# Symbolic Differentiation and Code Generation for *Modelica* Models

**SiScLab-18**

Andrea Hanke, Akshay Paranjape

February 26, 2018

## Abstract

Simulation is a widespread tool in many of today's industries. The act of simulating something first requires that a mathematical model is developed. The modeling language *Modelica* is used by many industries to represent such models. Each model represents the key characteristics, behaviours and functions of the selected physical or abstract system or process. Automobile companies such as Audi, BMW, Daimler etc. design for example energy efficient vehicles, or improve air-conditioning systems, using simulation models, which often contain differential equations. In this project we have restricted ourself to a small part of *Modelica*: The algorithm section in a function definition. In this report we explain our approach of calculating the derivatives of those functions. We used the concept of an Abstract Syntax Tree (AST) to interpret the input equation specified in *Modelica*'s algorithm section and differentiated it using the basic concept of symbolic differentiation i.e. the chain rule. At last, we generated C++ code for the Jacobian of the input equation from *Modelica*.

## 1 Introduction

Akshay Paranjape

*Modelica* is an object-oriented mathematical modeling language for component-oriented modeling of complex systems, e.g., systems containing mechanical, electrical, electronic, hydraulic, thermal, control, electric power or process-oriented subcomponents [1]. *Modelica* is a modeling language rather than a conventional programming language. It differs from semantic specification of ordinary programming language since modelica equation based models are not programmed in usual sense. Instead, modelica is a language primarily used to specify relations between different objects in a modeled system.

```
function SiScLab
    input Real x;
    output Real y;
algorithm
    y := 4*x*x + 3*x;
end SiScLab;
```

Figure 1: *Modelica* sample function

In this project, we restrict ourself to very small subset of Modelica, suited to formulate simple scalar *Modelica* function of type $F : R^n \to R$. In reality, *Modelica* is used to deal with complex equations. Figure 1, shows an example of *Modelica* function. The dynamic run time behaviour of modelica is based on hand implementation of programming language (such as C, C++, Java) of the primitives referred to by the modelica language. In this project, our aim is to generate code for evaluation of the model equation of *Modelica* and its Jacobian using symbolic differentiation.

Generated Code for *Modelica* model

```
void SiSc_Jacobian(double &dy_dx,
   const double x)
  {
    dy_dx = 8.0*x + 3
  }
```

We first build computation graph tree to interpret the input of *Modelica* model.This is done using Abstract Syntax Tree (AST) explained in Section 2.1. ASTs can further be used to evaluate model equation and find it Jacobian. Later, we generate C++ code for the differentiated equation which is explained in Section 4.

# 2 Approach

Akshay Paranjape

## 2.1 Abstract Syntax Tree

As mentioned in the problem statement, we have our input as an equation. In order to store this input in intermediate representation of data structure we make use Abstract Syntax Tree (AST). In our situation, AST is a binary tree representation. It holds the key tokens from the input stream and record grammatical relationships.

ASTs have following properties :

- They are dense, i.e. they do not contain any unnecessary node.
- Convenient : They are easy to walk. The visitor pattern explained in Section 3.2 are used by us to visit the tree.
- They emphasize operators, operands, and the relationship between them rather than artifacts from the grammar [2].

ASTs differ from parse tree. A parse tree is an ordered rooted tree that represent the syntactic structure of a string according to some context-free grammar. It contains all the information of the input whereas ASTs support more on the abstract relationships between the components of the equation. Different languages may have different parse tree but same abstract syntax tree.

Lets start with a simple example, lets make an AST for $x = 0$. There is only one operation in $x = 0$; and hence, just one subtree (see Figure 2 )

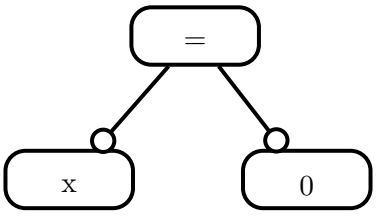


Figure 2: Abstract Syntax Tree for $x = 0$

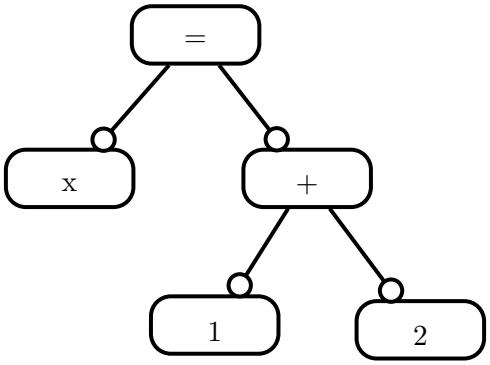


Figure 3: Abstract Syntax Tree for $x = 1 + 2$

Assignment $x = 1 + 2$;, on the other hand, has two operations: an assignment and an addition. We know that there should be two sub-trees, one for each operation. The semantics of assignment dictate that the right-side expression be evaluated before the assignment (=). To encode `X` happens before `Y` we make `X` lower than `Y` in the tree. To perform the assignment, we first need the value of the right child. This resembles the precedence of the = and + operators (+ has higher precedence). Take a look at Figure 3 We make use of this information in Section 3. (cf. operation order in C++ implementation). The same rule applies for operators within the same expression. Operators with higher precedence appear lower in the AST. For example, in equation $y = 4 * x^2 + 3 * x$, multiplication is lower in tree as compared to addition (see Figure 4).

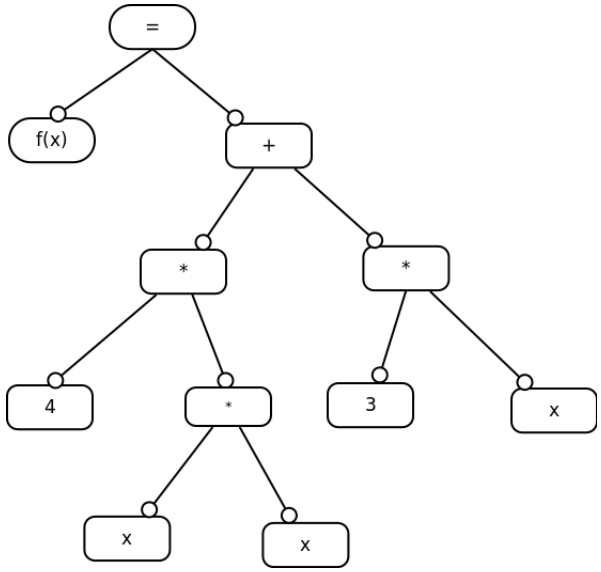


Figure 4: Abstract Syntax Tree for $y = 4 * x^2 + 3 * x$

### 2.2 Visitors Pattern

We build AST in C++ language using it's polymorphic and inheritance property. Class members and member functions are described in Section 3.1. Later task of problem statement is to use our previously build AST, to differentiate the input equation using the principles of symbolic differentiation. This involves operation to be performed on class objects in the AST. We want to avoid polluting the classes with this operations. One such technique for such purpose is Visitor Design Pattern.

In C++ implementation for AST (cf. Section 3.1), we have abstract functions which implies that the member function can we overridden by children classes. Visitor Pattern lets you define a new operation without changing the classes of the element on which it operates. This technique, for which the operation executed depends on : the request name and the type of two receivers (the type of Visitor and type of the element it visits) is called double dispatch.

## 3 Realization

Akshay Paranjape

Previous section has described the approach that we are going to follow, this section explains our implementation in detail.

We build our AST using the inheritance property to define class and sub-classes. Figure 5 shows the tree diagram. Class Node is the root class parent to all the sub-classes, which says that every node and leaves in our tree is an object of class Node.

`Node`

```
class Node
{
private :
    Node* parent ;
    bool isLeftChild
    bool isRightChild

public :
    void setParent();
    virtual Node* getRightChild();
    virtual Node* getLeftChild();
    virtual Node* copyNode();
}
```

Every Node has member parent pointer which point to its parent node. Root node has pointer set as NULL. It has two boolean member which tells whether the class has left or right child.

Member function of `Node` class `setParent` sets the parent for the object. When we make a `new Node` the parent is always set as `NULL`. `getRightChild` and `getLeftChild` are both `virtual` function implying that the derived classes with the same member function will have their own implementation. `CopyNode` member function is very important for Visitor Pattern where we have to build a new tree (for example in case of Differentiation).

As seen in Figure 5 class Node has further derived classes namely Function Node, Number Node, Variable Node, Binary Node and Unary Node. Our input equations are of the type `functon name = expression` . Class Func-

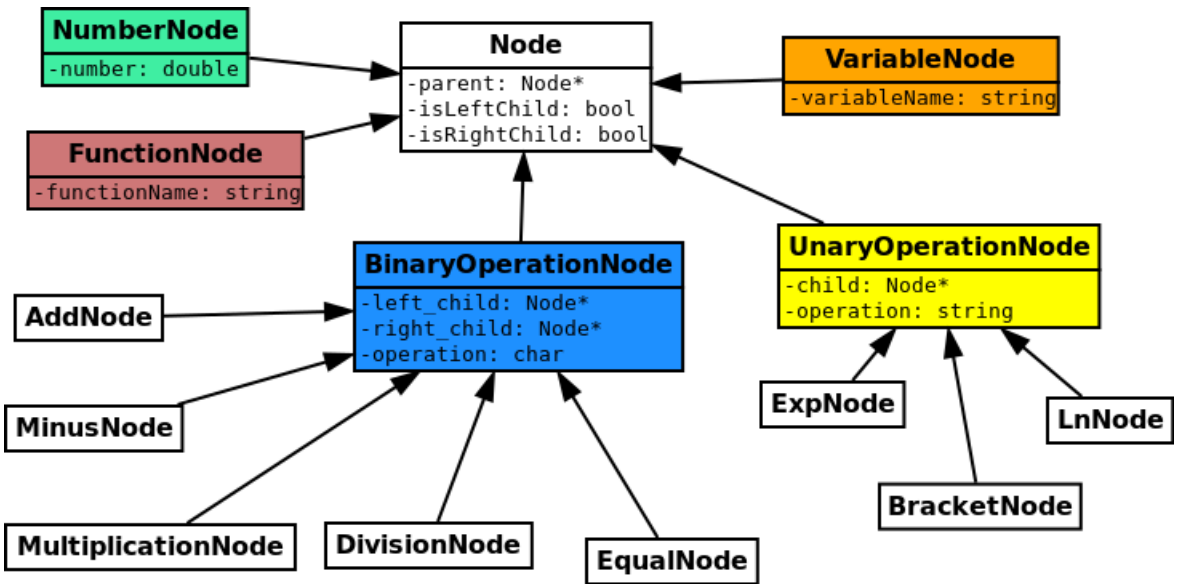


Figure 5: Class Diagram for Abstract Syntax Tree

tionNode has member `string functionName` which stores the name of the function. `Class NumberNode` and `class VariableNode` are the leaves of the tree which stores number and variable in the expression respectively.

Class BinaryOperationNode includes all the Binary operation like addition, subtraction, multiplication , division, exponential in the order of precedence as described in Section 2.1.

`BinaryOperationNode`

```
class BinaryOperationNode
      : public Node
{
private:
      Node * left;
      Node * right;
      char operation;
public:
  BinaryOperationNode();
  ~BinaryOperationNode();
  void setOperation(char op);
};
```

BinaryNode has further derived classes for different operation as can be seen in Figure 5. We may have unary operation like logarithm, exponential, therefore we also included UnaryOperationNode.

### 3.1 Graphical Process of building building AST

Here, we have described the process of building an AST with the help of an example. Let say we want to build AST for the equation $y = 4 * x^2 + 3 * x$. According to the algorithm, our code searches for equal sign and stores it in a `BinaryOperationNode` (also the root Node of the tree) and stores the name of the function in `FunctionNode`. Then it searches for operation in the expression and build the sub-tree (see Figure 6) and finally we get the complete AST as shown in Figure 7. Different colors have been used to denote different types of Node.

### 3.2 Implementation of Visitor Classes

As mentioned in Section 2.1, we are using visitor pattern technique to print and differentiate our equation stored in AST. To implement the visitor pattern we create a Visitor class hierarchy that defines a `virtual visit()` method in the abstract base class for each concrete derived class in the node hierarchy[reference to design patter]. Each visit method accept a single argument pointer to original Node derived class.

We add a `virtual accept(Visitor*)` method

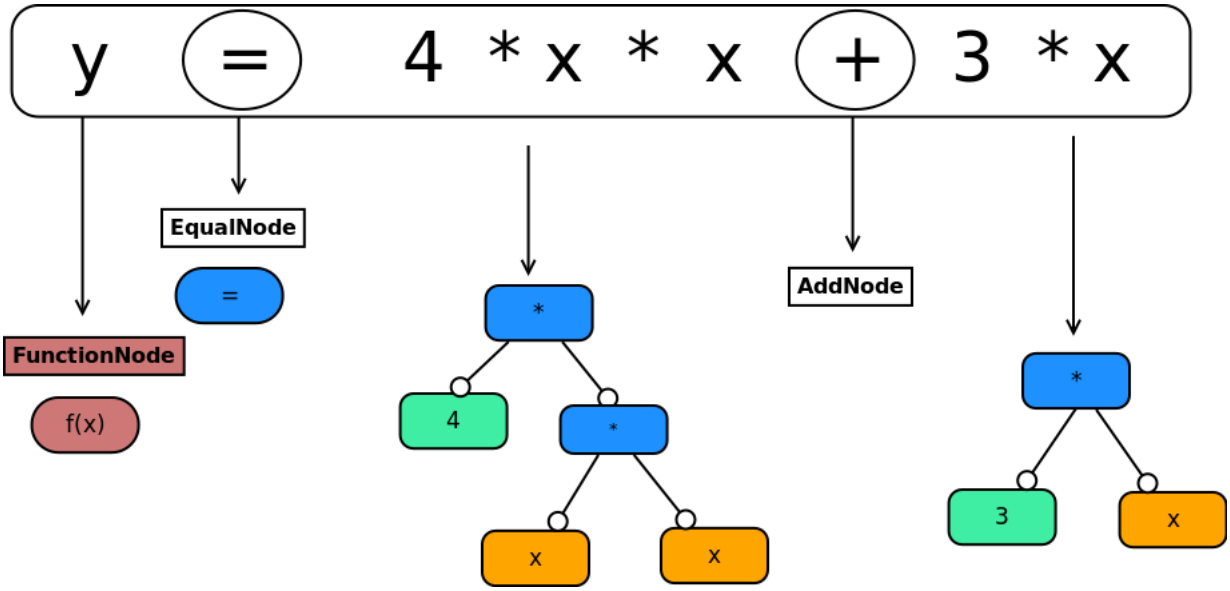


Figure 6: Building AST for $y = 4 * x^2 + 3 * x$

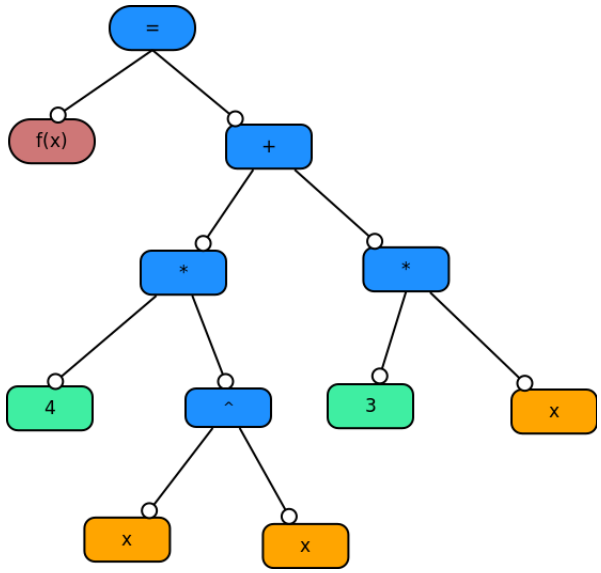


Figure 7: Final AST for $y = 4 * x^2 + 3 * x$. Different types of nodes can be identified from their color.

to the base class of original Node hierarchy (see Section 3.1 for class Node). Pointer to the abstract base class of Visitor hierarchy acts as an the argument for `accept()` method. In our case, we have two derived classes for our parent class Visitor - `PrintVisitor` and `DiffVisitor` for printing and differentiating the equation respectively. Process of differentiation is explained in next section.

Visitor class

```
class Visitor
{
private:
    Node* currentParent = nullptr;
public:
    Visitor()
    virtual void visit(NumberNode*) = 0;
    virtual void visit(VariableNode*)=0;
    virtual void visit(AddNode*)=0;
    virtual void visit(DivisionNode*)=0;
    virtual string getDiffName();
    virtual Node* getDiffTree();
    void setCurrentParent(Node*);
};
```

## Symbolic Differentiation of the Input Equation

Andrea Hanke

Using the class structure described above, the first step is to parse the *Modelica* code and build an AST from the given input equation. Since

the focus of this work is on the symbolic differentiation of a *Modelica* function, and not on the parsing itself, our code has implemented a very simple parser which takes an input of the form $A := B$, where $A$ is the function output, and $B$ is a mathematical expression like $x + 2 * xx/4.23 - x^y 2$. The allowed operators "+", "−", "·", "/", and "" have each to be surrounded by either a variable or a numerical value. Variables have to start with a letter and can also end with one or more numbers. Numerical values can be integers or floating point numbers with a format like 123.456.

Using the Visitor pattern described above, we implemented the derived class `DiffVisitor`. This subclass of `Visitor` visits each node of the AST and - using the following differentiation rules - builds a new AST, which represents the derivative of the input function.

## Differentiation rules for the AST

Every node type has a different differentiation rule. For that reason, the visitor pattern was chosen, as its double dispatch allows a straight forward differentiation between the types of nodes and the types of visitors (differentiation is done by `DiffVisitor`. Let us first consider the simple rules for the three types of leave nodes our code uses: `FunctionNode`, `NumberNode`, and `VariableNode`. Given that we want to differentiate after a variable of `varName` = "x", the rules for the specific nodes are as follows:

**FunctionNode** The `DiffVisitor` takes the `functionName` of the `FunctionNode` (for example "f") and makes a new string (for example "df_dx"):

diff. FunctionNode

```
string newFName = "d";
newFName += functionName;
newFName += "_d";
newFName += varName;
```

Finally, a new `FunctionNode` with `newFName` as its `functionName` is built.

**NumberNode** Since any derivative of a constant is 0, the `DiffVisitor` simply builds a new `NumberNode` with the value 0.

**VariableNode** For this node, the `DiffVisitor` checks, if the name of the variable we want to differentiate after `varName` is the same as the name of the variable stored in the node `variableName`.

diff. VariableNode

```
double newValue=0;
if(varName == variableName)
    newValue = 1;
```

Then, since a derivative of $x$ after $x$ is 1, and a derivative of $x$ after $y$ is 0 (as long as $x! = x(y)$), a new `NumberNode` is built with `newValue` as its value.

The remaining types of nodes (subclasses of `UnaryOperationNode` and `BinaryOperationNode` have somewhat more complicated rules. To be able to differentiate any kind of subtree from one of the unary operations $\langle op \rangle$, the general situation

$$\langle op \rangle (v(x)) \tag{1}$$

has been assumed, to work out the rules corresponding to each operation. The following section works out the differentiation rules for the three `UnaryOperationNode`s, that are currently implemented in our code.

**BracketNode** The simplest of all operation differentiation rules, the differentiation of a `BracketNode`

$$\frac{\mathrm{d}\left(u(x)\right)}{\mathrm{d}x} = \left(\frac{\mathrm{d}u(x)}{\mathrm{d}x}\right) \tag{2}$$

simply demands building a new `BracketNode` and then differentiating its single child. In general, whenever a child is differentiated, at least one new Node is created. In addition to building a new tree structure corresponding to the given Node, `DiffVisitor` also handles setting the correct parent-child relationship for every child it differentiates (i.e. for every new Node it creates).

**ExpNode** Differentiating an expression like $e^{u(x)}$ yields:

$$\frac{\mathrm{d}e^{u(x)}}{\mathrm{d}x} = e^{u(x)} \cdot \frac{\mathrm{d}u(x)}{\mathrm{d}x} \tag{3}$$

Therefore, a copy of the entire original node, including its subtree, is created and set as the left child of a new `MultiplicationNode`. The right child of that `MultiplicationNode` is determined by the differentiation of the orginal child. Take a look at Fig. 8 for a visual representation. Notice that the derivative is not the direct child of the `MultiplicationNode`. Inbetween there is a `BracketNode`, which makes sure that for example in the case of $u(x) = x + 3 * x * x$, the multiplication is applied to the complete derivative. In the following, there will be more additional `BracketNodes` for the same reason, unless otherwise stated.

**LnNode** Differentiating $\ln(u(x))$ yields:

$$\frac{\mathrm{d}\ln(u(x))}{\mathrm{d}x} = \frac{\frac{\mathrm{d}u(x)}{\mathrm{d}x}}{u(x)} \tag{4}$$

So a `DivisionNode` is built with a copy of the original child as its right child and the differentiated child as its left child (see Fig. 8).

Similar to the above calculations, we shall consider the general situation

$$(u(x))\langle op\rangle(v(x)) \tag{5}$$

to calculate the derivative of a binary operation $\langle op\rangle$. The following section works out the differentiation rules for the five `BinaryOperationNodes`, that are currently implemented in our code.

**AddNode** Since

$$\frac{\mathrm{d}u(x) + v(x)}{\mathrm{d}x} = \frac{\mathrm{d}u(x)}{\mathrm{d}x} + \frac{\mathrm{d}v(x)}{\mathrm{d}x} \tag{6}$$

a new `AddNode` is built and then the DiffVisitor visits each child of the original `AddNode` (see also Fig. 9).

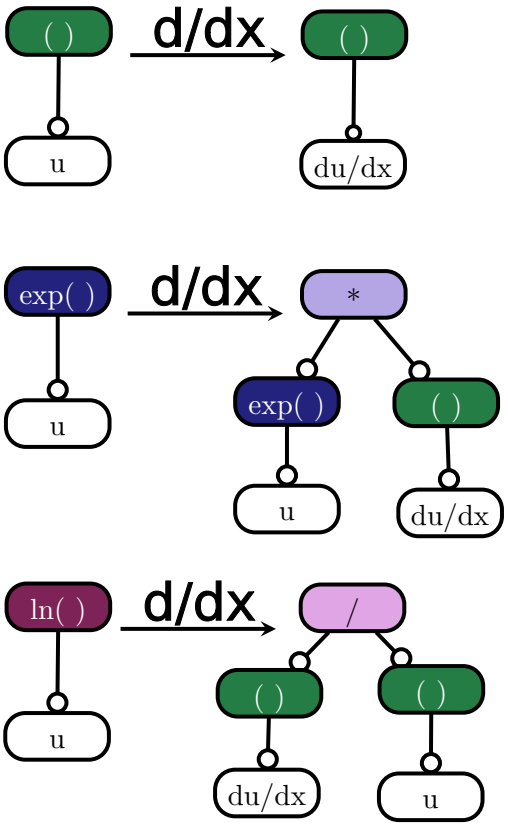


Figure 8: Differentiation of the different unary nodes (from top to bottom: `BracketNode`, `ExponentialNode`, `LnNode`. Each left tree corresponds to the undifferentiated, original AST, each right tree corresponds to the newly built, differentiated AST.

**MinusNode** In principle identical to the `AddNode`, differentiating a `MinusNode` yields:

$$\frac{\mathrm{d}\left(u(x) - v(x)\right)}{\mathrm{d}x} = \frac{\mathrm{d}u(x)}{\mathrm{d}x} - \frac{\mathrm{d}v(x)}{\mathrm{d}x} \tag{7}$$

A new `MinusNode` is built and then the DiffVisitor visits each child of the original `MinusNode` (see also Fig. 9).

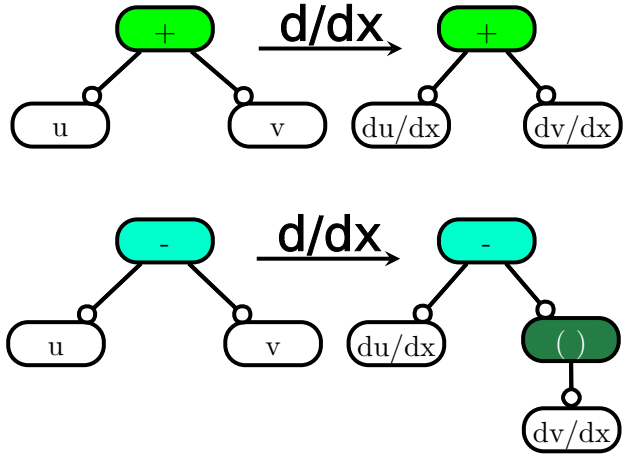


Figure 9: Differentiation of the `PlusNode` (top) and `MinusNode` (bottom). Each left tree corresponds to the undifferentiated, original AST, each right tree corresponds to the newly built, differentiated AST.

**MultiplicationNode** The derivative of a general multiplication comes out to :

$$\frac{\mathrm{d}\left(u(x)\cdot v(x)\right)}{dx} = \frac{du(x)}{dx}\cdot v(x) + u(x)\cdot\frac{dv(x)}{dx} \tag{8}$$

Hence we have to build more new nodes: an `AddNode` and two `MultiplicationNode`s. To each `MultiplicationNode` one of the original, not differentiated child nodes (including all grandchildren) is copied. Then the DiffVisitor visits each child of the original `MultiplicationNode` (see also Fig. 10).

**DivisionNode** A division's derivative is:

$$\frac{d\frac{u(x)}{v(x)}}{dx} = \frac{\frac{du(x)}{dx}\cdot v(x) - u(x)\cdot\frac{dv(x)}{dx}}{v(x)\cdot v(x)} \tag{9}$$

Much like with the derivative of the multiplication, new `BinaryOperationNode`s, and copies of the original child nodes are created. In addition to the previous case, a new `DivisionNode` is created. Its left and right child are new `BracketNode`s, and the left `BracketNode` has essentially the structure of the derivative of the `MultiplicationNode` as its child. The right `BracketNode` gets a new `MultiplicationNode` with two copies of the original left child as kids (as illustrated in Fig. 11).

**ExponentialNode** The derivative of a general exponentiation can be calculated as:

$$\begin{aligned}\frac{\mathrm{d}u(x)^{v(x)}}{\mathrm{d}x} &= \frac{\mathrm{d}e^{\ln(u(x))\cdot v(x)}}{\mathrm{d}x}\\ &= e^{\ln(u(x))\cdot v(x)}\cdot\frac{\mathrm{d}\ln(u(x))\cdot v(x)}{\mathrm{d}x}\\ &= u(x)^{v(x)}\\ &\cdot\left(\frac{\mathrm{d}u(x)}{\mathrm{d}x}\cdot\frac{v(x)}{u(x)} + \ln(u(x))\cdot\frac{\mathrm{d}v(x)}{\mathrm{d}x}\right)\end{aligned} \tag{10}$$

Again, the corresponding tree is created: The original structure of the tree is copied and set as the left child to a new `MultiplicationNode`. A new `BracketNode` is created as the right child of that `MultiplicationNode`. An `AddNode` as the single child to the `BracketNode` and two `MultiplicationNode`s are built, and then the `DiffVisitor` visits each child of the original `AddNode` (see also Fig. 12).

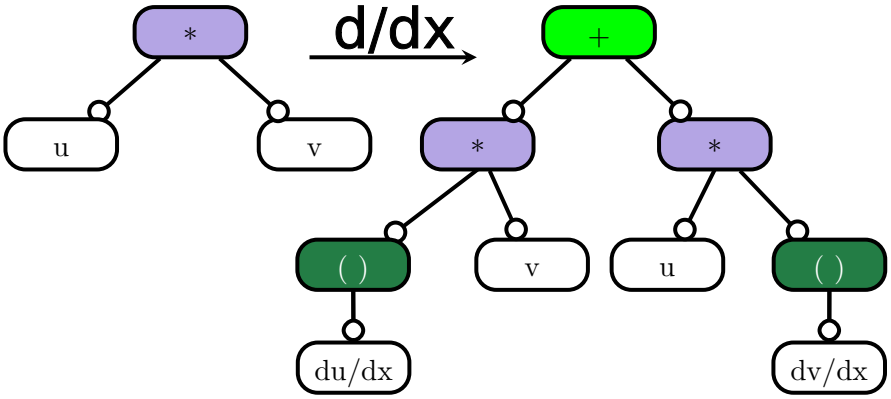


Figure 10: Differentiation of the `MultiplicationNode`. The left tree corresponds to the undifferentiated, original AST, the right tree corresponds to the newly built, differentiated AST.

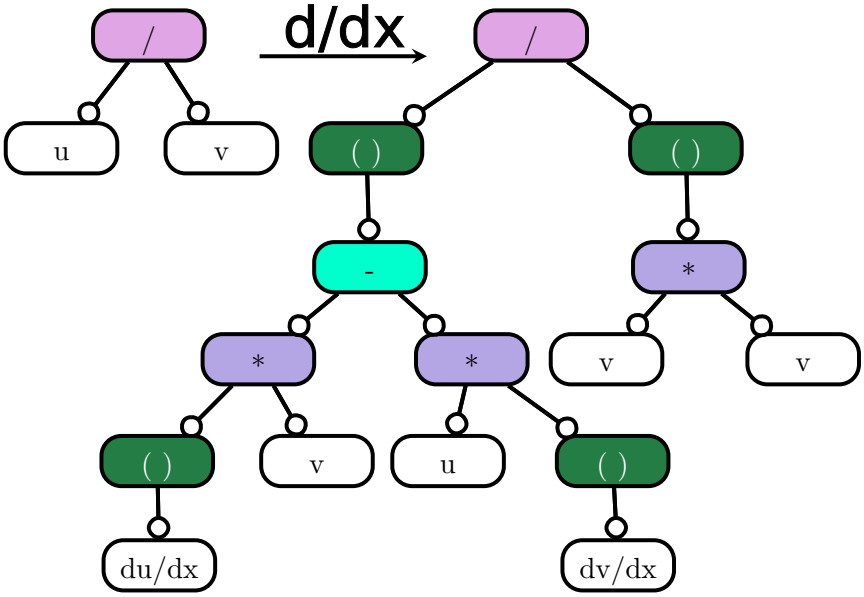


Figure 11: Differentiation of the `DivisionNode`. The left tree corresponds to the undifferentiated, original AST, the right tree corresponds to the newly built, differentiated AST.

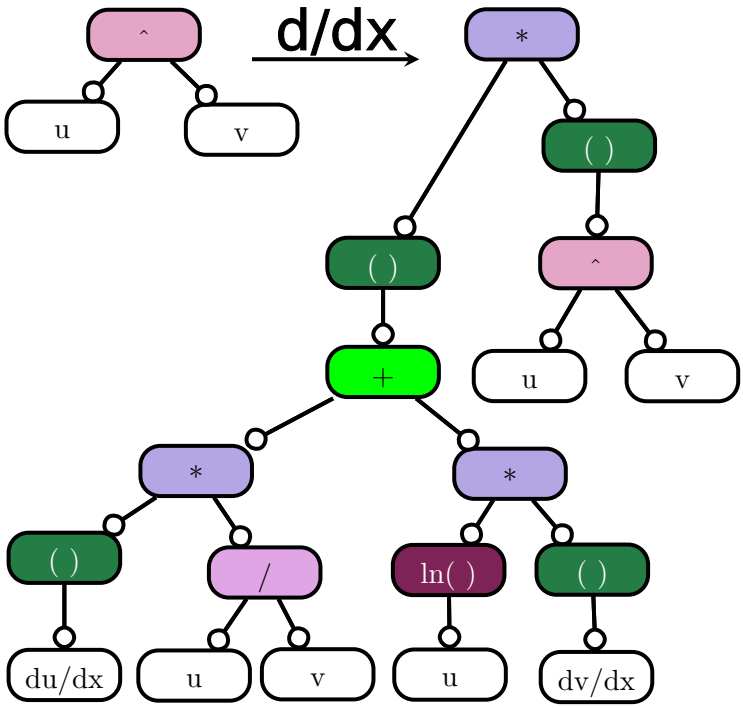


Figure 12: Differentiation of the `ExponentialNode`. The left tree corresponds to the undifferentiated, original AST, the right tree corresponds to the newly built, differentiated AST.

# 4 Generating the C++ Jacobian function

Andrea Hanke

## 4.1 Printing an AST

After differentiating the given input equation symbolically, the second step is to generate the C++ function which calculates the first derivative (i.e. the Jacobian) of the input equation at any given point. For this we implemented a second subclass of the `Visitor` class: the `PrintVisitor`. This visitor starts at the root of the tree (the `EqualNode`), and visits and "prints" the entire tree. In detail, the `PrintVisitor` appends - depending on the node type - different strings or characters to an internal string variable `completeResult`. For example, if the `PrintVisitor` visits a `FunctionNode`, simply the `functionName` is appended to `completeResult`:

Visit and Print a `FunctionNode`

```
void PrintVisitor::visit(
    FunctionNode* functionNode
  )
{
 completeResult
   += functionNode
     ->getFunctionName();
 completeResult += "␣" ;
}
```

Likewise, visiting a `VariableNode` simply appends the `variableName`, and visiting a `NumberNode` appends the `value`. For nodes with children, we usually have to choose a very specific sequence of appending some string and visiting the child node(s), in order to get valid C++ code. For example, if the `PrintVisitor` visits the `UnaryOperationNode BracketNode`, an opening bracket is appended to `completeResult`, the single child visited and then the closing bracket is appended:

Visit and Print a `BracketNode`

```
void PrintVisitor::visit(
    BracketNode* bracketNode
  )
{
 completeResult += "(";
 bracketNode
  ->getChild()->accept(this);
 completeResult += ") ";
}
```

In case of a `BinaryOperationNode`, both child nodes have to be visited. For this the tree is visited in the in-order traversal fashion. For instance, if the `PrintVisitor` visits a `MultiplicationNode`, it first visits the left child, appends `*` to the string and then visits the right child:

Visit and Print a `MultiplicationNode`

```
void PrintVisitor::visit(
    MultiplicationNode* multNode
  )
{
 multNode
  ->getLeftChild()->accept(this);
 completeResult += "* ";
 multNode
  ->getRightChild()->accept(this);
}
```

Note that in order to produce valid C++ code, some visits are not quite as straightforward. For instance, to print an `ExponentialNode`, the C++ function `pow(base, exponent)` has to be used:

Visit and Print an `ExponentialNode`

```
void PrintVisitor::visit(
    ExponentialNode* exponentialNode
  )
{

 completeResult += "pow(";
 exponentialNode
  ->getLeftChild()->accept(this);
 completeResult += ", ";
 exponentialNode
  ->getRightChild()->accept(this);
 completeResult += ")";
}
```

In case of a `BinaryOperationNode`, both child nodes have to be visited. For this the tree is visited in the in-order traversal fashion.

## 4.2 Printing the C++ code of the Jacobian

For a general function $F : \mathbb{R}^m \to \mathbb{R}^n$, the Jacobian is a $m \times n$-matrix. The current version of our code is specific for the case of $n = 1$. To obtain the Jacobian, all different variables of $F$ are kept track of while building the AST from the original *Modelica* input. After the (undifferentiated) AST is generated, the `DiffVisitor` visits that tree and differentiates it for each variable $F$ depends on. The `PrintVisitor` then prints each of those differentiated ASTs, and with some additional text wrapping around that output, a complete C++ function is obtained (see the appendix for the complete code):

Generating the C++ code of the Jacobian (shortened version)

```
string generateCJacobian(
    Node* tree,
    set<string>* setOfVariables,
    string functionName
  )
{

 string result = "void SiSc_Jacobian(";


 set<string>::iterator
   it=setOfVariables->begin();
 //generate function arguments

 result += "){\n";

 for(
     it=setOfVariables->begin();
     it!=setOfVariables->end();
     ++it
   )
```

```
  {
   //diff. after the variable *it
   DiffVisitor* differ
     = new DiffVisitor(*it);
   tree->accept(differ);
   Node* diffTree
     = differ->getDiffTree();
   PrintVisitor* printer
     = new PrintVisitor();
   diffTree->accept(printer);
   result += "\t";
   result += printer->getResult();
   result +=";\n";

   delete differ;
   delete printer;
   delete diffTree;
  }

 result += "}\n";

 }
 //Now, result is a
 //complet C++ function
 return result;
}
```

With this, the step from a *Modelica* function to a C++ function corresponding to the Jacobian of that *Modelica* function is complete. Similarly to the above code snippet, our code also generates the C++ function corresponding to the original *Modelica* function by simply letting a `PrintVisitor` visit the (undifferentiated) AST (look at the appendix for the full code).

Generating the C++ code of the original *Modelica* function (shortened version)

```
string generateCfunction(
   Node* tree,
   set<string>* setOfVariables,
   string functionName
  )
{
 string result
   = "void SiSc_function(double &";
 //functionName is the name
 //of the function of the tree

 result += functionName;

 //generate function arguments
 result += "){\n\t";

 PrintVisitor* printer
   = new PrintVisitor();
 tree->accept(printer);

 result += printer->getResult();

 result +=";\n}";
 delete printer;
 return result;
}
```

## Outlook

Since our code is symbolically differentiating a function, in principle we are able to differentiate up to any order without loss of accuracy. However, with every differentiation the resulting AST becomes larger, because the tree is not simplified. Therefore, in future work the tree and the calculation of the derivative (e.g. checking if the next node is a `NumberNode`) needs to be optimized. Furthermore, this code should be merged with the already existing *Modelica* parser.

## Conclusion

We successfully implemented symbolic differentiation with an abstract syntax tree for equations from *Modelica*. Using object oriented programming and the visitor pattern allowed us to simplify coding the differentiation and printing of the tree. Furthermore, this makes the codes simple to extent, enabling to easily implement possible future extensions. Furthermore, our code generates C++ code, which can calculate the numerical values of both the *Modelica* function and its derivative at any given point.

## References

[1] Modelica and the Modelica Association. 26 Feb. 2018, www.modelica.org/.

[2] Parr,Terence Chapter 4 : Language Implementation Pattern 2009

# Appendix 1

Generating the C++ code of the Jacobian

```
string generateCJacobian(Node* tree, set<string>* setOfVariables,
                         string functionName){

    string result = "void SiSc_Jacobian(";
    int numberOfParameters = setOfVariables->size();
    if(numberOfParameters>0){
         string diffFunctionName = "d";
        diffFunctionName += functionName;
        diffFunctionName += "_d";
        set<string>::iterator it=setOfVariables->begin();
        for ( ; it!=setOfVariables->end(); ++it){

            result += "double &";
            result += diffFunctionName;
            result += *it;
            result += ", ";

        }

        it=setOfVariables->begin();
        result += "const double ";
        result += *it;
        ++it;
        for ( ; it!=setOfVariables->end(); ++it){

            result += ", const double ";
            result += *it;

        }
        result += "){\n";

        for(it=setOfVariables->begin(); it!=setOfVariables->end(); ++it){
            DiffVisitor* differ = new DiffVisitor(*it);
            tree->accept(differ);
            Node* diffTree = differ->getDiffTree();
            PrintVisitor* printer = new PrintVisitor();
            diffTree->accept(printer);
            result += "\t";
            result += printer->getResult();
            result +=";\n";

            delete differ;
            delete printer;
            delete diffTree;
```

```
        }

        result += "}\n";

    }else{
        result += "){\n\t//Function does not depend on Variables!!!\n}\n";
    }
    return result;
}
```

Generating the C++ code of the original *Modelica* function

```
string generateCfunction(Node* tree, set<string>* setOfVariables,
                                string functionName){
    string result = "void SiSc_function(double &";
    //functionName is the name of the function of the tree
    result += functionName;
    for ( set<string>::iterator it=setOfVariables->begin();
            it!=setOfVariables->end(); ++it){

        result += ", const double ";
        result += *it;
    }

    result += "){\n\t";

    PrintVisitor* printer = new PrintVisitor();
    tree->accept(printer);

    result += printer->getResult();

    result +=";\n}";
    delete printer;
    return result;
}
```