lab1 实验报告

本实验已实现正确的词法和语法分析功能

一、编译与测试命令

编译可以使用：

> make parser（或直接make）

通过makefile提供的方式可以对本实验项目进行编译，生成程序名为parser，此外我们写了一个简单的脚本用于对 `../Test/` 中的所有单元测试进行一键测试，使用

> ./test.sh

进行测试，可加文件名实现制定测试文件运行，例如单独测试test_bison.cmm文件时可以使用命令：

> ./test.sh test_bison

二、实现过程

1、flex部分

- 书写用于词法分析的正则表达式(几个比较重要的正则表达式和书写过程中的注意事项)
  - 书写正则表达式的顺序应使得flex能进行某些**优先匹配**，比如“==”和“=”，应当将前者写在前面，否则就会将“==”识别为两个“=”。类似的情况还出现在ID和部分保留符之中。
  - 使用以下的正则表达式对**多行注释**进行匹配，可以满足,当有嵌套时总是识别最近的 `*/` ：
    > "/*" ([^*] | * + [^*/]) ** +"/"
  - 对于八进制，十六进制，科学计数法，也要书写正则表达式，下面是科学计数法的正则表达式：
    > ({digit}+.{digit}+|.{digit}+|{digit}+.)(e|E)?[-+]?{digit}+

    由于交给语法分析的属性值应是十进制和浮点数，所以需要进行相应的进制转换，基本思路是使用 `char * p=yytext` 得到相应字符串指针后进行简单的字符串到数值的转换，将最终结果赋值给 `yylval.node->i_val` 即可,相应的进制转换代码可以在代码中查看
- 创建终结符的叶子节点，用于语法分析中建立语法树
  关键的创建节点语句:
  > yylval.node=create_node("FLOAT",FLOAT,yylineno);

（创建名为FLOAT的树节点）

flex通过对全局变量yylval进行赋值，从而实现将词法单元的属性值交给bison继续使用

2、bison部分

- 对照实验语法抄写产生式，声明符号类型和优先级都只需按照pdf的讲解进行，不再赘述。
- 语法树相关
  - `yylval的声明: %union{ Tnode* node;} \\指针类型`
  - 当发生归约时应创建新的节点，并进行插入操作

    `$$=create_node("ExtDefList",0,@$.first_line);Ninsert($$,2,$1,$2);`
  - 多叉树数据结构的实现

```
struct treenode{
        char name[16];   //语法单元名称
        int line;        //所在行
        int type;        //词法单元类型
        struct treenode *firstchild;  //子结点
        struct treenode *nextbro;     //兄弟节点
        union{
                int i_val;
                float f_val;
                char *s_val;
};


Tnode *create_node(char *,int ,int );
void insert_node(Tnode *p,Tnode* newn); //将newn节点插入p节点的子结点中
void Ninsert(Tnode* p,int n,...); //不定长参数，用于连续插入多个节点
void dfs(Tnode* s,int dep); //遍历并打印语法树
```

- ***错误恢复***

  这一部分是比较麻烦且繁琐的，因为很难实现一个万能的错误恢复机制。基本的错误恢复通过在各处产生式中加入error符号，例如 `Stmt → error SEMI` 可以实现对语句中错误的恢复，但这还远远不够！代码中可能会出现一些复杂的情况（比如难以解决的分号和换行问题）我们将会在下文中讨论一个无法尽善尽美的例子。

  此外，虽然Bison默认error后成功移入三个符号才继续正常的语法分析的策略，但这会遇到一些连续错误的检测问题，在这一部分中我们使用 `yyerrok;` 进行立即错误恢复（如果没有比较特殊的连续错误也可以不使用）， `yyerrok;` 的副作用是可能会把一个错误之后的语法单元认为是另一个独立的错误，由于本次实验说明每行至多只有一个错误，我们使用了一个记录上一次报错的行数的trick使得每行的确至多只输出一次错误即可，虽然似乎有些不对劲，但我们相信这种曲折的策略能够较好地在既匹配实验要求的测试（一行只有一个错）的情况下，更好地处理一些连续错误以及换行的复杂问题，下面给出了我们遇到的一个例子：
  - **一个无法尽善尽美的例子**

```
//global 以下三行代码都是在全局定义中
1.int i
2.int j
3.int k=1;
```

对于上述的三行代码，按照讲义的要求我们应当报三个错误：

```
错误1 error B at 2 : missing ';'
错误2 error B at 3 : missing ';'
错误3 error B at 3 : unexpected '='   //全局中只能定义不能赋值
```

- 错误恢复产生式假设1：

  ```
  ExtDef:
    | Specifier error SEMI
      | error SEMI
  ```

  则只会报 `错误1` 一个错误，因为只有当找到一个SEMI，才能真正完成再同步，所以2、3行的词法单元都直接被丢弃

- 假设2

  ```
  ExtDef:
    | Specifier error SEMI {yyerrok;}
      | error SEMI          {yyerrok;}
      | Specifier error     {yyerrok;}
  ```

  这里使用yyerrok是因为错误在三个单元内，此时 `错误1，2，3` 都能找出来,**但在第三行的等号会重复两次报错**，因为当读入 `int k=` 时发生错误，且直接按 `Specifier error` 进行归约，所以此时不需要丢弃任何的词法单元。于是‘=’再一次被读入, `=1;` 报错，并按 `error SEMI` 归约。

  对于上面的这种同一个符号报两次错的情况，可以通过修改 `yyerror` 函数解决（增加一个行标志位记录上一次报错的位置，用于一行只输出一个错误），这也是我们最终决定采用的方案。

（强如gcc也只会报一个错：