Sworn declaration

I hereby declare, under oath, that this homework has been my independent work

and has not been aided with any prohibited means. I declare, to the best of my

knowledge and belief, that all passages taken from published and unpublished

sources or documents have been reproduced whether as original, slightly changed

or in thought, have been mentioned as such at the corresponding places of the

thesis, by citation, where the extent of the original quotes is indicated.

A homework that violates the above statement will be rejected.

This is readme about how to use the Boolean expression solver that I built using Z3 solver.

There are 2 classes of components involved in this:

* The syntax nodes and tree classes which represent the parsed expression
* The actual syntax tree evaluator class called “BooleanExpressionsProver” which takes a “SyntaxTree” class and proves if it’s satisfiable

The possible operations are:

* Addition binary operation represented by the “add” symbol
* Substraction binary operation represented by the “sub” symbol
* Multiplication binary operation represented by the “mul” symbol
* Division binary operation represented by the “div” symbol
* Boolean and binary operation represented by the “and” symbol
* Boolean or binary operation represented by the “or” symbol
* Boolean not unary operation represented by the “not” symbol
* Greater than binary operation represented by “gt” symbol
* Greater or equal than binary operation represented by the “gte” symbol
* Lesser than binary operation represented by the “ls” symbol
* Lesser or equal than binary operation represented by the “lse” symbol
* Minus unary operation represented by the symbol “min”
* Block syntax representing an operator or other expression surrounded by “(“ and “)”
* Any domain unary operation represented by the symbol “any”
* Exists domain unary operation represented by the symbol “exists”

I used words instead of the proper symbols for addition and comparison operations because it was easier to parse them. Too lazy to fix them but they work properly

For example an addition operation will look like “x add y”. A block operator will look like this “(a add b) gt 16”. I also applied the greater than operation “gt”. These operations can be chained so for example a greater than operator can take a block operator as the left argument.

When you pass an expression to the parser it creates a SyntaxTree out of it. It contains the root node. The root node can be a BlockSyntaxNode or any other binary or unary operation.

The syntax nodes types are:

* “LiteralSyntaxNode” which represents mostly a keyword such as the keywords for the operators like “and”, “or”. It can also be a parenthesis.
* “SymbolSyntaxNode” which represents variables in the expression. It has an important property called the “Id” which is used to identify the symbol uniquely
* “BlockExpressionSyntaxNode” which represents another expression enclosed by parentheses “(“ and “)”. Used to change the way the SyntaxTree looks like and how it will be evaluated
* “UnaryExpressionSyntaxNode” which represents a unary operation like Boolean “not” or arithmetic “min” operations
* “BinaryExpressionSyntaxNode” which represents a binary operation like arithmetic operation “add”
* “ConstantValueSyntaxNode” which represents a hardcoded constant value in the expression like a number “1231” or a Boolean value like “true”
* “ValueSyntaxNode” which is represents the base type for all the syntax nodes which return another value like the “add” operation or another “SymbolSyntaxNode”. Has a special property called “DomainValue” which represent the domain of a variable inside the ”ValueSyntaxNode”. A “SymbolSyntaxNode” or a “BlockExpressionSyntaxNode” are derived from “ValueSyntaxNode”
* “DomainValueSyntaxNode” which represents a special node to restrict a variable inside a “ValueSyntaxNode”. It can contain the “any” or “exists” operations

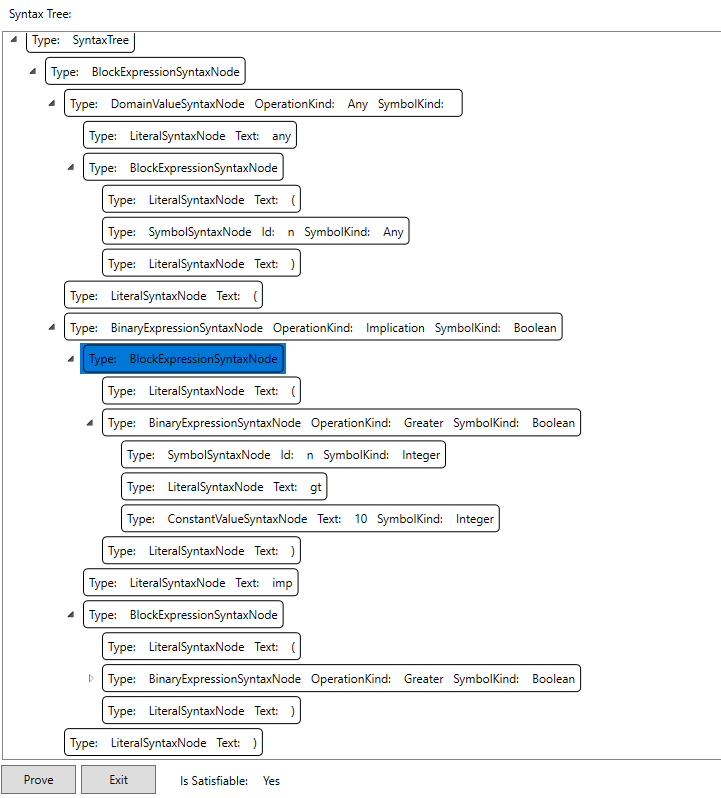
A “SyntaxTree” is built using the nodes mentioned above. The nodes mentioned above can have child nodes. For example a “BinaryExpressionSyntaxNode” has as children the left value, the actual operation symbol or literal called “LiteralSyntaxNode” and the right value.

The left and right values are any kind of syntax nodes derived from “ValueSyntaxNode”

I used the C# compiler tools codenamed “Roslyn” for another project and they had the same kind of names and structure. I got the idea from them but used my own classes and my own parser. The actual C# language has hundreds of syntax nodes. And the parser has 20000 lines in total. The parser is not generated by any sort of tool and is manually written. You can see the individual commits in GitHub: <https://github.com/dotnet/roslyn/blob/master/src/Compilers/CSharp/Portable/Parser/LanguageParser.cs>

Now in the application that I made you can actually see the SyntaxTree structure in the expandable treeview.

For example if we have the expression “any (n) ((n gt 10) imp (n gt 5))” the expanded generated SyntaxTree will appear like this in the user interface:



You can see how a node has children. The nodes that are shifted one step to the right are children from the first node from the top which is shifted less to the right as them.

You can also see a label called “Is Satisfiable” which shows the result of the evaluation of the syntax tree. In case of expression in our example it is satisfiable. If n is greater than 10, then of course n will be also greater than 5.

The “any” and “exists” domain operators are really tricky because they get attached to the first available value expression to their right. To force them to get attached to the proper value expression we need to add a BlockSyntaxNode with parenthesis to the right as we added in the example.

If we have another expression like this: “any (x) x add 5” it will apply the “any” domain operator to the x symbol and not the entire add operation. We need to encompass the “add” operation inside a “BlockSyntaxNode” like this: “any (x) (x add 5)”.

At the end you can see the source files for everything.