编译原理第一次实验报告

181860087 唐业

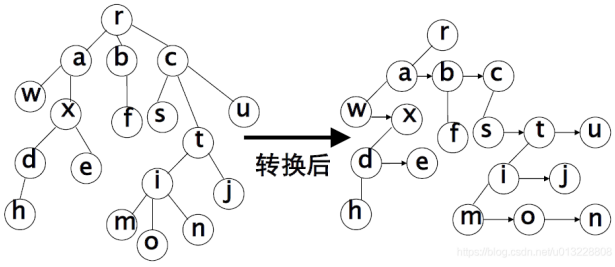
181860102 王印可　encore.w@qq.com

任务号：3　选做要求1.3——识别注释

实现的功能

- 错误识别
  1. 词法错误：当输入文件中出现C--词法中未定义的字符以及其它不符合定义的字符时，我们的程序能够输出错误类别（Error type A）与错误行号，并附以说明文字。
  2. 语法错误：当输入文件出现不符合C--语法的格式时，我们的程序能够输出错误类别（Error type B）并输出具体错误信息（包括行号与错误内容）。
- 语法树输出
  - 当且仅当输入文件中没有词法错误与语法错误时，我们的程序会解析该文件并生成一棵语法树。
  - 对于每一个语法树节点
    - 若当前结点是语法单元且没有产生空串，我们会打印它的名称与行号
    - 若当前结点是语法单元且产生空串，我们不会打印该语法单元的信息
    - 如果当前节点是词法单元，我们只打印名称，并对ID、TYPE、INT、FLOAT类型额外打印词素、具体类型、数值

- 实现方法
  1. 首先，我们在Flex源文件中定义了正则表达式，在规则部分描述了对于不同正则表达式应进行的具体操作，由此来检测词法错误并识别词法单元。
  2. 语法树是一棵多叉树，在这里我们使用了二叉树的数据结构，如图下



即每个节点只有子节点和兄弟节点两个指针变量。（例：图中m，o，n节点均为i节点的子节点，然而i节点仅有一个指向m的子节点指针，o与n节点通过m的兄弟节点指针相连以充当i的子节点。）

  3. 我们封装出上述节点类型treeNode，同时在Bison源文件中将yylval定义成treeNode类型。有了treeNode结构，我们就可以定义相关的创建插入函数，如下方代码所示

```
struct treeNode {   //treeNode类型
    char name[32];  //词法单元正则名
    char val[32];   //词法单元具体值
    int lineno;     //行号
    int type;       //节点类型值（我们自定义的类型数值，方便后续输出）
    node child;     //子结点指针
    node sibling;   //兄弟结点指针
};
//相应函数
node createNode(char name_[], char val_[], int lineno_, int
type_);//创建节点
void insertNode(node root_, int args, ...);//向root_插入多个子节点
void printNode(node root_); //输出root_节点信息
void preOrder(node root_, int depth);//前序遍历多叉树并输出节点信息
```

4. 有了上述封装，我们在Bison源文件中定义词法单元。当我们在Flex源文件中进行词法分析时，创建treeNode节点并返回Bison源文件中相应的终结符词法单元。而当我们在Bison源文件中进行语法分析时，我们在规则部分声明具体的C--语法并调用相应的创建插入动作即可。以下方CompSt为例

```
 CompSt :
LC DefList StmtList RC {
  $$ = createNode("CompSt", " ", @$.first_line, 0);
    //创建名为CompSt的节点
    insertNode($$, 4, $1, $2, $3, $4);
    //将子节点LC，DefList，StmtList，RC插入父节点CompSt
};
```

由此整棵语法树在解析文件的过程中一步步构建出来，当词法与语法均正确时，我们便可前序输出整棵树，根据具体要求附上行号或属性值等。

5. 最后我们还需要处理语法错误。出现语法错误时，程序会先调用yyerror输出`Error type B:syntax error`的报警。同时我们利用特殊符号`error`来分辨不同的错误情况，调用自定义的错误输出函数来显示更多错误细节。以下方丢失分号错误为例

```
Specifier DecList error {
    syntaxError += 1;
    my_yyerror("missing \';\' or invalid token");
    //调用my_yyerror输出具体错误细节
};
```

编译运行

- 如何编译?
  1. 进入目录 `Compiler/Lab/Code`
  2. 在控制台运行指令`make`
- 如何运行?
  - 命令行输入`make test`或`./parser  xxx.cmm` (xxx.cmm代表自己编写的测试输入文件)

程序亮点

- 在发生语法错误时，我们的程序能够输出行号与具体的错误信息，如图下

```
ubuntu@ubuntu-virtual-machine:~/Desktop/Compiler/Lab/Code$ make test
./parser ../Test/test1.cmm
Error type B at Line 4: syntax error: near 'int'. (missing ';' or invalid token)
Error type B at Line 5: syntax error: near ';'. (invalid expression between 'ret
urn' and ';')
```

  首先我们会输出Error type B、行号、并指出语法错误附近的一个token

  接下来在括号中我们又会打印出细节的错误

- 我们对语法树的操作封装地非常精简，对于插入操作

```
void insertNode(node root_, int args, ...);
```

  我们使用了C语言中的宏va_list,可以通过args来确定参数数量，并从...中依次取数据，这样可以一次向父节点插入很多子节点，极大程度降低了冗余性。

总结反思

1. 对于`malloc()`在堆区创建的变量必须初始化，否则如果指针没有初始化为`NULL`，但在堆区不断扩大的过程中，会出现野指针的问题；
2. 对于错误恢复的部分，仅仅用讲义中给到的几个规则不够，bison在shift-reduce的过程中会吞掉很多原本想要恢复的tokens。例如括号嵌套的语法错误，多出来的`*/`会产生`error`调用`yyerror()`，而我们在`stmt`中定义了`error SEMI`的规则，会将多出来的`*/`和下一条`stmt`一直到分号为止一起视为`error`并恢复。