Assignment 2. Sets.

For this assignment an interpreter has to be written for the "command-language" shown further down. All "sentences" in this command-language are assignments for operations with sets of natural numbers.

Proceed as follows:

1. Think of which objects are necessary for this assignment, and specify the interface for each of these objects.
2. Implement the class List according to the specification given in the `ListInterface`. The necessary class `Node` is also given.
3. Give the specification and implementation of an ADT for a sorted list. The specification of this list should specify that its elements are sorted monotonically increasing. The implementation should, by using inheritance, implement this sorted list as a subclass of the class List.
   In this exercise it is required to implement all objects you have specified yourself and in which any amount of other object can be stored, with this sorted list.
4. Specify if useful a parameterized class. Use the interface `Data` and the class `VPException` to assign the right properties for those objects that you need to specify yourself. Use instances of the class VPException in case you want to throw an exception.
5. Make, using the objects, a design for the interpreter. Use the method of recursive descent (see the example at the end).
6. Get approval for the self-made interfaces, as well as the design of the interpreter, at the meeting with the assistant.
7. Once your design and interfaces have been approved, and you have implemented the interfaces, you can start with programming. First, completely write-out the parser part of your design. Which means, write a program that reads lines of input, and does nothing in case of a correct command, but returns a clear error-message in case there are incorrect commands.
8. Only when the parser works, must you proceed to extend it to an interpreter. The interpreter must recognize and execute commands written in the command-language specified below.

You can find the previously mentioned interfaces ListInterface and Data and the classes Node and VPException on Blackboard.

With the aid of the command-language specified further down, we can manipulate the natural numbers and the variables that are contained in such sets. The natural numbers can consist of any length. Only identifiers are allowed as names for variables.

There are four operators that can be used on sets in this language:

$$
\begin{array}{llllll}
+ & : & A + B & = & \{\, x \mid x \in A \lor x \in B \,\} & \text{union} \\
* & : & A * B & = & \{\, x \mid x \in A \land x \in B \,\} & \text{intersection} \\
- & : & A - B & = & \{\, x \mid x \in A \land x \notin B \,\} & \text{complement} \\
| & : & A \mid B & = & \{\, x \mid x \in A + B \land x \notin A * B \,\} & \text{symmetric difference} \\
& & & = & \{\, x \mid x \in (A + B) - (A \star B) \,\} &
\end{array}
$$

The operator '*' has a higher priority than '+', '|' and '-', whom have the same priority. The operators are left-associative.
Example: $A - B * Set1 + D$ is to be evaluated as $(A - (B * Set1)) + D$.

There are two sorts of **commands** available. For each of those an example is given:

1. Set1 = A + B − Set1 * (D + Set2)

    Calculate the set that is the result of the expression to the right of the '=' -sign and associate the variable with the identifier Set1 with this Set.

2. ? Set1 + Set2

    Calculate the set that is the result of the expression, and print the elements of this set: separated by spaces on a single line on the standard output.
    Do not print any other text or commas because this will interfere with the automated test.

Syntax command-language

We use the EBNF-notation for describing the syntax-portion of the command-language grammar. Furthermore the signs '<' and '>' are used for descriptions (for example <eof>).

program = { statement } <eof> ;
    A program is any arbitrary number of statements (commands) ended by the end of file.

statement = assignment | print_statement | comment ;
    A statement is an assignment-statement, a print-statement or a comment-line.

assignment = identifier '=' expression <eoln> ;

An assignment statement is an identifier, followed by the '=' sign, after which there is an expression, followed by an end-of-line.

print_statement = '?' expression <eoln> ;
    A print statement is a '?' followed by an expression and ended with an end-of-line.

comment = '/' <a line of text> <eoln> ;
    A comment is a line of text that starts with the '/' -sign and is closed by an end-of-line.

identifier = letter { letter | number } ;
    An identifier starts with a letter and only consists of letters and numbers.

expression = term { additive_operator term } ;
    An expression is a term, followed by zero or more terms. All terms are separated by an additive-operator.

term = factor { multiplicative_operator factor } ;
    A term is a factor, followed by 0 or more factors. All factors are separated by a multiplicative-operator.

factor = identifier | complex_factor | set ;
    A factor is an identifier, a complex factor or a set.

complex_factor = '(' expression ')' ;
    A complex factor is an expression between round brackets.

set = '{' row_natural_numbers '}' ;
    A set is a row of natural numbers between accolades.

row_natural_numbers = [natural_number { ',' natural_number }] ;
    A row of natural numbers is empty or a summation of one or more natural numbers separated by commas.

additive_operator = '+' | '|' | '−' ;
    An additive operator is a '+', a '|' or a '-' sign.

multiplicative_operator = '*' ;
    A multiplicative operation is a '*' -sign.

natural_number = positive_number | zero ;
    A natural number is a positive number or zero.

positive_number = not_zero { number } ;

A positive number does not start with a zero, does not have a sign, and has 1 or more numbers.

number = zero | not_zero ;
    A number is a zero or not a zero.

zero = '0' ;
    Zero is the number 0.

not_zero = '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' ;
    Not-zero is the number from the range 1 up to and including 9.

letter = 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' | 'L' | 'M' |
        'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z' |
        'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' |
        'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z' ;
A letter is a capital letter or a small letter.

Remarks

1. Spaces in natural numbers and identifiers are not allowed. Elsewhere, spaces have no function and they can be added to any line to increase readability.
2. When a faulty statement is discovered, then a clear error message needs to be printed, followed by a new-line. Explanatory error messages are also necessary to discover what the program exactly does in the test-phase (this is usually something else than what it should do).
3. The program has to read from standard input and write to standard output. The output must consist of one-line error-messages or one-line with numbers, separated with spaces and ended with an end-of-line after the last number.
4. Comments have to be ignored.

Recursive descent

We illustrate how to make a design by showing how a factor can be recognized with the method "recursive descent". With "recursive descent" the procedures that parse the input call each other recursively.

We can, for example, design a method factor() that:
1. returns a set of natural numbers as the result of the function if there is grammatically correct input, and
2. gives an exception, and throws an VPException if there is grammatically incorrect input.

When the method `factor()` parses a complex factor, the right expression has to be parsed, for which a term has to be parsed, for which another correct factor has to be parsed. So after 3 calls in between, `factor()` is called again (in other words there is recursion).

A first sketch of the method `factor()` is:

```
Set factor () throws VPException {
/* factor() reads, if possible, a correct factor of the input.
   If this succeeds the factor is evaluated and the resulting
   set is returned.
   If this fails, then an error-message is given and a
   VPException is thrown.
*/
    if (the next character is a letter) {
        read an identifier
        retrieve the set that belongs with that identifier
    } else
    if (the next character is '{') {
        read a set
    } else
    if (the next character is '(') {
        determine the set that is the result of the complex factor
    } else
        give a clear error-message
        throw VPEXception
    }
    RETURN the value of the set
}
```

Also, pay attention to the similarity between the syntax-definition of a factor and the design of `factor()`. The syntax-rules are a very thorough analysis/description of the input and only have to be converted to a Java description during the design process.

Handing in the assignment

When the program is completely done and has been tested thoroughly (at the very least use this testinput), you can upload all Java files as a single zip-file on Practool.