Report on Lab 1
编译方法

在 `Code` 文件夹下输入命令 `make` 即可自动编译生成 `parser` 文件。

功能简介

文件结构如下所示：

```
Code
├── Makefile
├── lexical.l
├── syntax.y
├── syntaxtree.c
└── syntaxtree.h
```

其中 `lexical.h` 存放与词法分析相关的 `flex` 代码；`syntax.y` 存放与语法分析相关的 `bison` 代码和 `main` 函数；`syntaxtree.h` 和 `syntaxtree.c` 用于存放与语法树相关的结构体定义和函数。

词法分析

对于ID、INT和FLOAT，我使用了这样的正则表达式：

```
INT 0|([1-9][0-9]*)
FLOAT {INT}+"."{DIGIT}+
ID (_|{LETTER})(_|{LETTER}|{DIGIT})*
```

在分析时需要注意优先级，例如先匹配 `/*`、最后匹配ID等。

在匹配到 `/*` 时，我将不断向后读取内容直到遇到第一个 `*/`，并将中间的内容全部丢弃。

语法分析

在语法分析中，我将所有节点的类型自定义为语法树的节点指针。在推导成功后，我构造一个语法树的节点，并将当前词素的值赋为指向构造出的节点的指针，即 `$$ = Constructor(...)`。

在错误处理中，我在不产生移入规约冲突的前提下加入了尽可能多的错误处理。当产生冲突时尽量选择抽象层较高的词素进行错误处理。同时，我设计了一些包含多个错误的测试样例，在错误

恢复不能正常工作时，我尝试将所有错误改正后，根据正确情况下的语法树查找应当在何处进行错误恢复，并返回到 `syntax.y` 中对应位置进行改正。

有时，加入的错误处理语句本身与推导式产生了冲突（例如 `CompSt : LC DefList StmtList RC` 和 `CompSt : LC error RC`）。此时可以单独执行 `bison` 并加上 `-Wcounterexamples` 得到一个会产生冲突的例子，方便进一步分析和设计错误处理。

语法树

我的语法树如下设计：

- 存在且仅存在一个根节点
- 每个节点在保存子节点时仅保存第一个
- 每个节点以链表形式保存其兄弟节点（以上两条见代码中注释图例）
- 每个节点保存：词素名称、词素内容（若为ID, FLOAT, INT, TYPE）、是否为终结符、行号

如下所示：

```
typedef struct Node {
    char *lex_name;
    char *content;
    int type;
    int line;

    struct Node* fa;
    struct Node* next;
    struct Node* son;

    union {
        unsigned int int_value;
        float float_value;
    };
    /*
    [<fa>]
     |
    [son]---------------------------[son->next]---...
     |                                    |
    [son->son]-[son->son->next]-...  [son->next->son]-[son->next->son-
>next]-...
    */
} Node;
```

其提供如下四个成员函数：

```
Node* Constructor(char *lex_name, char *content, int type, int line);
void BuildSon(Node* fa, int count, ...);
void OutPutTree(Node* node, int depth);
void SetRoot(Node* node);
```

在为节点添加儿子时，考虑到语法树构建的特点，程序会 `assert` 当前父节点没有儿子、当前所有儿子节点没有兄弟。`Node *root` 定义在 `syntaxtree.c` 中，并在 `syntax.y` 中通过 `extern` 导入符号。其余实现较为常规，见代码。