

# 哈尔滨工业大学

## 编译原理 2024 春

### 实验一

学院:	计算学部
姓名:	徐柯炎
学号:	2021110683
指导教师:	单丽莉

#### 一、实验目的

- 巩固对词法分析与语法分析的基本功能和原理的认识。
- 能够应用自动机的知识进行词法分析与语法分析。
- 理解并处理词法分析与语法分析中的异常和错误。

#### 二、实验环境

- GNU Linux Release: Ubuntu 12.04, kernel version 3.2.0 - 29。
- GCC version 4.6.3。
- GNU Flex version 2.5.35。
- GNU Bison version 2.5。

#### 三、实验内容

编写一个程序对使用 C--语言书写的源代码进行词法和语法分析，并打印分析结果。实验要求使用词法分析工具 GNU Flex 和语法分析工具 GNU Bison，并使用 C 语言来完成。

程序要能够查出 C--源代码中可能包含的下述几类错误：

- 词法错误（错误类型 A）：出现 C--词法中未定义的字符以及任何不符合 C--词法单元定义的字符；
- 语法错误（错误类型 B）。

#### 四、实验过程

##### （一）程序功能

##### 1. 完成了抽象语法树的简单实现

首先我实现了抽象语法树的结构体，如图所示：

其中每一行的功能分别为：

- 1) int line: 当前语法单元的行号；
- 2) char\* name: 语法单元的名称；
- 3) int n: 语法单元子结点的个数；
- 4) struct TREE \*child[maxNum]: 子节点指针数组；
- 5) union: 联合体，用来存放终结节点的数值。

```
struct TREE{
    int line;
    char* name;
    int n; //子结点的个数
    struct TREE *child[maxNum];
    union{
        char* ID;
        int INT;
        float FLOAT;
    };
};
```

接下来的所有程序的实现都依赖于这个数据结构，当词法分析和语法分析完成后，将采用先序遍历的方法来打印整个语法分析树。

## 2. 词法分析

词法分析是语法分析的基础，关于词法分析，我实现了以下功能：

- (1) 能够正确识别十进制数、八进制数、十六进制数并能进行进制转换；
- (2) 能够正确识别浮点数；
- (3) 能够正确识别各类标识符、关键字和标点符号。

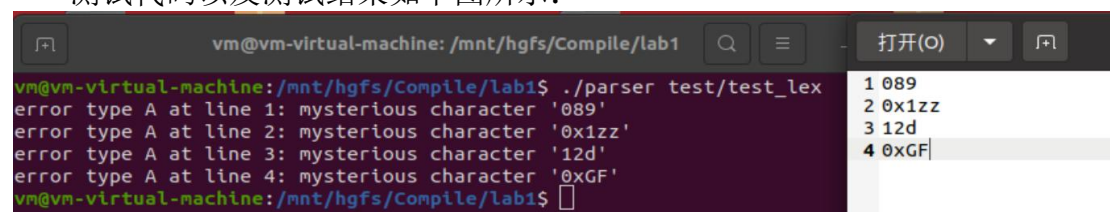
特别地：

- (1) 能够正确识别指数形式的浮点数；
- (2) 可以判断出错误的八进制数、十六进制数、浮点数、指数形式数。
- (3) 能够识别所有的注释符号，并判断出错误的注释符号。

识别错误的各类数字代码如下图所示：

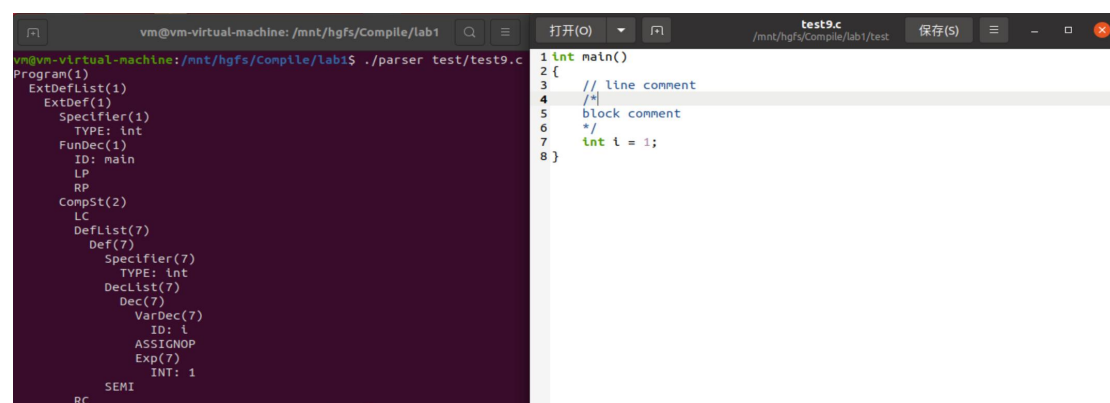
```
INT8_ERROR      0[0-7]*[8-9]+[0-7]*
INT16_ERROR     0[Xx][a-fA-F0-9]*[g-zG-Z]+[a-fA-F0-9]*
FLOAT_ERROR     [0]+[0|[1-9][0-9]*)\.[0-9]+\.[0-9]+|[0-9]+\.[0-9]+[Ee][+-]?[0-9]*|([+-]?([0-9]*\.[0-9]+|[0-9]+\.[0-9]*))
NUM_ERROR       {INT8_ERROR}|{INT16_ERROR}|{FLOAT_ERROR}
```

测试代码以及测试结果如下图所示：



```
vm@vm-virtual-machine: /mnt/hgfs/Compile/lab1
vm@vm-virtual-machine: /mnt/hgfs/Compile/lab1$ ./parser test/test_lex
error type A at line 1: mysterious character '089'
error type A at line 2: mysterious character '0x1zz'
error type A at line 3: mysterious character '12d'
error type A at line 4: mysterious character '0xGF'
vm@vm-virtual-machine: /mnt/hgfs/Compile/lab1$
```

本程序还能正确识别注释符号，这里单行的注释采用正则表达式实现，而多行注释采用如下方法：当遇到'/\*'时，执行 read\_comment()函数，贪婪匹配'\*/'。测试选做第五个样例如下图所示：



```
vm@vm-virtual-machine: /mnt/hgfs/Compile/lab1$ ./parser test/test9.c
Program(1)
  ExtDefList(1)
    ExtDef(1)
      Specifier(1)
        TYPE: int
      FunDec(1)
        ID: main
        LP
        RP
        CompSt(2)
          LC
          DefList(7)
            Def(7)
              Specifier(7)
                TYPE: int
              DeclList(7)
                Dec(7)
                  VarDec(7)
                    ID: i
                    ASSIGNOP
                    Exp(7)
                      INT: 1
          SEMI
          RC
```

可以发现能正确打印语法树，

## 3. 语法分析

关于语法分析，我实现了以下功能：

- (1) 使用 CFG 正确构建抽象语法树；
- (2) 使用先序遍历正确打印抽象语法树的各个节点以及数值。

特别地：

- (1) 实现较高层次的模块化，简化代码，增加代码的可读性。

具体来说，我实现了一个可传入可变数量的参数的函数，此函数用来构建语法单元节点，如下图所示：

```

treeNode cre_gram_unit(char* name, int yyline, int num, ...){
    treeNode res = NULL;
    if(num == 0){
        res = newNode(name, num, yyline, NULL);
    }
    else{
        va_list nodes;
        va_start(nodes, num);
        treeNode *childList = (treeNode*)malloc(num*sizeof(treeNode));
        for(int i=0; i<num; i++){
            childList[i] = va_arg(nodes, treeNode);
        }
        res = newNode(name, num, yyline, childList);
    }
    nodeList[nodeNum]=res;
    nodeNum++;
    return res;
}

```

此函数通过 `va_list` 来传入 `num` 个可选参数，也就是这个语法树的子节点，然后循环读入各个子节点，和父节点建立联系，从而实现此语法单元节点的构建。

## （二）编译过程

这里使用 `shell` 脚本来编译我们的 `flex`、`bison` 以及 `c` 语言的代码，并且实现测试代码的运行，如下图所示：

```

12 echo "====="
13 echo "Compiling Lexical Analyzer....."
14 flex -o code/lex.yy.c ${lex_file}
15 # gcc main.c code/lex.yy.c -lfl -o scanner
16 # ./scanner test1.c
17
18 echo "====="
19 echo "Compiling Syntax Analyzer....."
20 bison -d -o code/Syntax.tab.c ${syntax_file}
21
22 echo "====="
23 echo "Compiling Main File....."
24 gcc ${main_file} code/Syntax.tab.c -lfl -ly -o ${out_file}
25
26 echo "====="
27 echo "Test 1"
28 ./parser test/test1.c

```

其中实现程序的代码放在 `code` 文件夹中，测试代码放在 `test` 文件夹中，最后编译生成程序 `parser`，通过运行此程序完成测试。

## 五、实验总结

- (1) 学会了如何使用 `flex` 和 `bison` 编写程序分析词法和语法；
- (2) 复习了词法分析和语法分析的过程；