



燕山大学

编译原理实验报告

# Compilers Principles Experiment Report

学 院：信息科学与工程学院（软件学院）

班 级：18 级软件工程 6 班

姓 名：乔翱

学 号：201811040809

指导教师：王林 郝晓冰 李可

教 务 处

2020 年 9 月



# 目 录

|                               |           |
|-------------------------------|-----------|
| <b>实验 1 词法分析</b>              | <b>1</b>  |
| 1.1 实验目的                      | 1         |
| 1.2 实验任务                      | 1         |
| 1.3 实验内容                      | 1         |
| 1.4 心得体会                      | 10        |
| <b>实验 2 自顶向下的语法分析程序</b>       | <b>12</b> |
| 2.1 实验目的                      | 12        |
| 2.2 实验任务                      | 12        |
| 2.3 实验内容                      | 12        |
| 2.4 心得体会                      | 23        |
| <b>实验 3 基于 LR (0) 方法的语法分析</b> | <b>25</b> |
| 3.1 实验目的                      | 25        |
| 3.2 实验任务                      | 25        |
| 3.3 实验内容                      | 25        |
| 3.4 心得体会                      | 36        |
| <b>实验 4 语义分析和中间代码生成</b>       | <b>37</b> |
| 4.1 实验目的                      | 37        |
| 4.2 实验任务                      | 37        |
| 4.3 实验内容                      | 37        |
| 4.4 心得体会                      | 46        |

## 实验 1 词法分析

### 1.1 实验目的

- (1) 理解有穷自动机及其应用。
- (2) 掌握 NFA 到 DFA 的等价变换方法、DFA 最小化的方法。
- (3) 掌握设计、编码、调试词法分析程序的技术和方法。

### 1.2 实验任务

编写一个程序对输入的源代码进行词法分析，并打印分析结果。

借助词法分析工具 GNU Flex，编写一个对使用 C 语言书写的源代码进行词法分析，并使用 C 语言完成。

### 1.3 实验内容

#### 1.3.1 算法描述

对于本次实验，主要是编写 flex 源代码。而对于 flex 源代码的编写主要分为三个部分。第一个部分是定义部分，在阅读了 C 文法后，在此部分编写了 C 文法中常用的语法单元的正则表达式，包括十进制整数、十六进制整数、八进制整数、浮点数、标识符、加减乘除运算符等词法单元的正则表达式。一个正则表达式由特定字符串构成，或者由其他正则表达式通过并运算、连接运算、Kleene 闭包构成。

Flex 源代码的第二部分为规则部分，它由正则表达式和响应的响应函数构成。由于最后要输出正确的结果或者错误的信息，所以对于此部分，把匹配到的词法单元的类型、文本内容、行号、是否是未定义词法单元等信息存储到了一个结构体中，方便在主函数中输出结果

Flex 源码的第三部分是用户自定义代码部分，在这部分声明了一个结构体，用来存储词法的相关信息。在主函数中，首先判断所扫描的文法是否有错误，有错误的话就输出错误信息（错误类型和行号），没有错误才输出正确的信息。

#### 1.3.2 程序结构

此词法分析程序的结构比较简单，主要分为三部分，输入，根据词法规则匹配词法单元，输出。词法分析程序的程序结构图如下图所示：

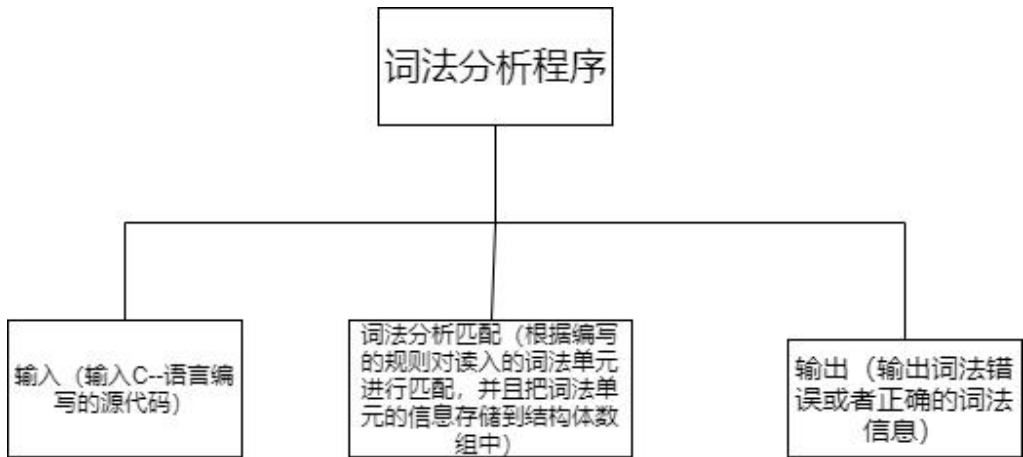


图 1.3.1 词法分析程序程序结构图

1.3.3 主要变量说明

对于本次实验程序的编写，主要的变量是一个结构体数组，这个结构体数组用来存储词法单元的信息，其中包括当前读入的词法单元，词法单元的类型，词法单元对应行号以及是否是未定义的词法的单元。

定义的结构体如下所示：

```
struct {
    char a[MAXTEXT];
    char type[MAXTYPE];
    int line;
    int flag;
} words[MAXWORD];
```

在此结构体中，char 类型的数组 a 用来存放当前读入的词法单元，char 类型的数组 type 用来存放读入的词法单元的类型，int 类型的变量 line 用来存放当前读入的词法单元的行号，int 类型的变量 flag 用来表示该词法单元是否是未定义的。

其他在本次实验中使用的主要变量，如下表所示：

表 1.1 实验一中主要变量说明

| 变量名      | 变量类型     | 含义                             |
|----------|----------|--------------------------------|
| MAXWORD  | 整型 (int) | 识别到的词法单元的最大数量。                 |
| MAXTYPE  | 整型 (int) | 识别到的词法单元的类型的最长度。               |
| MAXTEXT  | 整型 (int) | 识别到的词法单元的最大长度。                 |
| error    | 整型 (int) | 初始值为 0，用来记录是否词法分析过程中是否发现错误。    |
| num      | 整型 (int) | 初始值为 0，用来记录识别到的词法单元的个数。        |
| yytext   | char*    | Flex 内部提供的变量，表示当前词法单元所对应的词素。   |
| yylineno | 整型 (int) | Flex 内部提供的变量，表示行号，在每行结束时自动加 1。 |

### 1.3.4 程序清单

对于 flex 源码的编写主要分为三个部分，首先是定义部分，即正则表达式的编写，其中对 C 语法中的语法单元的正则表达式都进行了编写，包括整数（十进制整数、八进制整数、十六进制整数）、浮点数、运算符、标识符等，核心代码如下所示：

```

INT_DEX [1-9][0-9]*|[0]
INT_HEX [0][Xx]([1-9][0-9]*|[0])
INT_OCT [0][0-7]
FLOAT [0-9]*[.][0-9]+([eE][+-]?[0-9]*|[0])?f?
ID ([_]|a-z|A-Z|[_]|a-z|A-Z|0-9){0,32}
SEMI [;]
COMMA [,]
ASSIGNOP [=]
RELOP [>]|<|>|=|<|=|=|=|!=|^=)
PLUS [+]
MINUS [-]
STAR [*]
DIV [/]
AND [&]&
OR [|]||

```

```
DOT [.]  
NOT [!]  
TYPE int|float  
LP \  
RP \  
LB \  
RB \  
LC \  
RC \  
STRUCT struct  
RETURN return  
IF if  
ELSE else  
WHILE while  
SPACE [ \n\r]+  
NOTE \\\/.*  
ERROR_ID ([0-9][_a-zA-Z0-9]*){0,32}  
%%
```

Flex 源代码第二部分是规则部分，由正则表达式和相应的响应函数组成，响应函数中把词法单元的相关信息存储到结构体中（词法单元，词法单元的类型，词法单元对应的行号，词法单元是否是未定义的），对于识别到未定义的词法单元则需要置 error 的值为 1，方便在主函数中进行输出正确信息或者错误信息，相关的核心代码如下所示：

```
{INT_OCT} {  
    strcpy(words[num].a, yytext);  
    strcpy(words[num].type, "INT_OCT: ");  
    words[num].flag=1;  
    words[num].line=yylineno;  
    num++;  
}  
{FLOAT} {  
    strcpy(words[num].a, yytext);  
    strcpy(words[num].type, "FLOAT: ");  
    words[num].flag=1;  
    words[num].line=yylineno;  
    num++;  
}  
{TYPE} {  
    strcpy(words[num].a, yytext);
```

```
strcpy(words[num].type, "TYPE: ");
words[num].flag=1;
words[num].line=yylineno;
num++;
}
{ID} {
strcpy(words[num].a, yytext);
strcpy(words[num].type, "ID: ");
words[num].flag=1;
words[num].line=yylineno;
num++;
}
{NOTE} {
strcpy(words[num].a, yytext);
strcpy(words[num].type, "NOTE: ");
words[num].flag=1;
words[num].line=yylineno;
num++;
}
{ERROR_ID} {
strcpy(words[num].a, yytext);
strcpy(words[num].type, "ERROR_ID: ");
words[num].flag=2;
words[num].line=yylineno;
num++;
error=1;
}
. {strcpy(words[num].a, yytext);
words[num].flag=0;
words[num].line=yylineno;
num++;
error=1;
}
```

Flex 源代码的第三部分是用户自定义部分，这部分的代码包括变量、函数或者头文件以及 main 函数部分，定义变量、引入头文件的核心代码如下（相关变量的含义见 1.3.3）：

```
%{
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
```



```
#define MAXWORD 10005
#define MAXTEXT 32
#define MAXTYPE 100
int error=0;
struct {
    char a[MAXTEXT];
    char type[MAXTYPE];
    int line;
    int flag;
} words[MAXWORD];
int num=0;
%}
%option yylineno
```

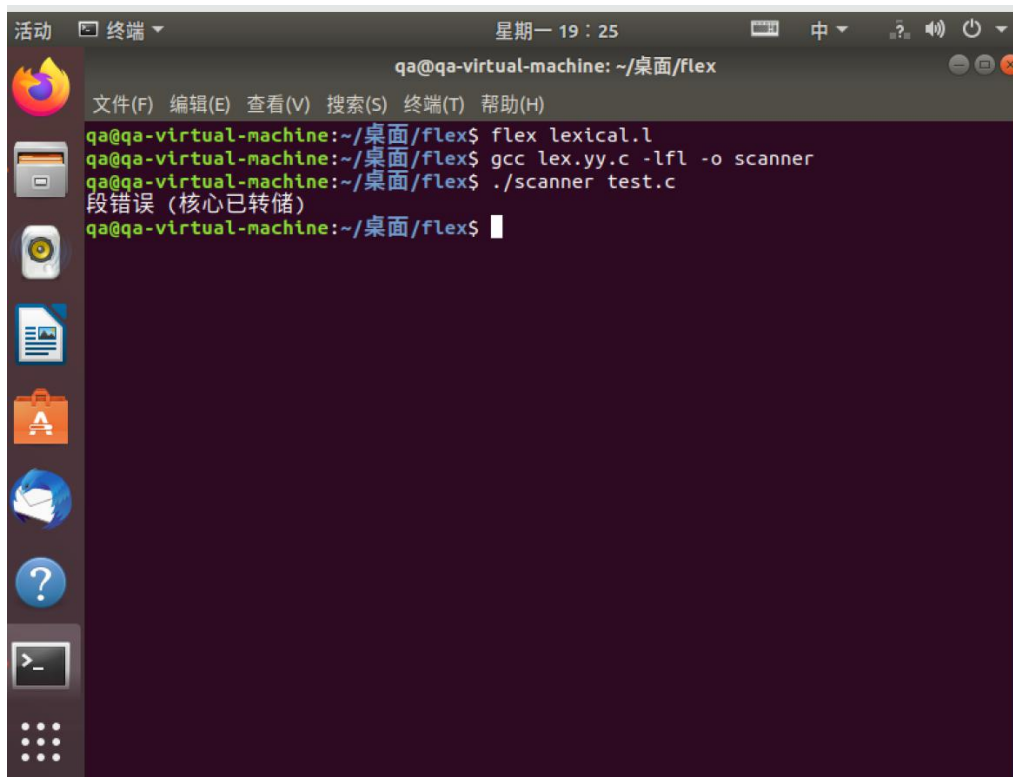
主函数中主要是先判断词法分析过程中是否识别到了未定义的词法单元，如果有则输出错误信息（Error type A），如果没有错误就输出每个词法单元及其类型，主函数的核心代码如下：

```
int main(int argc, char** argv) {
    if (argc > 1) {
        if (!(yyin = fopen(argv[1], "r"))) {
            perror(argv[1]);
            return 1;
        }
    }
    while (yylex());
    for(int i=0;i<num;i++)
    {
        if(error==1) {
            if(words[i].flag==0)
                printf("Error type A at Line %d: Mysterious characters %s\n", words[i].line, words[i].a);
            if(words[i].flag==2)
                printf("Error type A at Line %d: invalid ID %s\n", words[i].line, words[i].a);
        }
        else{
            printf("%s%s\n", words[i].type, words[i].a);
        }
    }
    return 0;
}
```

### 1.3.5 调试情况

在本次实验的调试过程中主要遇到了两个问题一个是对于某些词法单元的正则表达式的编写存在问题，导致匹配的时候出错，例如对于标识符 ID 的正则表达式的编写考虑不周，就会把某些标识符认为是未定义的词法错误。通过查阅资料，学习到了如何正确编写正则表达式，最终可以正确匹配。

另一个就是由于 C 语言中没有 string 类型，只能用 char 类型的数组来代替 string 类型，但是在利用 C 语言中的 strcpy 函数对字符串进行复制的时候遇到了问题，一直报段错误，经过查阅资料以及多次调试发现是由于未开辟空间导致的，最终成功解决此问题。



```
qa@qa-virtual-machine: ~/桌面/flex
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
qa@qa-virtual-machine:~/桌面/flex$ flex lexical.l
qa@qa-virtual-machine:~/桌面/flex$ gcc lex.yy.c -lfl -o scanner
qa@qa-virtual-machine:~/桌面/flex$ ./scanner test.c
段错误 (核心已转储)
qa@qa-virtual-machine:~/桌面/flex$
```

图 1.3.2 调试过程中出现段错误

### 1.3.6 运行结果

先利用 flex 对 lexical.l 进行编译。

```
flex lexical.l
```

编译好的结果会保存在当前目录下的 lex.yy.c 文件中，这个文件本质上就是一

份 C 语言的源代码。接着对 C 语言源代码进行编译。

```
gcc lex.yy.c -lfl -o scanner
```

输出程序命名为 scanner，利用 scanner 程序进行词法分析，假设测试文件名为 test.c。

```
./scanner test.c
```

输入多组测试用例，包括有词法错误的和没有词法错误的测试用例进行测试，运行结果截图：

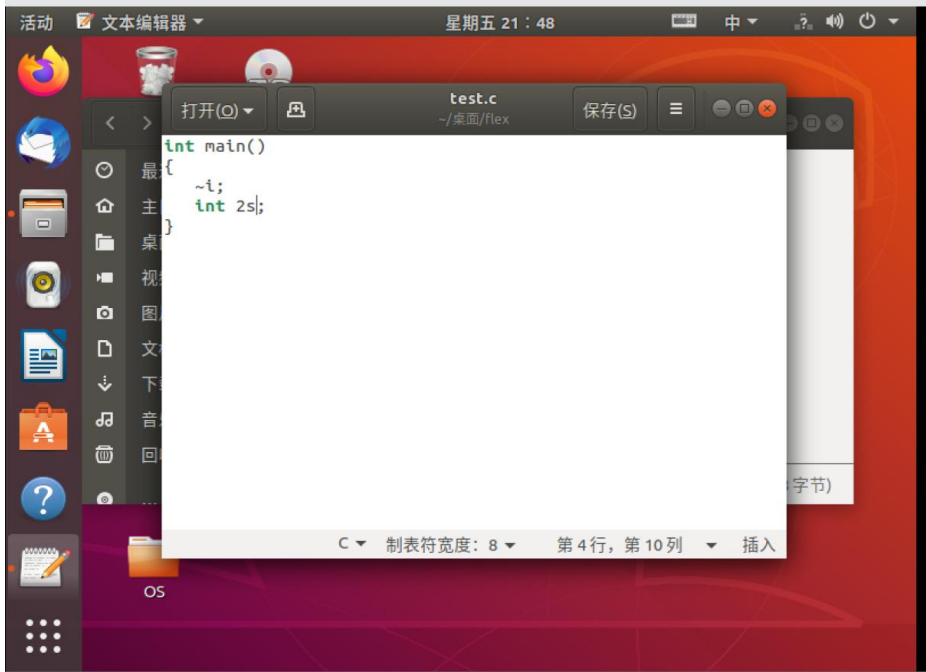


图 1.3.3 实验 1 测试用例 1（无词法错误）



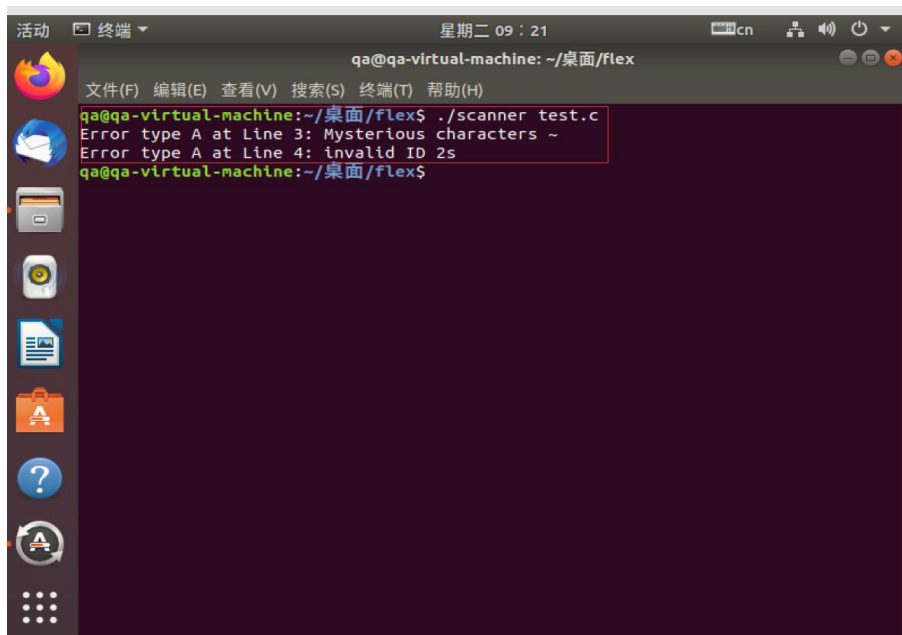
```
qa@qa-virtual-machine: ~/桌面/flex
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
qa@qa-virtual-machine:~/桌面/flex$ flex lexical.l
qa@qa-virtual-machine:~/桌面/flex$ gcc lex.yy.c -lfl -o scanner
qa@qa-virtual-machine:~/桌面/flex$ ./scanner test.c
STRUCT: struct
ID: node
LC: {
TYPE: float
ID: s
ASSIGNOP: =
FLOAT: 2.3
SEMI: ;
RC: }
SEMI: ;
TYPE: int
ID: main
LP: (
RP: )
LC: {
TYPE: int
ID: i
ASSIGNOP: =
INT_DEX: 1
SEMI: ;
NOTE: //type:int
IF: if
LP: (
ID: i
RELOP: >
```

图 1.3.4 实验 1 测试用例 1 运行结果



```
int main()
{
    ~i;
    int 2s;
}
```

图 1.3.5 实验 1 测试用例 2（有词法错误）



```
qa@qa-virtual-machine: ~/桌面/flex$ ./scanner test.c
Error type A at Line 3: Mysterious characters ~
Error type A at Line 4: invalid ID 2s
qa@qa-virtual-machine: ~/桌面/flex$
```

图 1.3.6 实验 1 测试用例 2 运行结果

## 1.4 心得体会

通过本次实验，利用 flex 编写了一个词法分析程序，加深了对词法分析的理解，了解并且掌握了如何利用 flex 编写一个词法分析程序。

词法分析的主要任务是将输入文件中的字符流组织成为词法单元流，在某些字符不符合程序设计语言此法规范时它也要有能力报告相应的错误。词法分析程序是编译器所有模块中位移读入并处理输入文件中每一个字符的模块，它使得后面的语法分析阶段能在更高抽象层次上运作，而不必纠结于字符串处理这样的细节问题。

开始写实验的时候确实无从下手，不知道怎么使用 flex 来编写这个词法分析程序，仔细阅读了实验指导书以及上网查询了相关的一系列资料有了大概的思路，然后开始尝试编写。

在词法分析程序的编写中遇到了很多问题，对于本次词法分析程序的编写，我采取的方法是把分析到的相关信息存储到一个自己定义的结构体数组中，对于其中字符串的复制操作的编写遇到了困难。因为 C 语言中没有 string 类型，只有 char 类型，对于 char 数组的复制最终经过查阅资料，了解到 C 语言中的 strcpy 函数可以用来对于 char 数组的复制。

对于本次实验还存在着一些不足，例如：能识别的字符有限，还应该改善程序能识别更多的字符。还可以对词法分析的识别结果输出到文件进行保存以便后续语法分析阶段的使用。在之后的学习中，要一步步对此词法分析程序进行完善改进。本实验是利用 flex 开发的词法分析程序，在之后的学习过程中，还要尝试不利用 flex 选择一种高级语言来开发一个词法分析程序。

总体来说，通过本次实验，收获颇多，锻炼了自己阅读资料的能力，在短时间内学会使用一种工具，锻炼了自己的编码能力，锻炼了自己的逻辑思维。

## 实验 2 自顶向下的语法分析程序

### 2.1 实验目的

- (1) 熟练掌握 LL(1) 分析表的构造方法。
- (2) 掌握设计、编制和调试典型的语法分析程序，进一步掌握常用的语法分析方法。

### 2.2 实验任务

根据 LL(1) 分析法自己编写一个语法分析程序，语言不限，文法不限。

### 2.3 实验内容

#### 2.3.1 算法描述

##### 1、输入

首先需要先输入产生式的个数，接着输入产生式，空产生式用\$代替，把输入的文法存到 LL1 类中的成员变量 grammar 中，该变量是一个结构体用来存储每条产生式。在输入的时候区分出非终结符和终结符分别存起来，并且调用成员函数 getFirst 和 getFollow 求出所有非终结符的 first 集和 follow 集保存起来。

##### 2、求非终结符的 first 集

(1)、产生式右端的第一个符号是终结符，直接把该终结符加入到左部的非终结符的 first 集中。

(2)、产生式右端的第一个符号是非终结符，则把他的 first 集的非空元素加入到产生式左部的非终结符的 first 集中。

(3)、如果产生式右部都是非终结符，并且每个非终结符的 first 集中都包含空（即产生式右部能推出空），则把空加入到产生式左部的非终结符的 first 集中。

##### 3、求非终结符的 follow 集

(1)、遍历每条产生式，如果该非终结符的后面跟着一个终结符则把该终结符加入到非终结符的 follow 集中。

(2)、遍历每条产生式，如果该非终结符的后面跟着一个非终结符，则把后面的非终结符的 first 集中非空的元素加入到该非终结符的 follow 集合中。

(3)、如果该非终结符的后面为空或者能推出空，则把产生式左部的非终结符

的 follow 集加入到该非终结符的 follow 集中。

#### 4、求预测分析表

首先先初始化预测分析表，初始值设为-1，遍历每一条产生式。

(1)、如果产生式右边的第一个符号是终结符并且不是空，就在预测分析表的相应位置记录产生式的下标。

(2) 如果产生式右边的第一个符号是空，求出产生式左部的非终结符的 follow 集，并且在 follow 集求出的元素在预测分析表的相应位置记录产生式的下标。

(3)、如果产生式右边第一个符号是非终结符，求出他的 first 集中所有非空元素中在预测表的相应位置记录下产生式的下标。

(4)、如果产生式右部能推出空，求出产生式左部的非终结符的 follow 集，并且在 follow 集求出的元素在预测分析表的相应位置记录产生式的下标。

#### 5、分析输入的句子

首先先把输入的字符串反向压入栈中，在分析栈中压入起始符号。当放剩余输入串的栈的 size 大于 0 的时候，取出两个栈的栈顶元素。

(1)、假如两个栈的栈顶元素相等并且都是“#”，分析成功，输出“Accepted!”；

(2)、假如两个栈的栈顶元素相等并且不是“#”，两个栈分别弹出栈顶元素。

(3)、假如分析栈顶弹出非终结符，存放剩余串的栈顶弹出终结符，并且在预测分析表中对应的位置不是-1，则进行推导，分析栈弹出，压入预测分析表中对应的产生式的右部，剩余输入串中也弹出一个元素。

(4)、其他情况输出“Error”。

### 2.3.2 程序结构

对于本实验采取面向对象的方法进行编写，编写了一个 LL1 类，其中包含了存储 first 集、follow 集等成员变量，还包含了 getFirst、getFollow 等方法。

程序的类图如下：



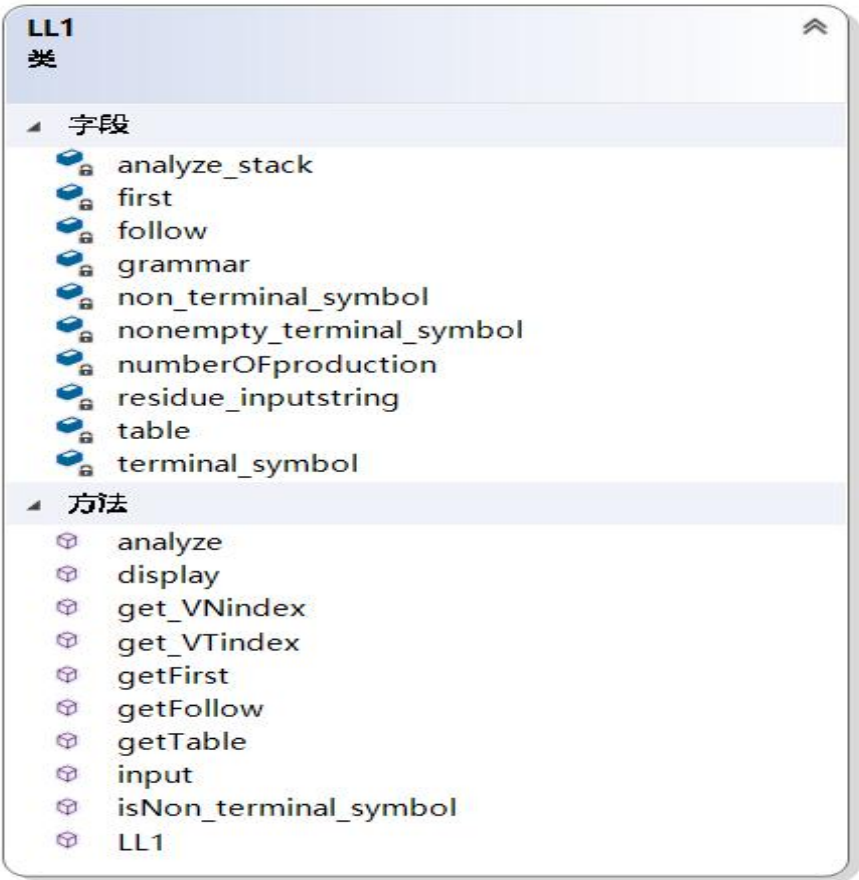


图 2.3.1 LL1 类类图

本程序的结构程序结构图如下：

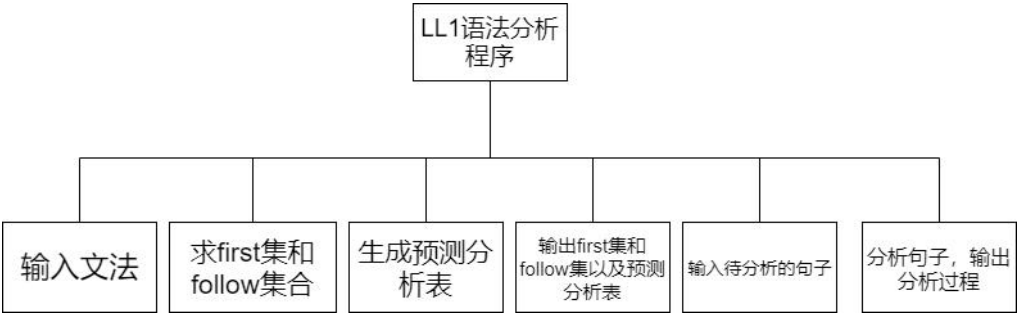


图 2.3.2 LL1 语法分析程序程序结构图

2.3.3 主要变量说明

对于本程序中的变量，主要是一个结构体用来存放文法，以及一个类，类中的成员变量用来存放 first 集和 follow 集预测分析表等，类中的成员函数用来输入、输出、求 first 集、求 follow 集、求预测分析表、分析字符串等。

存文法的结构体中有两个成员变量，一个 char 类型的变量存文法左部，试听类型的变量存文法的右部，该结构体如下所示：

```
struct Grammar
{
    char production_left;
    string production_right;
};
```

对于 LL1 类中的成员变量以及成员函数的说明如下表所示：

表 2.1 LL1 语法分析程序主要变量说明表

| 成员变量（private）                 |              |                       |
|-------------------------------|--------------|-----------------------|
| 变量名                           | 变量类型         | 说明                    |
| numberOfproduction            | int          | 文法产生式的个数              |
| terminal_symbol               | vector<char> | 终结符集合                 |
| terminal_symbol               | vector<char> | 非终结符集合                |
| nonempty_terminal_symbol      | vector<char> | 除去\$的终结符              |
| grammar[505]                  | Grammar      | 存文法                   |
| first[505]                    | set<char>    | first 集               |
| follow[505]                   | set<char>    | follow 集              |
| analyze_stack                 | vector<char> | 分析栈                   |
| residue_inputstring           | vector<char> | 剩余输入串                 |
| table[505][505]               | int          | 预测分析表                 |
| 成员函数（public）                  |              |                       |
| 函数名                           | 函数返回值类型      | 说明                    |
| isNon_terminal_symbol(char c) | bool         | 判断是否是非终结符，默认大写字母为非终结符 |
| get_VTindex(char c);          | int          | 获得 c 在终结符集合中的下标       |

|                              |      |                           |
|------------------------------|------|---------------------------|
| get_VNindex(char c)          | int  | 获得 c 在非终结符集合中的下标          |
| getFirst(char c)             | void | 得到 first 集合               |
| getFollow(char c)            | void | 得到 follow 集合              |
| input()                      | void | 输入，并且计算 first 集和 follow 集 |
| display()                    | void | 显示 first 和 follow 以及预测分析表 |
| getTable()                   | void | 得到预测分析表                   |
| analyze(string symbolstring) | void | 分析输入的字符串                  |

### 2.3.4 程序清单

求 first 集核心代码：

```
void LL1::getFirst(char c)
{
    int empty = 0;
    int flag = 0; //记录非终结符是否能推出$
    for (int i = 0; i < numberOfproduction; i++)
    {
        if (grammar[i].production_left == c)
        {
            if (!isNon_terminal_symbol(grammar[i].production_right[0]))
                //对于终结符，直接加入 first
            {
                first[get_VTindex(c)].insert(grammar[i].production_right[0]);
            }
            else
            {
                for (int j = 0; j < grammar[i].production_right.length(); j++)
                {
                    if (!isNon_terminal_symbol(grammar[i].production_right[j]))
                    {
                        first[get_VTindex(c)].insert(grammar[i].production_right[j]);
                        break;
                    }
                    getFirst(grammar[i].production_right[j]);
                }
            }
        }
    }
}
```

```

for (it = first[get_VTindex(grammar[i].production_right[j])].begin();
it != first[get_VTindex(grammar[i].production_right[j])].end(); it++)
    {
        if (*it == '$')
            flag = 1;
        else
            first[get_VTindex(c)].insert(*it); //非$就加入 FIRST
    }
    if (flag == 0)
        break;
    else
    {
        empty ++;
        flag = 0;
    }
}
if (empty == grammar[i].production_right.length()) //产生式右部能推出$
    first[get_VTindex(c)].insert('$');
}
}
}
}

```

求 follow 集核心代码:

```

void LL1::getFollow(char c) //求非终结符的 follow 集 (改进版本)
{
    for (int i = 0; i < numberOfproduction; i++)
    {
        int index = -1;
        int len = grammar[i].production_right.length();
        for (int j = 0; j < len; j++)
        {
            if (grammar[i].production_right[j] == c)
            {
                index = j;
                break;
            }
        }
    }
}

```

```
    }  
}  
if (index != -1 && index < len - 1)  
{  
    char next = grammar[i].production_right[index + 1];  
    if (!isNon_terminal_symbol(next))//如果后面的是终结符，直接加入  
    {  
        follow[get_VTindex(c)].insert(next);  
    }  
    else//把后面的非终结符的 first 集除了$加入  
    {  
        int flag = 0;  
        set<char>::iterator it;  
        for (it = first[get_VTindex(next)].begin(); it !=  
first[get_VTindex(next)].end(); it++)  
        {  
            if (*it == '$')  
                flag = 1;  
            else  
                follow[get_VTindex(c)].insert(*it);  
        }  
        if (flag==1 && grammar[i].production_left != c)  
        {  
            getFollow(grammar[i].production_left);  
            set<char>::iterator it;  
            char tmp = grammar[i].production_left;  
            for (it = follow[get_VTindex(tmp)].begin(); it !=  
follow[get_VTindex(tmp)].end(); it++)  
                follow[get_VTindex(c)].insert(*it);  
        }  
    }  
}
```

```

    }
    else if (index != -1 && index == len - 1 && c !=
grammar[i].production_left)
    {
        getFollow(grammar[i].production_left);
        set<char>::iterator it;
        char temp = grammar[i].production_left;
for      (it      =      follow[get_VTindex(temp)].begin();      it      !=
follow[get_VTindex(temp)].end(); it++)
            follow[get_VTindex(c)].insert(*it);
    }
}
}
}

```

求预测分析表核心代码:

```

void LL1::getTable()
{
    memset(table, -1, sizeof(table));
    for (int i = 0; i < numberOfproduction; i++)
    {
        char temp = grammar[i].production_right[0];
        if (!isNon_terminal_symbol(temp))//产生式右边是一个终结符
        {
            if (temp != '$')
table[get_VTindex(grammar[i].production_left)][get_VNindex(temp)] = i;
            if (temp == '$')
            {
                set<char>::iterator it;
                for (it = follow[get_VTindex(grammar[i].production_left)].begin();
it != follow[get_VTindex(grammar[i].production_left)].end(); it++)
                    {
                        table[get_VTindex(grammar[i].production_left)][get_VNindex(*it)] = i;
                    }
            }
        }
        else//产生式右边是一个非终结符
        {

```

```

        set<char>::iterator ti;
        for (ti = first[get_VTindex(temp)].begin(); ti !=
first[get_VTindex(temp)].end(); ti++)
        {
            table[get_VTindex(grammar[i].production_left)][get_VNindex(*ti)] = i;
        }
        if (first[get_VTindex(temp)].count('$') != 0)
        {
            set<char>::iterator it;
            for(it= follow[get_VTindex(grammar[i].production_left)].begin();
it != follow[get_VTindex(grammar[i].production_left)].end(); it++)
            {
                table[get_VTindex(grammar[i].production_left)][get_VNindex(*it)] = i;
            }
        }
    }
}

```

分析字符串核心代码:

```

void LL1::analyze(string symbolstring)
{
    cout<<"*****          分    析    字    符    串
*****"<<endl;
    cout << setw(15) << "分析栈" << setw(15) << "剩余输入串" << setw(15) <<
"产生式" << endl;
    for (int i = symbolstring.length() - 1; i >= 0; i--)
        residue_inputstring.push_back(symbolstring[i]);
    analyze_stack.push_back('#');
    analyze_stack.push_back(non_terminal_symbol[0]);
    while (residue_inputstring.size() > 0)
    {
        string outs = "";
        for (int i = 0; i < analyze_stack.size(); i++)
            outs += analyze_stack[i];
        cout << setw(15) << outs;
        outs = "";
        for (int i = residue_inputstring.size() - 1; i >= 0; i--)
            outs += residue_inputstring[i];
        cout << setw(15) << outs;
        char c1 = analyze_stack[analyze_stack.size() - 1];
        char c2 = residue_inputstring[residue_inputstring.size() - 1];
    }
}

```

```

        if (c1 == c2 && c1 == '#')
        {
            cout << setw(15) << "Accepted!" << endl;
            return;
        }
        if (c1 == c2)
        {
            analyze_stack.pop_back();
            residue_inputstring.pop_back();
            cout << setw(15) << c1 << "匹配" << endl;
        }
        else if (table[get_VTindex(c1)][get_VNindex(c2)] != -1)
        {
            int index= table[get_VTindex(c1)][get_VNindex(c2)];
            analyze_stack.pop_back();
            if (grammar[index].production_right != "$")
            {
                for (int i = grammar[index].production_right.length() - 1;
i >= 0; i--)

                analyze_stack.push_back(grammar[index].production_right[i]);
            }
            cout << setw(15) << grammar[index].production_right << endl;
        }
        else
        {
            cout << setw(15) << "Error!" << endl;
            return;
        }
    }
}

```

### 2.3.5 调试情况

对于本次 LL1 语法分析程序的编写，难点主要是求 first 集和求 follow 集，对于 follow 集的求取，在开始的时候考虑不全，没有考虑到文法右部该非终结符后的非终结符能够推出空的情况，导致后面的预测分析表以及分析字符串都出错。经过反复地多次调试才改正了这个错误。使得求出的非终结符的 follow 集正确。

对于本程序有一些代码的编写有些繁琐，还有待改进，比如对于 first 集和 follow 集的存储可以考虑采用 map 来存，这样比用数组来存更加方便简洁，这也是



在调试过程中发现的一个问题。

### 2.3.6 运行结果

输入了多组文法和字符串对程序进行测试，部分测试运行结果如下所示：

```

F:\编译实验\18级编译原理第一次实验课资料\18软六-乔翊-201811040809\实验2\LL1.exe
S->AT
A->BU
T->$
U->*BU
U->$
B->(S)
B->m
T->+AT
*****FIRST集合*****
S: ( m
A: ( m
T: $ +
B: ( m
U: $ *
*****FOLLOW集合*****
S: # )
A: # ) +
T: # )
B: # ) * +
U: # ) +
*****预测分析表*****
          $      *      (      )      m      +      #
S:      Error   Error   AT    Error   AT    Error   Error
A:      Error   Error   BU    Error   BU    Error   Error
T:      Error   Error   Error $    Error   +AT   $
B:      Error   Error   (S)   Error   m    Error   Error
U:      Error   *BU    Error   $    Error   $      $
请输入字符串:
m+m*m#
分析栈      剩余输入串      产生式
#S          m+m*m#         AT
#TA          m+m*m#         BU
#TUB         m+m*m#         m
#TUm         m+m*m#         m匹配
#TU          +m*m#         $
#T           +m*m#         +AT
#TA+         +m*m#         +匹配
#TA          m*m#          BU
#TUB         m*m#          m
#TUm         m*m#          m匹配
#TU          *m#           *BU
#TUB*        *m#           *匹配
#TUB         m#            m
#TUm         m#            m匹配
#TU          #             $
#T           #             $
#            #             Accepted!

```

图 2.3.3 LL1 语法分析程序测试用例 1 运行结果

```

F:\编译实验\18级编译原理第一次实验课资料\18软六-乔翱-201811040809\实验2\LL1.exe
输入的产生的个数:
8
请输入产生式:
E->TK
K->+TK
K->$
T->FM
M->*FM
M->$
F->i
F->(E)
*****FIRST集合*****
E: ( i
T: ( i
K: $ +
F: ( i
M: $ *
*****FOLLOW集合*****
E: # )
T: # ) +
K: # )
F: # ) * +
M: # ) +
*****预测分析表*****
          +      $      *      i      (      )      #
E:      Error   Error   Error   TK      TK      Error   Error
T:      Error   Error   Error   FM      FM      Error   Error
K:      +TK     Error   Error   Error   Error   $      $
F:      Error   Error   Error   i      (E)     Error   Error
M:      $      Error   *FM   Error   Error   $      $
请输入字符串:
i+ii*
          分析栈      剩余输入串      产生式
          #E          i+ii*          TK
          #KT          i+ii*          FM
          #KMF          i+ii*          i
          #KMi          i+ii*          i匹配
          #KM           +ii*          $
          #K            +ii*          +TK
          #KT+          +ii*          +匹配
          #KT            ii*          FM
          #KMF            ii*          i
          #KMi            ii*          i匹配
          #KM              i*          Error!

```

图 2.3.4 LL1 语法分析程序测试用例 2 运行结果

## 2.4 心得体会

本次实验，采用 C++ 语言编写了一个 LL1 语法分析器，采用面向对象的方法，编写了一个类，进行对词法的分析。

通过本次实验，对于 LL1 文法有了更深刻的认识，并且对于自顶向下的语法分析过程的理解有了进一步的提升。在此次实验的编写过程中，对于 FIRST 集和 FOLLOW

集的求取，在开始的时候遇到了一些问题，尤其是对于 FOLLOW 集合的求取，在开始的时候对于某些文法求取非终结符的 FOLLOW 集总是出现错误，对 FOLLOW 集的理解不是很深刻，导致某些情况下的 FOLLOW 集没有求，后来经过上网查询资料、查询书籍以及自己的思考，逐渐明白了求解的过程，在求 FOLLOW 集时全面考虑，不漏掉某些情况。最终正确求出 FIRST 集和 FOLLOW 集后，使用二者成功构建预测分析表，并且判断输入的句子是否是该文法的句子。对于实验的要求全部完成。

对于本次实验编写的 LL1 分析器，还存在着一些问题以及有待改进的地方，例如对于含左公因子以及左递归的文法不能消除左递归和左公因子。还有对于界面的设计不太美观，可以考虑使用 MFC 进行一个界面的简单设计开发。这些问题是在之后的学习过程中需要进一步深入研究改进的。

总的来说，通过这次实验，收获还是很大的，对 LL1 文法分析有了更加深刻的认识，并且对 C++ 语言的使用也有了很大的提高，锻炼了自己的动手能力，提高了自己的编程水平。但是还是存在很多的不足，这些需要自己总结经验，在接下来的学习中弥补这些不足，提高自己的能力。

## 实验 3 基于 LR (0) 方法的语法分析

### 3.1 实验目的

- (1) 掌握 LR(0) 分析表的构造方法。
- (2) 掌握设计、编制和调试典型的语法分析程序，进一步掌握常用的语法分析方法。
- (3) 理解语法分析在编译程序中的作用。

### 3.2 实验任务

借助语法分析工具 GNU Bison，编写一个对使用 C++ 语言书写的源代码进行语法分析（C++ 语言的文法参见附录 A），并使用 C 语言完成。

### 3.3 实验内容

#### 3.3.1 算法描述

##### 1、编写 Bison 源代码

Bison 源代码主要分为三个部分。

第一部分是定义部分，在这部分声明了 C++ 语法中的词法单元的定义，终结符都定义为 %token，没有定义为 %token 的为非终结符。并且需要声明一个 %union {} 将所有的可能的类型都包含进去。在 %token 部分用 <> 声明每一个词法单元对应的属性值的类型为结构体类型，此结构体为自己编写的树节点的结构体，方便之后进行树的建立和输出。对于非终结符使用 %type 加上尖括号的办法确定它们的类型。当所有需要确定类型的符号的类型都被定下来之后，规则部分里的 \$\$、\$1 等就自动地带有了相应的类型，不再需要显示地为其指定类型了。最后在这部分还需要声明终结符的优先级，对终结符的结合性进行规定。

第二部分是规则部分，在此部分声明了具体的语法和相应的语义动作。在此部分书写每一条产生式以及对应的语义动作，语义动作会在每条产生式归约后执行，这部分的语义动作都是生成一个节点，为了之后建立语法分析树以及输出语法分析树来做准备。对于语法错误，也在这部分声明，语法错误的产生式的语义动作中，调用自己编写的 grammarerror 函数，用来之后在主函数中输出语法错误，并且用 error 来记录是否有语法错误以及语法错误的数量。

第三部分是用户函数部分，这部分主要声明了引用的头文件以及一些需要用到的变量，例如 `error` 变量用来记录语法错误。

## 2、修改 flex 源代码

语法分析阶段的输入可以是之前 flex 编写的词法分析程序的输出的词法单元，只需要修改之前编写的 flex 源代码，在之前编写的规则的响应函数中都返回相应的词法单元。并且需要为每个匹配到的终结符建立一个节点。由于在 bison 源代码中进行了对词法单元属性值的类型的修改，所以直接调用建立语法树的函数来建立叶子节点。

## 3、构建语法树以及遍历语法树

对于此部分构建语法树的算法经过多次尝试并没有很好的思路，最终是参考 CSDN 中的一篇博客中的代码进行改进。首先先是声明一个结构体代表语法树的节点，此结构体用来存储行号、语法单元的名字、左孩子、右孩子、以及用来存放 ID/TYPE/INTEGER/FLOAT 结点的值

建立语法分析树的函数使用的是变长参数列表，第一个参数是语法单元的名字，第二个参数是参数的个数，假如为 0，代表的是终结符或者空的语法单元，第一个变长参数代表行号（产生空的语法单元行号为-1），假如不为 0，代表的是非终结符，变长参数均为语法分析树的节点，取变长参数第一个节点设为当前节点的左孩子，其他节点依次设置成兄弟节点。

先序遍历语法树进行输出，输出时注意子节点相对父节点需要缩进两个空格，产生空的语法单元不需要打印，打印语法单元的名字，如果是 ID/TYPE/INTEGER 语法单元，还需要打印 `yytext` 的值。完后递归调用，遍历左子树和右子树。

## 4、错误处理

Bison 函数中自带有 `yyerror` 函数可以用来处理错误，但是此函数并不能满足本实验的要求，于是编写了 `gramerror` 函数用来处理错误，当发生词法错误的时候调用此函数，此函数主要用来输出错误类型（Error type B）、错误发生的行号以及说明文字。

### 3.3.2 程序结构

该程序的结构比较简单，程序结构图如下图所示：

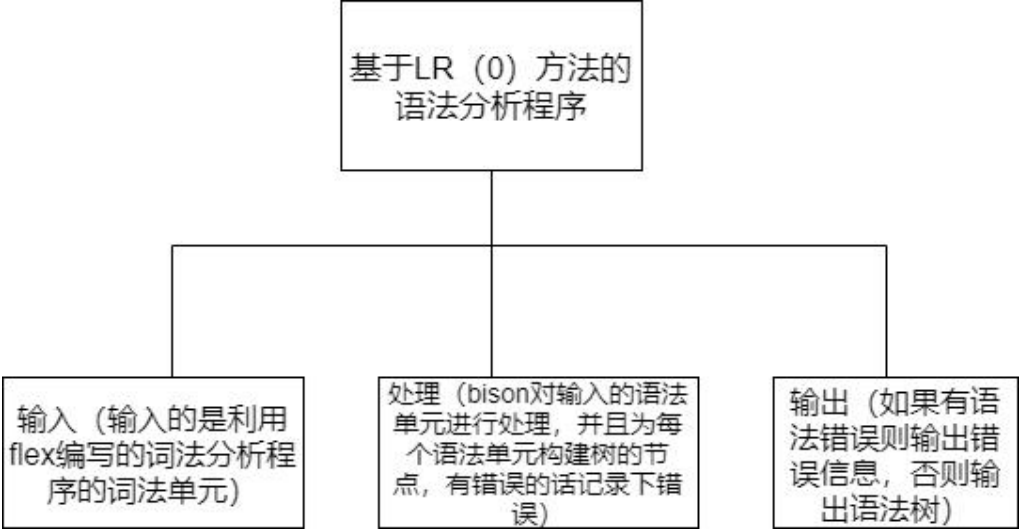


图 3.3.1 基于 LR (0) 语法分析程序程序结构图

3.3.3 主要变量说明

本次实验的主要变量是建立数时使用的结构体，对于该部分的编写，自己尝试了很久都没能正确运行出程序，最终借鉴了 CSDN 上的一篇博客。

语法树节点的结构体主要用来存储行号、语法单元的名字、左孩子、右孩子以及一个联合体用来存放 ID/TYPE/INTEGER/FLOAT 结点的值，结构体声明如下：

```
struct ast
{
    int line; //行号
    char* name; //语法单元的名字
    struct ast *l; //左孩子
    struct ast *r; //右孩子
    union //用来存放 ID/TYPE/INTEGER/FLOAT 结点的值
    {
        char* type;
        int i;
        float f;
    };
};
```

另外，在 bison 源代码中声明 error 变量，用来记录是否有语法错误以及语法

错误的个数，方便在主函数进行输出，如果 error 为 0，输出语法树，否则，输出语法错误的信息。

### 3.3.4 程序清单

Flex 部分源代码 (lexical.y) 如下：

```
%{
#include "stdio.h"
#include "stdlib.h"
# include "gramtree.h"
#include "syntax.tab.h"
%}

%option yylineno
TYPE int|float
PLUS \+
MINUS -
INTEGER [1-9]+[0-9]*|0
FLOAT [0-9]+\.[0-9]*
ID [a-zA-Z][a-zA-Z_0-9]*
SPACE [ \t\r]*
EOL \n
SEMI ;
COMMA ,
ASSIGNOP =
RELOP >|<|>=|<=|==|!=
STAR \*
DIV \/

%%

int|float {yylval.a=newast("TYPE",0,yylineno);return TYPE;}
{PLUS} {yylval.a=newast("PLUS",0,yylineno); return PLUS;}
{MINUS} {yylval.a=newast("MINUS",0,yylineno); return MINUS;}
{INTEGER} {yylval.a=newast("INTEGER",0,yylineno); return INTEGER;}
{ID} {yylval.a=newast("ID",0,yylineno); return ID;}
```

```

{SPACE} {}
{EOL} {}
{SEMI} {yyval.a=newast("SEMI",0,yylineno); return SEMI;}
{COMMA} {yyval.a=newast("COMMA",0,yylineno); return COMMA;}
{ASSIGNOP} {yyval.a=newast("ASSIGNOP",0,yylineno); return ASSIGNOP;}
{RELOP} {yyval.a=newast("RELOP",0,yylineno); return RELOP;}
{STAR} {yyval.a=newast("STAR",0,yylineno); return STAR;}
{DIV} {yyval.a=newast("DIV",0,yylineno); return DIV;}
. { printf("Error type A at line %d: Mystirious charachter
' %s' \n",yylineno,yytext);}
%%
int yywrap()
{
    return 1;
}

```

bison 部分源代码 (syntax.y) :

```

%{
#include<unistd.h>
#include<stdio.h>
#include "gramtree.h"
int error=0;
%}
%union{
struct ast* a;
double d;
}
%token <a> INTEGER FLOAT
%token <a> TYPE STRUCT RETURN IF ELSE WHILE ID SPACE SEMI COMMA ASSIGNOP RELOP
PLUS
MINUS STAR DIV AND OR DOT NOT LP RP LB RB LC RC AERROR
%token <a> EOL

```



```

%type <a> Program ExtDefList ExtDef ExtDeclList Specifire StructSpecifire
OptTag Tag VarDec FunDec VarList ParamDec Compst StmtList Stmt DefList Def
DeclList Dec Exp Args
%%
Program:|ExtDefList {$$=newast("Program",1,$1);
if(error==0){eval($$,0);}}      ;
FunDec:ID LP VarList RP {$$=newast("FunDec",4,$1,$2,$3,$4);}
      |ID error VarList RP{gramerror("(");error++;}
      |ID LP VarList error{gramerror(")");error++;}
      |ID LP RP {$$=newast("FunDec",3,$1,$2,$3);}
      |ID error RP {gramerror("(");error++;}
      |ID LP error {gramerror(")");error++;}    ;
Stmt:Exp SEMI {$$=newast("Stmt",2,$1,$2);}
      |Exp error{gramerror(";");error++;}
      |Compst {$$=newast("Stmt",1,$1);}
      |RETURN Exp SEMI {$$=newast("Stmt",3,$1,$2,$3);}
      |IF LP Exp RP Stmt {$$=newast("Stmt",5,$1,$2,$3,$4,$5);}
      |IF          LP          Exp          RP          Stmt          ELSE          Stmt
{$$=newast("Stmt",7,$1,$2,$3,$4,$5,$6,$7);}
      |IF error Exp RP Stmt {gramerror("(");error++;}
      |IF LP Exp error Stmt ELSE Stmt {gramerror(")");error++;}
      |WHILE LP Exp RP Stmt {$$=newast("Stmt",5,$1,$2,$3,$4,$5);}
      |WHILE error Exp RP Stmt{ gramerror("(");error++;}
      |WHILE LP Exp error Stmt{ gramerror(")");error++;}    ;
Exp:Exp ASSIGNOP Exp{$$=newast("Exp",3,$1,$2,$3);}
      |Exp AND Exp{$$=newast("Exp",3,$1,$2,$3);}
      |Exp OR Exp{$$=newast("Exp",3,$1,$2,$3);}
      |Exp RELOP Exp{$$=newast("Exp",3,$1,$2,$3);}
      |Exp PLUS Exp{$$=newast("Exp",3,$1,$2,$3);}
      |Exp MINUS Exp{$$=newast("Exp",3,$1,$2,$3);}
      |Exp STAR Exp{$$=newast("Exp",3,$1,$2,$3);}

```

```

|Exp DIV Exp {$$=newast("Exp", 3, $1, $2, $3);}
|LP Exp RP {$$=newast("Exp", 3, $1, $2, $3);}
|MINUS Exp {$$=newast("Exp", 2, $1, $2);}
|NOT Exp {$$=newast("Exp", 2, $1, $2);}
|ID LP Args RP {$$=newast("Exp", 4, $1, $2, $3, $4);}
|ID LP RP {$$=newast("Exp", 3, $1, $2, $3);}
|Exp LB Exp RB {$$=newast("Exp", 4, $1, $2, $3, $4);}
|Exp DOT ID {$$=newast("Exp", 3, $1, $2, $3);}
|ID {$$=newast("Exp", 1, $1);}
|INTEGER {$$=newast("Exp", 1, $1);}
|FLOAT {$$=newast("Exp", 1, $1);}          ;

%%

```

建立多叉树的头文件 (gramtree.h) :

```

extern int yylineno;
extern char* yytext;
/*语法树结点*/
struct ast
{
    int line;
    char* name;
    struct ast *l;
    struct ast *r;
    union
    {
        char* type;
        int i;
        float f;
    };
};

struct ast *newast(char* name, int num, ...);
void eval(struct ast*, int level);

```

主函数:

```
# include<stdio.h>
# include<stdlib.h>
# include<stdarg.h>
# include"gramtree.h"
# include"syntax.tab.h"
int i;
struct ast *newast(char* name,int num,...)//语法树建立
{
    va_list valist; //变长参数列表
    struct ast *a=(struct ast*)malloc(sizeof(struct ast));//新生成的父节点
    struct ast *temp=(struct ast*)malloc(sizeof(struct ast));
    a->name=name;//语法单元名字
    va_start(valist,num);//初始化变长参数为 num 后的参数
    if(num>0)//num>0 代表当前词法单元为非终结符，变长参数为语法树结点
    {
        temp=va_arg(valist, struct ast*);
        a->l=temp;
        a->line=temp->line;
        if(num>=2)
        {
            for(i=0; i<num-1; ++i)          {
                temp->r=va_arg(valist,struct ast*);
                temp=temp->r;
            }
        }
    }
    else //当前词法单元为终结符或空
    {
        int t=va_arg(valist, int);
        a->line=t;
    }
}
```

```

        if ((!strcmp(a->name, "ID")) || (!strcmp(a->name, "TYPE")))
{char*t;t=(char*)malloc(sizeof(char* )*40);strcpy(t, yytext);a->type=t;}
        else if(!strcmp(a->name, "INTEGER")) {a->i=atoi(yytext);}
        else {}
    }
    return a;
}

void eval(struct ast *a,int level)//先序遍历抽象语法树
{
    if(a!=NULL)
    {
        for(i=0; i<level; ++i)//孩子结点相对父节点缩进 2 个空格
            printf("  ");
        if(a->line!=-1){ //产生空的语法单元不需要打印信息
printf("%s",a->name);
if ((!strcmp(a->name, "ID")) || (!strcmp(a->name, "TYPE")))printf(":%s",a->type);
            else if(!strcmp(a->name, "INTEGER"))printf(":%d",a->i);
            else
                printf("(%d)",a->line);
        }
        printf("\n");
        eval(a->l, level+1);//遍历左子树
        eval(a->r, level);//遍历右子树
    }
}

void gramerror(char*s)//自己编写的错误处理函数，输出错误 (Error type B)
{
printf("Error type B at line %d:Missing : %s\n",yylineno,s);
}

int main()

```

```
{  
  
    return yyparse();  
  
}
```

### 3.3.5 调试情况

本次实验对于 bison 的掌握并不是特别难，难点在于语法树的建立，对于数据结构的掌握并不是很好，所以在做实验时遇到了很大的困难，调试了很久都不能正确输出语法树，只能输出错误的语法信息，最终是借鉴 CSDN 中的一篇博客进行语法树代码的编写，从中学习到了如何建立多叉树以及插入节点，本语法树是自底向上建树的。从中还学到了如何使用变长参数列表。对于变长参数之前只是在 Java 中接触过，并没有在 C 语言或者 C++ 中接触过。通过这篇博客也学会了在 C 语言中使用变长参数列表。

### 3.3.6 运行结果

分别输入多组测试样例（有语法错误的和没有语法错误的）进行测试，部分运行截图如下所示：

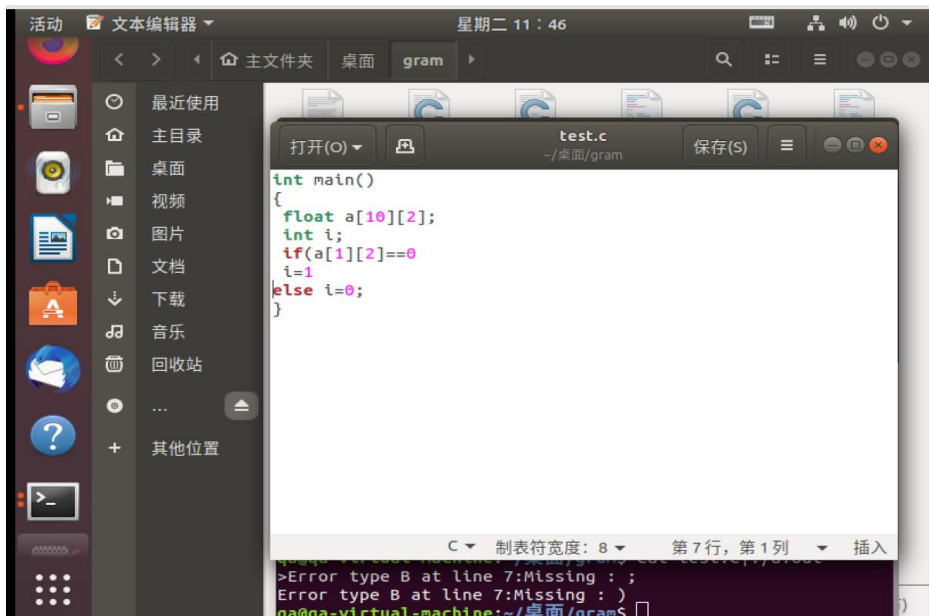


图 3.3.2 LR0 语法分析程序测试用例 1 运行结果



图 3.3.3 LR0 语法分析程序测试用例 2

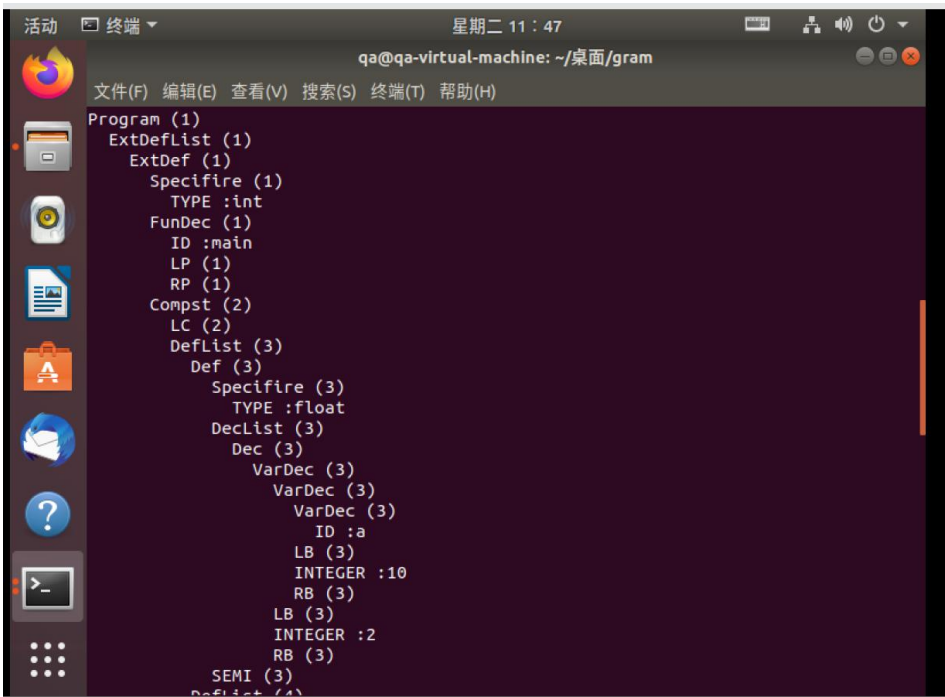


图 3.3.4 LR0 语法分析程序测试用例 2 运行结果

### 3.4 心得体会

语法分析程序的主要任务是读入词法单元流、判断输入程序是否匹配程序设计语言的语法规则，并在匹配规范的情况下构建起输入程序的静态结构。语法分析使得编译器的后续阶段看到的输入程序不再是一串字符流或者单词流，而是一个结构整齐、处理方便的数据对象。

通过编写此语法分析程序，对自底向上的语法分析程序有了更好的理解，本实验采取 bison 进行开发，主要是掌握了如何使用 bison 开发工具，bison 开发工具的掌握较为简单，难点在于语法分析树的建立。对于语法分析树的建立采取了很多办法都没有解决，最终上网查资料，参考了 CSDN 上的一篇博客建立语法分析树。对于多叉树的创建插入掌握不扎实。通过此实验也复习了数据结构中多叉树的相关知识，对于多叉树的知识很多都已经忘记了，也导致这次实验未能自己独立编写出多叉树，在以后的学习过程中要吸取经验教训，多多复习数据结构的知识，多多动手编写程序，而不是只看，锻炼自己的动手能力，提高自己的编程水平。

总之在本次实验中，对自底向上的语法分析程序有了进一步的深刻理解，并且对于多叉树的建立和插入以及先序遍历复习了一下，在之后的学习过程中还需要对此程序进行完善改进。

## 实验 4 语义分析和中间代码生成

### 4.1 实验目的

- (1) 熟悉语义分析和中间代码生成过程。
- (2) 加深对语义翻译的理解。

### 4.2 实验任务

此次实验任务有两个：首先是语义分析任务，即在词法分析和语法分析程序的基础上编写一个程序，对输入的源代码进行语义分析和类型检查，并打印分析结果；然后，在语义正确的基础上，实现一种中间代码的生成。

任务 1：审查每一个语法结构的静态语义，即验证语法正确的结构是否有意义。此部分不再借助已有工具，需手写代码来完成。

任务 2：在词法分析、语法分析和语义分析程序的基础上，将输入源代码翻译成中间代码。

编写一个中间代码生成程序，能将算术表达式等翻译成逆波兰、三元组或四元组形式（选择其中一种形式即可）

### 4.3 实验内容

#### 4.3.1 算法描述

##### 1、静态语义分析

对于静态语义分析，主要是对于符号表的建立以及查询，在实验三的基础上进行修改，定义了变量符号表、函数符号表、数组符号表、结构体符号表，以及创立各个符号表和对于各个符号表的查询。对于该符号表采用链表这种数据结构，设置头尾指针连接各个节点，进行创建和查询。并且修改 bison 源代码，在每个语义动作中查询符号表验证是否有重定义等语义错误。

##### 2、将算术表达式转换成逆波兰表达式

逆波兰表达式是运算对象在前，运算符在后面，也就是后缀表达式形式。对于前缀表达式转换成后缀表达式采用栈来实现。对于逆波兰表达式更有利于计算机程序进行计算，扫描后缀表达式，如果扫描到的是运算对象，就压入栈中，如果扫描到的是单目运算符，对栈顶元素，执行该运算，将运算结果代替该元素进栈，如果扫



描到的是双目运算符，则对栈顶的两个运算对象进行该运算并且将运算结果代替这两个运算对象进栈，最后的运算结果就是栈顶元素。

前缀表达式转换后缀表达式的算法如下：

扫描输入的字符串。

- (1)、如果当前字符是数字或者字母，就把该数字或者字母加到结果中；
- (2)、如果当前字符是左括号则压入栈中；
- (3)、如果当前字符是右括号则匹配之前压入的左括号，并且把这一对括号删掉；
- (4)、如果当前字符的优先级高于栈顶字符的优先级则压栈；
- (5)、如果当前字符的优先级低于或者等于栈顶字符的优先级，则把栈顶元素加到结果中，当前字符入栈。

最后把栈里面的元素加到结果中，返回结果。

#### 4.3.2 程序结构

对于语义分析程序，需要在每个语义动作中检查是否有重定义等语义错误，该程序的程序结构图如下图所示：

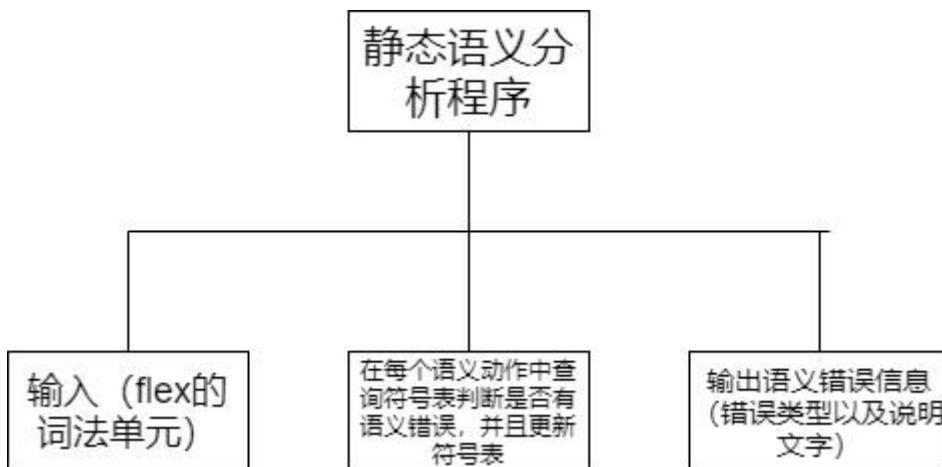


图 4.3.1 语义分析程序程序结构图

对于将算术表达式转换为逆波兰表达式的程序，程序结构较为简单，只需要利用栈这种数据结构就能轻松完成该程序。

#### 4.3.3 主要变量说明

语义分析程序中的变量主要是符号表节点的结构体，各结构体如下所示（变量

含义在注释中说明)：

```
/*变量符号表的结点*/
struct var
{ char* name;//变量名
  char* type;//变量类型
  struct var *next;//指针 }*varhead,*vartail;
/*函数符号表的结点*/
struct func {
  int tag;//0 表示未定义，1 表示定义
  char* name;//函数名
  char* type;//函数类型
  char* rtype;//实际返回值类型
  int pnum;//形参数个数
  struct func *next; }*funchead,*functail;
  int rpnum;//记录函数实参个数
/*数组符号表的结点*/
struct array
{ char* name;//数组名
  char* type;//数组类型
  struct array *next; }*arrayhead,*arraytail;
/*结构体符号表的结点*/
struct struc {
  char* name;//结构体名
  char* type;//数组类型
  struct struc *next;
}*strucehead,*structail;
```

将算术表达式转换成逆波兰表达式程序中的变量如下表所示：

表 4.3 逆波兰表达式程序中的主要变量

| 变量名                     | 类型           | 含义             |
|-------------------------|--------------|----------------|
| s                       | stack <char> | 在转后缀表达式的时候存放字符 |
| inversePolandexpression | string       | 转换后的逆波兰表达式结果   |

#### 4.3.4 程序清单

对于符号表的建立都大同小异，此处展示变量符号表语建立和查询代码：

```
void newvar(int num,...)//1)创建变量符号表
{
    va_list valist; //定义变长参数列表
    struct var *a=(struct var*)malloc(sizeof(struct var));//新生成的父节点
    struct ast *temp=(struct ast*)malloc(sizeof(struct ast));
    va_start(valist,num);//初始化变长参数为 num 后的参数
    temp=va_arg(valist, struct ast*);//取变长参数列表中的第一个结点
    a->type=temp->content;
    temp=va_arg(valist, struct ast*);//取变长参数列表中的第二个结点
    a->name=temp->content;
    vartail->next=a;
    vartail=a;
}

int exitvar(struct ast* tp){
    struct var* p=(struct var*)malloc(sizeof(struct var*));
    p=varhead->next;
    int flag=0;
    while(p!=NULL)
    {
        if(!strcmp(p->name,tp->content))
        {
            flag=1;    //存在返回 1
            return 1;
        }
        p=p->next;
    }
    if(!flag)
    {
        return 0;//不存在返回 0
    }
}
```

```

    }
}
char* typevar(struct ast*tp)//3) 查找变量类型
{
    struct var* p=(struct var*)malloc(sizeof(struct var*));
    p=varhead->next;
    while(p!=NULL)
    {
        if(!strcmp(p->name, tp->content))
            return p->type;//返回变量类型
        p=p->next;
    }
}

```

需要修改 Bison 源代码中的语义动作，此处展示对于变量的语义错误的检查的代码：

```

ExtDef:Specifire ExtDecList SEMI //变量定义:检查是否重定义 Error type 3
{
    $$=newast("ExtDef", 3, $1, $2, $3);
    if(exitvar($2)) printf("Error type 3 at Line %d:Redefined Variable
's' \n", yylineno, $2->content);
    else newvar(2, $1, $2);
}
|Specifire SEMI {$$=newast("ExtDef", 2, $1, $2);}
|Specifire FunDec Compst //函数定义:检查实际返回类型与函数类型是
否匹配 Error type 8 {
    $$=newast("ExtDef", 3, $1, $2, $3);
    newfunc(4, $1);
} ;
/*Local Definitions*/
DefList:Def DefList {$$=newast("DefList", 2, $1, $2);}
| {$$=newast("DefList", 0, -1);}
;
Def:Specifire DecList SEMI //变量或数组定义:检查变量是否重定义 Error type
3
{
    $$=newast("Def", 3, $1, $2, $3);
    if(exitvar($2) || exitarray($2)) printf("Error type 3 at
Line %d:Redefined Variable 's' \n", yylineno, $2->content);
}

```

```

else if($2->tag==4) newarray(2,$1,$2);
else newvar(2,$1,$2);
}
;

```

将算术表达式转换成逆波兰表达式的核心代码：

```

stack <char> s;
int getPriority(char c) //获取符号的优先级
{
    switch (c)
    {
        case '+':
        case '-':
            return 1;
        case '*':
        case '/':
            return 2;
        case '%':
            return 3;
        case '#':
        default:
            return 0;
    }
}
bool judgePriority(char c1, char c2) //判断两个符号的优先级
{
    if (getPriority(c1) > getPriority(c2))
    {
        return true;
    }
    else
    {
        return false;
    }
}
bool deleteBracket(stack<char>& s)
{
    stack<char> t;
    while (!s.empty() && s.top() != '(') //如果栈顶不是右括号
    {
        t.push(s.top());
        s.pop();
    }
    if (s.empty()) //证明没有与右括号匹配的左括号
    {
        return false;
    }
}

```

```

else if (s.top() == '(')//如果匹配上了左括号
{
    s.pop();//弹出左括号
    while (!t.empty())//把(之上的元素压入栈
    {
        s.push(t.top());
        t.pop();
    }
    return true;
}
string translate(string prefix){
    string result;
    s.push('#');
    for (int i = 0; i < prefix.length(); i++)
    {
        if (prefix[i] == '#') { break; }
        if (isalpha(prefix[i]) || isdigit(prefix[i]))//当前字符为字母或数字
        {
            result += prefix[i];
        }
        else if (prefix[i] == '(')//左括号, 入栈
        {
            s.push(prefix[i]);
        }
        else if (prefix[i] == ')')//右括号, 匹配左括号, 并将它们都去掉, 去掉
        一对括号
        {
            if (!deleteBracket(s))
            {
                return "error";
            }
        }
        else
        {
            if (judgePriority(prefix[i], s.top()))//若当前字符的优先级高于栈顶字
            符的优先级
            {
                s.push(prefix[i]); //入栈
            }
            else
            {
                while (!judgePriority(prefix[i], s.top()) && s.top() !=
                '#' && s.top() != '(')
                {
                    result += s.top();

```

```
        s.pop();
    }
    s.push(prefix[i]);
}
}
}
while (!s.empty() && s.top() != '#')
{
    result += s.top();
    s.pop();
}
return result;
}
```

#### 4.3.5 调试情况

对于算术表达式转换成逆波兰表达式的程序编写较为容易，基本上一次编写成功，因为之前做过类似的算法题，运用栈可以很轻松的编写好该程序，没有经过大量的调试。

而对于语义分析程序的编写，由于数据结果基础不牢，其次对于 bison 的掌握不是很好，废了很长的时间也没能编写成功，最后借鉴的网上的一篇博客的代码，但也只是能够读懂他的代码，并且成功运行，他的代码中还有一些错误，没能修改，对于此任务没有独立完成，在接下来的时间里，对于此任务要能够自己独立完成。

#### 4.3.6 运行结果

对于语义分析程序设置了多个含有语义错误的代码进行测试，部分测试结果如下图所示：

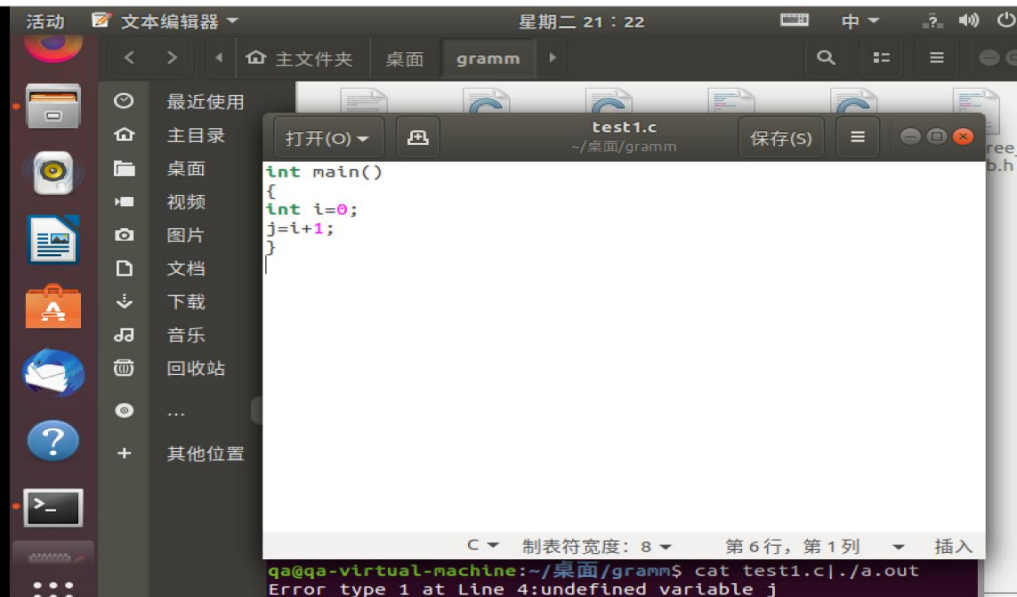


图 4.3.2 语义分析测试 1 运行结果

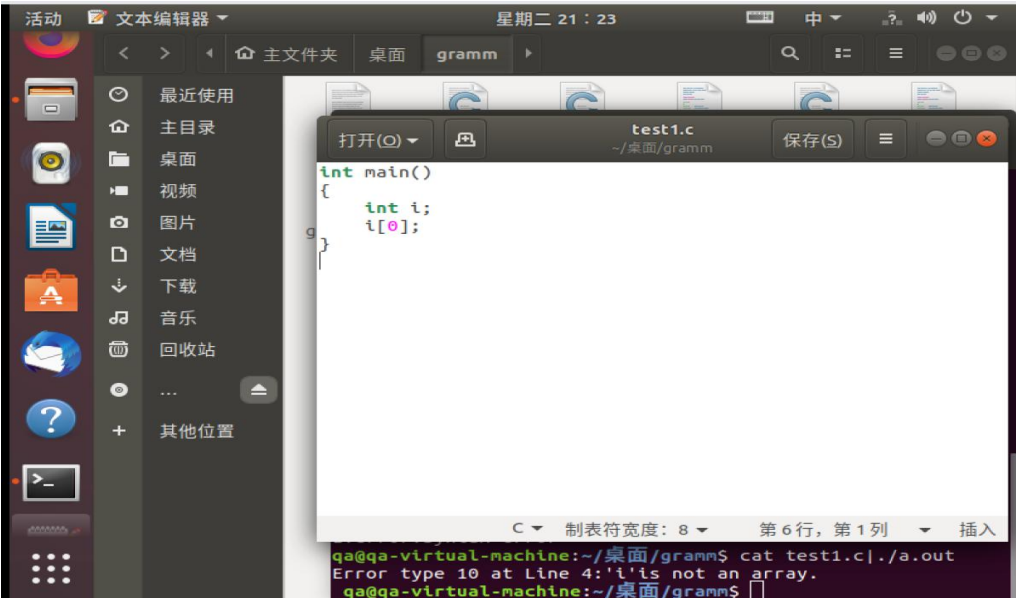


图 4.3.3 语义分析测试 2 运行结果

对于算术表达式（前缀表达式）转换为逆波兰表达式（后缀表达式）输入多组算术表达式测试，程序运行结果如下图所示：



```
F:\编译实验\18级编译原理第一次实验课资料\18软六-乔翱-201811040809\实验4\reversePolish.exe
请输入需要转换的中缀表达式，输入0退出
5+4*9
逆波兰表达式为
549*+
请输入需要转换的中缀表达式，输入0退出
(a+4+(d*4)+5)+8
逆波兰表达式为
a4+d4*+5+8+
请输入需要转换的中缀表达式，输入0退出
(2+8%9)*5+7
逆波兰表达式为
289%5*+7+
请输入需要转换的中缀表达式，输入0退出
0

-----
Process exited after 85.95 seconds with return value 0
请按任意键继续. . .
```

图 4.3.4 算术表达式转逆波兰表达式程序运行结果

## 4.4 心得体会

通过本次实验对语义分析以及中间代码生成有了更加深刻的理解，通过这次实验对于自己的编码能力有了很大的提高。但是本次实验还有很多有待改进的地方，对于静态语义分析能找出的错误比较有限，有些错误类型不能发现，对于中间代码生成的程序，只是简单的把算术表达式转换为逆波兰表达式，没有实现转换成中间代码，这是在之后的学习过程中需要改进的地方。

通过编译原理的这四次课程实验，分别编写了词法分析程、自顶向下的语法分析程序、自底向上的语法分析程序、语义分析和中间代码生成程序，在之后的学习过程中要把这几部分连接起来，做一个小的编译器。通过这四次实验，对编译原理课程学到的理论知识有了更好的理解，更有助于对于编译原理课程的理解。

封面设计： 贾丽

地 址： 中国河北省秦皇岛市河北大街 438 号

邮 编： 066004

电 话： 0335-8057068

传 真： 0335-8057068

网 址： <http://jwc.ysu.edu.cn>