**Assignment 2. Sets.**

For this assignment an interpreter has to be written for a calculator with the "command-language" shown further down. All "sentences" in this command-language are statements for operations with sets of arbitrary big natural numbers. The calculator should be able to store an arbitrary number of variables (A variable being a combination of a name and a value.).

Proceed as follows:

1. Make a specification for an Identifier ADT and a Set<E> ADT. The type Identifier will be the type of the name and the type Set<E> will be the type of the value of each variable stored by the calculator.
2. From the API use (1) the class BigInteger to implement the arbitrary big natural numbers and (2) the class HashMap<K, V> to store the arbitrary number of variables.
3. Implement the class List<E> according to the specification given in the ListInterface<E>. The given class Node has to be used in the implementation of this class List. We provide a class ListTest which you can optionally use to test your implementation. You can find this class on Blackboard. Instructions on how to use it are at the end of this document.
4. Implement the class Set<E> with the class List<E>.
5. Use instances of the class APException in case you want to throw an exception.
6. Make, using the objects, a design for the interpreter. Use the method of recursive descent (see the example at the end).
7. Get approval for the self-made interfaces, as well as the design of the interpreter, at the meetings with the assistant.
8. 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.
9. 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 interface ListInterface, the class Node and the class APException on Blackboard.

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

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

| | | | | | |
|---|---|---|---|---|---|
| $+$ | : | $A + B$ | $=$ | $\{ x \mid x \in A \vee x \in B \}$ | union |
| $*$ | : | $A * B$ | $=$ | $\{ x \mid x \in A \wedge x \in B \}$ | intersection |
| $-$ | : | $A - B$ | $=$ | $\{ x \mid x \in A \wedge x \notin B \}$ | complement |
| $\mid$ | : | $A \mid B$ | $=$ | $\{ x \mid x \in A + B \wedge x \notin A * B \}$ | symmetric difference |
| | | | $=$ | $\{ x \mid x \in (A + B) - (A * B) \}$ | |

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 **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 like this are necessary in the test-phase to discover what the program exactly does (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 ending 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". "recursive descent" means that the methods 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 APException 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 APException {
/* factor() reads, if possible, a correct factor of the input.
   If this succeeds this factor is evaluated and the resulting
   set is returned.
   If this fails, then an error-message is given and a
   APException 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 APException
    }
    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 during the design process.

# Testing List implementation

In order to help you implement the ListInterface, we provide you with a unit test *ListTest.java*. You can download it from Canvas. **Usage of this test is completely optional.** You won't get any points for using it, but testing might help you to implement the list faster and with less errors.

If you name your implementation of the ListInterface *List.java*, you can automatically use these tests (if you have a different class name, you need to change all occurrences of the type "List" in the test file accordingly).

## *Test framework: JUnit*

The unit test uses JUnit (junit.org). Follow this tutorial on how to use JUnit in Eclipse. If you want to learn more about JUnit, look at the documentation.

## *Quickstart*

In short, you need to follow these steps to use our test suite in Eclipse:

1. Right-click on the package in which you want to create the test and select *New > JUnit Test Case*.
2. Fill "ListTest" into the name field and click *Finish*.
3. Eclipse prompts you to add JUnit to the build path. Approve this by clicking *OK*.
4. Paste the content from *ListTest.java* on Blackboard into this newly generated file.
5. Right-click on the newly generated file *ListTest.java* and select *Run as > JUnit Test*. Eclipse will open a list of tests which were passed and failed.

If anything is unclear follow the detailed tutorial at https://courses.cs.washington.edu/courses/cse143/11wi/eclipse-tutorial/junit.shtml

## *More testing*

The provided tests are not complete. You can add additional tests for lists in the same file. If you want to create a test for another class, you can use the file as a template.