

# 《编译器构造实验》lab1报告

20331041 徐锬达 xusd3@mail2.sysu.edu.cn

## 1 实现功能

### 1 词法分析

匹配合法的词法单元，建立对应的终结符节点，用于语法树的构建。

对于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，输出“Mysterious characters”的词法错误的提示信息。

### 实现的具体要求

- 识别八进制数和十六进制数

对于错误的八进制数和十六进制数，输出对应的词法错误提示信息。

```
{ill_oct}      { lexical_error_handler("Illegal octal number");      }
{ill_hex}      { lexical_error_handler("Illegal hexadecimal number"); }
```

- 识别指数形式的浮点数

对于错误的指数形式的浮点数，输出对应的词法错误提示信息。

```
{ill_float}    { lexical_error_handler("Illegal floating point number"); }
```

- 识别 // 和 /\*...\*/ 形式的注释

滤除合法的注释内容。

对于 /\*...\*/ 形式的注释，若缺少 \*/，即对于悬挂的 /\*，输出 Unmatched "/\*" 的词法错误提示信息；  
若缺少 /\*，即对于悬挂的 \*/，会因被识别为乘号与除号而输出语法错误提示信息。

### 2 语法分析

对于错误的语法模式，重写 yyerror() 函数，输出“Syntax error”的语法错误提示信息。

### 二义性与冲突处理

- 通过规定运算符的优先级与结合性解决 Exp 里的二义性问题

- 定义一个比 ELSE 优先级更低的 LOWER\_THAN\_ELSE 运算符，降低归约相对于移入 ELSE 的优先级，解决 if-else 语句的移入-归约冲突

```
%right ASSIGNOP
%left OR
...
%left LP RP
%nonassoc LOWER_THAN_ELSE
%nonassoc ELSE
```

## 错误恢复

书写包含 `error` 的产生式，检查输入文件中的各种语法错误。

通常让结束符 `;` 以及括号 `{`、`}`、`(`、`)` 等作为错误恢复的同步符号。

对于一些特定的语法错误，识别并输出特定的语法错误提示信息，比如 `"Missing ")"`、`"Missing "]"`、`"Missing ";"`。

```
EXP :
    | ID LP error RP { myerror("Missing \")\"); }
```

## 语法树

以**二叉树**的结构实现语法树的多叉树数据结构，即包含第一个子节点和兄弟节点，声明部分写在 `syntax_tree.h`。

```
struct Node* first_child;    // 第一个子节点
struct Node* sibling;        // 第一个兄弟节点
```

属性值的类型修改为树节点类型，通过先序遍历打印语法树，行号通过 `yylineno` 维护。

```
void preorder_print(struct Node* node, int depth);
```

插入终结符节点和非终结符节点，后者节点数不确定，采用可变参数写法。

```
struct Node* insert_terminal_node(const char* type, enum TOKEN_TYPE token_type, const char*
value);
struct Node* insert_nonterminal_node(const char* type, int lineno, int num, ...);
```

## 2 编译

---

两种编译方式，进入 `Code` 文件夹：

### 1 逐步手动编译

- 编译Flex源代码，运行：`flex lexical.l`  
编译生成 `lex.yy.c`。
- 编译Bison源代码，运行：`bison -d syntax.y`  
`-d` 的含义是将编译的结果分拆成 `syntax.tab.c` 和 `syntax.tab.h` 两个文件。  
`lexical.l` 包含了对 `syntax.tab.h` 的引用。
- 整合编译，运行：`gcc main.c syntax.tab.c syntax_tree.c -lfl -ly -o parser`  
`lex.yy.c` 已经被 `syntax.tab.c` 引用，因此最后把 `main.c`、`syntax.tab.c` 和 `syntax_tree.c` 放到一起编译，生成 `parser`

### 2 Makefile自动编译

- `Makefile` 文件自动编译，运行：`make`
- 清理编译生成文件，运行：`make clean`

## 3 测试

---

在 `Code` 文件夹下：

### 1 单个文件手动测试，运行：`./parser test.cmm`

若 `test.cmm` 与 `parser` 不在同一文件夹下，需自行添加路径。

### 2 多个文件测试脚本测试，运行：`./test.sh`

将多个测试文件（后缀必须是 `.cmm`）统一放在 `Test` 文件夹下，运行测试脚本 `test.sh`。

注意默认情况是 `parser` 在 `Code` 文件夹下，`Code` 文件夹与 `Test` 文件夹同级。不同则自行修改脚本中的 `path`（测试文件所在文件夹相对路径）。