Project: a Simple MAth Calculator (SMAC)

This project is about making a **S**imple **MA**th **C**alculator (**SMAC** in the sequel) with some interesting features. A tricky part of the project is provided to you and you must use the provided classes in your project.

SMAC functionalities

You are to implement the following functionalities. You are strongly encouraged to develop your code step by step, implementing each functionality one after the other in the order presented below. You can do all or only part of the sections described below. The sections are fairly independent so you can try to share the work in the team. Be aware that sharing work on the same project is tricky as you have to define a clear and stable interface between all the team members in order to be able to put the different parts together and make them work in the same context.

1. The core functionalities

SMAC works as a mathematical expression calculator. After started, the user can type in mathematical expressions, and SMAC will evaluate them and print the result. This is an example of the basic use of SMAC (user input is in bold):

```
Welcome to SMAC

> 2 + 3
5

> 5 + 3 * 4
17

> exit
```

```
Thank you for using SMAC
```

After started, SMAC prompts the user (with the sign >) and waits for the input. The user can then submit a mathematical expression, then press the enter key and get the result. SMAC can handle complex expressions with parenthesis:

```
> (2*5 - 7)*(9 + 7)/4
12
```

SMAC is using operator precedence to avoid non necessary parenthesis. For example, the following expression:

```
> 4*5 + 15/3 - 2^3
17
```

is equivalent to:

```
> ((4*5) + (15/3)) - (2^3)
17
```

In this part, you should only try to evaluate mathematical expressions made from numbers, operators (addition, subtraction, unary minus, multiplication, division, power) and parenthesis. In an another part, we will introduce more mathematical functions (like sin, cos, tan, etc.)

2. Managing the precision

The user can set the decimal precision to be used to format values on output by using the setprecision command:

```
> 1/3
0.333333

> setprecision 2
precision set to 2

> 1/3
0.33
```

The command setprecision called with no argument will show the current precision:

```
> setprecision
current precision is 2
```

3. Using variables

SMAC allows the user to define and use variables in mathematical expression. To define and assign to a variable, we use the let keyword together with the = sign:

```
> let x = 3.5
3.5

> let y = 10
10
```

The previous statements define two variables named x and y which are assigned to 3.5 and 10. From now on, we can use those variables in mathematical expressions:

```
> 2*x + y
17
```

The value assign to a variable can be the result of a complex mathematical expression (eventually involving variables):

```
> let z = 2*x + y
17

> z
17
```

It is possible to display the list of all variables by using the let command alone:

```
> let
x = 3.5
y = 10
z = 17
```

To change the value of a variable, simply use the let command again:

```
> z
17

> let z = 31
31

> z
31
```

Names for variable are made of letters and digits and the character underscore ('_') but the name must start by a letter. For example, these are valid variable names:

```
x   alpha   beta1  temp123   a_Long_Variable_Name
```

although these are not:

```
1x  _max  y-1  :W
```

Checking the lexical correctness of variable names is done by the token reader which is provided to you as a supporting file.

SMAC allows the user to delete a variable previously defined and assigned using the reset command:

```
> x
3.5

> reset x
x has been reset

> x
error: x is not a variable

> let
y = 10
z = 31
```

It is possible to reset multiple variables with one reset command:

```
> reset y z
y has been reset
z has been reset

> let
no variable defined
```

Calling reset with no argument will reset all the variables:

```
> let a1 = 100
100

> let a2 = 200
200

> let a3 = 300
300

> let
a1 = 100
a2 = 200
a3 = 300

> reset
a1 has been reset
a2 has been reset
a3 has been reset

> let
no variable defined
```

4. Managing errors

SMAC will show lexical, syntax or computation errors to the user by printing a message showing the error type. For example, using an invalid character will cause an error:

```
> let ! = 1
Lexical error: ! is not a valid character
```

Using an invalid keyword causes also an error:

```
> Let x = 1
Error: Let is not a variable
```

In the last example, the user typed in "Let" (capital 'L') instead of "let" (lower case letters) so SMAC thinks "Let" must be a variable name, but there is no variable of that name. Lexical errors are token error (for example, illegal variable name or unknown character). Syntax errors are related to the structure of the expression. For example, typing an incorrect mathematical expression will give the following error:

```
> 3 + 7 * / 2
Syntax error: malformed expression
```

Computation errors are likely to be limited to the division by 0:

```
> 5 / 0
Error: division by zero
```

5. Keeping track of the last value

Each time you compute a mathematical expression, its value is stored in a special variable. You can access the content of the variable using the keyword last:

```
> 5*(8-2)
30

> last
30
```

You can use the keyword last inside a mathematical expression or as the value to be assigned to a variable:

```
> 4*5 + 15/3 - 2^3
17

> (last + 3)/2
10

> let ten = last
ten = 10
```

The keyword last is not a regular variable, so you can't reset its value:

```
> reset last
Syntax error: last is not a variable
```

The keyword last refers to the last computed value and is not affected by non numerical command:

```
> 2^10
1024

> let
ten = 10

> last
1024
```

6. Storing variables

Sometimes it is useful to save some variable values for a future SMAC session. You can do so by using the save command:

```
> let x = 10
10

> let y = 20
20

> save "myfile"
variables saved in myfile

> exit

Thank you for using SMAC
```

After we exit from SMAC, we can start it again and load the content of the file "myfile" using the load command:

```
Welcome to SMAC

> let
no variable defined

> load "myfile"
myfile loaded

> let
x = 10
y = 20
```

Instead of saving all variables, it is possible to selected the ones you wish to save by simply passing them as extra arguments to the save command:

```
> let z = 30
30

> let
x = 10
y = 20
z = 30

> save "just-z-file" z
variables saved in just-z-file
```

A file created by the save command contains the SMAC let commands to (re)assign saved values to variables when loading the file. For example, the content of the file "myfile" from previous example is:

```
let x = 10
let y = 20
```

So the load command is just reading that file line by line and is evaluating each line as if they were input interactively by the user. Notice that the file "just-z-file" from the previous example should contains only one line:

```
let z = 30
```

The path of the file we save the variables into is always relative (like in **"just-z-file"**) and so the file must be saved in a default folder. The path of that folder should be easy to change in your code (i.e. should be defined as a constant in the main method). Using the command saved allows the user to see all the files saved until now:

```
> saved
myfile
just-z-file
```

7. Logging a session

SMAC allows you to keep track of the calculations you are performing by using the log command. When called, this command makes SMAC writing in a text file all the input and output occurring in the current session:

```
> log "mylog"
logging session to mylog

>> let x = 10.5
10.5

>> 10*x - 5
100
```

Notice that after we use the log command, the SMAC prompt changed from ">" to ">>" to remind us that we are currently logging the session. A call to the log function with no argument will display the name of the file the current session is been logging to:

```
>> log
mylog
```

To stop the logging, just use the log command with the special keyword end as argument:

```
>> log end
session was logged to mylog

> x
10.5
```

After we stop logging the session, the SMAC prompt gets its initial value back. If we look at the content of the file "mylog" we see all the input and output from the logged session:

```
>> let x = 10.5
10.5

>> 10*x - 5
100

>> log
mylog
```

As for the save command, the path of the file we save the session into is relative (like in **"mylog"**) and the file must be saved in the same default folder like the one for the files storing the saved variables. Using the command logged allows the user to see all the relative logged files created until now:

```
> logged
mylog
```

To distinguish between saved files and logged files, you should use two sub-folders in the main default save folder we mention before.

8. Adding more mathematical functions

Until now, we only shown the use of operators in the mathematical expressions SMAC is handling. You can try to update part 1 by adding the common mathematical functions like sin, cos or tan. Introducing mathematical function should not lead to any change in your mathematical expression evaluator: function like sin or cos can be viewed as prefix operators (like the unary minus) and can be easily evaluated because they have their parameter in parenthesis. You can introduce as many mathematical functions as you wish (sin, cos, tan, abs, etc.).

SMAC implementation

Lexical parser

The lexical parser is the part of the program which is reading the input string from the user and split that string in tokens. This part is provided in the following files:

- Token.java: a file for the Token datatype
- Tokenizer.java: a file for the lexical parser
- LexicalErrorException: a file for lexical exception
- TokenException.java: a file for token exception
- TestTokenizer.java: a sample program to see how the lexical parser works

You must include those classes in your project. It is likely you won't have to create new tokens but only get some from the Tokenizer class. The code in main.cpp is demonstrating how to use those classes.
The lexical parsing of the input string consists in splitting the string in pieces called tokens. Given a string, a Tokenizer object on that string is able to deliver the list of tokens in the string one by one. A token can have one of the following type:

- ERR: this token is produced when an error in encountered
- EOL: this token is the end of line
- IDENT: this token is an identifier. This kind of token contains the identifier (as a string)
- VALUE: this token is a double value. This kind of token contains the value (as a double)
- STRING: this token is a string. This kind of token contains the string (as a string)
- EQUAL: this token is the '=' delimiter (equal)
- OPAREN: this token is the '(' delimiter (open parenthesis)
- CPAREN: this token is the ')' delimiter (close parenthesis)
- EXP: this token is the '^' delimiter (the power operator)
- MULT: this token is the '*' delimiter (the multiply operator)
- DIV: this token is the '/' delimiter (the division operator)
- PLUS: this token is the '+' delimiter (the plus operator)
- MINUS: this token is the binary '-' delimiter (the binary minus operator). This token is returned when the **binary** minus operator is encountered
- UMINUS: this token is the unary '-' delimiter (the unary minus operator). This token is returned when the **unary** minus operator is encountered

Mathematical expression evaluator

The algorithm you are to use to evaluate mathematical expressions in SMAC will be explained in class. This algorithm is using two stacks, one stack of numbers and one stack for operators and parenthesis. You can use the Stack class from the Java API. The algorithm takes a tokenizer as input and works as follow:

1. While there are still tokens to be read in,
1.1 Get the next token.
1.2 If the token is:
1.2.1 a number: push it onto the value stack.
1.2.2 a variable: get its value, and push onto the value stack.
1.2.3 a left parenthesis: push it onto the operator stack.
1.2.4 a right parenthesis:
1 While the thing on top of the operator stack is not a left parenthesis,
1 Pop the operator from the operator stack.
2 Pop the value stack one or twice, getting one or two operands (depending on the arity of the operator)
3 Apply the operator to the operands, in the correct order.
4 Push the result onto the value stack.
2 Pop the left parenthesis from the operator stack
*1.2.5 An operator (call it **thisOp**):*
*1 While the operator stack is not empty, and the top thing on the operator stack has the same or greater precedence as **thisOp**,*
1 Pop the operator from the operator stack.
2 Pop the value stack one or twice, getting one or two operands (depending on the arity of the operator)
3 Apply the operator to the operands, in the correct order.
4 Push the result onto the value stack.
*2 Push **thisOp** onto the operator stack.*
2. While the operator stack is not empty,
1 Pop the operator from the operator stack.
2 Pop the value stack one or twice, getting one or two operands (depending on the arity of the operator)
3 Apply the operator to the operands, in the correct order.
4 Push the result onto the value stack.
3. At this point the operator stack should be empty, and the value stack should have only one value in it, which is the final result.

Syntax analysis

The first step to evaluate an expression in SMAC is to do the lexical parsing of the input string provided by the user. This is done using the provided classes Token and Tokenizer. Then you are to analyze the stream of tokens to check if the expression or command input by the user is valid. This is usually a rather complex process but for SMAC the syntax analysis remains easy to perform. Basically, it works as follow:

- peek the first token
- if this token is a number or an open parenthesis, process the mathematical expression
- if this token is an identifier (a name) then:
  - if that identifier is a variable or the keyword last, process the mathematical expression
  - if that identifier is not a variable, it must be a keyword, so process the command accordingly
- if none of the previous case apply, then it must be a syntax error

For some expression like the definition/assignment, there are two parts in the expression: the keyword, the name of the variable and the "=" sign first, followed by a mathematical expression

Storing variables

The easiest way to manage variables is to use the map container from the STL.
The map container allows you to pair a string (the variable) with a double (the value of the variable). For keywords you can use the setcontainer because given a string, you only need to know if that string is in the set of the keywords, which is one the basic functionalities on a set.

Managing errors

To manage errors you may use exception together with the try-catch construct. Although we didn't study exception in class, you can try to learn this concept by yourself and use it in a simple way in the project.

Project structure

Your program should be structured using different classes. For example, you should have a class Evaluator to evaluate an input line typed in by the user. As a separate class, you should have a MathematicalEvaluator class to evaluate mathematical expression. Because you need to handle operators in the mathematical expression evaluator, you should have a FunOp class to implement the concept of operator/function. According to the mathematical expression evaluator algorithm, a FunOp object should have a name (like "+") an arity (the arity is the number of operands, like 2) and a priority (an integer).

- LexicalErrorException.java
- TestTokenizer.java
- Token.java
- TokenException.java
- Tokenizer.java

Grading summary

Participants	1
Submitted	1
Needs grading	1

View all submissions | Grade