看了一下指导书和代码，以下是我的理解，如果有不对的地方请指正：

## 作业的目标

输入自定义函数和表达式，并化简表达式。

## 代码框架

1. 大功能区分得很好，词法分析、语法分析到后面的简化、合并步骤都区分出来了，之后仅需要通过修改或替换相应的方法，就能更换实现。这一部分可不做调整。

2. 但是，要实现更好地存储解析数据和合并加减乘除，

框架上目前还面临一些问题：

（1）框架中对<因子>的存储还比较原始，后续的**解析、简化操作可能遇到困难**。

（2）抽象语法树部分的实现似乎比较神奇。

（3）合并、去0等操作是解析完成之后、通过操作解析结果的字符串完成的。这一步感觉回到了面向过程的处理思路，**丢掉了解析时获取的信息**。

## 一些解决思路

### 存储部分

1. **因子**

目前<因子>的存储抽象为 Factor 接口，但Factor接口中没有声明任何方法，**使得Factor的作用仅限于：标记某个类是<因子>。**

因此，无法实现对<因子>调用同一方法，从而对<因子>进行统一操作。**实现统一操作的目的是，要让处理的逻辑尽可能简化，通过简单的组合来实现复杂的功能。**所以要提高Factor接口的表达能力，需要对<因子>进行抽象。

先分析目前指导书中存在的<因子>，包括：

\* 变量因子

\* 幂函数

\* 指数函数

\* 自定义函数

\* 常数因子

\* 表达式因子

可以留意到，这些<因子>至少需要包含一个 常量或变量<Token> ，可能包含多个操作符<Token>。我们可以从<Token>的角度入手，将因子抽象为：

Factor :

- 变量或常量<Token>

- 操作符 op <Token>

为了简化Factor的表达，可以假设<因子>最多仅由一个操作符组成。此时Factor可以用来表示 ：常量因子、变量因子

**（2）表达式**

当需要表达包含多个操作符的表达式时，可以将多个Factor组合形成Expr：(已知目前没有三目操作符）

Expr:

- 左操作数 Factor / Expr

- 操作符 op <Token>

- 右操作数 Factor / Expr

此时Expr可以用来表达：幂函数、指数函数、表达式因子、项

**（3） 自定义函数。**

针对需要展开的自定义函数，可定义：

Function:

- 参数1 Factor

- 参数2 Factor

- 参数3 Factor

### 合并加减乘除

抱歉抽象语法树部分的实现没太看懂，好像是把项和操作符分别塞到两个列表里。

我先说一下我对抽象语法树的理解：抽象语法树的分支节点是操作符op，叶子节点是操作数。假设分支节点和叶子节点的都有一个处理方法***parse()***

***叶子节点调用 parse(): 直接返回叶子节点的值***

***分支节点调用 parse(): 返回 叶子节点1 +/-/\*/÷/^ 叶子节点2 的值***

所以只要在根节点调用parse()，就能递归解析处理整棵抽象语法树。

**合并加减乘除本质上也是一种递归处理抽象语法树的过程**。

接下来探讨一下部分合并操作的细节：

**（1）合并**

如（5+3）\*（2-3）=> -1，可以理解为抽象语法树，在根节点调用parse()时，递归调用每层分支节点的parse，计算加减乘除，并返回加减乘除结果。对于不可合并的<因子>，如幂函数、指数函数等，则保留原有形式直接返回。

示例：

exp((1\*x+2)\*3)^(4+5\*x) = exp(3\*x+6) \* (4+5\*x)



* 1. **去除前缀0：**

如叶子节点值为 0000000012，调用parse()时直接返回12

### 3.总结

假设我们用Factor、Expr、Function组成抽象语法树，只需要为这三者提供同一个parse接口即可。在对应的parse()接口中，直接进行化简操作即可。

稍详细的实现可以参考如下伪代码（reference.java）：

// 处理接口

interface Parser {

    public ParseResult parse(); // 返回解析的字符串

}

// 缓存解析结果

class ParseResult {

    // 缓存parse()方法的处理结果，可以是化简后的表达式。

    ArrayList<...> result; // 这里的数据类型仍需要斟酌，主要是为了方便合并左右的表达式。甚至可以存一棵化简之后的抽象语法树

    // 处理result, 返回解析结果

    public String toString(){ ... }

}

// 操作符

interface Op {

    public String operate(Parser param1, Parser param2);  // 操作方法

}

// 以下为操作符示例

// +

class Add implements Op {

    @Override

    public String operate(Parser param1, Parser param2){ ... }

}

// -

class Sub implements Op {

    @Override

    public String operate(Parser param1, Parser param2){ ... }

}

// 表示： 常量因子、变量因子

class Factor implements Parser {

    public String NumOrVar; // 常量因子 / 变量因子，变量类型暂时取String

    public Op op;

    @Override

    public ParseResult parse(){

        // 在这里添加操作符功能实现

        // 也可添加 消去前缀0 和 + 等简化逻辑

        return  op.operate(NumOrVar);

    }

}

// 表示：幂函数、指数函数、表达式因子、项

class Expr implements Parser {

    public Parser factor1;  // 左操作数

    public Op op;           // 操作符

    public Parser factor2;  // 右操作数

    @Override

    public ParseResult parse(){

        // 化简等其它处理

        // [注意]这里比较麻烦的是 ：当左操作数和右操作数都是一个带有变量因子的表达式，需要遍历两侧操作数的全部因子，逐个变量因子执行合并

        return op.operate(factor1, factor2);

    }

}

// 表示：自定义函数

class Function implements Parser{

    public String param1;

    public String param2;

    public String param3;

    @Override

    public ParseResult parse(){

        // 化简处理

        Parser p =  new Expr(....); // 将param1 、 param2 、 param3 塞到一个Expr里

        return p.parse();

    }

}

// 抽象语法树

class AST {

    public Parser root; // 根节点

    public AST(){ ... } // 构造函数

    // 解析，并返回转化结果

    public String parse(

        ParseResult result = root.parse();

        return result.toString()

    )

}