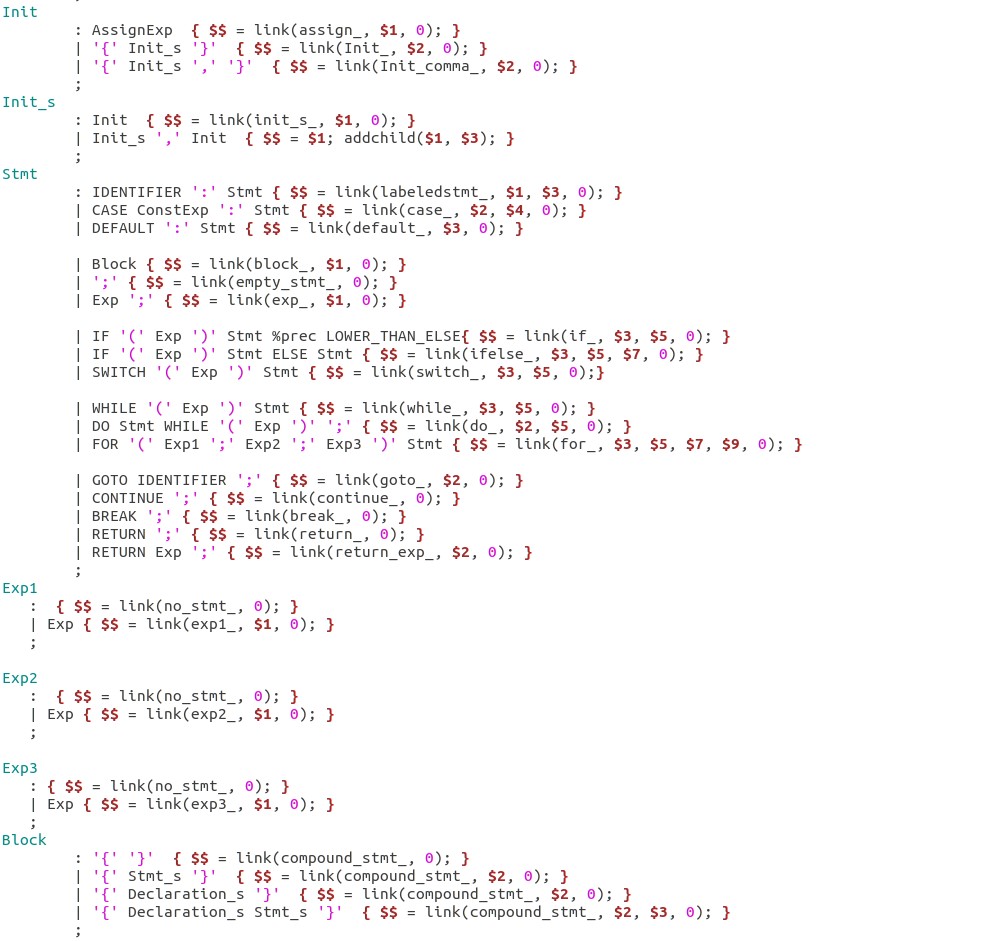


图14 Bison规则段6

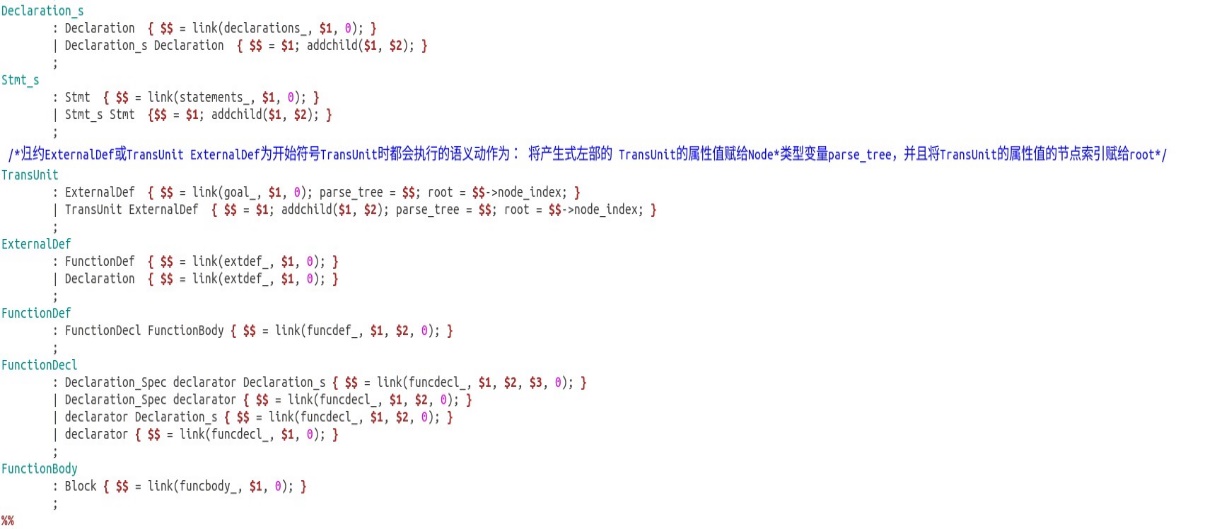


图15 Bison规则段7

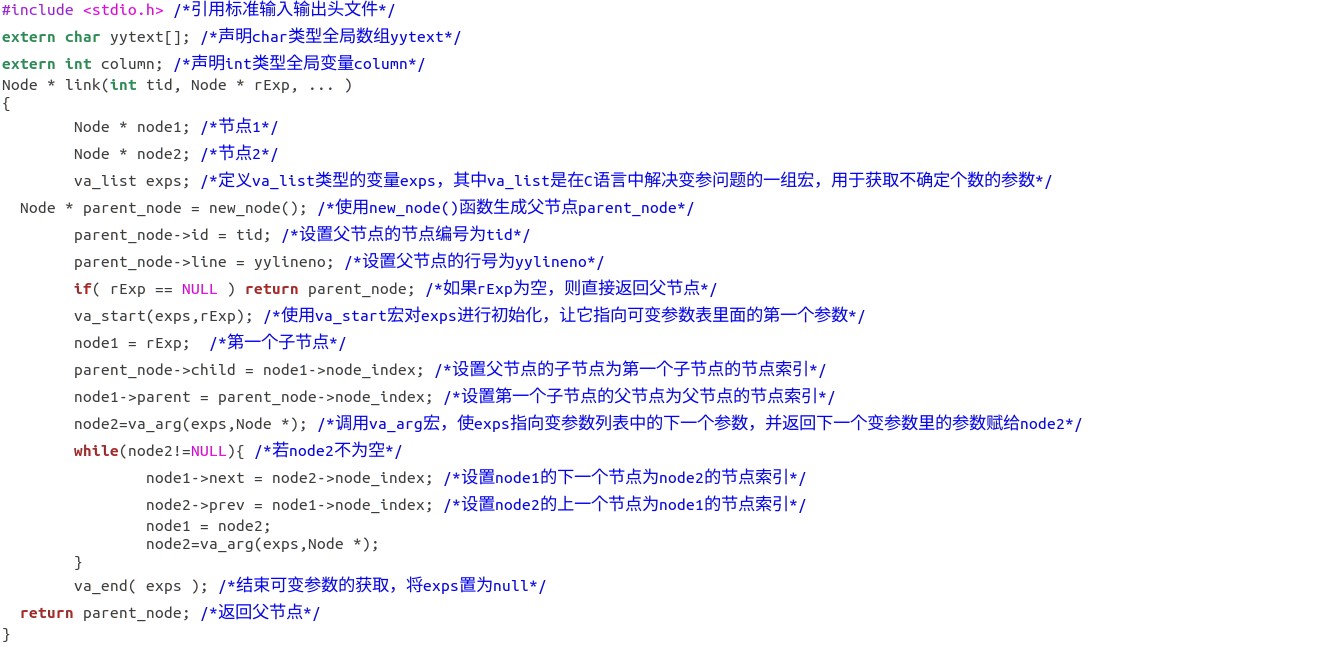


图16 Bison用户代码段

# 7 实验步骤

## 7.1 Windows环境

(1) 打开Developer Command Prompt for VS 2019命令窗口，依次输入e:、cd E:\Flex\GnuWin32\bin进入安装目录下，再依次输入flex input.lex、bison -d cgrammar-new.y，生成lex.yy.c、cgrammar-new.tab.h、cgrammar-new.tab.c文件，但发现在执行命令bison -d cgrammar-new.y后，Bison提示cgrammar-new.y文件中出现了移进-规约冲突。

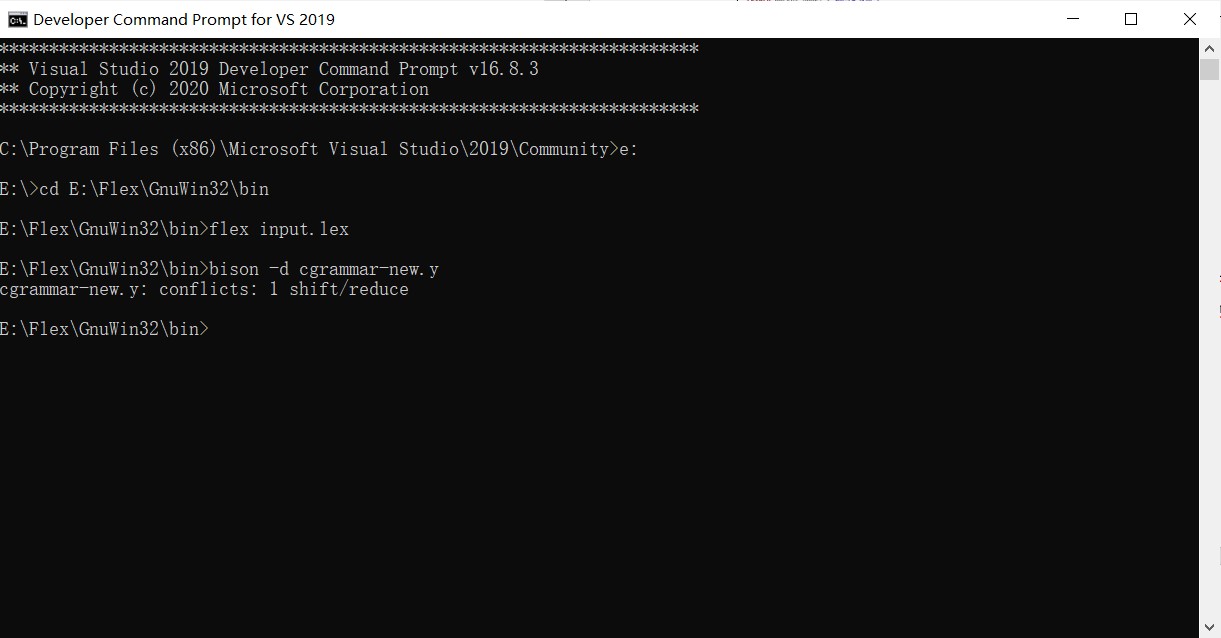


图17 输入命令及结果

(2) 以Bison的"-v"选项生成状态机描述文件cgrammar-new.output，即执行命令：bison -v cgrammar-new.y，查看生成的cgrammar-new.output文件具体内容。

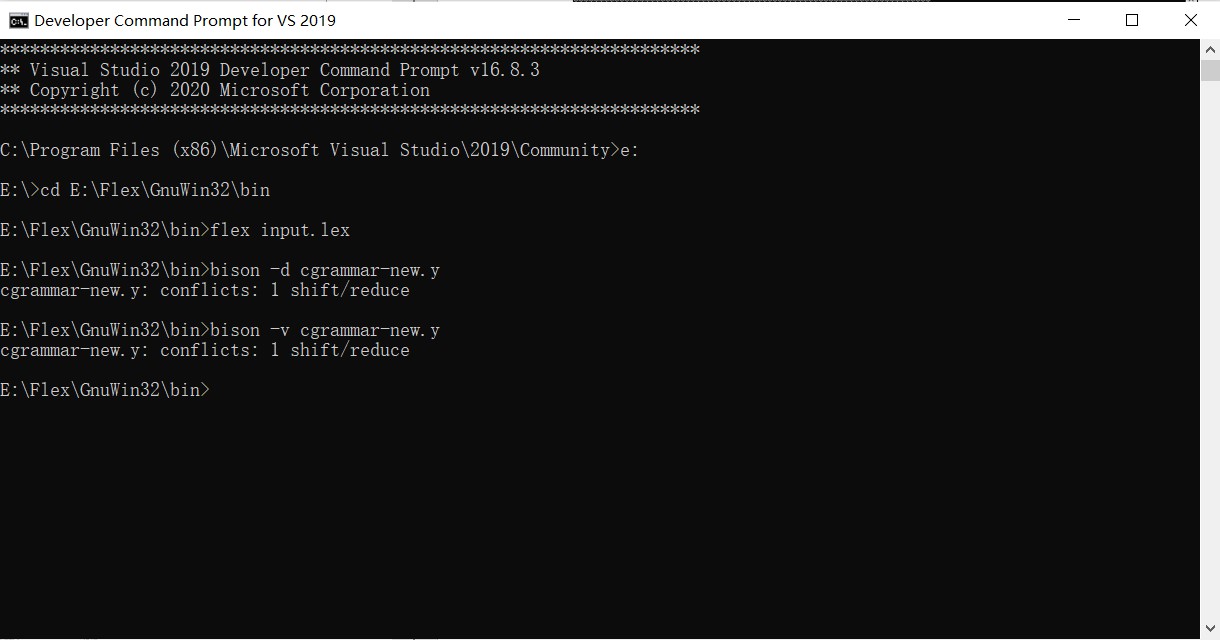


图18 输入命令

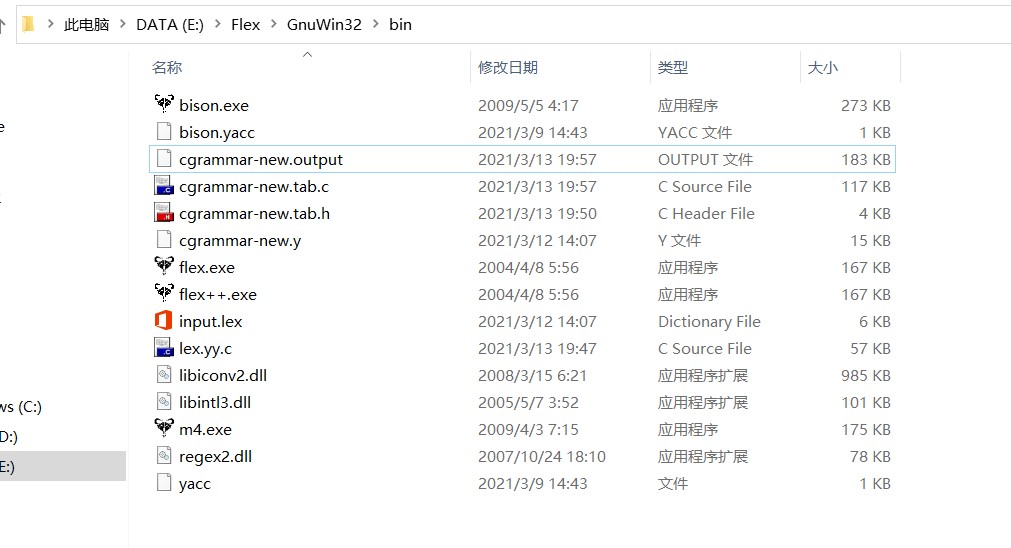


图19 生成文件



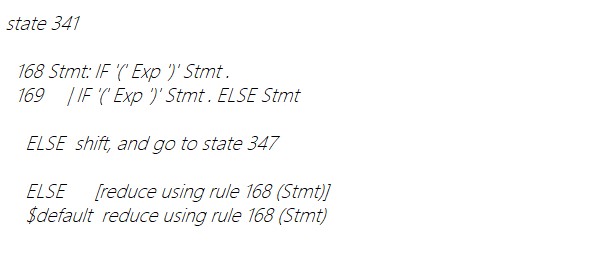


图20 查看文件内容

(3) 查看文件内容后发现产生if-else的二义性语句，即当ELSE终结符读入后作为一个预读终结符时，堆栈中的内容(假设输入是合法的)正好可以归约到第一条规则上。但是把它移进堆栈也是合理的，因为那样根据第二条规则就会导致最后的归约。按照已有的惯例通常把else子句连接到最临近的if语句，因而要让Bison选择移进而不是归约。

(4) 为了解决冲突，需要在cgrammar-new.y文件中的头部加入%nonassoc LOWER\_THAN\_ELSE、%nonassoc ELSE的说明，并在对应产生式规则中加入%prec LOWER\_THAN\_ELSE。

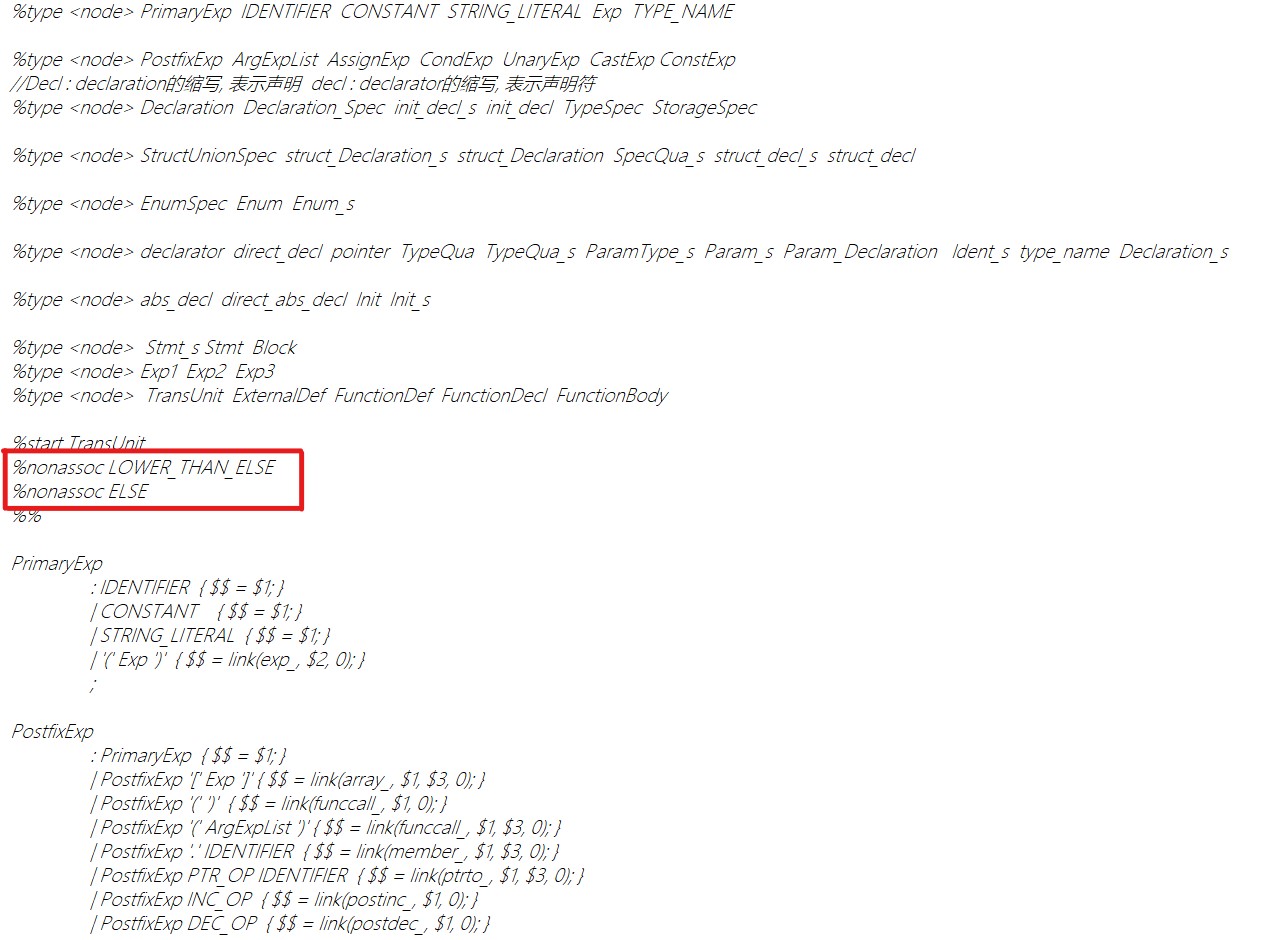


图21 修改文件内容1

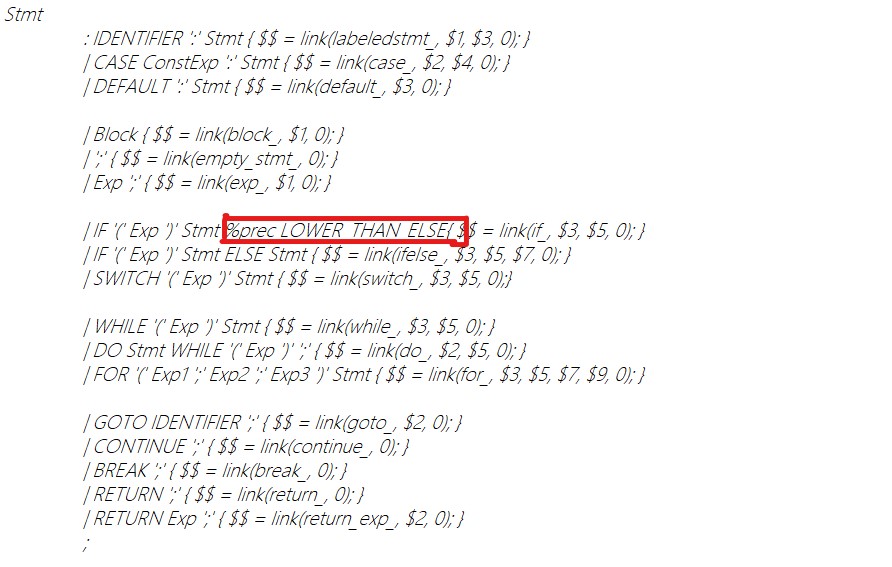


图22 修改文件内容2

(5) 重新依次输入flex input.lex、bison -d cgrammar-new.y，生成lex.yy.c、cgrammar-new.tab.h、cgrammar-new.tab.c文件，发现没有冲突提示。再输入命令cl lex.yy.c cgrammar-new.tab.c main.c parser.c -o”2\_2.exe”使用cl.exe编译器进行编译生成2\_2.exe文件。



图23 输入命令和结果

(6) 从编译结果可以看到，使用cl.exe编译后出现了错误。为了使其能顺利编译，需要修改lex.yy.cc文件。将lex.yy.c中从1634行开始的“#ifdef cplusplus static int yyinput() #else static int input() #endif”改成“static int yyinput()”，并且在2159行处添加声明定义”int yylineno;”。

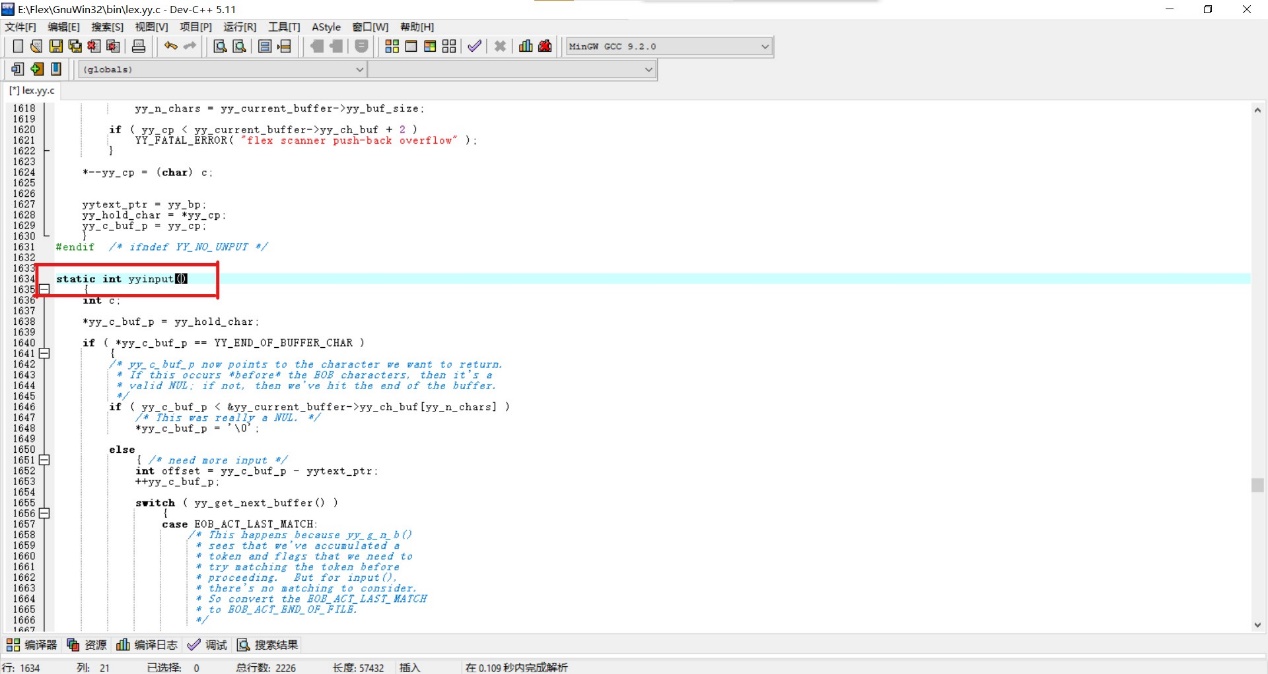


图24 修改文件内容1



图25 修改文件内容2

(7) 修改后重新编译，又出现了错误提示。因此需要继续对lex.yy.c进行修改，将lex.yy.c中从行开始的“#ifdef cplusplus return yyinput(); #else return input(); #endif”修改为“return yyinput();”。

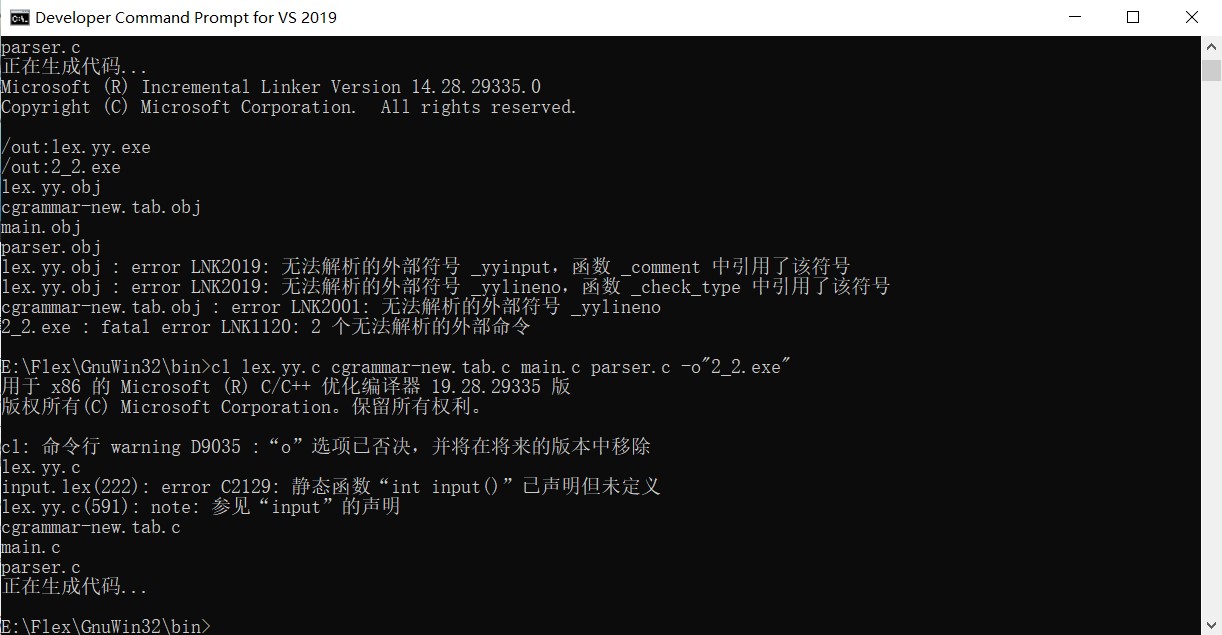


图26 重新编译结果

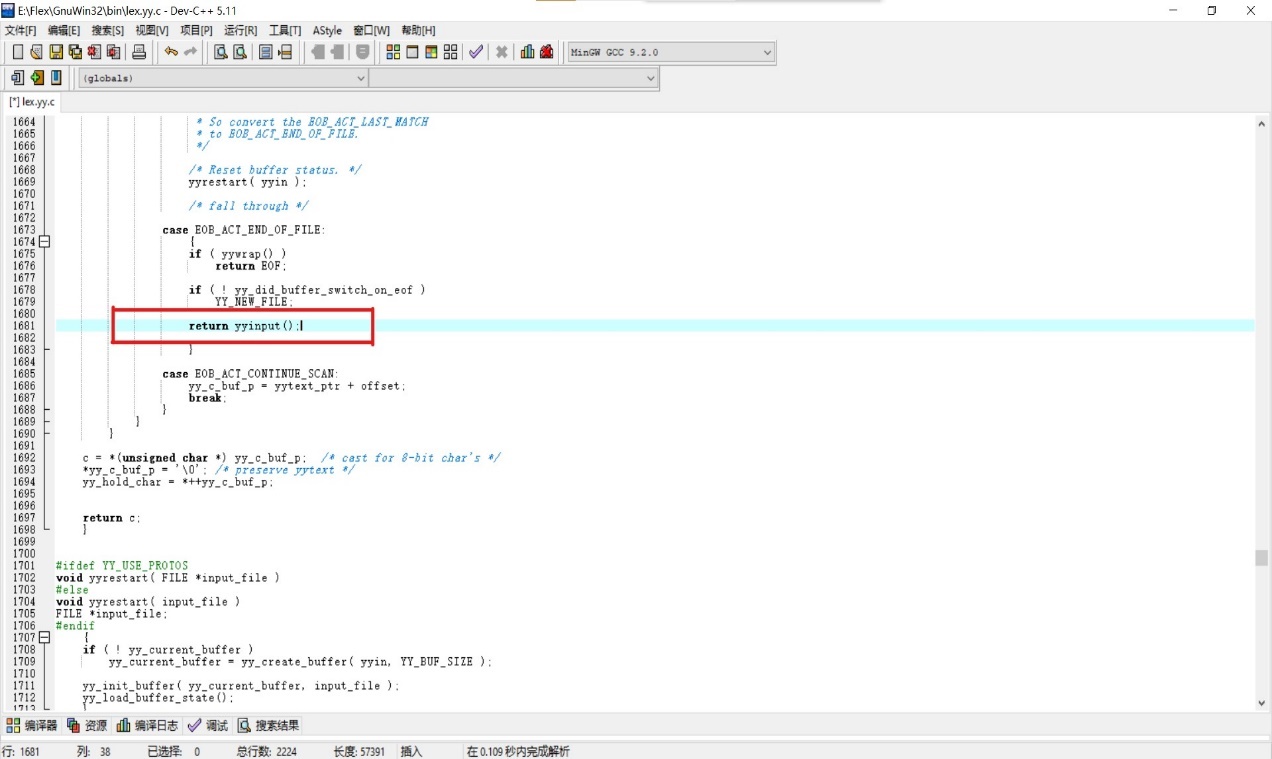
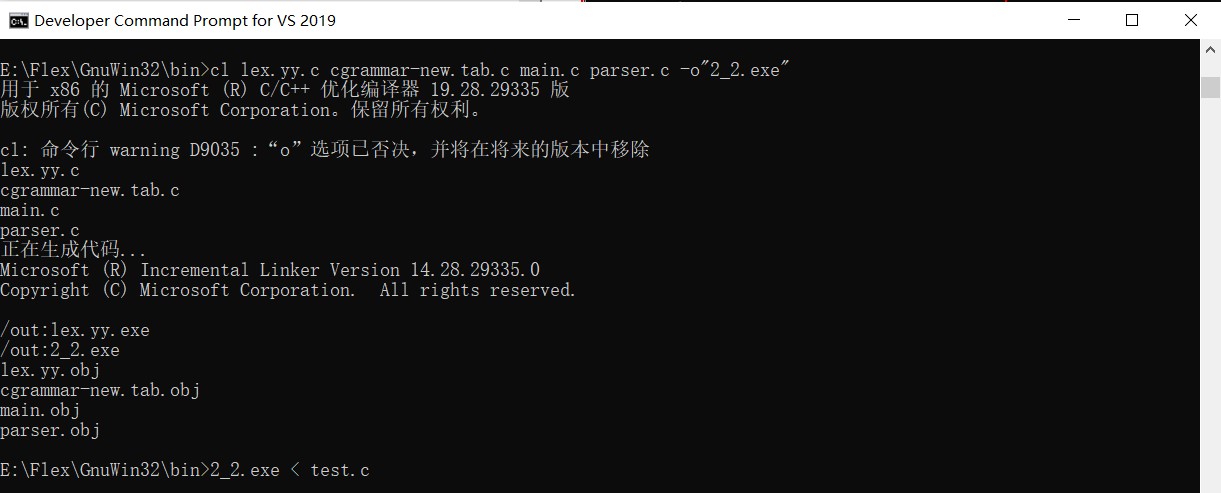


图27 再次修改文件内容

(8) 修改完成后，再次进行编译，可以发现编译通过。最后输入命令2\_2.exe < test.c对源文件test.c进行分析得到输出结果和输出文件out.txt，打开文件看到生成的符号表和抽象语法树。



图表 28 编译通过并输入测试命令

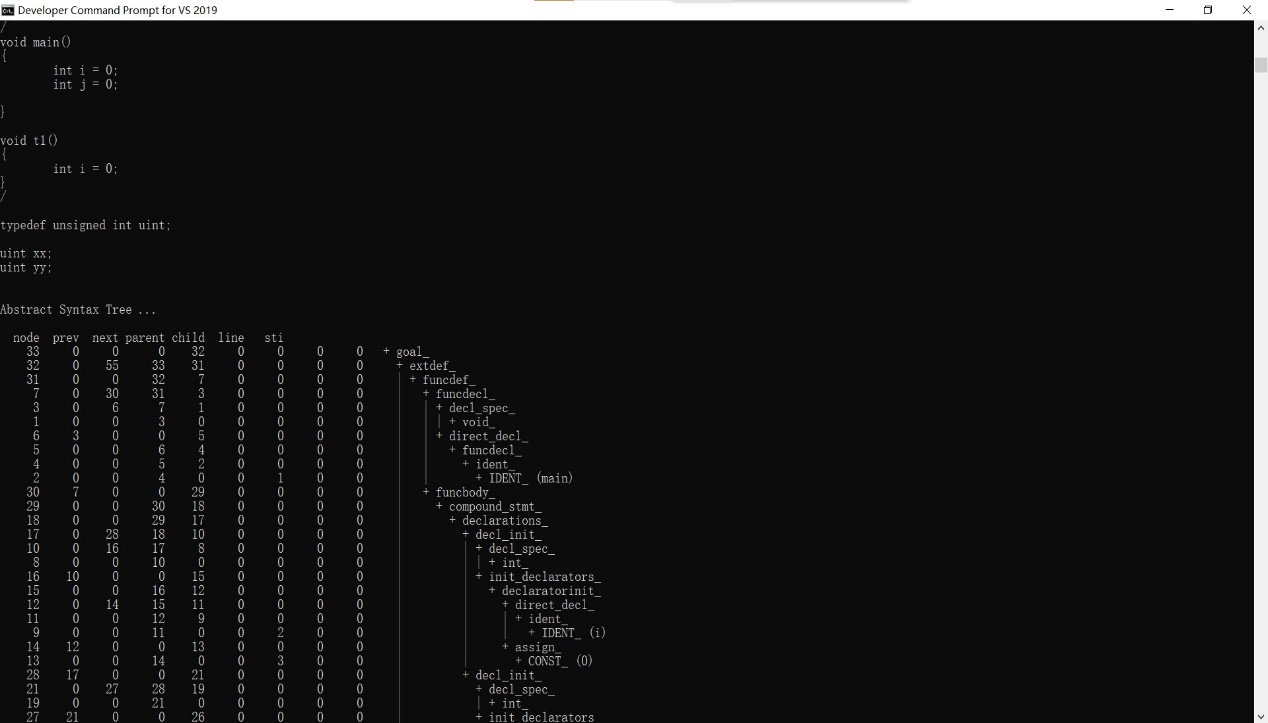


图29 输出结果1

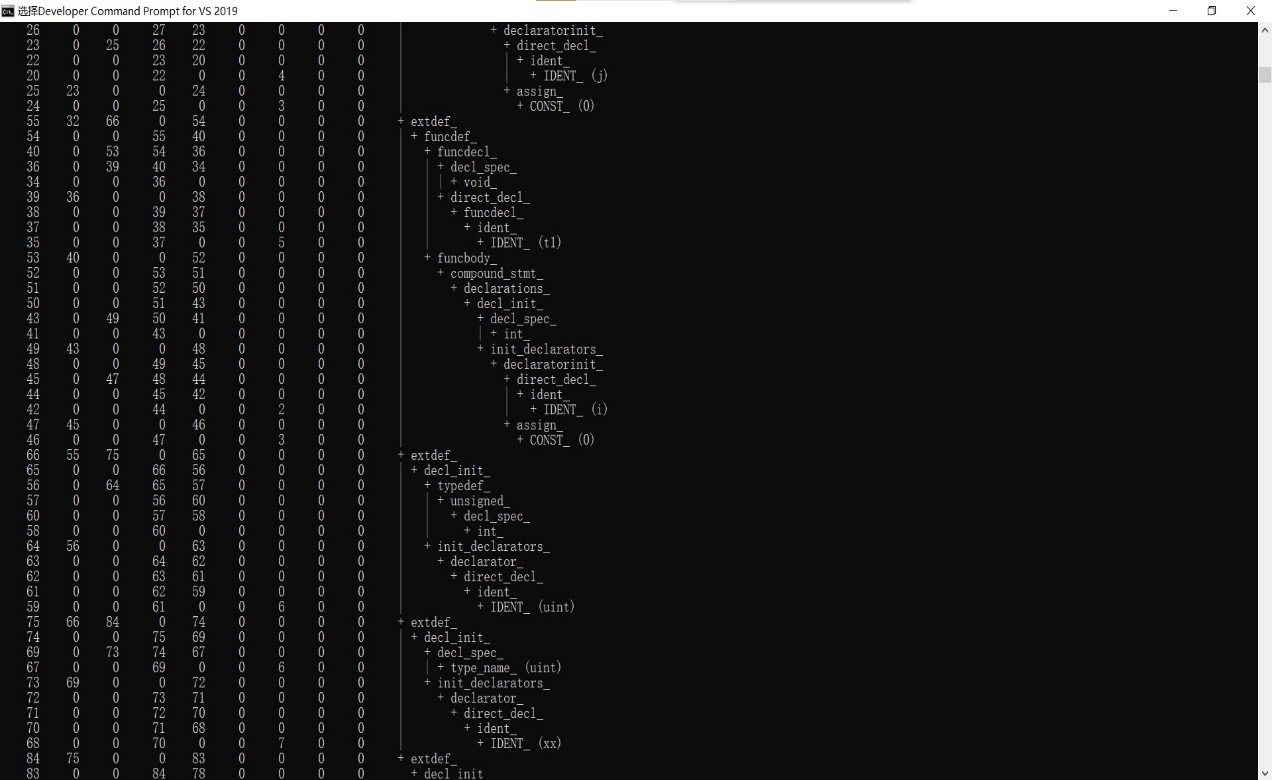


图30 输出结果2

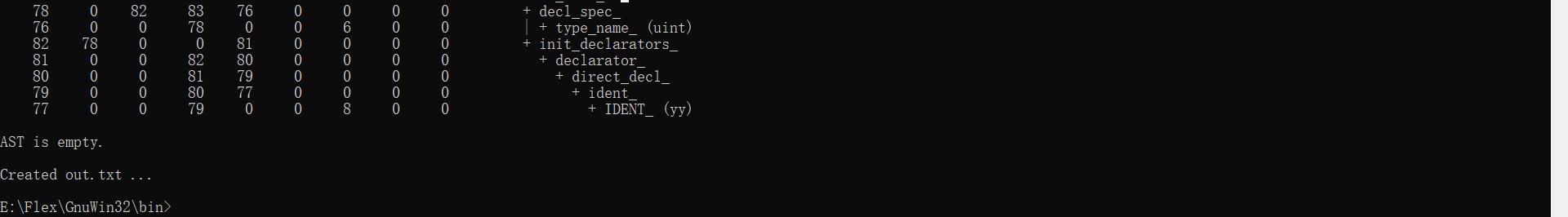


图31 输出结果3

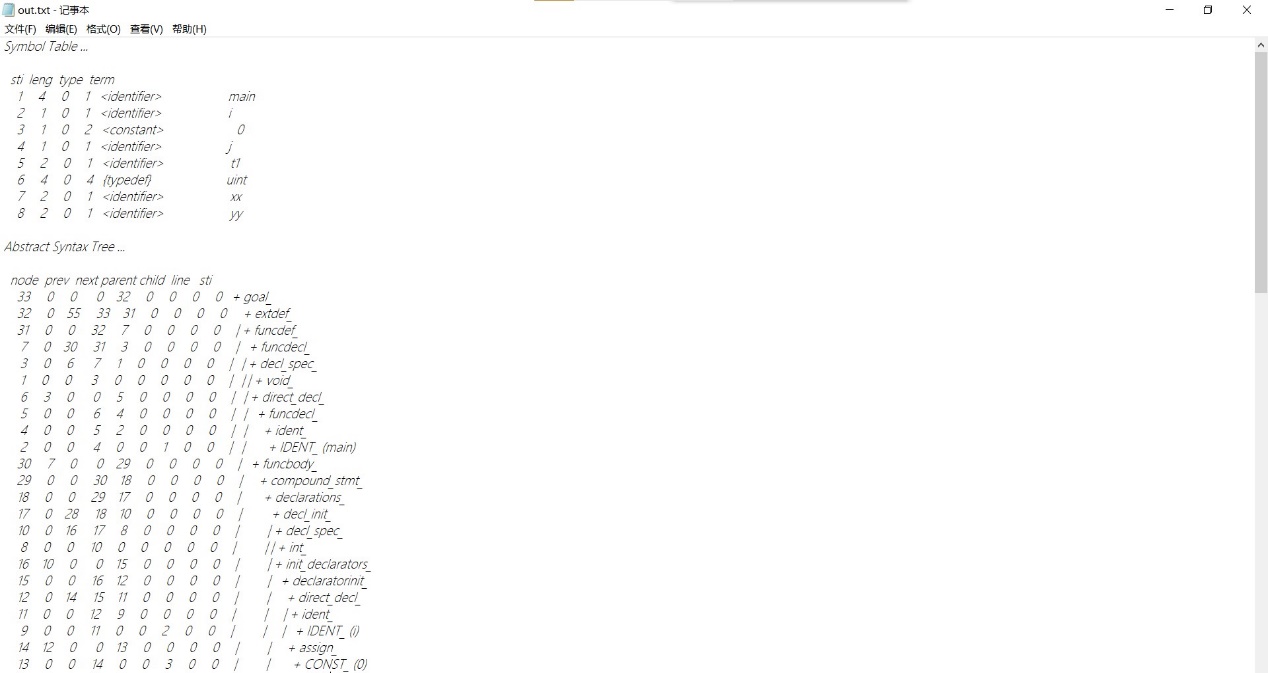


图32 out.txt文件内容1

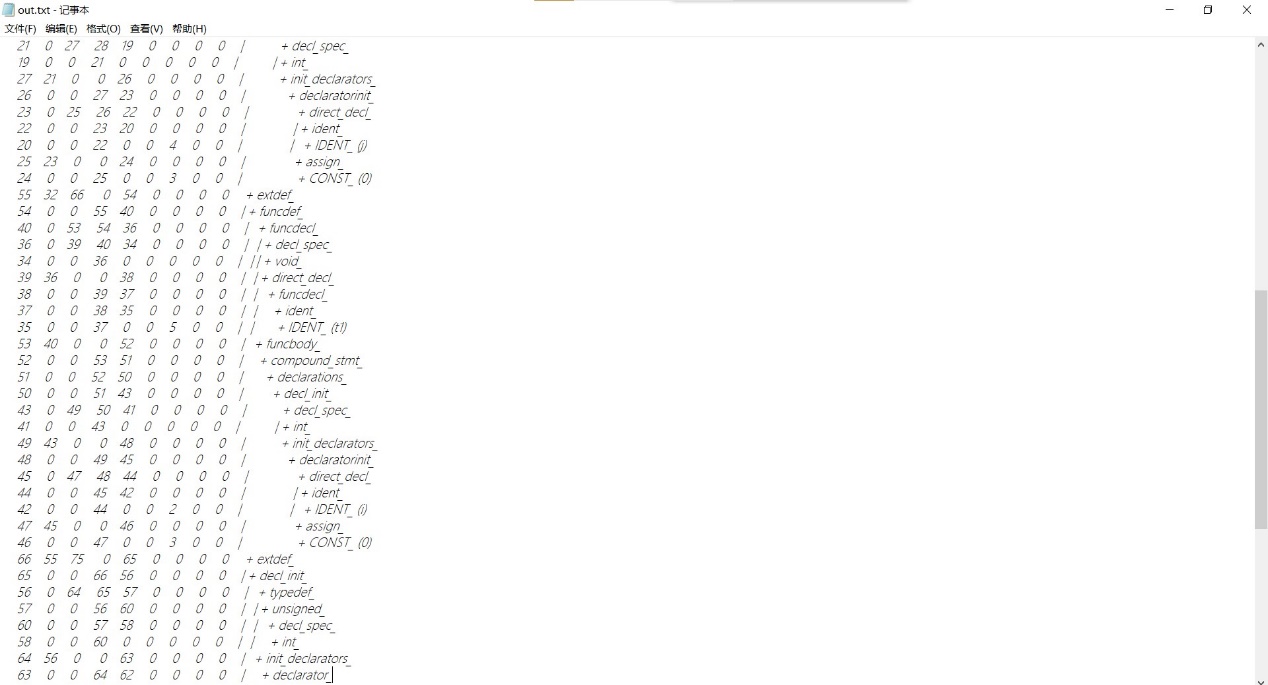


图33 out.txt文件内容2

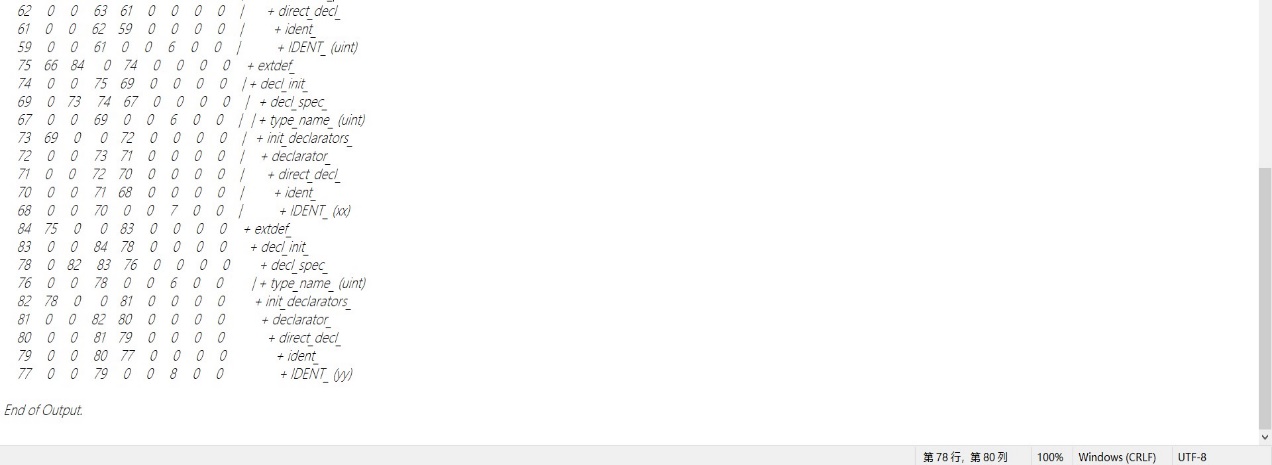


图34 out.txt文件内容3

## 7.2 Linux环境

(1) 打开命令窗口，输入cd Desktop命令进入桌面目录，再依次输入flex input.lex、 bison -vdty cgrammar-new.y命令生成lex.yy.c、y.tab.c、y.tab.h和y.output文件，会发现cgrammar-new.y文件内出现了移进-规约冲突提示，打开y.output文件查看具体内容。

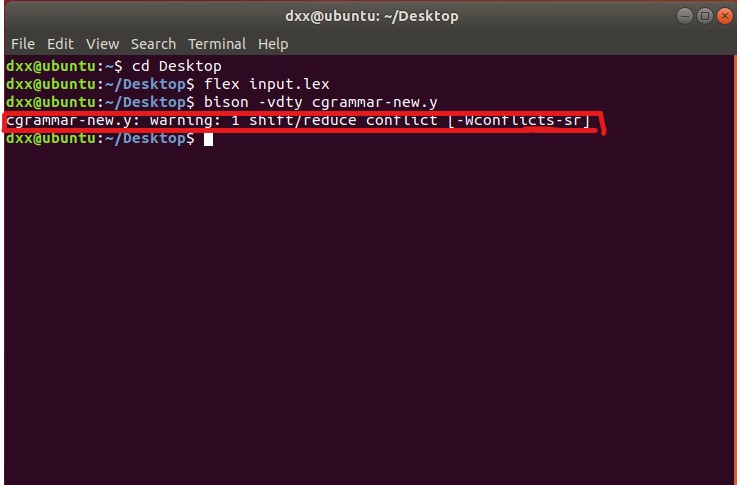


图35 输入命令及结果





图36 查看文件内容

(2) 冲突分析以及解决方式同Windows环境下一致，在此不再赘述。修改cgrammar-new.y文件中的内容见下图：

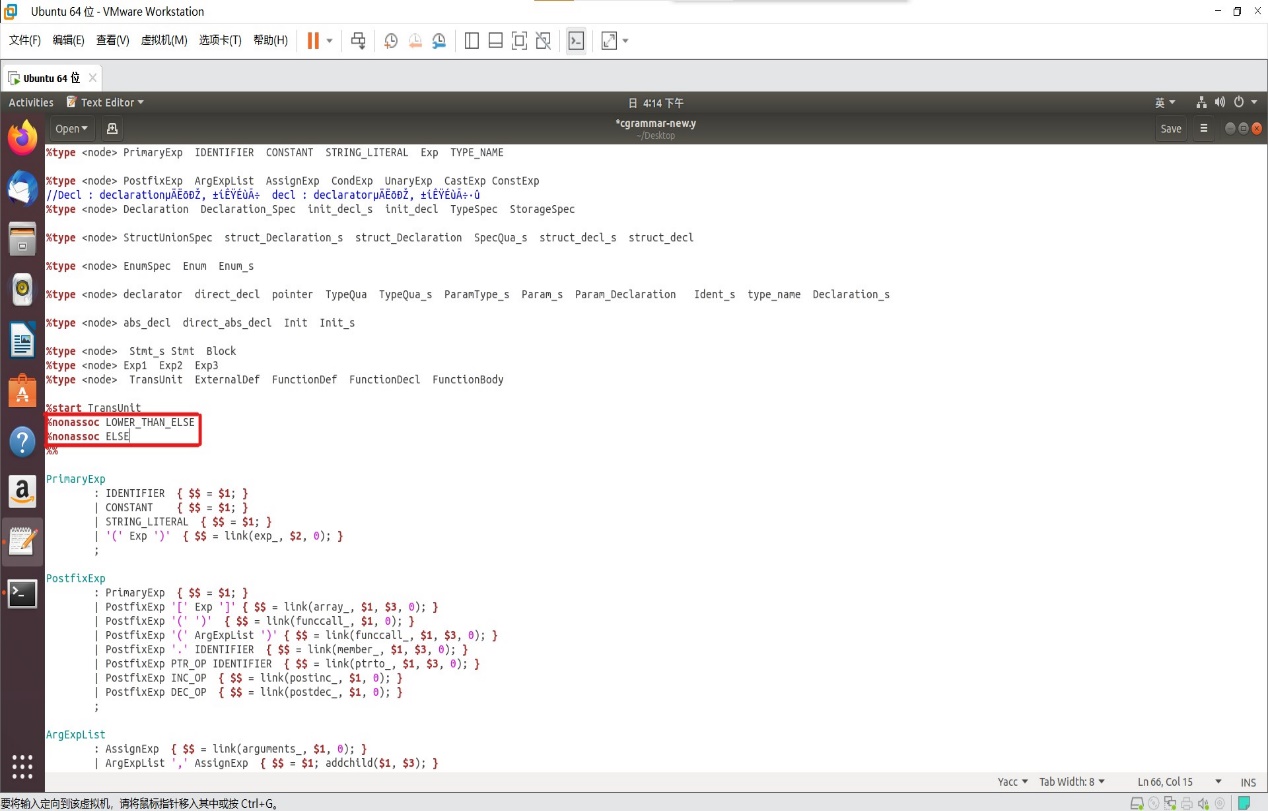


图37 修改文件内容1

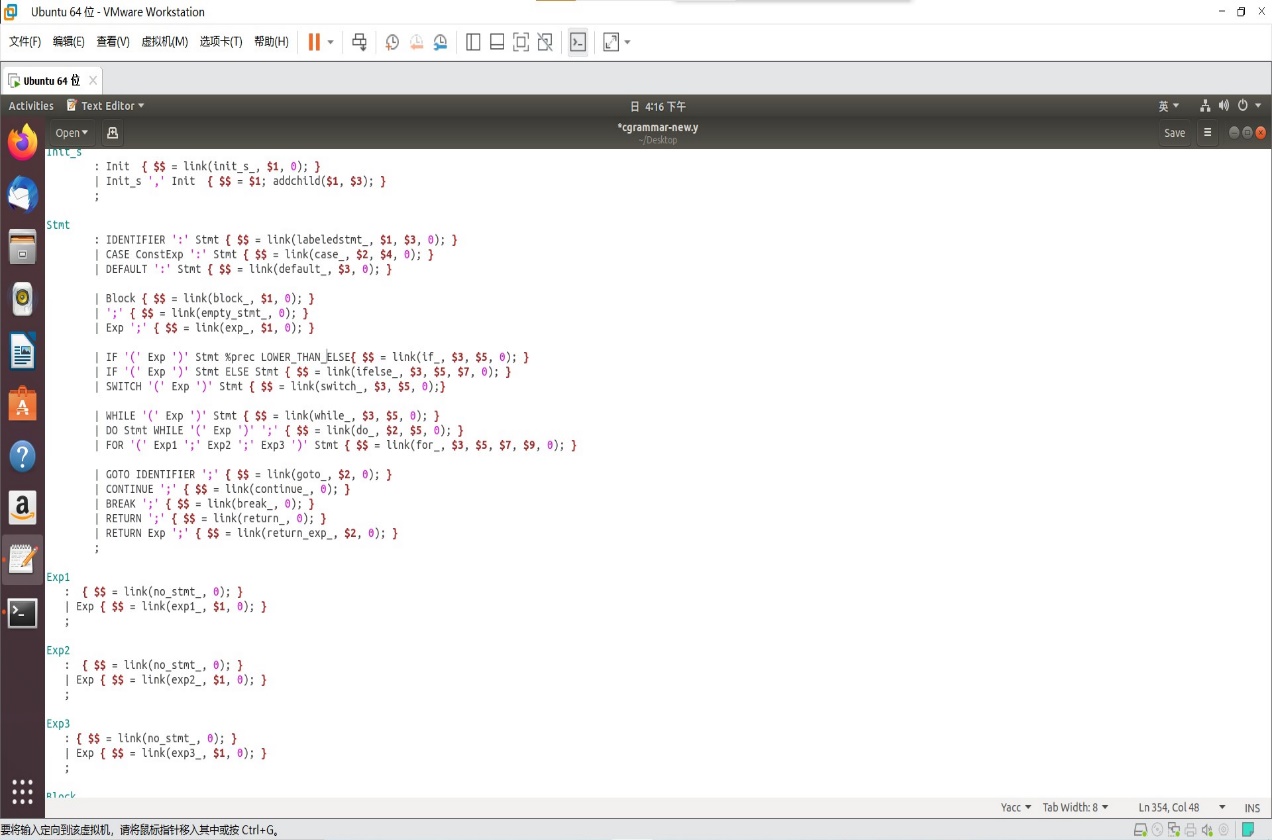


图38 修改文件内容2

(3) 修改完成后，重新再依次输入flex input.lex、 bison -vdty cgrammar-new.y命令，发现不再出现冲突提示。紧接着输入命令cc -o parser y.tab.c main.c lex.yy.c parser.c生成parser文件，发现出现编译错误。为解决编译问题，其文件修改内容与在Windows环境下相同。

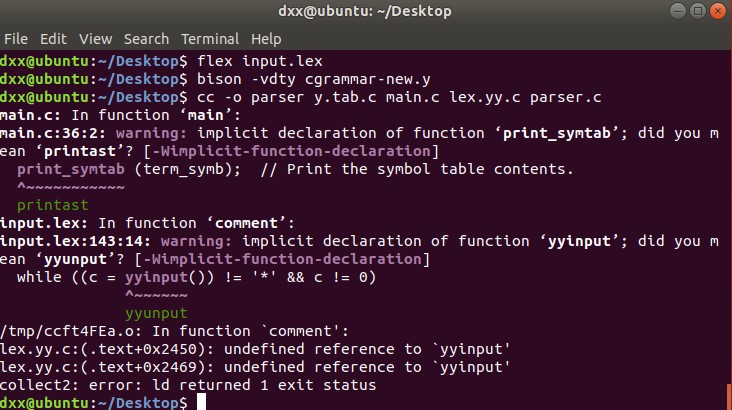


图39 编译错误警报

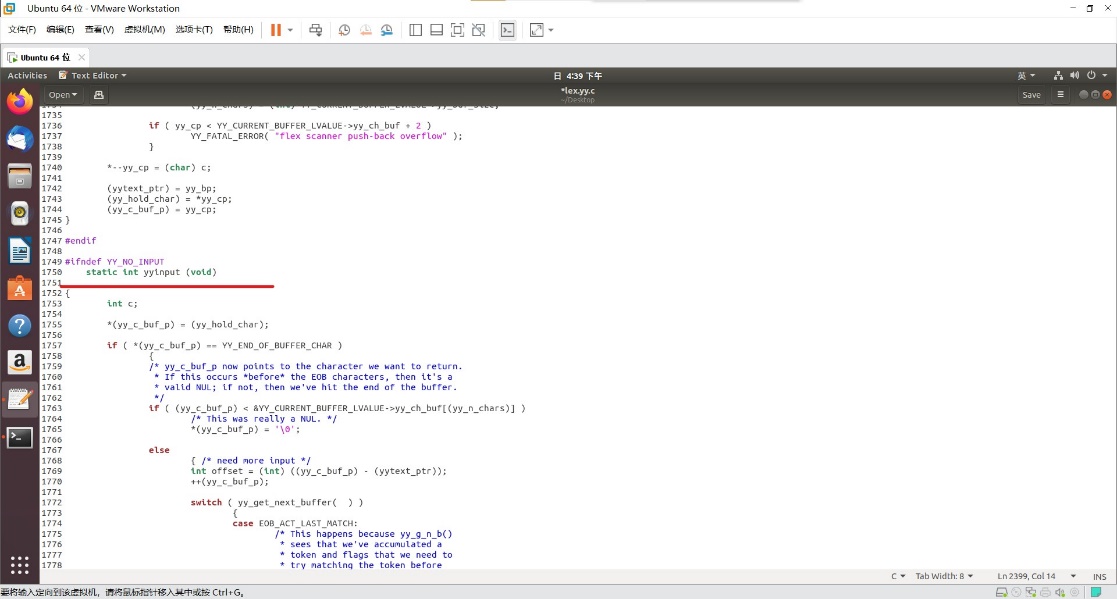


图40 修改文件内容1

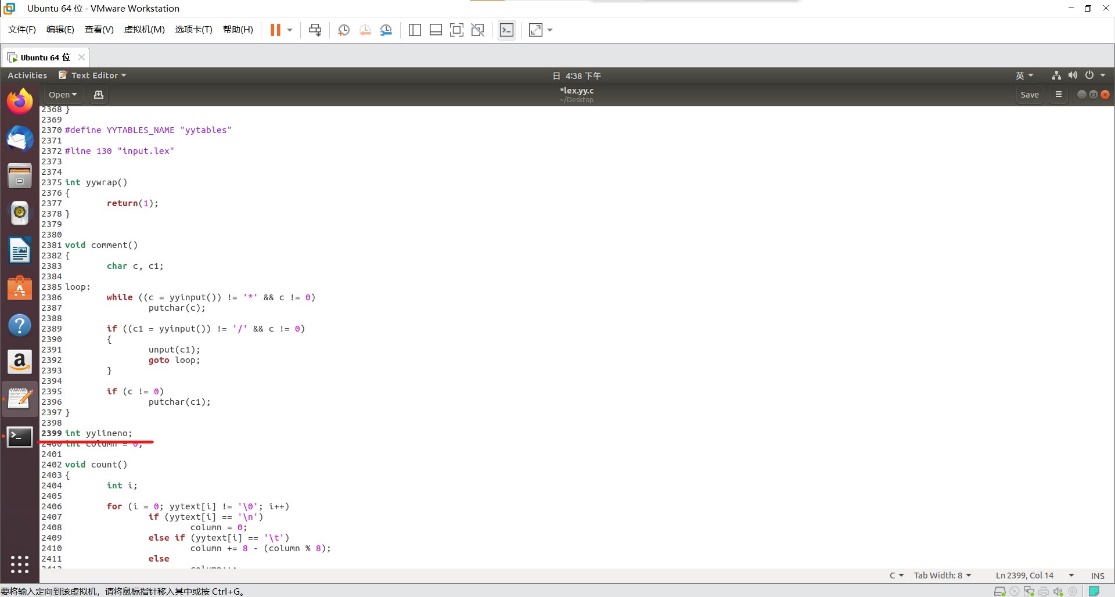


图41 修改文件内容2

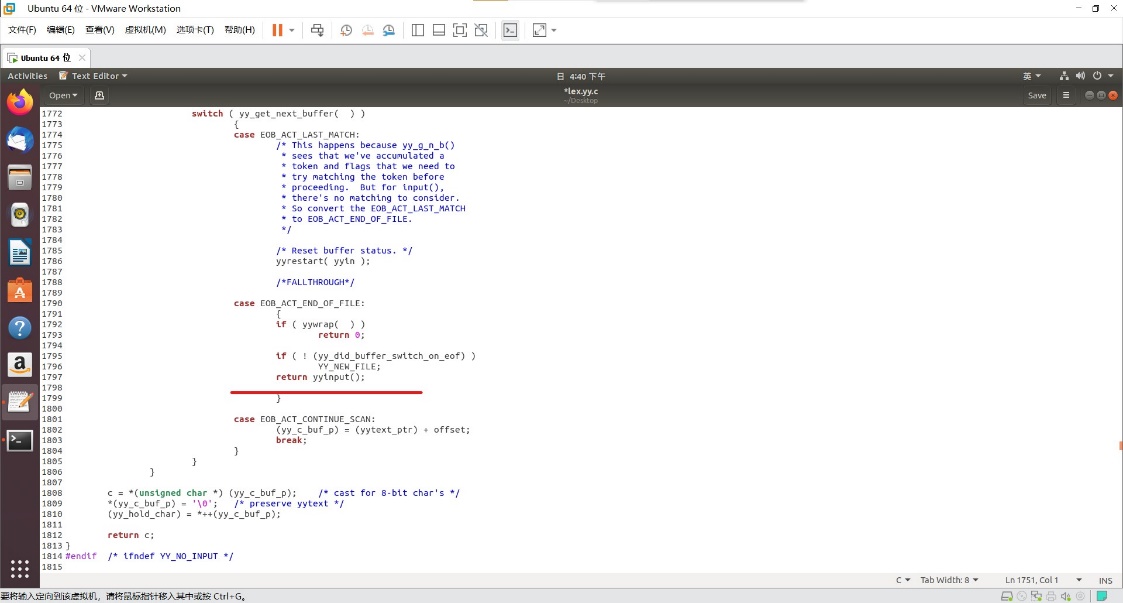


图42 修改文件内容3

(4) 修改后重新编译，生成parser文件。最后输入./parser < test.c，对源文件test.c进行分析得到输出结果和输出文件out.txt，打开文件看到生成的符号表和抽象语法树。

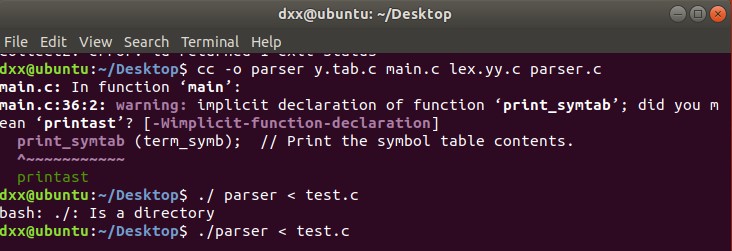


图43 编译通过并输入测试命令

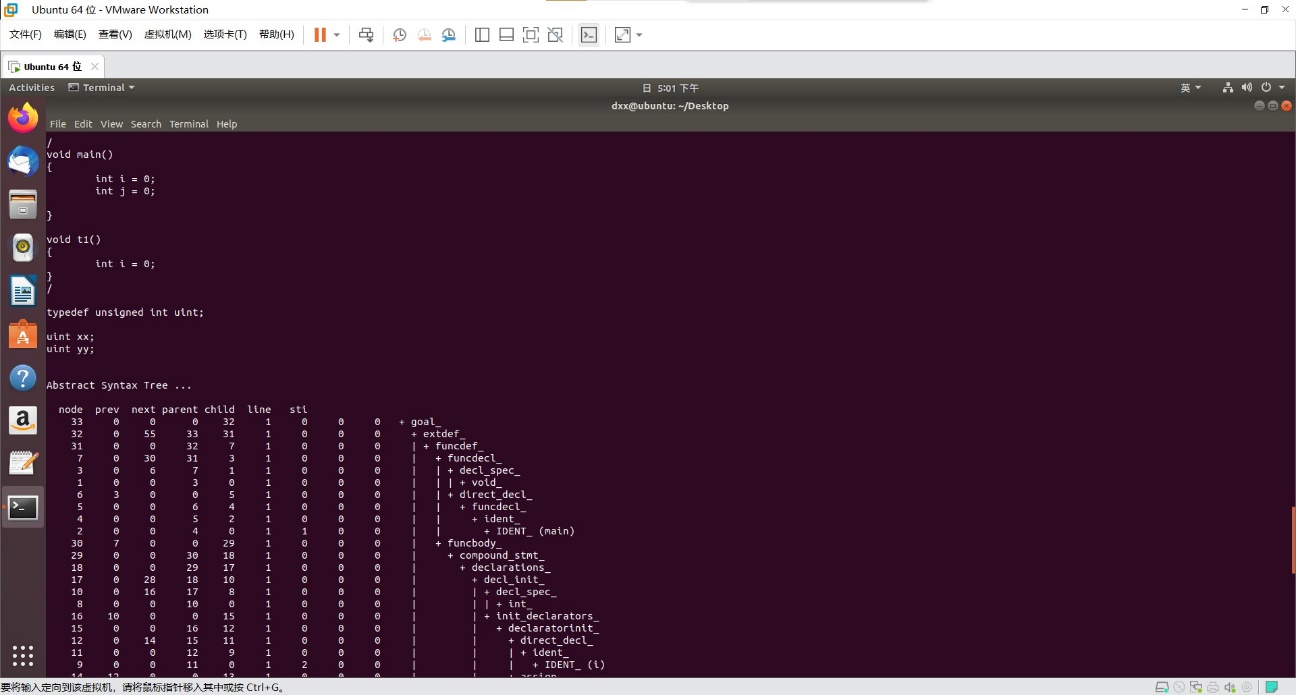


图44 输出结果1

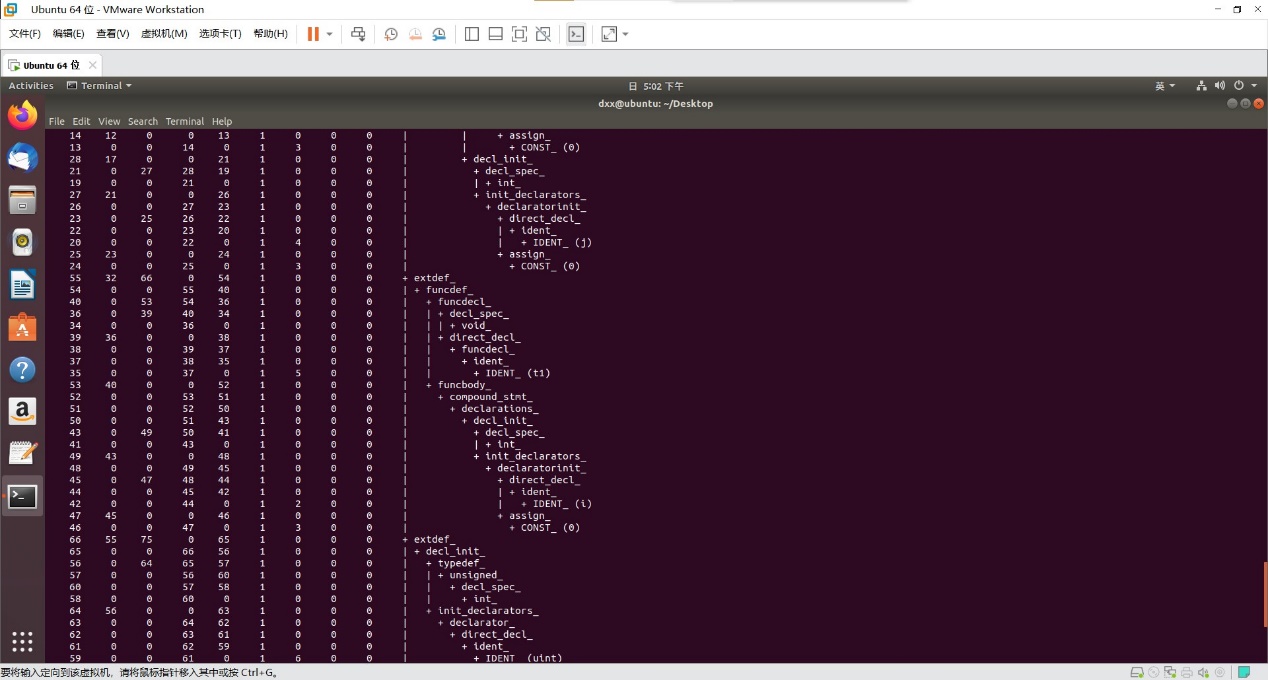


图45 输出结果2

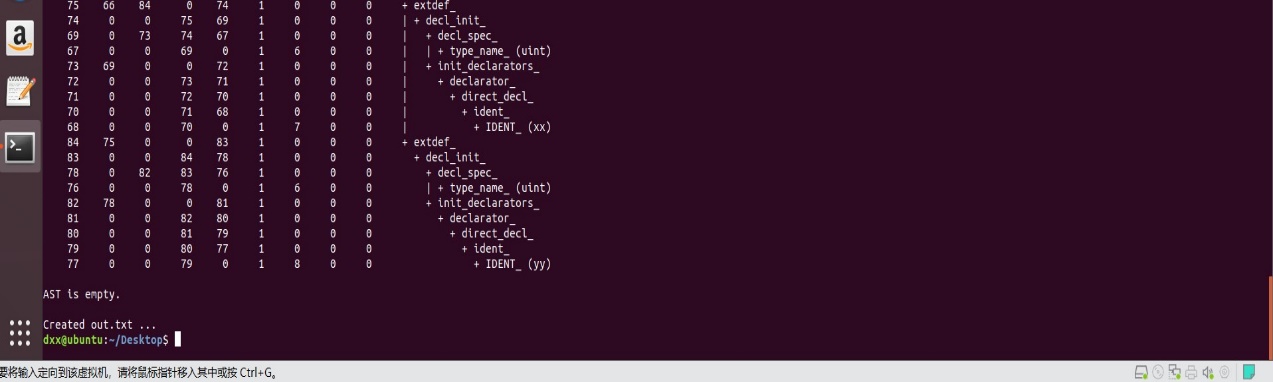


图46 输出结果3

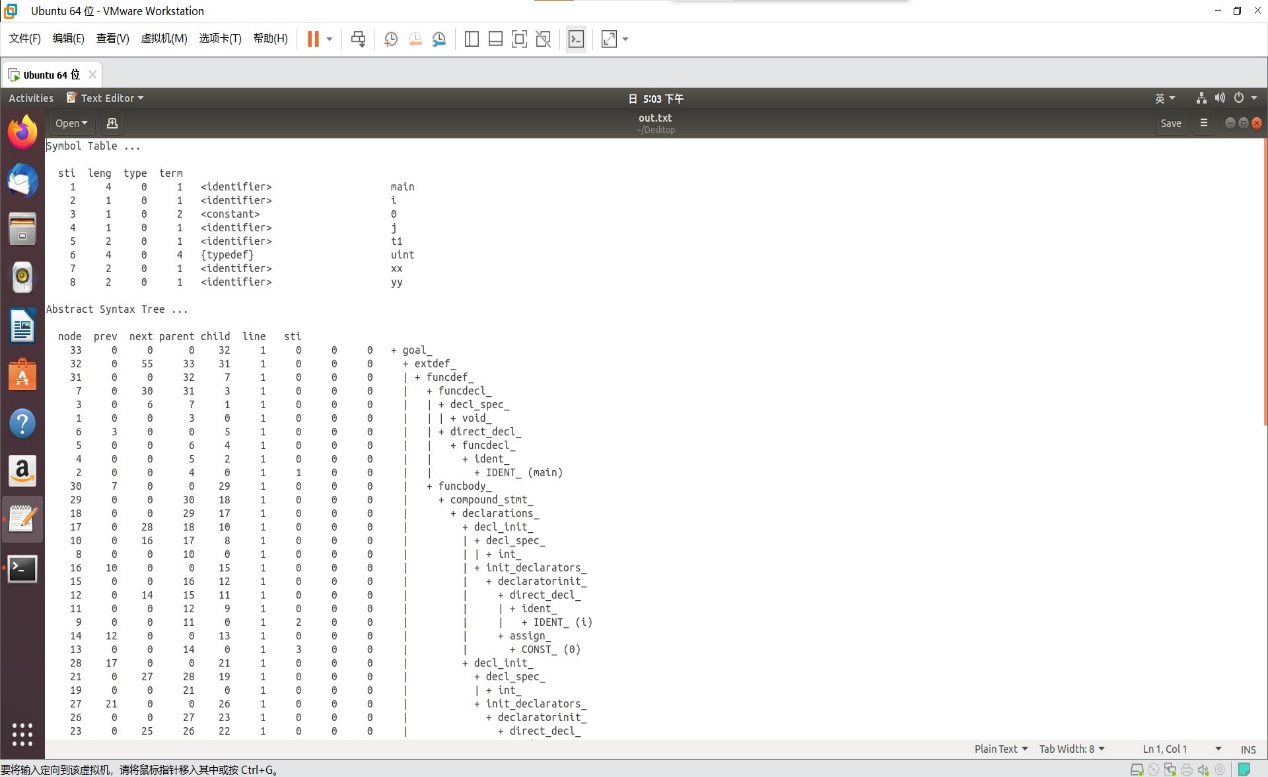


图47 out.txt文件内容1

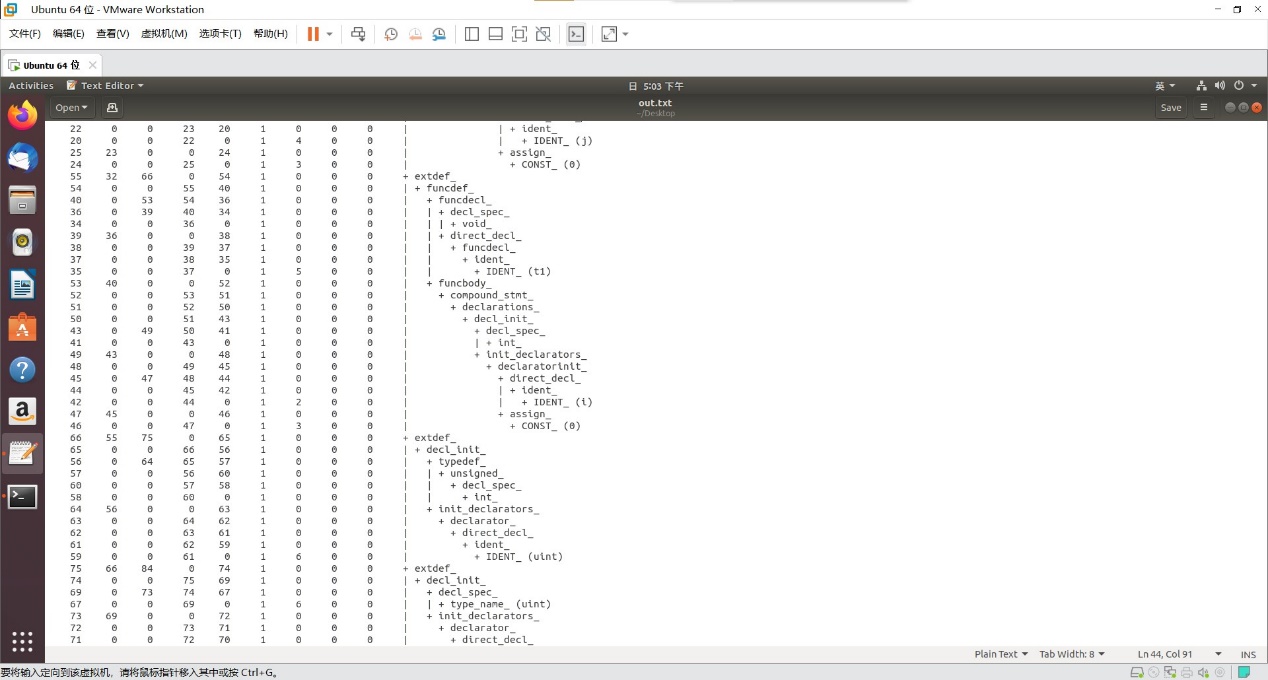


图48 out.txt文件内容2

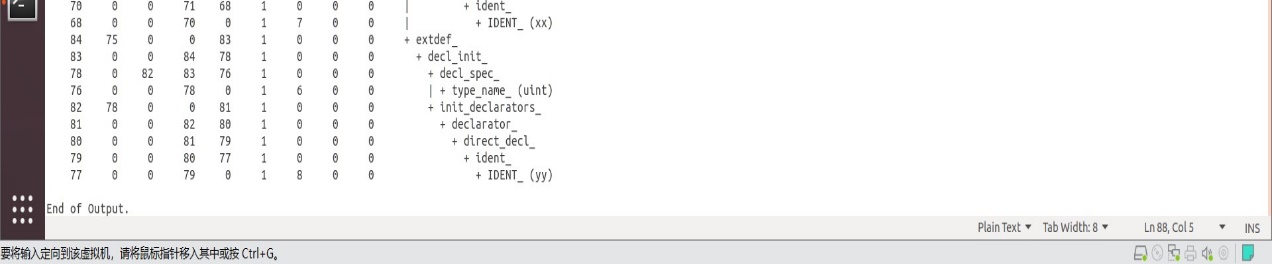


图49 out.txt文件内容3

(5) 注意：由于系统差异的问题在Linux系统下的input.lex文件中要将#include "cgrammar-new.tab.h"改为#include "y.tab.h"，将#include <io.h>改为#include <sys/io.h>。

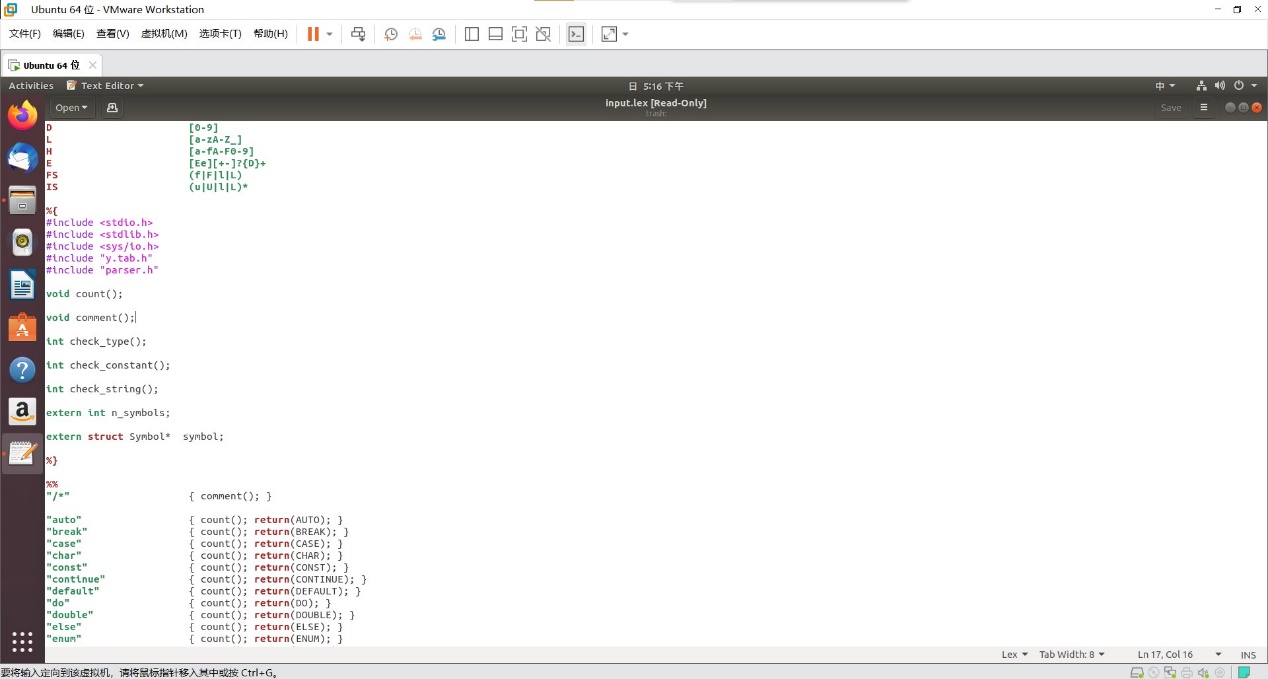


图50 其他修改

# 8 分析输出结果

对输出结果中符号表和抽象语法树的生成过程进行分析，在输出的符号表中，每一行输出的数据从左到右依次是符号表索引、符号长度、符号类型(由于type 并没有在input.lex和cgrammar-new.y被重新赋值，因此均为0)、终结符编号(1 代表identifier，2代表constant，3代表string，4代表typedef)和符号本身。符号表中一共有8个不同的符号，与test.c文件进行比较可以发现，除了保留字之外的所有单词都被输出，与预期相符合。在输出的抽象语法树中，每一行输出的数据从左到右依次是节点编号、该节点的前一个节点、该节点的下一个节点、该节点的父节点、该节点的子节点、该节点所在的行号、该节点的符号表索引、节点名称。可以发现，从上到下每一个节点的节点名称都是从开始符号的节点名称一步步推导到终结符，符合上下文无关文法，并使用“|”和“+”使语法分析树的建立过程十分清晰易懂。叶子节点处的终结符的数量也与test.c源程序相符合，说明语法分析树是正确的。

# 9 实验总结

此次实验遇到的最大的难题就是理解input.lex、cgrammar-new.y文件中的相关代码，由于文件代码量比较大，定义的标识符比较多，各个文件之间的变量和函数的引用或调用关系复杂，导致花费很长时间才弄清楚代码之间的相互联系并搞清楚抽象语法树的生成过程。其次的困难就是解决编译错误问题，在Windows环境和Linux环境下出现的编译错误有相同之处，但也有不同的地方。通过查阅相关资料并参考模板最终还是编译成功。

通过此次实验，更加熟练掌握了Flex/Bison的语法规则，以及相应代码的编写，也提高了自己解读代码的能力，对如何使用Flex和Bison编写某一种语言的分析器有了更深刻的认识，为下面的综合性实验奠定了基础。