# 编译原理 实验二 实验报告

181250117 秦锐鑫

# 设计与实现

## 实现功能

1. 识别大部分的语法错误
2. 在没有错误的时候打印出正确的语法树

## 实现方式

### 两次遍历 在没有错误的时候打印出正确的语法树

通过设置 syntax_error_flag 标记初始化为 0 来识别是否有语法错误，如果出现错误，将 syntax_error_flag 置为 1；第一次结束后 syntax_error_flag 仍为 0 在遍历一次并打印出语法树

### 错误恢复

syntax.y 文件
在所有有 SEMI 的产生式最后添加 error SEMI
在 Exp 的产生式最后添加 Exp LB error RB
在 CompSt 的产生式最后添加 error RC
在 FunDec 的产生式最后添加 error RP

### 构建语法树

在 lexical.l 中声明定义节点 tree_node 结构体，用于构建语法树，树节点包括 节点类型 node_type，节点名称 node_name，子节点 child_node 使用结构体 tree_node 指针数组，子节点数量 child_num，行号 line_no，以及用于存储数值类型的具体值 int_val float_val。

利用枚举和字符串数组 typedef enum NT{ ... } NODE_TYPE; const char* type_name[] = { ... }; 采用表驱动的方式来存储与使用 语法/词法单元名称

编写 创建普通节点函数，创建词法单元节点函数，以及遍历函数，其中前两者使用可变参数 ... 。创建普通节点函数使用可变参数以实现同时传入多个子节点的父节点的创建；创建词法单元节点函数使用可变参数以实现不同类型词法单元所需要保存内容的传递

```
tree_node* create_node(NODE_TYPE enum_type, int lineno,int childnum,...);
tree_node* create_token_node(NODE_TYPE enum_type, int lineno,int
val_selector,...);
void traverse(struct tree_node* root,int cur_deep);
```

# 有趣的现象 or 印象深刻的 bug

一开始将节点结构体声明定义在 syntax.y 中，导致 lexical.l 中无法使用节点结构体，于是把节点结构体声明定义在 lexical.l 因为 syntax.y 会 #include "lex.yy.c" 所以可以使用到节点结构体。

说来惭愧，c语言中定义一个结构体类型后如 struct tree_node{...}， 如果没有使用 typedef 则不能直接使用 tree_node，而需要使用 struct tree_node，一般会用 typedef 起别名，后直接使用别名。一开始没注意，总是报错。

```
error: unknown type name ‘tree_node’
     tree_node node;
```

c语言中不允许有同名函数，一开始想使用同名函数不同参数实现不同数量子节点的创建，但会报错，于是改成可变参数函数。