MaTheX2Java : Online LaTeX to Java converter

Compilers 2018 / 2019

Francisco Maria[*]
Department of Computer Science
Faculdade de Engenharia da Universidade do Porto
Rua Dr. Roberto Frias
4200-465
Porto, Portugal
up201607928@fe.up.pt

July 19, 2019

Abstract

LaTeX[2] is an application to write well-prepared documents, which has been widely used for communication and publication of scientific documents in many fields, for instance, mathematics, computer science, political science, philosophy or physics. With the growing use of this application in the specific fields of mathematics and computer science, it arouse the necessity of converting mathematic formulas into programming code to calculate or to simply have a real implementation of the formulas in programming code.

***K*eywords** Compiler · ANTLR4 · JavaScript · NodeJS · LaTeX · Java · Amsmath

1 Introduction

MatheX2Java is a project which started to be developed in late March 2019. It is a web application which converts LaTeX code - from the amsmath package - to Java code, in real-time. It converts mathematic formulas into java methods which can then be invoked and executed.

It's first design was very simple, which only considered raw amsmath code. In a second, deeper approach to the project specification, the application design was upgraded, considering the peculiar properties of the Java programming language. Hence, it was implemented an auxiliary grammar which allows annotations, in the form of LaTex comments to specify important details of how the Java code must be generated.

2 Grammars

MaTheX2Java's has two dedicated grammars: the annotation and the amsmath grammars. The user input must correctly match these in order to successfully generate the Java code. Each of these grammars have important elements which should be noted and strictly followed so that the code generation is successful.

It is also important to specify that both of these grammars support a number of different Java data types, allowing the user to define the types of the variables and the generated method return types as they wish. The table 1 specifies the data types which are accepted by the grammars:

[*]francismaria.github.io
[2]https://www.latex-project.org/

Accepted Data Types
int
float
double
long
short
real

Table 1: MaTheX2Java's accepted data types

Note: each of these data types is treated as *case-insensitive*.

The data type "real" is the *default* data type for MaTheX2Java's code generation. It represents a "double".

This data type was created so that a user with none or little experience with the Java programming language can still specify how it wants a variable or the method return type generated without having to care about the different Java data types. This way, the user needs not to know what the other data types represent in the Java language in order to generate Java code.

When a variable is not specified of its type, then it is automatically assigned the *default* "real" data type to it, and it is then treated as a *"double"-type* variable.

2.1 Annotations

The annotations are **optional** LaTeX comments which specify important details of the Java code generation. It allows a user to specify a name for a method, a method return type and to specify the types of the variables which are involved in a method, as well as memory allocation for *array-type* variables.

This way, we can group the annotations in two distinct groups of operation:

2.1.1 Method Annotations

Correspond to specific details about the method that will be created to represent the mathematic formula, more precisely, the name and the return type of the java method that will represent the implementation of the formula.

This type of annotation must be declared before the 'begin' tag of the amsmath equation definition.

- **Name** : if present, assigns the specified name to the generated Java method. Otherwise, it assigns a custom (template) name.

```
% name : myEquation
\begin{equation}
    ...
\end{equation}
```

- **Return** : if present, sets the return type of the generated method to the specified input type. Otherwise, it sets the return type of the generated method to the default type - "real".

```
% return : int
\begin{equation}
    ...
\end{equation}
```

2.1.2 Variable Annotations

Correspond to specific details about how the variables in the method will be declared in the generated Java code.

A user may define the type of the variable and, if the variable is an array, it also defines the memory allocation of the array.

These types of annotations are written before the equation line, in the following format:

% **[** *variable (, variables)** **] :** *data type*

Each variable inside the same brackets will have the same specified data type. This annotation allows *multiline* comments to specify different variable types in a *user-friendly* way. This way, one can write these types of annotations in the following way:

```
\begin{equation}
    % [x, y] : int
    % [z, w] : short
    a = 3 + x + y - z * w
\end{equation}
```

- **Single type**: allows to define the declares type of a variable.

```
\begin{equation}
    % [x] : int
    ...
\end{equation}
```

- **Multi type**: allows to define the declares type of a variable.

```
\begin{equation}
    % [x, y] : int, [w] : long
    % [z] : short
    ...
\end{equation}
```

- **Array type**: allows to define variables as arrays, as well as the space allocated for each dimension.

```
\begin{equation}
    % [x] : int[10]
    % [y] : short[10][5]
    ...
\end{equation}
```

2.2 Amsmath

The amsmath grammar matches all of the accepted amsmath elements. It allows a number of operations, including simple and complex mathematics functions, although these have specific rules which are listed in the next section. All of the equations are converted to methods in the generated Java code, allowing to be individually invoked according to the user's will, assuring flexibility to the user.

The table 2 specifies the accepted amsmath operations:

`\frac`	`\sqrt`	`\ln`
`\exp`	`\log`	`\sum`
`\sin`	`\cos`	`\tan`
`\sinh`	`\cosh`	`\tanh`
`\int`	`\iint`	`\iiint`
`\arcsin`	`\arccos`	`\arctan`

Table 2: MaTheX2Java's accepted amsmath operations

Note: the `\int`, `\iint`, `\iiint` tags are recognized by the grammar but are not yet implemented in the code generation.

3 Rules

MaTheX2Java requires special rules when dealing with certain operations. In this section these rules are specified and described.

3.1 Variables

Variables are defined as a sequence of letters (*case-sensitive*), numbers and *underscores*, which always start by a letter. The variables need not to be defined in the annotations. If one is not defined, then it is treated as a scalar variable.

Every variable in the right side of the equation is an argument to the method and is declared in the main function of the generated Java code class. Using this design, the user may use the same variable for different equations, hence the variable is only defined once, in the main method. Whenever the user wants to change the value of this variable, it only needs to change it once, rather than changing each corresponding local variable of each method it appears.

The left-side variable, represents the result of the mathematic operation, it will hold its value. It is a local variable for the method. Thus, it is not possible to use this variable in the right-side of the equation.

Below are correct and error examples of the variables declaration.

```
\begin{equation}
    result = c          --> c is a scalar variable
\end{equation}


\begin{equation}
    % [c] : int[10]
    result = c_{2}      --> c is a one-dimension array variable
\end{equation}


\begin{equation}
    % [c] : int[10][10]
    result = c_{0, 5}   --> c is a two-dimension array variable
\end{equation}


\begin{equation}
    result = c_{2}      --> this will raise an error.
                        --> c is a scalar variable
                        --> (it was not defined as an array variable)
\end{equation}


\begin{equation}
    % [c] : int[10]     --> this will raise an error.
    result = c_{14}     --> c is a one-dimension array variable with allocated space
                        --> of 10 elements. Thus, an out-of-bounds violation occurs.
\end{equation}


\begin{equation}
    % [c] : int[10][5]  --> this will raise an error.
    result = c_{1}      --> c is a two-dimension array variable but is being treated
                        --> in the equation as a one-dimension array variable.
\end{equation}
```

3.2 Powers

The use of powers is accepted by MaTheX2Java. Every element may contain any power, using the following rule:

(base_element) ^ { power_element }

Note: the parenthesis around the base_element may be disregarded if the element is a single member (not a composition of members).

3.3 Logarithms

The following amsmath logarithm operations are allowed: \log and \ln. Both of these operations must obey the following syntax:

[log | ln] (_ { [variable | integer] })? { [members] }

The lower part of the logarithms is **optional** and it can only be a variable or an integer. The body of the algorithm supports every other operation available in the amsmath grammar and may even contain mathematic operations between the members (addition, subtraction, etc.).

```
% name: logExample
\begin{equation}
    result = \log{0.5} + \ln{0.4 + x - \sin{0.1}}
\end{equation}
```

Note: In the current MaTheX2Java release, the lower part of the logarithm is not fully operational due to some Java restrictions, thus, the lower logarithm part is not accepted and thus it shall not be used.

3.4 Trigonometry

Trigonometric functions are crucial elements within the mathematic formulas, thus MaTheX2Java's also accepts them. A high number of different trigonometric functions are currently supported by MaTheX2Java. These are presented in table 3.

To use any of these functions, the user should specify them obeying the following syntax:

[function_tag] { [members] }

Below is an example of the use of trigonometric functions.

```
% name: trigonometryExample
\begin{equation}
    result = \sin{2 + b} - 4 * \arctan{0.2} - 8 \times[3] \cos{2 - b + \arcsin{0.1}}
\end{equation}
```

It shall be noted that the members (arguments) to the trigonometric functions can be operations of other accepted amsmath elements, even other trigonometric functions.

\sin	\cos	\tan
\sinh	\cosh	\tanh
\arcsin	\arccos	\arctan

Table 3: MaTheX2Java's supported trigonometric functions

[3]'\times' and '*' are the symbols used for the multiplication operation. The user has the will to choose the one it is most confortable with.

3.5 Summation

Summation is an important feature of MaTheX2Java. Its implementation requires a separate method which will be invoked from the method representing the equation it is in, where a for cycle will be executed with the correct specifications of the summation. The method representing the summation will have important specific elements to its correct operation: a **lower bound** and an **upper bound**.

$$\sum_{\textit{lower bound}}^{\textit{upper bound}} \texttt{summation\_body} \tag{1}$$

3.5.1 Lower Bound

In this MaTheX2Java current release, the **lower bound** specification must follow the below syntax:

$$\textit{lower bound} = \textit{value}$$

The value of the **lower bound** must be an **integer** or a **variable**, only.

3.5.2 Upper Bound

The summation **upper bound** must be an **integer**, **variable** or **infinity**, no operations are allowed in this release. However, complex operations on the summation upper bound will be available in a future release.

$$\textit{upper bound} = \textit{value}$$

3.5.3 Summation Body

There are no restrictions for the summation body elements. It may have any of the elements that the equation body can have. Every operation is available inside the summation body, in fact there can be summations inside summations.

3.5.4 Summations inside Summations

This is an important feature of MaTheX2Java, the ability of having summations inside summations. The implementation of this feature relies on having a sequence of having separate summation functions invoking other (summation) functions, inside their summations body.

Note: Each summation is a separate method.

$$\sum_{i = v1}^{u1} \sum_{j = v2}^{u2} \dots \tag{2}$$

In "chain" summations, each summation lower bound variable is passed to the child summation as an argument, in order to be accessed in the child summation body. This way, it can be assured that the index variables (lower bounds) of the parent summations can be accessed by the child summations.

With this design the user will have the autonomy to edit the values of each summation in a comfortable, *user-friendly* way.

```
% name: singleSummationExample
\begin{equation}
    % [a] : int[10]
    result = \sum_{i = 0}^{10}{a_{i}}
\end{equation}
```

3.6 Factorial

The factorial function is also supported by MaTheX2Java. Its argument may be a single element or a set of operations of elements. To specify the factorial function over some element, the symbol "**!**" must be used after the element (right-side). Below is the factorial function syntax:

(*element (operator element)?*)**!**

Note: the parenthesis around the elements shall only be used when there is an operation. For single elements, then no parenthesis are needed.

It shall be noted that only **integers** and **variables** are accepted as an argument to the factorial method. If a variable, is not of the the type "integer", then an automatic cast to an "int" will be added.

3.7 Roots

Mathematic roots are accepted by the MaTheX2Java's application. To specify a root, one should follow the below syntax:

\sqrt ([root])? body

Note that the root element inside the brackets is **optional**. If none is specified, then MaTheX2Java automatically assumes that it is a square root (root = 2).

The only members accepted as a root element are **integers** or **variables**, any other element is not accepted.

If a variable is not of the **integer** type, then an automatic cast to an "int" is added, allowing the user to operate with the variable in the desired type and to automatically have a correct type as a root member.

3.8 Method Invocation

In order to allow the generated equations methods to interact with one another, MaTheX2Java supports the invocation of different method within the same generated file. This allows to have a well-structured and scalable file, separating the different generated formulas to later reuse.

To invoke a method, one should define it before the method that is invoking it. Below is an usage example.

```
% name : func
\begin{equation}
% [c] : short
% [d] : int           --> This method, to be invoked, must be written before
a = c + d             --> the method that is invoking it
\end{equation}

\begin{equation}
% [l]: short
% [d] : int
a = func(l, d) + k   --> Invocation of the "func" method
\end{equation}
```

To invoke the "func" method, the user shall define this equation method before and, when in the current method definition, invoke the method with the specified name and with the correct arguments (in the illustrated example, the arguments to the method "**func**" is "**c**" and "**d**").

A method, to be successfully invoked, shall be specified with the correct arguments type, otherwise an error will be raised and an error message will appear. In the above case, if "**l**" was not of the type "**short**" an error would be raised and the java file generation would fail.

4 Future Improvements

This application will be available online. It is first needed to resolve some web development issues, for instance, security against possible attacks to the correct operation of the application, and some update on the styling of the web page, as well as to finish the implementation of the logarithms lower part.

Some improvements in the grammars are also required, for example, the recognition of pi summations, more flexibility to the user, allowing to define the specific bytes of the variables or even their initialization value, including the array variables.

5 Conclusion

This application will have a positive impact on users who use LaTeX to write mathematic formulas and computer science/mathematic researchers and students who need real implementations and simulations of the formulas in an easy and confortable way.

By using MaTheX2Java, one can easily transform complex mathematic formulas into well-defined, structured Java code, and reuse that Java code when needed. The fact that the application can generate Java code in seconds is a major advantage to any user who wants a correct, effective and functional implementation of its mathematic formulas.

The academic environment will make good use of this application as it will allow the users tohave a new simple approach to transform LaTeX mathematic formulas representation code into fully operational Java code.

It is expected that the application will soon be released to the public with its own homepage, and possibly make MaTheX2Java an *open-source* project in order to encourage the users to participate and contribute in its developement and to clearly expose the code in which is built.

6 Acknowledgments

To my Family, Girlfriend and Friends, for their endless Love.

A Appendix : Use Cases

Some example use cases :

A.1 Power Series

$$\sum_{n=0}^{\infty} b_n x^n \tag{3}$$

The correspondent MatheX2Java input code:

```
\begin{equation}
    % [b] : int[10]
    result = \sum_{n = 0}^{\infty}{b_{n} \times x^{n}}
\end{equation}
```

It can be noted that the variable *b* needs to be declared as an array or else it will raise an error. The variable n, as it not declared then it will be declared in java code as a double, which is the default type for variables that were not defined in annotations.

Listing 1: Power series generated code

```
import java.lang.Math;
import java.util.Arrays;

public class GeneratedAmsmath_18_7_2019_23_18_43 {

        public static int powerSeries(int[] b, double x){
                int result;

                result = (int) (summation_MX2J_0(0, b, x));

                return result;
        }

        public static double summation_MX2J_0(int lowerBound, int[] b, double
            x){
                double sum = 0;
                int n = lowerBound;
                final int INFINITY = 10;

                for(; n < INFINITY; n++){
                        sum += b[n] * Math.pow(x, n);
                }

                return sum;
        }

        public static void main(String[] args){

                // ---- powerSeries ----

                int b[] = new int[10];
                Arrays.fill(b, 1); // initializes all the array elements to 1
                double x = 1.0;

                powerSeries(b,x);
        }
}
```

A.2 Multiple Summation

$$\sum_{i=0}^{10}\sum_{j=0}^{5}\sum_{k=2}^{N} b_k + j - i \tag{4}$$

Below is the input code for this mathematic formula.

```
% name : tripleSummation
% return : int
\begin{equation}
% [b] : int[10]
result = \sum_{i = 0}^{10}{ \sum_{j = 0}^{5}{ \sum_{k = 2}^{N}{ b_{k} + j - i} } }
```

The generated code is presented below.

Listing 2: Multiple summation generated code

```
import java.lang.Math;
import java.util.Arrays;

public class GeneratedAmsmath_18_7_2019_23_23_57 {

        public static int powerSeries(int[] b, int N){
                int result;

                result = (int) (summation_MX2J_0(0, 10, (int)N, b));

                return result;
        }

        public static double summation_MX2J_0(int lowerBound, int upperBound,
            int N, int[] b){
                double sum = 0;
                int i = lowerBound;

                for(; i < upperBound; i++){
                        sum += summation_MX2J_1(0, 5, i, N, b);
                }

                return sum;
        }

        public static double summation_MX2J_1(int lowerBound, int upperBound,
            int i, int N, int[] b){
                double sum = 0;
                int j = lowerBound;

                for(; j < upperBound; j++){
                        sum += summation_MX2J_2(0, N, j, i, b);
                }

                return sum;
        }

        public static double summation_MX2J_2(int lowerBound, int N, int j,
            int i, int[] b){
                double sum = 0;
                int k = lowerBound;
                for(; k < N; k++){
                        sum += b[k] + j - i;
                }

                return sum;
        }

        public static void main(String[] args){

                // --- powerSeries ---

                int b[] = new int[10];
                Arrays.fill(b, 1); // initializes all the array elements to 1
                int N = 1;

                powerSeries(b,N);
        }
}
```

A.3 Second Degree Equation

$$ax^2 + bx + c \tag{5}$$

Below is the input code for this mathematic formula. Note that **\times** and the symbol "*" can be used for the multiplication operation.

```
% name: secondDegreeExample
\begin{equation}
result = a \times x^{2} + b * x + c
\end{equation}
```

The generated code is presented below.

Listing 3: Second degree equation generated code

```java
import java.lang.Math;
import java.util.Arrays;

public class GeneratedAmsmath_18_7_2019_23_37_9 {

        public static double secondDegreeExample(double a, double x, double b,
             double c){
                double result;

                result = (double) (a * Math.pow(x, 2) + b * x + c);

                return result;
        }

        public static void main(String[] args){

                // --- secondDegreeExample ---

                double a = 1.0;
                double x = 1.0;
                double b = 1.0;
                double c = 1.0;

                secondDegreeExample(a,x,b,c);
        }
}
```

A.4 Sin(x) MacLaurin series

$$\sum_{n=0}^{\infty} \frac{x^{2n+1}}{(2n+1)!} \tag{6}$$

Below is the input code for this mathematic formula.

```
% name: sinxMaclaurinSeries
\begin{equation}
result = \sum_{n = 0}^{\infty}{ \frac{x^{2 \times n + 1}}{(2 \times n + 1)!}}
\end{equation}
```

The generated code is presented below.

Listing 4: Sin(x) generated code

```
public static double sinxMaclaurinSeries(double x){
        double result;

        result = (double) (summation_MX2J_0(0, x));

        return result;
}

public static double summation_MX2J_0(int lowerBound, double x){
        double sum = 0;
        int n = lowerBound;
        final int INFINITY = 10;

        for(; n < INFINITY; n++){
                sum += ((Math.pow(x, 2 * n + 1))/(factorial_MX2J((int)
                    (2 * n + 1))));
        }

        return sum;
}

public static int factorial_MX2J(int n){
        if(n < 0)
                return 0;
        else if(n == 0)
                return 1;
        else
                return (n * factorial_MX2J(n-1));
}

public static void main(String[] args){

        // ---- sinxMaclaurinSeries ----

        double x = 1.0;

        System.out.println(sinxMaclaurinSeries(x));
}
```

A.5 Arctan(x) MacLaurin series

$$\sum_{n=0}^{\infty} \frac{x^{2n+1}}{(2n+1)!} \tag{7}$$

Below is the input code for this mathematic formula.

```
% name: arctanxMaclaurinSeries
\begin{equation}
result = \sum_{n = 0}^{\infty}{ \frac{ (-1)^{n} \times x^{2 \times n + 1} }{2 \times n + 1}}
\end{equation}
```

The generated code is presented below.

Listing 5: Arctan(X) generated code

```
import java.lang.Math;
import java.util.Arrays;

public class GeneratedAmsmath_19_7_2019_0_2_49 {

        public static double arctanxMaclaurinSeries(double x){
                double result;

                result = (double) (summation_MX2J_0(0, x));

                return result;
        }

        public static double summation_MX2J_0(int lowerBound, double x){
                double sum = 0;
                int n = lowerBound;
                final int INFINITY = 10;

                for(; n < INFINITY; n++){
                        sum += ((Math.pow((-1), n) * Math.pow(x, 2 * n + 1))
                            /(2 * n + 1));
                }

                return sum;
        }

        public static void main(String[] args){

                // --- arctanxMaclaurinSeries ---

                double x = 1.0;

                arctanxMaclaurinSeries(x);
        }
}
```