**CHEMICAL EQUATION BALANCER**

The program calculates the coefficients to balance your given chemical equation. The algorithm used is Gauss–Jordan elimination, slightly modified to operate using only integer coefficients (not fractions).

**ERROR MESSAGES**

* **Syntax Error**

Your input does not describe a proper chemical equation. Check each letter carefully, and follow the examples as a guide to the correct syntax.

* **All-Zero Solution**

The only mathematical solution to your equation has all coefficients set to zero, which is a trivial solution for every chemical equation. For example, C → N2 has no solution because the only solution is 0C → 0N2.

* **Multiple Independent Solutions**

There exist multiple solutions to your equation that are not simply multiples of each other. Your equation can be considered as two or more independent equations added together. For example, H + O → H2 + O2 has no unique solution because two solutions are 2H + 4O → H2 + 2O2 and 6H + 2O → 3H2 + O2, which are not multiples of each other. Furthermore, the equation can be separated as H → H2 and O → O2, each of which does have a unique solution.

* **Arithmetic Overflow**

Your equation used numbers that are too big, or a term has an element that occurs too many times, or the internal calculation used numbers that are too big. I don’t expect this error to occur for real-world chemical formulas, only deliberately contrived ones. There is no workaround; the code would need to be rewritten to use bigints.

**HOW IT WORKS**

**Data Representation**

* A raw chemical element is a string of letters that begins with an uppercase letter and is followed by any number of lowercase letters. The regular expression is /[A-Z][a-z]\*/. For example, these are elements: H, Na, Uuq. The exception is that e by itself represents an electron.
* An ChemElem object is a chemical element with a positive integer repetition count. For example: Fe, H2.
* A Group object is a parenthesized list of ChemElem or Group objects, with a positive integer repetition count for the whole group. For example: (H2O)6, (C(OH)3)2.
* A Term object is a list of ChemElem or Group objects, with an integer electric charge for the whole term. For example: H3O+, S2−.
* An Equation object is a list of Term objects for the left side and a list of Term objects for the right side. For example: C + O2 → CO2.

In the code, this functionality is found in Equation, Term, Group, ChemElem.

**Parsing The Input**

Your text input is parsed by a hand-written tokenizer and a recursive descent parser with one token of look-ahead. The parser takes a lot of code to implement and is ugly, but it is robust. The parsed result is your chemical equation in the internal object representation. In the code, this functionality is found in parseEquation, parseTerm, parseGroup, parseElement, parseCount, and Tokenizer.

**Setting Up The Matrix**

The idea is to set up a system of linear equations to represent the balancing problem. Each term in the equation gets a variable. Each different element, and also electric charge, gets an equation. In the code, this functionality is found in buildMatrix, Equation/Term/Group/ChemElem.getElements, Term/Group/ChemElem.countElement.

For example, the equation H2 + O2 → H2O has 3 terms and 2 elements (H, O). We give a variable to represent the coefficient for each term, and we get *a*H2 + *b*O2 → *c*H2O. Now we make an equation for each unique element. For H, the equation balances only if 2*a* + 0*b* = 2*c*. In our matrix, we actually represent this equation as 2*a* + 0*b* − 2*c* = 0, using the row of integers [2, 0, −2]. For O, the equation is 0*a* + 2*b* = 1*c*, and we get the matrix row [0, 2, −1]. Even though none of the terms in this example are electrically charged, we always add an equation for electric charge anyway. In this case, the equation is 0*a* + 0*b* = 0*c*.

There is one more step in setting up the matrix. So far, all the equations form a homogeneous linear system. This means setting all the variables to zero is a solution, and each solution multiplied by any real number is also a solution. By convention in chemistry, we want the solution where all the coefficients are the smallest possible positive integers. So we add a (somewhat arbitrary) non-homogeneous equation stating that the first term should have a coefficient of 1, to break the symmetry. The matrix gains a column on the right, initially filled with zeros. Then the matrix gains a row of the form [1, 0, ..., 0, 1].

**Solving The Matrix**

The standard Gauss–Jordan elimination algorithm is used to bring the matrix to reduced row echelon form (RREF), but with an important modification. My algorithm always works with integers (not fractions or floats) and is exact. After each operation, each row in the matrix is put into simplified form, i.e. the GCD of all the numbers in the row is 1 or 0. The final matrix is in quasi-RREF where the leading coefficients need not be 1, but all the zeros are the same as in RREF. The key idea that makes integer-only operation possible is that when eliminating, the two rows involved are brought to a suitable common multiple. For example, to use *x* = [3, 1, 4, 5] to eliminate the first column from *y* = [2, 7, 1, 8], we compute 3*y* − 2*x* = [0, 19, −5, 14]. In the code, this functionality is found in Matrix, Matrix.gaussJordanEliminate.

**Extracting The Answer**

If the chemical equation has *n* terms and there is a unique solution, then the solved matrix will have *n* non-zero leading coefficients. From the way the matrix was set up, the solution is normalized so that the first term has a coefficient of 1. But if this results in another term having a fractional coefficient, then all the coefficients are multiplied by the smallest positive integer so that all coefficients are integers. What actually happens in the code is that the lowest common multiple (LCM) of all the leading coefficients is computed, and then the top left *n* × *n* of the matrix is made to become the LCM times the *n* × *n* identity matrix. In the code, this functionality is found in extractCoefficients.

**Exact Syntax**

To parse the formula, the string is first converted into a sequence of tokens, and then these tokens are parsed into a tree structure.

**Tokens**

These are the tokens and their definitions in regular expressions:

* Space: / +/ (tokenized but filtered out)
* Name: /[A-Za-z][a-z]\*/ (except “e” being a special name that starts with a lowercase letter)
* Number: /[0-9]+/
* Punctuation: /[+\-−^=()]/ (each letter is a different token, and both - and − mean minus)

**Grammar**

The following context-free grammar (CFG) describes the set of syntactically correct chemical formulas:

* Equation (root) = Term (Plus Term)\* Equal Term (Plus Term)\*
* Term = (ChemElem | Group)+ (Caret Number? (Plus | Minus))?
* Group = LeftParen (ChemElem | Group)+ RightParen Number?
* ChemElem = Name Number?