

**编译原理实验课程报告**

**题目：MiniC语言编译器设计与实现**

班 级 信安 1805

姓 名 肖昂

学 号 U201812212

指导教师 刘铭

报告日期 2020.11.22

网络空间安全学院

**要 求**

1、报告本人独立完成，内容真实，如发现抄袭，成绩无效；如果引用查阅的资料，需将资料出处列入参考文献，其格式请按华中科技大学本科毕业论文规范，并在正文中标注参考文献序号；

2、按编译原理实验内容，应包含：语言的文法、语言的词法及语法分析、语义分析、中间代码生成、目标代码生成；

3、应说明编译器编写调试时的系统环境、各编译器部分主要采用的设计思路、关键过程、设计的结果（主要源码文件功能、数据结构、函数说明）；同时，对设计实现中遇到的问题、解决过程进行记录和小节；并对完成的编译器过程进行总结，明确指出其优点、不足；

4、评分标准：4个主要实验环节按任务书要求完成；采用的方法合适、设计合理；能体出分析问题、灵活运用知识解决实际问题的能力；报告条理清晰、语句通顺、格式规范；

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **格式**  **规范** | **词法**  **语法** | **语义**  **分析** | **中间**  **代码** | **目标**  **代码** | **总分** |
| **20** | **30** | **30** | **10** | **10** | **100** |
|  |  |  |  |  |  |

**目 录**

[1 选题背景 1](#_Toc56962094)

[1.1 总体任务 1](#_Toc56962095)

[1.2 目标 1](#_Toc56962096)

[1.3 主要技术 1](#_Toc56962097)

[2 实验一 词法分析和语法分析 2](#_Toc56962098)

[2.1 语法的文法描述 2](#_Toc56962099)

[2.2 词法的文法描述 2](#_Toc56962100)

[2.3 语法分析器设计 2](#_Toc56962101)

[2.4 词法分析器设计 2](#_Toc56962102)

[2.5 词法及语法分析器运行截图 2](#_Toc56962103)

[2.6 小结 2](#_Toc56962104)

[参考文献 3](#_Toc56962105)

1 选题背景

1.1 总体任务

根据密码学编程时的某个函数（例如计算最大公约数函数）需要完成的程序功能，运用编译原理课程中的知识，设计一个类C语言程序设计语言相应的词法及语法，并实现该语言的编译器，将源码翻译为能在模拟器上运行的目标代码。

1.2 目标

通过资料查阅、分析设计、编程实现等步骤，逐步培养学生解决工程问题的能力，提高独立思考、灵活运用理论知识以解决实际问题的能力；

通过构造简化编译器的过程，了解编译器各部分的理论知识，找出理论与实践中的差异；灵活运用第三方工具协助解决工程问题；综合运用数据结构、算法、汇编语言、C语言编程等前序课程的知识和技能。

1.3 语言定义

结合完成以下给出的是一个简化的C语言的文法，其称为mini-c，仅作为讲解参考；你可以根据自己的爱好和兴趣，拟定其他名称和语法规则。

G[program]:

program → ExtDefList

ExtDefList→ExtDef ExtDefList | ε

ExtDef→Specifier ExtDecList ; |Specifier FunDec CompSt

Specifier→int | float

ExtDecList→VarDec | VarDec , ExtDecList

VarDec→ID

FucDec→ID ( VarList ) | ID ( )

VarList→ParamDec , VarList | ParamDec

ParamDec→Specifier VarDec

CompSt→{ DefList StmList }

StmList→Stmt StmList | ε

Stmt→Exp ; | CompSt | return Exp ;

| if ( Exp ) Stmt | if ( Exp ) Stmt else Stmt | while ( Exp ) Stmt

DefList→Def DefList | ε

Def→Specifier DecList ;

DecList→Dec | Dec , DecList

Dec→VarDec | VarDec = Exp

Exp →Exp =Exp | Exp && Exp | Exp || Exp | Exp < Exp | Exp <= Exp

| Exp == Exp | Exp != Exp | Exp > Exp | Exp >= Exp

| Exp + Exp | Exp - Exp | Exp \* Exp | Exp / Exp | ID | INT | FLOAT

| ( Exp ) | - Exp | ! Exp | ID ( Args ) | ID ( )

Args→Exp , Args | Exp

说明：

program为文件开始符号；

ExtDefList指程序语句列表；

ExtDef指某一行的语句；

ExtDefList指后面多行语句；

Specifier为标识符；

ExtDecList 变量声明列表；

FunDec函数；

CompSt函数体；

FunDec函数声明；

VarList变量列表；

ParamDec参数声明；

VarDec变量声明；

Dec代表声明语句；

Stmt代表函数内部声明语句；

StmtList代表声明语句列表；

Exp代表表达式；

ID为对应的变量或函数名字。

1.4 主要技术

开发环境：Ubuntu16.04系统

关键技术：Flex、Bison、C语言基本知识、抽象语法树AST

**2 实验一 词法分析和语法分析**

实验一任务：

1、完成语言的语法描述和单词描述；完成语法规则、词法规则定义；

2、构造语法分析程序、词法分析程序；完成抽象语法树AST的设计；

3、构造合适的源程序样例，出其对应的抽象语法树；

4、完成错误提示功能。

2.1 语法的文法描述

定义非终结符的类型，结合bison的语法规则，%type定义非终结符的语义值类型，形式是%type <union的成员名> 非终结符。

定义的非终结符如下：

%type <ptr> program ExtDefList ExtDef Specifier ExtDecList FuncDec ArrayDec CompSt VarList VarDec ParamDec Stmt StmList DefList Def DecList Dec Exp Args

举一个例子进行说明，其中%type <ptr> program ExtDefList，这表示非终结符ExtDefList属性值的类型对应联合中成员ptr的类型，在本实验中对应一个树结点的指针。

其次，利用%token定义终结符的语义值类型。%token <type\_id> ID，表示识别出来一个标识符后，标识符的字符串串值保存在成员type\_id中。

%token <type\_int> INT//指定INT的语义值是type\_int,由词法分析得到的数值。

%token <type\_id> ID RELOP TYPE //指定ID,RELOP 的语义值是type\_id,由词法分析得到的标识符字符串。

%token <type\_float> FLOAT //指定ID的语义值是type\_id,由词法分析得到的标识符字符串。

%token <type\_char> CHAR

%token LP RP LC RC SEMI COMMA LB RB

%token PLUS MINUS STAR DIV ASSIGNOP AND OR NOT IF ELSE WHILE RETURN COMADD COMSUB FOR //+=comadd，-=comsub，for，=assignop

然后，定义运算符的优先级与结合性，如下表所示

表1优先级与结合性

|  |  |  |
| --- | --- | --- |
| 优先级 | 结合性 | 符号 |
| 高 | 左 | “+=”，”-=" |
|  | 左 | “=” |
|  | 左 | “||” |
|  | 左 | “&&” |
|  | 左 | ">"|"<"|">="|"<="|"=="|"!=" |
|  | 左 | “+”, “-“ |
|  | 左 | “\*”，“/” |
| 低 | 右 | “！”，“++”，“—” |

再次，是语法定义的核心部分：语法规则部分。语法规则由1.1中所定义的语法规则编写，具体实现代码见附件。

2.2 词法的文法描述

按照语法定义的终结符与非终结符如表所示。

|  |  |  |
| --- | --- | --- |
| 单词符号类型 | 单词种类码 | 正则表达式 |
| {id} | ID | [A-Za-z][A-Za-z0-9]\* |
| {int} | INT | [0-9]+ |
| {float} | FLOAT | ([0-9]\*\.[0-9]+)|([0-9]+\.) |
| "int" | TYPE |  |
| "float" | TYPE |  |
| "return" | RETURN |  |
| "if" | IF |  |
| "else" | ELSE |  |
| "while" | WHILE |  |
| "for" | FOR |  |
| ";" | SEMI |  |
| "," | COMMA |  |
| ">"|"<"|">="|"<="|"=="|"!=" | RELOP |  |
| "=" | ASSIGNOP |  |
| "+" | PLUS |  |
| “+=” | COMADD |  |
| “-=” | COMSUB |  |
| “++” | AUTOADD |  |
| “—" | AUTOSUB |  |
| "-" | MINUS |  |
| "\*" | STAR |  |
| "/" | DIV |  |
| "&&" | AND |  |
| "||" | OR |  |
| "!" | NOT |  |
| "(" | LP |  |
| ")" | RP |  |
| "{" | LC |  |
| "}" | RC |  |
| "[" | LB |  |
| "]" | RB |  |
| "//"[^\n]\* | 代表单行注释 |  |
| “/”\*(\s|.)\*?\*”/” | 代表多行注释 |  |

2.3 语法分析器设计

同Flex源代码类似，Bison源代码也分为三个部分，其作用与Flex源代码大致相同，其分为定义部分、规则部分、用户函数部分。

1. **定义部分**

所有的词法单元的定义都可以放到这部分。这段bison代码以“%{“和”%}“开头，被“%{“和”%}“包含的内容主要是对stdio.h的引用。接下来是一些以%token开头的此法单元（终结符）定义，如果需要采用Flex生成的yylex（），那么在这里定义的词法单元都可以作为Flex源代码里的返回值。与终结符相对的，所有未被定义为%token的符号都会被看作非终结符，这些非终结符要求必须在任意产生式的左边至少出现一次。

%{

#include "stdio.h"

#include "math.h"

#include "string.h"

#include "def.h"

extern int yylineno;

extern char \*yytext;

extern FILE \*yyin;

void yyerror(const char\* fmt, ...);

void display(struct node \*,int);

%}

此外，bison文件辅助定义部分如下：

%union {

  int type\_int;

  float type\_float;

  char type\_char;

  char type\_id[32];

  struct node \*ptr;

};

辅助声明部分利用%union将各种类型统一起来。Bison中默认将所有的语义值都定义为int类型，可以通过定义宏YYSTYPE来改变值的类型。如果有多个值类型，则需要通过在Bison声明中使用%union列举出所有的类型。

1. **规则部分**

这部分包括具体的语法和相应的语义动作。具体来讲是在书写产生式。第一个产生式左边的非终结符默认为初始符号。产生式里的箭头在这里用冒号”:”表示，一组产生式与另一组之间以分号”；“隔开。产生式里无论是终结符还是非中介都都各自对应一个属性值，乘胜是左边的非终结符对应的属性值用￥￥表示，右边的几个符号的属性值按从左到右的顺序一次对应位$1、$2、$3等。每条产生式的最后可以添加一组以花括号”{“和”}”括起来的语义动作，这组语义动作会在整条产生式的最后可以添加一组产生式规约完成之后执行，如果不明确指定语义动作，那么bison将采用默认的语义动作{$$=$1}。需要注意的是，在产生式中间添加语义动作在某些情况下有可能在原有语法中引入冲突，因此能使用时要特别谨慎。

1. **用户函数部分**

这部分的代码会被原封不动的拷贝到syntax.tab.c中，以方便用户自定义所需要的函数。如果想要对这部分所用到的变量、函数或者头文件进行声明，可以在定义部分之前使用”%{“和”%}”将要申明的内容添加进去。被”%{“和”%}”所包围的内容也会被一并拷贝到syntax.tab.c的最前面。用户自定义函数如下：

%%

int main(int argc, char \*argv[]){

  yyin=fopen(argv[1],"r");

*if* (!yyin)

    return 0;

  yylineno=1;

  yyparse();

  return 0;

}

*#include*<stdarg.h>

void yyerror(const char\* fmt, ...)

{

  va\_list ap;

  va\_start(ap, fmt);

  fprintf(stderr, "Grammar Error at Line %d Column %d: ", yylloc.first\_line,yylloc.first\_column);

  vfprintf(stderr, fmt, ap);

  fprintf(stderr, ".\n");

}

其中yyerror函数会在语法分析程序中每发现一个语法错误时被调用，其默认参数为“syntax error“。默认情况下yyerror智慧将传入的字符串参数打印到标准错误输出上，而自己也可以重新定义这个函数，从而使它打印一些别的内容。

1. **抽象语法树节点的建立**

在语法分析阶段，要生成建立抽象语法树AST。抽象语法树将词法分析之后生成的单词元素都按照一定的规则组装起来，再利用树的的结构表示出文件中各语法元素的关系。

首先是AST树结点的定义：

struct node {//以下对结点属性定义没有考虑存储效率,只是简单地列出要用到的一些属性

  enum node\_kind kind;//结点类型

  union {

    char type\_id[33]; //由标识符生成的叶结点

    int type\_int; //由整常数生成的叶结点

    char type\_char;//由字符型生成的叶节点

    float type\_float; //由浮点常数生成的叶结点

  };

  struct node \*ptr[3];//子树指针,由kind确定有多少棵子树

  int level;//层号

  int place; //表示结点对应的变量或运算结果临时变量在符号表的位置序号

  char Etrue[15],Efalse[15]; //对布尔表达式的翻译时,真假转移目标的标号

  char Snext[15]; //该结点对应语句执行后的下一条语句位置标号

  struct codenode \*code; //该结点中间代码链表头指针

  char op[10];

  int type;//结点对应值的类型

  int pos; //语法单位所在位置行号

  int offset; //偏移量

  int width; //各种数据占用的字节数

};

其次是实例化一个语法树节点：

struct node \*mknode(int kind,struct node \*first,struct node \*second, struct node \*third,int pos ){

  struct node \*tempnode = (struct node\*)malloc(sizeof(struct node));

  tempnode->kind = kind;

  tempnode->ptr[0] = first;

  tempnode->ptr[1] = second;

  tempnode->ptr[2] = third;

  tempnode->pos = pos;

*return* tempnode;

}

**5. 显示抽象语法树**

抽象语法树的遍历是树的先序遍历，将遍历的结果输出，对不同的节点输出结果不一样。Display的函数见附件。

2.4 词法分析器设计

使用flex工具编写lex文件，对指定的高级语言程序进行词法分析。

1. **定义部分**

定义部分实际上就是给后面某些可能经常用到的正则表达式取一个别名，从而简化词法规则的书写。定义部分的格式一般为:

*Name definition*

其中name是名字，definition是任意的正则表达式。

在lex文件中定义部分如下：

%{

*#include* "parser.tab.h"

*#include* <string.h>

*#include* "def.h"

    int yycolumn = 1;

*#define* YY\_USER\_ACTION yylloc.first\_line=yylloc.last\_line=yylineno; yylloc.first\_column=yycolumn; yylloc.last\_column=yycolumn+yyleng-1; yycolumn+=yyleng;

    typedef union{

        int type\_int;

        float type\_float;

        char type\_char;

        char type\_id[32];

        struct node \*ptr;

    }YYLVAL;

*#define* YYSTYPE YYLVAL

%}

1. **规则部分**

规则部分是由正则表达式和相应的响应函数组成，其格式为

*Pattern {action}*

其中pattern为正则表达式，其书写规则与前面部分的正则表达式定义相同。而action则为将要进行的具体操作，这些操作可以用一段C代码表示。Flex将按照这部分给出饿内容依次尝试每一个规则，尽可能匹配最长的输入串。如果有些内容不匹配任何规则，Flex默认之将其拷贝到标准输出，想要修改这个默认行为只需要在所有郭泽的最后加上一条“.” (即匹配任何输入)规则，然后在其对应的action部分书写想定义的行为即可。

规则部分的代码如下：

%%

*/\*注释处理 单行+多行\*/*

\/\/[^\n]\*  {;}*//匹配注释的正则表达式*

\/\\*(\s|.)\*?\\*\/ {;}*//匹配注释的正则表达式*

{int} {yylval.type\_int=atoi(yytext);return INT;}

{float} {yylval.type\_float=atof(yytext); return FLOAT;}

"int" {strcpy(yylval.type\_id,yytext); return TYPE;}

"float" {strcpy(yylval.type\_id,yytext); return TYPE;}

"char" {strcpy(yylval.type\_id,yytext); return TYPE;}

"return" {return RETURN;}

"if" {return IF;}

"else" {return ELSE;}

"while" {return WHILE;}

"for" {return FOR;}

{id} {strcpy(yylval.type\_id,yytext); return ID;}

";" {return SEMI;}

"," {return COMMA;}

">"|"<"|">="|"<="|"=="|"!=" {strcpy(yylval.type\_id,yytext); return RELOP;}

"=" {return ASSIGNOP;}

"+" {return PLUS;}

"-" {return MINUS;}

"+=" {return COMADD;}

"-=" {return COMSUB;}

"++" {return AUTOADD;}

"--" {return AUTOSUB;}

"\*" {return STAR;}

"/" {return DIV;}

"&&" {return AND;}

"||" {return OR;}

"!" {return NOT;}

"(" {return LP;}

")" {return RP;}

"[" {return LB;}

"]" {return RB;}

"{" {return LC;}

"}" {return RC;}

[\n] {yycolumn=1;}

[ \r\t] {;}

.   {printf("Error type A: Mysterious character\"%s\" at line %d,column %d\n",yytext,yylineno,yycolumn);}

其中的yytext的类型为char\*，它是flex提供的一个变量，里面保存了当前词法单元所对应的词素。

1. **用户自定义代码部分**

这部分代码会被原封不动的拷贝到lex.yy.c中，以方便用户自定义所需要执行的函数（包括之前的main函数）。如果用户想要对这部分用到的变量、函数或者头文件进行声明，可以前面的定义部分（即Flex源代码文件的第一部分）之前使用“%{“和”%}“符号将要声明的内容添加进去。被”%{“和”%}“所包围的内容也会被一并拷贝到lex.yy.c的最前面。

以下是用户子程序部分：

/\* 复制到lex.yy.c中,main冲突不能用了

void main(int argc,char \*argv[]){

    yylex();

    return;

}

\*/

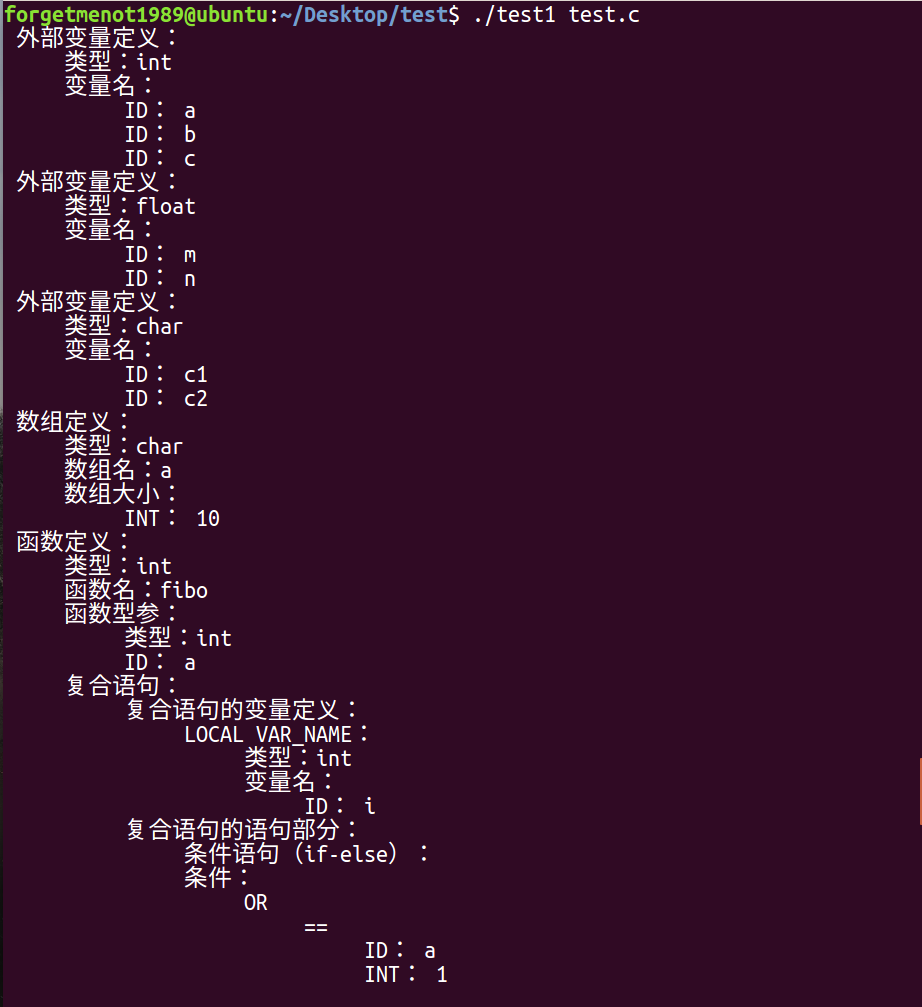
int yywrap(){

    return 1;

}

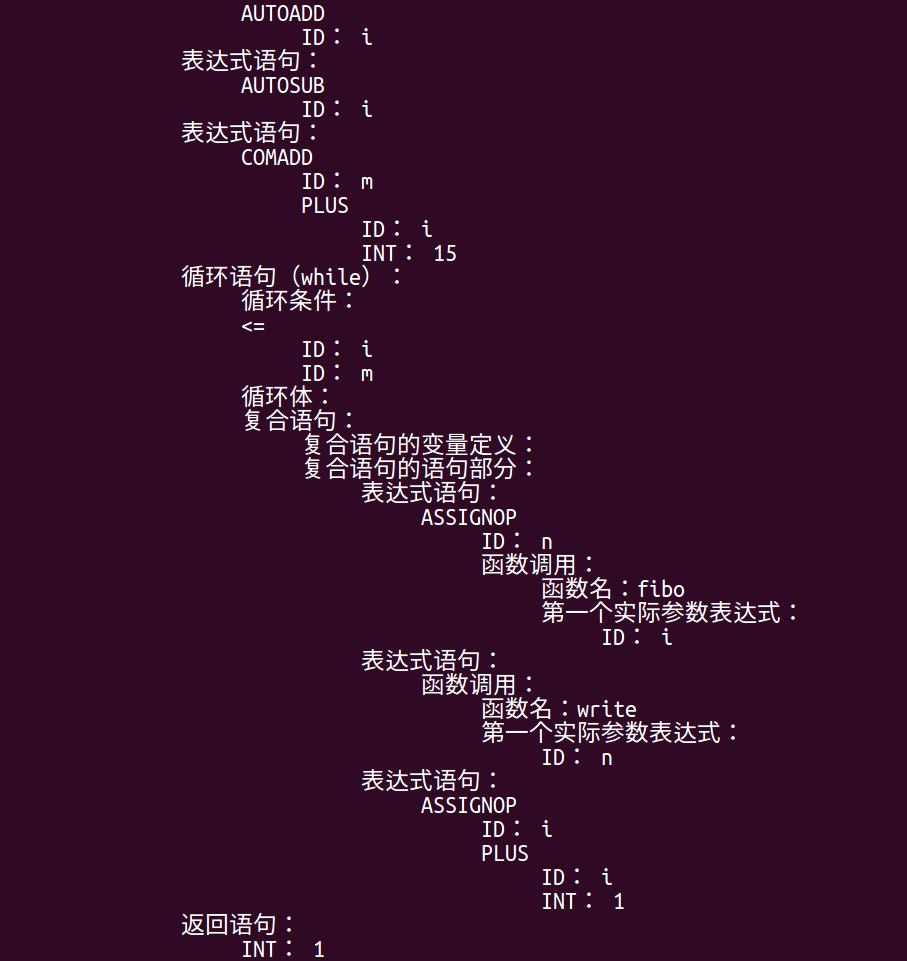
2.5 词法及语法分析器运行截图

下为对样例test.c文件分析的截图。

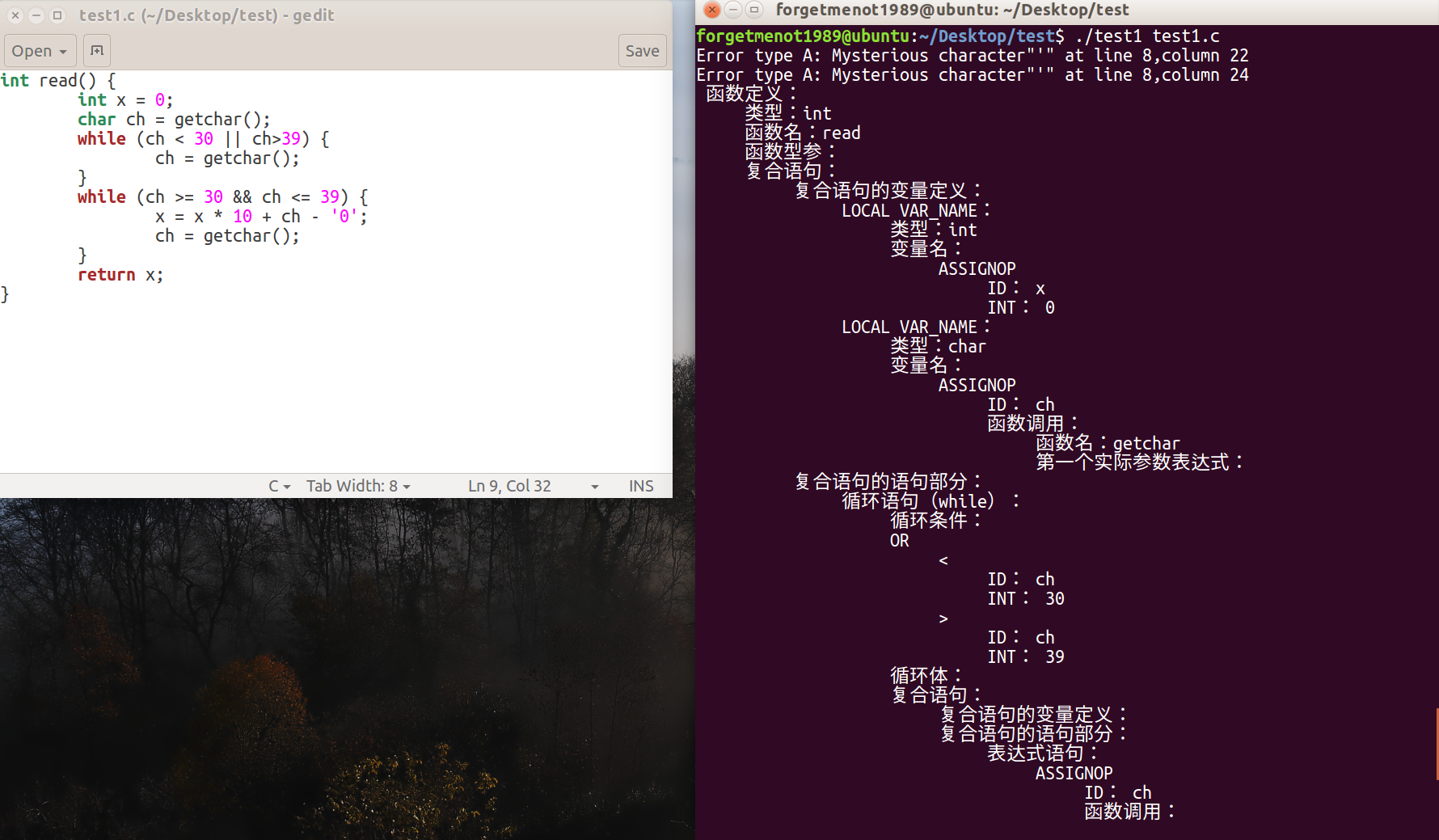


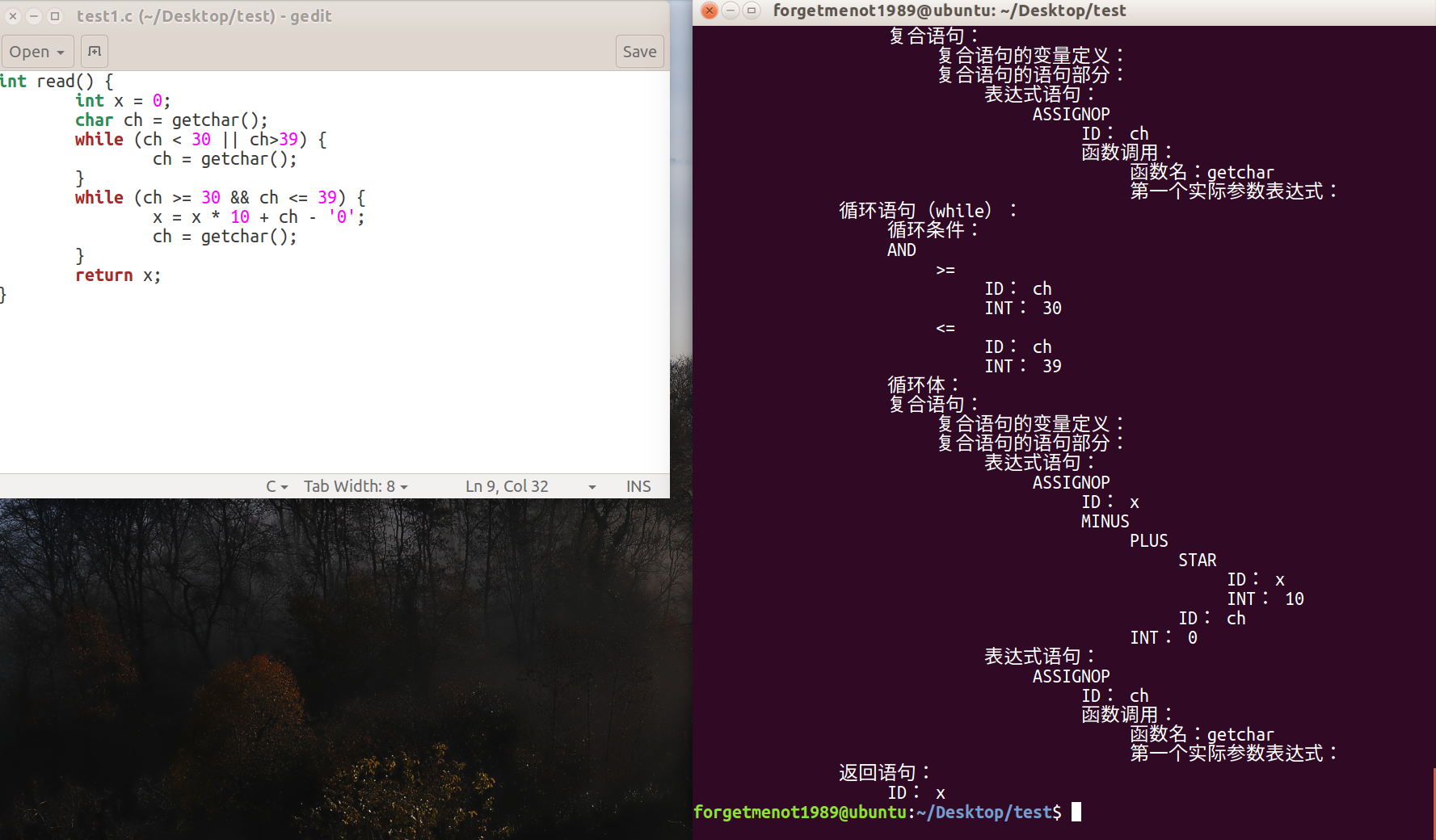






自己写一个test1.c测试





可见在遇到错误时仍然可以处理。

2.6 小结

本次实验我了解到如何用lex分析正则表达式，实现词法分析，还有利用bison写语法分析，添加新的规则可以进行错误处理。此外我们还需要display函数将分析器的分析结果显示出来。我个人觉得这个实验很难，看懂实验不容易，不过在做实验时发现自己写的test1.c竟然真的可以被分析出来还是很开心的。

本次实验只是编译原理实验的一个开始，只是学习过程的第一步，希望以后好好弄清楚其中原理，探究编译语言的乐趣。

参考文献

[1] 许畅 等编著. 《编译原理实践与指导教程》.机械工业出版社

[2] Jobn Levine著，陆军 译. 《Flex与Bison》.东南大学出版社

[3] 王生原等编著. 《编译原理（第3版）》.清华大学出版社